

КЛАССИКА COMPUTER SCIENCE

ФИЛОСОФИЯ JAVA



БРЮС ЭККЕЛЬ



4-е полное
издание





Bruce Eckel

Thinking in Java

4th Edition





БРЮС ЭККЕЛЬ

ФИЛОСОФИЯ JAVA

4-е полное
издание



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск

2015

ББК 32.973.2-018.1

УДК 004.3

Э38

- Эккель Б.**
Э38 **Философия Java. 4-е полное изд. — СПб.: Питер, 2015. — 1168 с.: ил. — (Серия «Классика computer science»).**

ISBN 978-5-496-01127-3

Впервые читатель может познакомиться с полной версией этого классического труда, который ранее на русском языке печатался в сокращении. Книга, выдержавшая в оригинале не одно переиздание, за глубокое и поистине философское изложение тонкостей языка Java считается одним из лучших пособий для программистов. Чтобы по-настоящему понять язык Java, необходимо рассматривать его не просто как набор неких команд и операторов, а понять его «философию», подход к решению задач, в сравнении с таковыми в других языках программирования. На этих страницах автор рассказывает об основных проблемах написания кода: в чем их природа и какой подход использует Java в их разрешении. Поэтому обсуждаемые в каждой главе черты языка неразрывно связаны с тем, как они используются для решения определенных задач.

12+ (Для детей старше 12 лет. В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1

УДК 004.3

Права на издание получены по соглашению с Prentice Hall, Inc. Upper Sadle River, New Jersey 07458. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-0131872486 англ.

ISBN 978-5-496-01127-3

© Prentice Hall, Inc., 2006

© Перевод на русский язык ООО Издательство «Питер», 2015

© Издание на русском языке, оформление ООО Издательство «Питер», 2015

Краткое содержание

Предисловие	25
Введение	33
Глава 1. Введение в объекты	40
Глава 2. Все является объектом.....	70
Глава 3. Операторы.....	95
Глава 4. Управляющие конструкции.....	127
Глава 5. Инициализация и завершение	143
Глава 6. Управление доступом.....	186
Глава 7. Повторное использование классов.....	206
Глава 8. Полиморфизм	237
Глава 9. Интерфейсы	263
Глава 10. Внутренние классы	288
Глава 11. Коллекции объектов	323
Глава 12. Обработка ошибок и исключения	365
Глава 13. Строки.....	413
Глава 14. Информация о типах.....	451

Глава 15. Обобщенные типы	501
Глава 16. Массивы	602
Глава 17. Подробнее о контейнерах	636
Глава 18. Система ввода-вывода Java	723
Глава 19. Перечислимые типы.....	811
Глава 20. Аннотации	849
Глава 21. Параллельное выполнение.....	887
Глава 22. Графический интерфейс	1039
Приложение А.....	1157
Приложение Б. Ресурсы	1159

Содержание

Предисловие	25
Java SE5 и SE6	26
Java SE6	27
Четвертое издание	27
Изменения	27
Замечания о дизайне обложки	29
Благодарности	29
Введение	33
Предпосылки	34
Изучение Java	34
Цели	35
Обучение по материалам книги	36
HTML-документация JDK	36
Упражнения	37
Сопроводительные материалы	37
Исходные тексты программ	38
Стандарты оформления кода	39
Ошибки	39
Глава 1. Введение в объекты	40
Развитие абстракции	41
Объект обладает интерфейсом	43
Объект предоставляет услуги	45
Скрытая реализация	46
Повторное использование реализации	47
Наследование	48
Отношение «является» в сравнении с «похоже»	51
Взаимозаменяемые объекты и полиморфизм	52

Однокорневая иерархия.....	55
Контейнеры	56
Параметризованные типы	57
Создание и время жизни объектов.....	58
Обработка исключений: борьба с ошибками	60
Параллельное выполнение.....	61
Java и Интернет	62
Что такое Web?	62
Вычисления «клиент—сервер»	62
Web как гигантский сервер	63
Программирование на стороне клиента	63
Модули расширения	64
Языки сценариев	65
Java.....	66
Альтернативы.....	66
.NET и C#	67
Интернет и интрасети.....	67
Программирование на стороне сервера	68
Резюме.....	69
Глава 2. Все является объектом	70
Для работы с объектами используются ссылки.....	70
Все объекты должны создаваться явно.....	71
Где хранятся данные	72
Особый случай: примитивные типы	73
Числа повышенной точности.....	74
Массивы в Java	74
Объекты никогда не приходится удалять.....	75
Ограничение области действия	75
Область действия объектов.....	76
Создание новых типов данных	76
Поля и методы	77
Значения по умолчанию для полей примитивных типов	78
Методы, аргументы и возвращаемые значения	78
Список аргументов.....	79
Создание программы на Java.....	80
Видимость имен	80
Использование внешних компонентов	81
Ключевое слово static.....	82
Ваша первая программа на Java.....	83
Компиляция и выполнение.....	85
Комментарии и встроенная документация	87
Документация в комментариях	87
Синтаксис.....	88
Встроенный HTML.....	88
Примеры тегов.....	89
@see: ссылка на другие классы.....	89

{@link пакет.класс#член_класса метка}	89
{@docRoot}	90
{@inheritDoc}	90
@version	90
@author	90
@since	90
@param	90
@return	90
@throws	91
@deprecated	91
Пример документации	91
Стиль оформления программ	92
Резюме	92
Упражнения	93
Глава 3. Операторы	95
Простые команды печати	95
Операторы Java	96
Приоритет	96
Присваивание	97
Совмещение имен во время вызова методов	98
Математические операторы	99
Унарные операторы плюс и минус	101
Автоувеличение и автоуменьшение	101
Операторы сравнения	103
Проверка объектов на равенство	103
Логические операторы	104
Ускоренное вычисление	105
Литералы	106
Экспоненциальная запись	108
Поразрядные операторы	109
Операторы сдвига	110
Тернарный оператор	113
Операторы + и += для String	114
Типичные ошибки при использовании операторов	115
Операторы приведения	116
Округление и усечение	117
Повышение	118
В Java отсутствует sizeof	118
Сводка операторов	118
Резюме	126
Глава 4. Управляющие конструкции	127
true и false	127
if-else	127
Циклы	128

do-while.....	129
for	129
Оператор-запятая	131
Синтаксис foreach.....	131
return	133
break и continue	134
Нехорошая команда goto	136
switch.....	140
Резюме	142
Глава 5. Инициализация и завершение	143
Конструктор гарантирует инициализацию	143
Перегрузка методов	145
Различение перегруженных методов.....	147
Перегрузка с примитивами	148
Перегрузка по возвращаемым значениям	151
Конструкторы по умолчанию.....	152
Ключевое слово this	153
Вызов конструкторов из конструкторов	155
Значение ключевого слова static	156
Очистка: финализация и уборка мусора	157
Для чего нужен метод finalize()?	158
Очистка — ваш долг	159
Условие «готовности»	159
Как работает уборщик мусора	161
Инициализация членов класса	164
Явная инициализация	165
Инициализация конструктором.....	167
Порядок инициализации.....	167
Инициализация статических данных	168
Явная инициализация статических членов.....	171
Инициализация нестатических данных экземпляра	172
Инициализация массивов	173
Списки аргументов переменной длины	177
Перечисления	182
Резюме.....	184
Глава 6. Управление доступом	186
Пакет как библиотечный модуль	187
Структура кода	188
Создание уникальных имен пакетов.....	190
Конфликты имен	192
Пользовательские библиотеки.....	193
Использование импортирования для изменения поведения	194
Предостережение при работе с пакетами	195
Спецификаторы доступа Java	195

Доступ в пределах пакета.....	195
public.....	196
Пакет по умолчанию	197
private.....	197
protected	198
Интерфейс и реализация	200
Доступ к классам	201
Резюме	205
Глава 7. Повторное использование классов.....	206
Синтаксис композиции	206
Синтаксис наследования	209
Инициализация базового класса.....	211
Конструкторы с аргументами	212
Делегирование	214
Сочетание композиции и наследования.....	215
Обеспечение правильного завершения.....	217
Соккрытие имен	220
Композиция в сравнении с наследованием	221
protected.....	222
Восходящее преобразование типов.....	224
Почему «восходящее преобразование»?	224
Снова о композиции с наследованием.....	225
Ключевое слово final.....	226
Неизменные данные	226
Пустые константы.....	228
Неизменные аргументы	229
Неизменные методы	229
Спецификаторы final и private.....	230
Неизменные классы	231
Предостережение	232
Инициализация и загрузка классов.....	233
Инициализация с наследованием	234
Резюме	235
Глава 8. Полиморфизм	237
Снова о восходящем преобразовании	238
Потеря типа объекта.....	239
Особенности.....	240
Связывание «метод-вызов».....	240
Получение нужного результата	241
Расширяемость	244
Проблема: «переопределение» закрытых методов.....	247
Проблема: поля и статические методы.....	248
Конструкторы и полиморфизм.....	249
Порядок вызова конструкторов.....	249

Наследование и завершающие действия	251
Поведение полиморфных методов при вызове из конструкторов	256
Ковариантность возвращаемых типов	258
Наследование при проектировании	259
Нисходящее преобразование и динамическое определение типов	260
Резюме	261
Глава 9. Интерфейсы	263
Абстрактные классы и методы	263
Интерфейсы	266
Отделение интерфейса от реализации	270
«Множественное наследование» в Java	274
Расширение интерфейса через наследование	276
Конфликты имен при совмещении интерфейсов	278
Интерфейсы как средство адаптации	279
Поля в интерфейсах	281
Инициализация полей интерфейсов	281
Вложенные интерфейсы	282
Интерфейсы и фабрики	285
Резюме	287
Глава 10. Внутренние классы	288
Создание внутренних классов	288
Ссылка на внешний класс	290
.this и .new	292
Внутренние классы и восходящее преобразование	293
Внутренние классы в методах и областях действия	295
Анонимные внутренние классы	297
Снова о паттерне «Фабричный метод»	301
Вложенные классы	303
Классы внутри интерфейсов	305
Доступ вовне из многократно вложенных классов	306
Внутренние классы: зачем?	306
Замыкания и обратные вызовы	309
Внутренние классы и система управления	311
Наследование от внутренних классов	317
Можно ли переопределить внутренний класс?	318
Локальные внутренние классы	319
Идентификаторы внутренних классов	321
Резюме	322
Глава 11. Коллекции объектов	323
Обобщенные типы и классы, безопасные по отношению к типам	324
Основные концепции	327
Добавление групп элементов	329
Вывод контейнеров	330

List.....	333
Итераторы	336
ListIterator	339
LinkedList	340
Стек	341
Множество	343
Map	346
Очередь	350
PriorityQueue	351
Collection и Iterator	353
Foreach и итераторы	356
Идиома «Метод-Адаптер»	358
Резюме	361
Глава 12. Обработка ошибок и исключения	365
Основные концепции	365
Основные исключения	366
Аргументы исключения	368
Перехват исключений	369
Блок try	369
Обработчики исключений	369
Прерывание и возобновление	370
Создание собственных исключений	370
Вывод информации об исключениях	373
Спецификация исключений	376
Перехват любого типа исключения	377
Трассировка стека	379
Повторное возбуждение исключения	379
Цепочки исключений	382
Стандартные исключения Java	385
Особый случай: RuntimeException	386
Завершение с помощью finally	387
Для чего нужен блок finally?	389
Использование finally при return	391
Ловушка: потерянное исключение	392
Ограничения исключений	394
Конструкторы	397
Отождествление исключений	402
Альтернативные решения	403
Предыстория	404
Перспективы	406
Передача исключений на консоль	408
Преобразование контролируемых исключений в неконтролируемые	408
Рекомендации по использованию исключений	411
Резюме	411

Глава 13. Строки	413
Постоянство строк	413
Перегрузка + и StringBuilder	414
Непреднамеренная рекурсия	418
Операции со строками	419
Форматирование вывода	421
printf()	421
System.out.format()	422
Класс Formatter	422
Форматные спецификаторы	423
Преобразования Formatter	425
String.format()	427
Вывод файла в шестнадцатеричном виде	428
Регулярные выражения	429
Основы	429
Создание регулярных выражений	432
Квантификаторы	433
CharSequence	434
Pattern и Matcher	434
find()	436
Группы	437
start() и end()	438
Флаги шаблонов	440
split()	442
Операции замены	442
reset()	444
Регулярные выражения и ввод-вывод в Java	445
Сканирование ввода	446
Ограничители Scanner	448
Сканирование с использованием регулярных выражений	449
StringTokenizer	450
Резюме	450
Глава 14. Информация о типах	451
Необходимость в динамическом определении типов (RTTI)	451
Объект Class	454
Литералы class	459
Ссылки на обобщенные классы	461
Новый синтаксис приведения типа	463
Проверка перед приведением типов	464
Использование литералов class	470
Динамическая проверка типа	472
Рекурсивный подсчет	473
Зарегистрированные фабрики	475
instanceof и сравнение объектов Class	478

Отражение: динамическая информация о классе	479
Извлечение информации о методах класса	480
Динамические заместители	483
Null-объекты	487
Фиктивные объекты и заглушки	493
Интерфейсы и информация типов	494
Резюме	499
Глава 15. Обобщенные типы	501
Сравнение с C++	502
Простые обобщения	503
Библиотека кортежей	504
Класс стека	507
RandomList	508
Обобщенные интерфейсы	509
Обобщенные методы	512
Использование автоматического определения аргументов-типов	513
Явное указание типа	515
Списки аргументов переменной длины и обобщенные методы	515
Обобщенный метод для использования с генераторами	516
Генератор общего назначения	517
Упрощение использования кортежей	518
Операции с множествами	520
Анонимные внутренние классы	523
Построение сложных моделей	524
Загадка стирания	526
Подход C++	528
Миграционная совместимость	530
Проблема стирания	531
Граничные ситуации	533
Компенсация стирания	536
Создание экземпляров типов	537
Массивы обобщений	540
Ограничения	544
Маски	548
Насколько умен компилятор?	550
Контравариантность	552
Неограниченные маски	555
Фиксация	560
Проблемы	561
Примитивы не могут использоваться как параметры-типы	561
Реализация параметризованных интерфейсов	563
Приведения типа и предупреждения	563
Перегрузка	565
Перехват интерфейса базовым классом	566
Самоограничиваемые типы	567

Необычные рекурсивные обобщения	567
Самоограничение	568
Ковариантность аргументов	570
Динамическая безопасность типов	573
Исключения	574
Примеси	576
Примеси в C++	576
Примеси с использованием интерфейсов	577
Использование паттерна «Декоратор»	579
Примеси и динамические заместители	580
Латентная типизация	582
Компенсация отсутствия латентной типизации	586
Отражение	586
Применение метода к последовательности	587
Если нужный интерфейс отсутствует	590
Моделирование латентной типизации с использованием адаптеров	591
Использование объектов функций как стратегий	594
Резюме	599
Дополнительная литература	601
Глава 16. Массивы	602
Особое отношение к массивам	602
Массивы как полноценные объекты	604
Возврат массива	607
Многомерные массивы	608
Массивы и обобщения	612
Создание тестовых данных	614
Arrays.fill()	614
Генераторы данных	615
Применение генераторов для создания массивов	620
Класс Arrays	624
Копирование массива	624
Сравнение массивов	625
Сравнения элементов массивов	626
Сортировка массива	630
Поиск в отсортированном массиве	631
Резюме	633
Глава 17. Подробнее о контейнерах	636
Полная таксономия контейнеров	636
Заполнение контейнеров	637
Решение с Generator	638
Генераторы Map	640
Использование классов Abstract	643
Функциональность Collection	650

Необязательные операции	653
Неподдерживаемые операции.....	654
Функциональность List.....	656
Set и порядок хранения	659
SortedSet	663
Очереди.....	664
Приоритетные очереди	665
Деки.....	666
Карты (Map).....	667
Производительность.....	669
SortedMap.....	672
LinkedHashMap	673
Хеширование и хеш-коды	674
Понимание hashCode()	677
Хеширование ради скорости.....	680
Переопределение hashCode()	683
Выбор реализации.....	688
Среда тестирования	689
Выбор List.....	693
Опасности микротестов.....	698
Выбор между множествами	700
Выбор между картами.....	701
Факторы, влияющие на производительность HashMap	704
Вспомогательные средства работы с коллекциями.....	705
Сортировка и поиск в списках.....	708
Получение неизменяемых коллекций и карт	710
Синхронизация коллекции или карты.....	711
Срочный отказ.....	712
Удержание ссылок.....	713
WeakHashMap.....	715
Контейнеры Java версий 1.0/1.1	716
Vector и Enumeration	716
Hashtable.....	717
Stack.....	717
BitSet.....	719
Резюме.....	721
Глава 18. Система ввода-вывода Java	723
Класс File.....	724
Получение содержимого каталогов.....	724
Анонимные внутренние классы.....	725
Вспомогательные средства для работы с каталогами	727
Проверка и создание каталогов	732
Ввод и вывод	734
Типы InputStream	734
Типы OutputStream.....	736

Добавление атрибутов и интерфейсов.....	737
Чтение из InputStream с использованием FilterInputStream.....	737
Запись в OutputStream с использованием FilterOutputStream	738
Классы Reader и Writer	739
Источники и приемники данных	740
Изменение поведения потока.....	741
Классы, оставленные без изменений	742
RandomAccessFile: сам по себе	742
Типичное использование потоков ввода-вывода	743
Буферизованное чтение из файла.....	743
Чтение из памяти.....	745
Форматированное чтение из памяти	745
Вывод в файл.....	746
Сокращенная запись для вывода в текстовые файлы.....	747
Сохранение и восстановление данных.....	748
Чтение/запись файлов с произвольным доступом.....	749
Каналы.....	751
Средства чтения и записи файлов.....	751
Чтение двоичных файлов.....	754
Стандартный ввод-вывод	755
Чтение из стандартного потока ввода	755
Замена System.out на PrintWriter	756
Перенаправление стандартного ввода-вывода.....	756
Управление процессами.....	757
Новый ввод-вывод (nio).....	759
Преобразование данных	762
Извлечение примитивов	765
Представления буферов	766
Данные о двух концах	769
Буферы и манипуляция данными.....	770
Подробнее о буферах.....	770
Отображаемые в память файлы.....	774
Производительность	775
Блокировка файлов.....	778
Блокирование части отображаемого файла.....	779
Сжатие данных	780
Простое сжатие в формате GZIP.....	781
Многофайловые архивы ZIP	782
Архивы Java ARchives (файлы JAR)	784
Сериализация объектов.....	786
Поиск класса.....	790
Управление сериализацией	791
Ключевое слово transient	795
Альтернатива для Externalizable.....	796
Версии	799
Долговременное хранение	799

XML.....	805
Предпочтения	807
Резюме.....	809
Глава 19. Перечислимые типы	811
Основные возможности перечислений.....	811
Статическое импортирование и перечисления	812
Добавление методов к перечислению	813
Переопределение методов перечисления.....	814
Перечисления в командах switch	815
Странности values()	816
Реализация, а не наследование.....	818
Случайный выбор.....	819
Использование интерфейсов для организации кода.....	820
Использование EnumSet вместо флагов	824
Использование EnumMap	826
Методы констант	828
Цепочка обязанностей	831
Конечные автоматы.....	835
Множественная диспетчеризация	839
Диспетчеризация с использованием перечислений	842
Использование методов констант.....	844
Диспетчеризация с EnumMap	846
Использование двумерного массива.....	846
Резюме	847
Глава 20. Аннотации.....	849
Базовый синтаксис.....	850
Определение аннотаций.....	850
Мета-аннотации	852
Написание обработчиков аннотаций.....	853
Элементы аннотаций.....	854
Ограничения значений по умолчанию.....	854
Генерирование внешних файлов.....	854
Альтернативные решения	857
Аннотации не поддерживают наследование	858
Реализация обработчика.....	858
Использование apt для обработки аннотаций.....	861
Использование паттерна «Посетитель» с apt	865
Использование аннотаций при модульном тестировании.....	868
Использование @Unit с обобщениями	876
«Семейства» не нужны	877
Реализация @Unit	877
Удаление тестового кода.....	883
Резюме	885

Глава 21. Параллельное выполнение..... 887

Многогранная параллельность.....	889
Ускорение выполнения.....	889
Улучшение структуры кода.....	891
Основы построения многопоточных программ.....	893
Определение задач.....	893
Класс Thread.....	895
Использование Executor.....	896
Возвращение значений из задач.....	899
Ожидание.....	900
Приоритет.....	902
Уступки.....	904
Потоки-демоны.....	904
Разновидности реализации.....	908
Терминология.....	913
Присоединение к потоку.....	914
Чуткие пользовательские интерфейсы.....	915
Группы потоков.....	916
Перехват исключений.....	917
Совместное использование ресурсов.....	919
Некорректный доступ к ресурсам.....	920
Разрешение спора за разделяемые ресурсы.....	922
Синхронизация EvenGenerator.....	925
Использование объектов Lock.....	925
Атомарность и видимость изменений.....	928
Атомарные классы.....	934
Критические секции.....	935
Синхронизация по другим объектам.....	940
Локальная память потоков.....	941
Завершение задач.....	943
Завершение при блокировке.....	946
Состояния потока.....	946
Переход в заблокированное состояние.....	947
Прерывание.....	947
Блокирование по мьютексу.....	952
Проверка прерывания.....	954
Взаимодействие между задачами.....	957
wait() и notifyAll().....	958
Пропущенные сигналы.....	962
notify() и notifyAll().....	963
Производители и потребители.....	965
Явное использование объектов Lock и Condition.....	969
Производители-потребители и очереди.....	971
Очередь BlockingQueue с элементами toast.....	973
Использование каналов для ввода-вывода между потоками.....	975
Взаимная блокировка.....	977

Новые библиотечные компоненты.....	982
CountDownLatch	983
Потоковая безопасность библиотеки	984
CyclicBarrier.....	985
DelayQueue	987
PriorityBlockingQueue.....	989
Управление оранжереей на базе ScheduledExecutor	992
Semaphore	995
Exchanger	998
Моделирование	1000
Модель кассира	1000
Моделирование ресторана.....	1005
Распределение работы	1009
Оптимизация	1014
Сравнение технологий мьютексов.....	1014
Контейнеры без блокировок.....	1021
Вопросы производительности.....	1022
Сравнение реализаций Map.....	1026
Оптимистическая блокировка	1028
ReadWriteLock.....	1030
Активные объекты.....	1032
Резюме.....	1036
Дополнительная литература.....	1038
Глава 22. Графический интерфейс	1039
Апплет	1041
Основы Swing	1042
Вспомогательный класс.....	1044
Создание кнопки.....	1045
Перехват событий.....	1046
Текстовые области.....	1048
Управление расположением компонентов	1050
BorderLayout.....	1050
FlowLayout	1051
GridLayout.....	1052
GridBagLayout	1053
Абсолютное позиционирование	1053
BoxLayout	1053
Лучший вариант?.....	1054
Модель событий библиотеки Swing	1054
Типы событий и слушателей.....	1055
Адаптеры слушателей упрощают задачу.....	1060
Отслеживание нескольких событий	1061
Компоненты Swing.....	1063
Кнопки.....	1064
Группы кнопок.....	1064

Значки.....	1066
Подсказки.....	1067
Текстовые поля.....	1068
Рамки.....	1069
Мини-редактор.....	1070
Флажки.....	1071
Переключатели.....	1073
Раскрывающиеся списки.....	1074
Списки.....	1075
Панель вкладок.....	1076
Окна сообщений.....	1077
Меню.....	1079
Всплывающие меню.....	1084
Рисование.....	1085
Диалоговые окна.....	1089
Диалоговые окна выбора файлов.....	1092
HTML для компонентов Swing.....	1094
Регуляторы и индикаторы выполнения.....	1095
Выбор внешнего вида и поведения программы.....	1096
Деревья, таблицы и буфер обмена.....	1098
JNLP и Java Web Start.....	1098
Параллельное выполнение и Swing.....	1103
Продолжительные задачи.....	1103
Визуальные потоки.....	1110
Визуальное программирование и компоненты JavaBean.....	1112
Что такое компонент JavaBean?.....	1113
Получение информации о компоненте Bean: инструмент Introspector.....	1115
Более сложный компонент Bean.....	1120
Компоненты JavaBean и синхронизация.....	1123
Упаковка компонента Bean.....	1127
Поддержка более сложных компонентов Bean.....	1128
Больше о компонентах Bean.....	1129
Альтернативы для Swing.....	1129
Построение веб-клиентов Flash с использованием Flex.....	1130
Первое приложение Flex.....	1131
Компилирование MXML.....	1132
MXML и ActionScript.....	1133
Контейнеры и элементы управления.....	1133
Эффекты и стили.....	1135
События.....	1136
Связывание с Java.....	1136
Модели данных и связывание данных.....	1139
Построение и развертывание.....	1140
Создание приложений SWT.....	1141
Установка SWT.....	1142
Первое приложение.....	1142

Устранение избыточного кода	1145
Меню	1146
Вкладки, кнопки и события	1147
Графика	1151
Параллельное выполнение в SWT	1152
SWT или Swing?	1155
Резюме	1155
Ресурсы	1156
Приложение А	1157
Приложения, доступные для загрузки	1157
Thinking in C: Foundations for Java	1157
Семинар «Разработка Объектов и Систем»	1158
Приложение Б. Ресурсы	1159
Программные средства	1159
Редакторы и среды разработки	1159
Книги	1160
Анализ и планирование	1161
Python	1163
Список моих книг	1164

*Посвящается
Дон*

Предисловие

Изначально я рассматривал JavaScript как «еще один язык программирования»; во многих отношениях так оно и есть.

Но с течением времени и углублением моих знаний я начал видеть, что исходный замысел этого языка отличался от других языков, которые я видел прежде.

Программирование состоит в управлении сложностью: сложность решаемой проблемы накладывается на сложность машины, на которой она решается. Именно из-за этих трудностей большинство программных проектов завершается неудачей. И до сих пор ни один из языков, которые я знаю, не был смоделирован и создан *прежде всего* для преодоления сложности разработки и сопровождения программ¹. Конечно, многие решения при создании языков были сделаны в расчете на управление сложностью, но при этом всегда находились другие аспекты, достаточно важные, чтобы учитывать это при проектировании языка. Все это неизбежно приводило к тому, что программист рано или поздно заходил в тупик. Например, язык C++ создавался в расчете на продуктивность и обратную совместимость с C (чтобы упростить переход с этого языка на C++). Оба решения, несомненно, полезны и стали одними из причин успеха C++, но также они выявили дополнительные трудности, что не позволило успешно воплотить в жизнь некоторые проекты (конечно, можно винить программистов и руководителей проекта, но если язык в силах помочь в устранении ошибок, почему бы этим не воспользоваться?). Как другой пример подойдет Visual Basic (VB), привязанный к языку BASIC, который изначально не был рассчитан на расширение, из-за чего все расширения языка, созданные для VB, имеют ужасный синтаксис, создающий массу проблем с сопровождением. Язык Perl был основан на awk, sed, grep и других средствах UNIX, которые он должен был заменить, и в результате при работе с Perl программист через какое-то время уже не может разобраться в собственном коде. С другой стороны, C++, VB, Perl и другие языки, подобные Smalltalk, *частично* фокусировались на преодолении трудностей и, как следствие, преуспели в решении определенных типов задач.

Больше всего удивило меня при ознакомлении с Java то, что его создатели, среди прочего, стремились сократить сложность с точки зрения *программиста*. Они словно говорили: «Мы стараемся сократить время и сложность получения надежного кода». На первых порах такое намерение приводило к созданию не очень быстрых программ (хотя со временем ситуация улучшилась), но оно действительно изумительно

¹ Но по моему мнению, язык Python наиболее близок к достижению этой цели (см. www.Python.org).

повлияло на сроки разработки программ; для разработки эквивалентной программы на C++ требуется вдвое больше или еще больше человеко-часов. Уже одно это приводит к экономии колоссальных денег и уймы времени, но Java на этом не останавливается. Творцы языка идут дальше и встраивают поддержку аспектов, которые стали играть важную роль в последнее время (таких, как многозадачность и сетевое программирование), в сам язык или его библиотеки, что упрощает решение этих задач. Наконец, Java энергично берется за действительно сложные проблемы: платформенно-независимые программы, динамическое изменение кода и даже безопасность; каждая из этих проблем может существенно повлиять на время разработки, от простой задержки до непреодолимого препятствия. Таким образом, несмотря на известные загвоздки с производительностью, перспективы Java потрясают: этот язык способен существенно повысить продуктивность нашей работы.

Во всех областях — при создании программ, командной разработке проектов, конструировании пользовательских интерфейсов, запуска программ на разных типах компьютеров, простом написании программ, использующих Интернет, — Java расширяет «пропускную способность» взаимодействий *между людьми*.

Я полагаю, что перегонка туда-сюда большого объема битов не есть главный результат информационной революции; нас ожидает истинный переворот, когда мы сможем с легкостью общаться друг с другом: один на один, в группах и, наконец, всепланетно. Я слышал предположение, что следующей революцией будет появление единого разума, образованного из критической массы людей и взаимосвязей между ними. Java может быть катализатором этой революции, а может и не быть, но по крайней мере вероятность такого влияния заставляет меня чувствовать, что я делаю что-то значимое, пытаюсь обучать этому языку.

Java SE5 и SE6

Это издание книги сильно выиграло от усовершенствований, внесенных в язык Java в пакете, который компания Sun сначала назвала JDK 1.5, затем переименовала в JDK5 или J2SE, и наконец убрала устаревшее обозначение «2» и заменила название на Java SE5. Многие изменения в языке Java SE5 были призваны сделать работу программиста более приятной. Как вы вскоре убедитесь, проектировщики Java не решили всех проблем, но сделали большой шаг в правильном направлении.

Одним из важных усовершенствований этого издания стала полная интеграция новых возможностей Java SE5/6 для их использования в книге. Это означает, что в этой книге мы пошли на довольно смелый шаг; материал совместим только с Java SE5/6, и большая часть приведенного кода не откомпилируется в предыдущих версиях Java. Если вы попытаетесь это сделать, система построения сообщит об ошибке и прекратит работу. Тем не менее я считаю, что преимущества компенсируют все неудобства.

Если же вы почему-либо привязаны к предыдущим версиям Java, я подстраховался, выложив бесплатные электронные версии предыдущих изданий книги на сайте www.MindView.net. По различным причинам я решил не распространять данное издание книги в электронной форме, ограничившись предыдущими изданиями.

Java SE6

Эта книга была грандиозным и трудоемким проектом, и еще до ее завершения вышла бета-версия Java SE6 (кодовое название *mustang*). И хотя в Java SE6 были внесены небольшие второстепенные изменения, которые позволили улучшить некоторые примеры, в основном новшества Java SE6 не отразились на содержимом книги; их главной целью было повышение скорости и возможности библиотек, выходящие за рамки материала.

Код, приведенный в книге, был успешно протестирован в предвыпускной версии Java SE6; вряд ли в официальном выпуске встретятся какие-то изменения, влияющие на материал. Если это все же произойдет, эти изменения будут отражены в исходном тексте программ, доступном на сайте www.MindView.net.

На обложке указано, что эта книга предназначена для Java SE5/6; это означает «написана для Java SE5 с учетом очень значительных изменений, появившихся в языке в этой версии, но материал в равной степени применим к Java SE6».

Четвертое издание

Работая над новым изданием книги, я стремился реализовать в нем все, что узнал с момента выхода последнего издания. Часто эти поучительные уроки позволяли мне исправить какую-нибудь досадную ошибку или просто оживить скучный материал. Нередко в ходе работы над новым изданием у меня появлялись увлекательные новые идеи, а досаду от выявления ошибок затмевала радость открытия и возможность выражения своих идей в более совершенной форме.

Изменения

Компакт-диск, который традиционно прилагался к книге, в этом издании отсутствует. Важная часть этого диска — мультимедийный семинар *Thinking in C* (созданный для MindView Чаком Алисоном) — теперь доступна в виде загружаемой Flash-презентации. Цель этого семинара — подготовка читателей, незнакомых с синтаксисом C, к пониманию материала книги. Хотя в двух главах книги приведено неплохое вводное описание синтаксиса, для неопытных читателей их может оказаться недостаточно. Семинар *Thinking in C* поможет таким читателям подняться на необходимый уровень.

Пришлось, например, полностью переписать главу «Параллельное выполнение» (некогда «Многопоточность»), так чтобы материал соответствовал основным нововведениям Java SE5, но при этом в нем были отражены основные концепции многопоточности. Без такого фундамента понять более сложные вопросы многозадачности очень трудно. Я провел много месяцев в потустороннем мире «многопоточности», и материал, приведенный в конце главы, не только закладывает фундамент, но и прокладывает дорогу в более сложные территории.

Практически для каждой новой возможности Java SE5 в книге появилась отдельная глава, а другие новшества были отражены в изменениях существующего материала.

Мое изучение паттернов проектирования также не стояло на месте, поэтому в книге вы найдете описания новых паттернов.

Структура материала также претерпела серьезные изменения. В основном это объяснялось спецификой учебного процесса вместе с пониманием того, что мое представление о «главах» необходимо пересмотреть. Я был склонен бессознательно считать, что тема должна быть «достаточно большой» для выделения в отдельную главу. Но на семинарах (и особенно при описании паттернов проектирования) оказалось, что участники лучше всего воспринимают материал, если после описания одного паттерна мы немедленно выполняли упражнение, даже если теория излагалась недолго (также выяснилось, что такой темп изложения больше подходит мне как учителю). Итак, в этой версии я постарался разбить главы по темам, не обращая внимания на их объем. Думаю, книга от этого стала только лучше.

Я также осознал всю важность тестирования кода. Если каждый раз при сборке и запуске своей системы вы не выполняете встроенных тестов, то у вас нет возможности определить, насколько надежен ваш код. Специально для примеров данной книги была создана инфраструктура модульного тестирования, позволяющая показывать и проверять результаты работы каждой программы. (Инфраструктура написана на Python; вы найдете ее в загружаемом коде книги по адресу www.MindView.net.) Общие вопросы тестирования рассматриваются в приложении по адресу <http://MindView.net/Books/BetterJava>; на мой взгляд, это основополагающие навыки, которыми должен владеть каждый программист.

Вдобавок я просмотрел все примеры книги и для каждого из них задал себе вопрос: «Почему я написал это так?» Поэтому в большинстве случаев были добавлены некоторые улучшения и исправления, так чтобы в примерах прослеживалась общая тема и они демонстрировали то, что я считаю лучшими практическими приемами написания кода Java (по крайней мере, в материале вводного уровня). Некоторые из уже существовавших примеров были значительно переработаны. Те, которые потеряли свое значение, были удалены, их место заняли новые примеры.

Я получил от читателей очень много прекрасных отзывов о первых трех изданиях книги, и мне это было очень приятно. Однако всегда были и есть жалобы, и по какой-то причине постоянно существует недовольство по поводу того, что «книга очень велика». По-моему, это не очень строгая критика, если «очень много страниц» — ваше единственное замечание. (Оно напомнило мне замечание императора Австрии Моцарту о его композиции: «Очень много нот!» Заметьте, что я никоим образом не сравниваю себя с Моцартом.) Вдобавок, я могу предположить, что подобные замечания исходят от людей, не знакомых еще с «громадностью» самого языка Java и не видевших других книг, посвященных предмету обсуждения. Кроме того, в этом издании я попытался убрать из книги устаревшие части (или по крайней мере те, без которых можно обойтись). Так или иначе, я просмотрел весь материал, удалил все лишнее, включил изменения и улучшил все, что только мог. Я легко расстался со старыми текстами, поскольку этот материал сохранился на сайте (www.MindView.net) в виде свободно доступных первых трех изданий, а также загружаемых приложений.

Для тех, кто все же не удовлетворен размером книги, я действительно приношу свои извинения. Верите вы или нет, но я очень старался уменьшить ее размер.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.

Книга «Философия Java» была написана Брюсом Эккелем достаточно давно, поэтому приведенные автором ссылки на сайты могли измениться. При подготовке русского издания мы постарались оставить в тексте только действующие адреса, но не можем гарантировать, что они не изменятся в дальнейшем.

Даже веб-страничка автора переехала с www.MindView.net на mindviewinc.com, так что не удивляйтесь, если какая-либо ссылка не будет работать. В этом случае попробуйте воспользоваться удобной для вас поисковой системой.

Благодарности

Во-первых, благодарю тех, кто работал со мной на семинарах, организовывал консультации и развивал преподавательский процесс: Дэйва Бартлетта (который немало потрудился над главой 15), Билла Веннерса, Чака Алисона, Джереми Мейера и Джейми Кинга. Я благодарен вам за терпение, которое вы проявили, пока я пытался сформировать команду из таких независимых людей, как мы.

Недавно, несомненно благодаря Интернету, я оказался связан с удивительно большим количеством людей, которые помогали мне в моих изысканиях, как правило, работая из своих домашних офисов. Будь это раньше, мне пришлось бы платить за гигантские офисные пространства, чтобы вместить всех этих людей, но теперь, благодаря Сети, факсам и отчасти телефону, я способен воспользоваться их помощью без особых затрат. В моих попытках научиться «играть в команде» вы мне очень помогли, и я надеюсь и дальше учиться тому, как улучшить свою собственную работу с помощью усилий других людей. Пола Стьюар оказала бесценную помощь, взявшись за мой беспорядочный бизнес и приведя его в порядок (спасибо, что подталкивала меня, Пола, когда я не хотел ничего делать). Джонатан Вилкокс перетряс всю структуру моей корпорации, переворачивая каждый «камешек», под которым мог скрываться «скорпион», и заставил нас пройти через процесс эффективного и юридически верного обустройства компании. Спасибо за вашу заботу и постоянство. Шерилин Кобо (которая нашла Полу) стала экспертом в обработке звука и значительной части мультимедийных курсов, а также бралась за другие задачи. Спасибо за твое упорство в борьбе с неподатливыми компьютерными делами. Ребята из Amaio (Прага) помогли мне в нескольких проектах. Дэниел Уилл-Харрис был вдохновителем работы через Интернет, и конечно же, он сыграл главную роль во всех моих решениях относительно дизайна.

За прошедшие годы Джеральд Вайнберг стал моим неофициальным учителем и наставником; я благодарен ему за это.

Эрвин Варга оказал огромную помощь с технической правкой 4-го издания — хотя другие участники помогали в работе над разными главами и примерами, Эрвин был главным техническим рецензентом книги; кроме того, он взялся за переработку руководства по решениям для 4-го издания. Эрвин нашел ошибки и сделал немало бесценных усовершенствований. Его скрупулезность и внимание к подробностям просто поражают. Безусловно, это лучший технический рецензент, с которым я когда-либо работал. Спасибо, Эрвин.

Мой блог на сайте Билла Веннерса www.Artima.com приносил много полезной информации, когда мне требовалось «обкатать» ту или иную идею. Спасибо читателям, чьи комментарии помогали прояснять описанные концепции, — в том числе Джеймсу Уотсону, Говарду Ловатту, Майклу Баркеру и другим (и особенно тем, кто помогал с обобщенными типами).

Спасибо Марку Уэлшу за неустанную поддержку.

Эван Кофски, как и прежде, способен сходу разъяснить многие заумные подробности настройки и сопровождения веб-серверов на базе Linux. Кроме того, он помогает в оптимизации и настройке безопасности сервера MindView.

Кафетерий Camp4 Coffee в Crested Butte, Колорадо, стал настоящим местом встреч во время преподавания семинаров. Пожалуй, это самая лучшая закусочная из всех, что я когда-либо видел. Спасибо моему приятелю Алу Смиту за то, что он создал это замечательное место, весьма интересно и приятно вспоминать его. И спасибо всем приветливым бариста Camp4.

Некоторые инструменты с открытыми текстами стали просто незаменимыми для меня в процессе разработки, и каждый раз, когда я их использую, я благодарю разработчиков. Cygwin (<http://www.cygwin.com>) решил бесчисленные проблемы, которые система Windows не могла или не хотела решить, и с каждым днем я все больше привязываюсь к нему (был бы у меня этот инструмент лет 15 назад, когда мой мозг был забит хитростями Gnu Emacs). Eclipse от IBM (<http://www.eclipse.org>) — чудесный подарок сообществу разработчиков, и я ожидаю от него много хорошего по мере его развития (когда это IBM вошла в моду? Кажется, я что-то упустил в этой жизни). JetBrains IntelliJ Idea продолжает прокладывать новые творческие пути в области инструментария разработчика.

В этом издании книги я начал использовать Enterprise Architect от Sparxsystems, и эта система стала моим любимым инструментом для работы с UML. Код системы форматирования кода Jalopy Марко Хансикера (www.triemax.com) неоднократно пригодился мне во время работы, а Марко помог приспособить его для моих специфических потребностей. Также в некоторых ситуациях пригодился созданный Славой Пстовым редактор JEdit и его плагины (www.jedit.org) — это неплохой редактор начального уровня для семинаров.

И конечно же, если я не наскучил напоминанием об этом во всех остальных местах, для решения своих задач я постоянно использовал Python (www.Python.org), детище моего приятеля Гвидо Ван Россума и его эксцентричных гениев. Я провел с ними несколько великолепных дней, занимаясь экстремальным программированием (Тим Питерс, я поместил ту мышку, что ты мне одолжил, в рамку, у нее есть официальное имя «Мышка

Тима»). Вам, ребята, надо поискать более приличные места для еды. (Спасибо также всему сообществу Python, совершенно изумительные люди.)

Множество людей посылало мне исправления для книги, и я признателен всем вам, но в особенности хочу поблагодарить (относительно первого издания): Кевина Раулersona (нашел тонны замечательных ошибок), Боба Резендеса (просто бесподобно), Джона Пинто, Джо Данте, Джо Шарпа (все трое были великолепны), Дэвида Комбса (множество грамматических исправлений), д-ра Роберта Стивенсона, Джона Кука, Франклина Чена, Зева Гринера, Дэвида Карра, Леандер Строшейн, Стива Кларка, Чарльза Ли, Остина Магера, Денниса Рота, Рок Оливейра, Дугласа Данна, Дейян Ристик, Нейл Галарно, Дэвида Мальковски, Стива Вилкинсона и многих других. Профессор Марк Мюрренс предпринял большие усилия для того, чтобы опубликовать и сделать доступной электронную версию первого издания книги в Европе.

Спасибо всем, кто помог мне переписать примеры для библиотеки Swing (для второго издания), и всем остальным: Джону Шварцу, Томасу Киршу, Рахиму Адаше, Раджишу Джайн, Рави Мантене, Бану Раджамани, Йенсу Брандту, Нитину Шираваму, Малькольму Дэвису и всем выразившим мне поддержку.

В 4-м издании Крис Гриндстафф сильно помог с разработкой раздела, посвященного SWT, а Шон Невилл написал первый проект материалов по Flex.

Крейг Брокшимдт и Ген Кийока — очень умные люди, которые стали моими друзьями и оказали на меня необычное влияние, поскольку они занимаются йогой и практикуют некоторые другие способы духовного развития — на мой взгляд, очень поучительные и воодушевляющие.

Для меня не удивительно, что понимание Delphi помогло мне понять Java, так как в обоих заложено много общих идей и решений. Мои друзья, специалисты по Delphi, помогли мне окунуться в мир этого прекрасного средства разработки программ. Это Марко Канту (еще один итальянец — может быть, это как-то связано со способностями к языкам программирования?), Нейл Рубенкинг (который был вегетарианцем и занимался дзеном и йогой, но только до тех пор, пока не увлекся компьютерами) и, конечно же, Зак Урлокер (администратор продуктов Delphi), давний приятель, с которым мы объездили весь свет. И конечно, все мы в долгу перед выдающимися способностями Андерса Хейлсберга, который продолжает трудиться над C# (этот язык, как вы узнаете, стал одним из источников вдохновения для создания Java SE5).

Поддержка и понимание моего друга Ричарда Хейла Шоу были очень нужны. (И Ким также.) Ричард вместе со мной проводил множество семинаров, и мы старались выработать идеальную обучающую программу для наших учеников.

Дизайн книги, обложки и фотографий выполнен моим другом Дэниэлем Уилл-Харрисом, известным автором и дизайнером (www.Will-Harris.com), который в школе делал надписи из переводных картинок, пока еще не изобрели компьютеры и настольные издательские системы, а также все время жаловался, что я бормочу трудясь над алгеброй. Впрочем, оригинал-макеты для печати я подготавливал сам, поэтому все типографские ошибки на моей совести. Для написания книги был использован Microsoft Word XP; для типографии текст готовился в Adobe Acrobat, печаталась книга прямо с файлов PDF. В качестве дани электронному веку финальные версии первого и второго изданий

книги были завершены за границей и посланы издателю по электронной почте — первое издание я закончил в Кейптауне, Южная Африка, а второе издание отправил из Праги. Третье и четвертое издания были закончены в Колорадо. Для основного текста использовалась гарнитура *Georgia*, для заголовков *Verdana*. Текст на обложке использует гарнитуру *ITC Rennie Mackintosh*.

Особые благодарности всем моим учителям и всем моим студентам (которые тоже учат меня).

Кошка Молли часто сидела у меня на коленях, пока я работал над книгой, тоже предоставляя свою поддержку — теплую и пушистую.

Ну и наконец друзья, которые так или иначе помогли мне: Эндрю Бинсток, Стив Синовски, Дж. Д. Гильдебрандт, Том Кеффер, Брайан Макэлхинни, Бринкли Барр, Билл Гейтс из журнала *Midnight Engineering Magazine*, Ларри Константайн и Люси Локвуд, Джин Вонг, Дэйв Мейер, Дэвид Интерсимон, Крис и Лора Странд, Алмквисты, Брэд Джервик, Мэрилин Цвитаник, Марк Мабри, семья Роббинсов, семья Мэлтэров (и Макмилланов), Майкл Вилк, Дэйв Стонер, Крэнстоны, Ларри Фогг, Майк и Карен Секвера, Гэри Эстмингер, Кевин и Сонда Донован, Честер и Шэннон Андерсены, Джо Лорди, Дэйв и Бренда Барлетты, Патти Гаст, Блейк, Аннет и Джейд, Рентшлерсы, Судексы, Дик, Патти и Ли Эккель, Линн и Тодд и их семьи. И конечно, спасибо маме и папе.

Введение

И человек по воле Прометея
Заговорил, и речь рождала мысль,
А мыслью измеряется весь мир...

*Перси Биши Шелли.
«Прометей освобожденный»*

Люди... во многом зависят от определенного языка, ставшего средством выражения в их обществе. Наивно было бы полагать, что человек способен приспособиться к реальности, не используя языка, и что этот язык представляет собой просто подручное средство решения каких-то проблем общения и мышления. На самом деле оказывается, что «реальный мир» по большей части бессознательно построен на языковых пристрастиях группы.

*Эдвард Сепир. «Статус лингвистики
как науки», 1929 г.*

Подобно любому человеческому языку, Java предоставляет способ выражения понятий и идей. Если способ был выбран успешно, то с ростом масштабов и сложности проблем он будет существенно превосходить другие способы по гибкости и простоте.

Язык Java не может рассматриваться как простая совокупность функциональных возможностей — некоторые из них ничего не значат в отдельности. Получить представление о целом как о совокупности частей можно только при рассмотрении *архитектуры*, а не при простом написании кода. И чтобы понять Java в этом смысле, необходимо проникнуться его задачами и задачами программирования в целом. В этой книге мы рассмотрим проблемы программирования, а также разберемся, почему они стали проблемами и какой подход использует Java в их решении. Поэтому раскрываемые в каждой главе возможности языка неразрывно связаны с тем, как этим языком решаются определенные задачи. Таким образом я надеюсь понемногу приблизить вас к тому, чтобы «менталитет Java» стал для вас естественным.

Я постарался помочь вам построить некую внутреннюю модель, которая бы помогла глубже понять язык; столкнувшись с какой-то головоломкой, вы подаете ее на вход своей модели языка и быстро получаете ответ.

Предпосылки

Предполагается, что читатель уже обладает определенным опытом программирования: он понимает, что программа представляет собой набор команд; имеет представление о концепциях подпрограммы/функции/макροопределения; управляющих командах (например, `if`) и циклических конструкциях типа «`while`» и т. п. Обо всем этом вы легко могли узнать из многих источников — программируя на макроязыке или работая с таким инструментом, как Perl. Если вы уже имеете достаточно опыта и не испытываете затруднений в понимании основных понятий программирования, то сможете работать с этой книгой. Конечно, книга будет *проще* для тех, кто использовал язык C и особенно C++; если вы незнакомы с этими языками, это не значит, что книга вам не подходит — однако приготовьтесь основательно поработать (мультимедийный семинар, который можно загрузить с сайта www.MindView.net, поможет быстро освоить основные понятия Java). Но вместе с тем, начну я с основных концепций и понятия объектно-ориентированного программирования (ООП) и базовых управляющих механизмов Java.

Несмотря на частые упоминания возможностей языков C и C++, они не являются неразрывной частью книги — скорее, они предназначены для того, чтобы помочь всем программистам увидеть связь Java с этими языками — от которых, в конце концов, и произошел язык Java. Я попытаюсь сделать эти связки проще и объяснять подробнее то, что незнакомый с C/C++ программист может не понять.

Изучение Java

Примерно в то самое время, когда вышла моя первая книга *Using C++* (Osborne/McGraw-Hill, 1989), я начал обучать этому языку. Преподавание языков программирования стало моей профессией; с 1987 года я видел немало кивающих голов и озадаченных выражений лиц во всех аудиториях мира. Но когда я начал давать корпоративные уроки для небольших групп, я кое-что понял. Даже те ученики, которые усердно кивали и улыбались, многое не понимали до конца. Я обнаружил, будучи несколько лет во главе направления C++ на конференции разработчиков программного обеспечения (а позднее создав и возглавив Java-направление), что я и другие преподаватели имеют обыкновение слишком быстро выдавать аудитории слишком большое число тем. Со временем из-за разницы в уровне подготовки аудитории и способа изложения материала я бы потерял немало учеников. Возможно, это чересчур, но так как я являюсь одним из убежденных противников традиционного чтения лекций (у большинства, я полагаю, это неприятие возникает из-за скуки), я хотел, чтобы за мной успевали все обучающиеся.

Какое-то время мне пришлось создать несколько разных презентаций за достаточно короткое время. Таким образом мне пришлось учиться посредством экспериментов и повторения (метод, хорошо работающий и при проектировании Java-программ). В конце концов я разработал курс, использующий все, что я вынес из своего преподавательского опыта. Моя компания MindView, Inc. сейчас проводит этот курс в форме открытых и корпоративных семинаров *Thinking In Java*; это наш основной вводный курс, подготавливающий основу для более сложных семинаров. Подробности можно узнать по адресу www.MindView.com. (Семинар для начинающих также доступен на отдельном компакт-диске. Информацию о нем можно найти по вышеуказанному адресу.)

Обратная связь, получаемая мной после проведения семинаров, помогает мне изменять и перерабатывать материал до тех пор, пока он не станет хорошим учебным пособием. Но эта книга не является просто собранием заметок с семинаров — я старался поместить сюда как можно больше информации и устроить ее таким образом, чтобы переход к следующей теме происходил по возможности логично. Более всего книга рассчитана на отдельного читателя, столкнувшегося с трудностями нового языка.

Цели

Подобно моей предыдущей книге *Философия C++*, при планировании этой книги я прежде всего ориентировался на метод изучения языка. Размышляя о каждой главе, я думаю о том, какие темы будут эффективно работать на усвоение материала во время семинара. Мнение аудитории помогает мне выявить трудные моменты, которые следует изложить более подробно.

В каждой главе я пытаюсь описать одну возможность (или небольшую группу логически связанных возможностей), не полагаясь на понятия, которые еще не упоминались ранее. Это позволяет усвоить каждую тему в контексте текущих знаний читателя, прежде чем двигаться дальше.

В данной книге я пытаюсь достичь следующего.

1. Последовательно представлять материал, чтобы читатель мог усвоить каждое понятие перед тем, как продолжить изучение. Я тщательно слежу за порядком изложения, чтобы каждая возможность была рассмотрена до того, как она будет использоваться в программе. Конечно, это не всегда удается; в таких ситуациях приводится краткое вводное описание.
2. Использовать примеры как можно более простые и наглядные. Иногда это мешает быть ближе к проблемам «реального» мира, но я обнаружил, что начинающим лучше, когда они понимают пример во всех подробностях, а не восхищаются грандиозностью решаемой задач. Также существуют достаточно строгие ограничения по объему кода, который можно усвоить в классе. Несомненно, меня будут критиковать за «игрушечные примеры», но я обращаюсь к ним в надежде на получение педагогического эффекта.
3. Рассказывать то, что на мой взгляд действительно важно для понимания языка, а не все, что я знаю. Я полагаю, что существует различная по важности информация и в

ее иерархии есть некоторые факты, которые 95 процентам программистов никогда не понадобятся и только путают людей, усложняя восприятие нового предмета. Возьмем пример из C: если вы помните таблицу приоритета операций (я не помню), то способны писать хитроумный код. Но если на то пошло, такой код только запутает программиста, которому приходится его читать или заниматься его сопровождением. Поэтому лучше забыть о приоритете и использовать круглые скобки там, где ситуация не настолько очевидна.

4. Постараться вести изложение целенаправленно, чтобы время лекции — и время между упражнениями — было малым. Это не только способствует активности и свежести восприятия аудитории на семинарах, но и позволяет читателю испытать удовольствие от выполненного дела после прочтения каждой части.
5. Сформировать прочную основу для того, чтобы читатель достаточно хорошо понял материал и смог продолжить изучение с помощью более трудных книг и курсов.

Обучение по материалам книги

Первое издание книги было создано на базе недельного семинара, которого в эпоху становления Java было достаточно для описания всего языка. Со временем язык Java рос, включая все больше возможностей и библиотеки, а я упрямо пытался изложить все за одну неделю. В какой-то момент клиент попросил меня ограничиться изложением «азов»; выполняя его просьбу, я обнаружил, что попытка втиснуть все в одну неделю только усложняет жизнь мне и посетителям семинаров. Язык Java перестал быть «простым» языком, которому можно научить за неделю.

Это осознание привело к значительной реорганизации книги, которая сейчас рассчитана на проведение двухнедельного семинара или курса обучения, рассчитанного на два семестра. Вводная часть завершается главой 9, но ее также можно дополнить знакомством с JDBC, сервлетами и JSP. Этот подготовительный курс составляет основу материалов компакт-диска *Hands-On Java*. В оставшейся части книги приводится материал промежуточного уровня, представленный на компакт-диске *Intermediate Thinking in Java*. Оба диска можно приобрести на сайте www.MindView.net.

HTML-документация JDK

Язык Java и его библиотеки от компании Sun Microsystems (бесплатно загружаемые по адресу <http://java.sun.com>) предоставляются с документацией в электронном виде, которая просматривается из любого браузера. Почти все книги, посвященные Java, повторяют эту документацию. Так как такая документация у вас уже имеется или вы можете загрузить ее, в книге она повторяться не будет (за исключением необходимых случаев), поскольку найти описание класса можно гораздо быстрее с помощью браузера, нежели листать в его поиске книгу (и потом, интерактивная документация обычно более свежая). Чаше вы увидите просто ссылку на «документацию JDK». Книга будет

предоставлять дополнительные описания для классов только в том случае, если это необходимо для понимания какого-то примера.

Упражнения

Я обнаружил, что простые упражнения исключительно полезны для полного понимания темы обучающимся, поэтому набор таких упражнений приводится в конце каждой главы.

Большинство упражнений спланированы так, чтобы их можно было пройти за разумный промежуток времени в классе, в то время как преподаватель будет наблюдать за их выполнением, проверяя, все ли студенты усваивают материал. Некоторые из них нетривиальны, хотя ни одно упражнение не создает особых трудностей.

Решения к отдельным упражнениям можно найти в электронном документе *The Thinking In Java Annotated Solution Guide*, доступном за небольшую плату на сайте www.MindView.net.

Сопроводительные материалы

Еще одним преимуществом данной книги является бесплатный мультимедийный семинар, который можно загрузить на сайте www.MindView.net. Это семинар *Thinking In C*, предлагающий введение в синтаксис, операторы и функции языка C, на котором основан синтаксис Java. В предыдущих изданиях книги этот семинар входил в состав компакт-диска *Foundations For Java*, но теперь этот семинар доступен для бесплатной загрузки. Дополнительно он включает первые семь лекций из второго издания семинара *Hands-On Java*.

Поначалу я задумывал поручить Чаку Аллисону создать семинар *Thinking In C* в виде отдельного продукта, но потом решил включить его во второе издание *Thinking In C++* и второе и третье издания *Thinking In Java*, так как на семинарах слишком часто появлялись люди, не обладающие достаточной подготовкой C. Видимо, они думали так: «Я буду классным программистом, что мне этот C, а лучше возьмусь за C++ или Java; нечего тратить время на C, лучше начну прямо с C++/Java». Посидев на занятиях, люди постепенно начинают открывать для себя тот факт, что требования к знанию синтаксиса C включены здесь совсем не напрасно.

Технологии изменились, и теперь *Thinking in C* стало разумнее оформить в виде презентации (вместо компакт-диска). Размещение семинара в Интернете помогает гарантировать, что все учащиеся будут обладать необходимой подготовкой.

Семинар *Thinking in C* позволит книге охватить большую аудиторию. Несмотря на то что в главе 3 рассматриваются фундаментальные элементы Java, заимствованные из C, семинар излагает материал не так быстро и выдвигает еще меньше требований к навыкам студента, чем книга.

Исходные тексты программ

Все исходные тексты программ для этой книги распространяются свободно, единым пакетом с сайта www.MindView.net, и защищены авторским правом. Этот сайт является официальным распространителем самых свежих версий; воспользуйтесь им для того, чтобы быть уверенным в том, что вы имеете последние варианты программ и электронной версии книги. Также существуют зеркальные копии электронной книги и кода на других сайтах (список можно найти на официальном сервере). Исходные тексты можно распространять в учебных классах и в иных образовательных целях.

Основная цель авторского права — обеспечить наличие необходимых ссылок на источник кода и предотвратить публикацию кода без разрешения. (Пока упоминается источник исходного текста, использование его в большинстве ситуаций не является проблемой.)

В каждом файле с исходным текстом вы найдете следующее замечание об авторском праве:

//:! Copyright.txt

Этот компьютерный исходный текст принадлежит компании MindView Inc. ©2006. Все права защищены.

Этим предоставляется разрешение использовать, копировать, изменять и распространять этот компьютерный исходный текст (Исходный Текст) и его документацию без платы и без письменного соглашения для целей, сформулированных ниже, при условии, что вышеупомянутое объявление об авторском праве, этот параграф и следующие пять пронумерованных параграфов появляются во всех копиях.

1. Разрешение дает право компилировать Исходный Текст и включать скомпилированный код исключительно в исполняемом формате в персональные и коммерческие программы.

2. Разрешение дает право использовать Исходный Текст без модификации в учебных ситуациях (классы), включая и материалы презентаций, при условии, что книга "Thinking In Java" указывается как первоисточник.

3. Разрешение на использование Исходных Текстов в печатных средствах информации можно получить, обратившись по адресу:

MindView, Inc. 5343 Valle Vista La Mesa, California 91941

Wayne@MindView.net

4. Исходный текст и документация защищены авторским правом компании MindView, Inc. Исходный текст предоставляется без явной или неявной гарантии любого вида, включая гарантию коммерческого применения, пригодность для индивидуального использования или ущемления чьих-либо прав. Компания MindView, Inc. не гарантирует, что работа любой программы, которая включает Исходный Текст, будет устойчивой или свободной от ошибок. Компания MindView не делает никаких предположений относительно пригодности Исходного Текста или любого программного обеспечения, которое включает в себя Исходный Текст, для любой цели. Полный риск относительно качества и эффективности любой программы, которая включает Исходный Текст, лежит на пользователе Исходного Текста. Пользователь понимает, что Исходный текст был разработан для исследовательских и учебных целей и ему советуется не полагаться во всех возможных случаях на Исходный Текст или на любую программу, которая включает Исходный текст. Если Исходный текст или любое созданное с его помощью программное обеспечение оказываются дефектными, пользователь берет стоимость всего необходимого обслуживания, ремонта и возмещения ущерба на себя.

5. В ЛЮБОМ СЛУЧАЕ КОМПАНИЯ MINDVIEW ИЛИ ЕЕ ИЗДАТЕЛЬ НЕ ОБЯЗАНЫ ВОЗМЕЩАТЬ УБЫТКИ ЛЮБОЙ СТОРОНЕ СОГЛАСНО ЛЮБОЙ ЮРИДИЧЕСКОЙ ТЕОРИИ В РЕЗУЛЬТАТЕ ПРЯМОГО, КОСВЕННОГО, СПЕЦИАЛЬНОГО, НЕПРЕДВИДЕННОГО ИЛИ ПОСЛЕДУЮЩЕГО УЩЕРБА, ВКЛЮЧАЯ ПОТЕРЯННУЮ ПРИБЫЛЬ, СРЫВ БИЗНЕСА, ПОТЕРЮ ДЕЛОВОЙ ИНФОРМАЦИИ ИЛИ ЛЮБУЮ ДРУГУЮ ДЕНЕЖНУЮ ПОТЕРЮ ИЛИ ПЕРСОНАЛЬНЫЙ

INC. ОТКАЗЫВАЕТСЯ ОТ ЛЮБЫХ ГАРАНТИЙ, ВКЛЮЧАЯ, НО НЕ ОГРАНИЧИВАЯСЬ ИМИ, ГАРАНТИИ ДЛЯ КОММЕРЧЕСКОГО И ИНДИВИДУАЛЬНОГО ИСПОЛЬЗОВАНИЯ. ИСХОДНЫЙ ТЕКСТ И ДОКУМЕНТАЦИЯ, ПРЕДСТАВЛЕННЫЕ ЗДЕСЬ, ПРЕДОСТАВЛЯЮТСЯ "КАК ЕСТЬ", БЕЗ ЛЮБЫХ СОПРОВОДИТЕЛЬНЫХ УСЛУГ ОТ КОМПАНИИ MINDVIEW, И КОМПАНИЯ MINDVIEW НЕ ИМЕЕТ НИКАКИХ ОБЯЗАТЕЛЬСТВ В ОТНОШЕНИИ ИХ ОБСЛУЖИВАНИЯ, ПОДДЕРЖКИ, МОДИФИКАЦИИ, РАСШИРЕНИЯ ИЛИ МОДИФИКАЦИИ.

Пожалуйста, обратите внимание, что компания MindView поддерживает веб-сайт, который является единственным пунктом распространения для электронных копий Исходного Текста, по адресу <http://www.MindView.net> (и официальные зеркальные сайты), где они свободно доступны на изложенных выше условиях.

Если вы считаете, что Исходный Текст содержит ошибку, сообщите о ней по системе обратной связи, доступной на сайте <http://www.MindView.net>.

///:~

Вы можете использовать код в ваших проектах и в классе (включая материалы показов), до тех пор пока сохраняется замечание об авторском праве, встроенное в каждый файл с исходным текстом.

Стандарты оформления кода

Я использую определенный стиль программирования для примеров в книге. Этот стиль эквивалентен стилю, который практикует компания Sun фактически во всех своих программах (можете посмотреть java.sun.com/docs/codeconv/index.html), его же придерживается большинство систем разработки программ для Java. Если вы знакомы с другими моими работами, то могли заметить, что стиль Sun ничем не отличается, — это радует меня, хотя я и не делал ничего для этого. Предмет манеры кодирования может стать весомым поводом для горячих дебатов, я же просто скажу, что не навязываю никому свой стиль; у меня есть личные причины для его использования. Поскольку Java является языком со свободным форматом, вы вправе писать так, как привыкли. Для решения проблемы стиля также можно воспользоваться таким инструментом, как *Jalopy* (www.triemax.com), — я пользовался им во время работы над книгой; это позволит вам переключиться на тот стиль форматирования, который лучше всего подходит лично вам.

Файлы с программным кодом, включенные в книгу, прошли автоматизированное тестирование, поэтому весь приводимый код должен работать и компилироваться без ошибок.

Материал книги предназначен для Java SE5/6 и тестировался с этими языками. Если вам потребуется узнать о более ранних версиях языка, которые в этом издании не рассматриваются, издания этой книги с 1-го по 3-е можно бесплатно загрузить по адресу www.MindView.net.

Ошибки

Сколько бы усилий ни прилагал писатель для поиска ошибок, все равно какие-то из них «просачиваются» и потом бросаются в глаза читателю. Если вы обнаружите что-то похожее на ошибку, используйте ссылку на сайте www.MindView.net для уведомления об ошибке и выражения вашего мнения в отношении возможного исправления. Ваша помощь приветствуется.

1

Введение в объекты

Мы препарируем природу, преобразуем ее в концепции и приписываем им смысл так, как мы это делаем во многом, потому что все мы являемся участниками соглашения, которое имеет силу в обществе, связанном речью, и которое закреплено в структуре языка... Мы не можем общаться вовсе, кроме как согласившись с установленными этим соглашением организацией и классификацией данных.

Бенджамин Ли Ворф (1897–1941)

Возникновением компьютерной революции мы обязаны машине. Поэтому наши языки программирования стараются быть ближе к этой машине.

Но в то же время компьютеры не столько механизмы, сколько средства усиления мысли («велосипеды для ума», как любил говорить Стив Джобс), и еще одно средство самовыражения. В результате инструменты программирования все меньше склоняются к машинам и все больше тяготеют к нашим умам, так же как и к другим формам выражения человеческих устремлений, как то: литература, живопись, скульптура, анимация и кинематограф. Объектно-ориентированное программирование (ООП) — часть превращения компьютера в средство самовыражения.

Эта глава познакомит вас с основами ООП, включая рассмотрение основных методов разработки программ. Она (и книга вообще) подразумевает наличие у вас опыта программирования на процедурном языке, не обязательно C. Если вам покажется, что перед прочтением этой книги вам не хватает познаний в программировании и синтаксисе C, воспользуйтесь мультимедийным семинаром *Thinking in C*, который можно загрузить с сайта www.MindView.net.

Настоящая глава содержит подготовительный и дополнительный материал. Многие читатели предпочитают сначала представить общую картину, а уже потом разбираться

в тонкостях ООП. Поэтому большинство идей в данной главе служат тому, чтобы дать вам цельное представление об ООП. Однако многие не воспринимают общей идеи до тех пор, пока не увидят конкретно, как все работает; такие люди нередко вязнут в общих словах, не имея перед собой примеров. Если вы принадлежите к последним и горите желанием приступить к основам языка, можете сразу перейти к следующей главе — это не помешает вам в написании программ или изучении языка. И все же чуть позже вам стоит вернуться к этой главе, чтобы расширить свой кругозор и понять, почему так важны объекты и какое место они занимают при проектировании программ.

Развитие абстракции

Все языки программирования построены на абстракции. Возможно, трудность решаемых задач напрямую зависит от типа и качества абстракции. Под словом «тип» я имею в виду ответ на вопрос: «Что конкретно мы абстрагируем?» Язык ассемблера есть небольшая абстракция от компьютера, на базе которого он работает. Многие так называемые «командные» языки, созданные вслед за ним (такие, как Fortran, BASIC и C), представляли собой абстракции следующего уровня. Эти языки обладали значительным преимуществом по сравнению с ассемблером, но их основная абстракция по-прежнему заставляет мыслить в контексте структуры компьютера, а не решаемой задачи. Программист должен установить связь между моделью машины (в «пространстве решения», которое представляет место, где реализуется решение, — например, компьютер) и моделью задачи, которую и нужно решать (в «пространстве задачи», которое является местом существования задачи, — например, прикладной областью). Для установления связи требуются усилия, оторванные от собственно языка программирования; в результате появляются программы, которые трудно писать и тяжело поддерживать. Мало того, это еще создало целую отрасль «методологий программирования».

Альтернативой моделированию машины является моделирование решаемой задачи. Ранние языки, подобные LISP и APL, выбирали особый подход к моделированию окружающего мира («Все задачи решаются списками» или «Алгоритмы решают все» соответственно). PROLOG трактует все проблемы как цепочки решений. Были созданы языки для программирования, основанного на системе ограничений, и специальные языки, в которых программирование осуществлялось посредством манипуляций с графическими конструкциями (область применения последних оказалась слишком узкой). Каждый из этих подходов хорош в определенной области решаемых задач, но стоит выйти из этой сферы, как использовать их становится затруднительно.

Объектный подход делает шаг вперед, предоставляя программисту средства для представления задачи в ее пространстве. Такой подход имеет достаточно общий характер и не накладывает ограничений на тип решаемой проблемы. Элементы пространства задачи и их представления в пространстве решения называются «объектами». (Вероятно, вам понадобятся и другие объекты, не имеющие аналогов в пространстве задачи.) Идея состоит в том, что программа может адаптироваться к специфике задачи посредством создания новых типов объектов так, что во время чтения кода, решающего задачу, вы одновременно видите слова, ее описывающие. Это более гибкая и мощная абстракция,

превосходящая по своим возможностям все, что существовало ранее¹. Таким образом, ООП позволяет описать задачу в контексте самой задачи, а не в контексте компьютера, на котором будет исполнено решение. Впрочем, связь с компьютером все же сохранилась. Каждый объект похож на маленький компьютер; у него есть состояние и операции, которые он позволяет проводить. Такая аналогия неплохо сочетается с внешним миром, который есть «реальность, данная нам в объектах», обладающих характеристиками и поведением.

Алан Кей подвел итог и вывел пять основных особенностей языка Smalltalk, первого удачного объектно-ориентированного языка, одного из предшественников Java. Эти характеристики представляют «чистый», академический подход к объектно-ориентированному программированию.

- ❑ **Все является объектом.** Представляйте себе объект как усовершенствованную переменную; он хранит данные, но к нему можно «обращаться с запросами», приказывая объекту выполнить операции над собой. Теоретически абсолютно любой компонент решаемой задачи (собака, здание, услуга и т. п.) может быть представлен в виде объекта.
- ❑ **Программа — это набор объектов, указывающих друг другу, что делать, посредством сообщений.** Чтобы обратиться с запросом к объекту, вы «посылаете ему сообщение». Более наглядно можно представить сообщение как запрос на вызов метода, принадлежащего определенному объекту.
- ❑ **Каждый объект имеет собственную «память», состоящую из других объектов.** Иными словами, вы создаете новый объект с помощью встраивания в него уже существующих объектов. Таким образом, можно сконструировать сколь угодно сложную программу, скрывая общую сложность за простотой отдельных объектов.
- ❑ **У каждого объекта есть тип.** В других терминах, каждый объект является экземпляром класса, где «класс» является синонимом понятия «тип». Важнейшее отличие классов друг от друга как раз и заключается в ответе на вопрос: «Какие сообщения можно объекту посылать?»
- ❑ **Все объекты определенного типа могут получать одинаковые сообщения.** Как мы вскоре убедимся, это очень важное обстоятельство. Так как объект типа «круг» также является объектом типа «фигура», справедливо утверждение, что «круг» заведомо способен принимать сообщения для «фигуры». А это значит, что можно писать код для фигур и быть уверенным в том, что он подойдет для всего, что подпадает под понятие фигуры. *Взаимозаменяемость* представляет одно из самых мощных понятий ООП.

Буч предложил еще более лаконичное описание объекта:

Объект обладает состоянием, поведением и индивидуальностью.

¹ Некоторые разработчики языков считают, что объектно-ориентированное программирование плохо подходит для решения некоторых задач, и выступают за объединение разных подходов в *мультипарадигменных* языках программирования. См. «Multiparadigm Programming in Leda» by Timothy Budd (Addison-Wesley, 1995).

Суть сказанного в том, что объект может иметь в своем распоряжении внутренние данные (которые и есть состояние объекта), методы (которые определяют поведение), и каждый объект можно уникальным образом отличить от любого другого объекта — говоря более конкретно, каждый объект обладает уникальным адресом в памяти¹.

Объект обладает интерфейсом

Вероятно, Аристотель был первым, кто внимательно изучил понятие *типа*; он говорил о «классе рыб и классе птиц». Концепция, что все объекты, будучи уникальными, в то же время являются частью класса объектов со сходными характеристиками и поведением, была использована в первом объектно-ориентированном языке Simula-67, с введением фундаментального ключевого слова `class`, которое вводило новый тип в программу.

Язык Simula, как подразумевает его имя, был создан для развития и моделирования ситуаций, подобных классической задаче «банковский кассир». У вас есть группы кассиров, клиентов, счетов, платежей и денежных единиц — много «объектов». Объекты, идентичные во всем, кроме внутреннего состояния во время работы программы, группируются в «классы объектов». Отсюда и пришло ключевое слово `class`. Создание абстрактных типов данных есть фундаментальное понятие во всем объектно-ориентированном программировании. Абстрактные типы данных действуют почти так же, как и встроенные типы: вы можете создавать переменные типов (называемые *объектами* или *экземплярами* в терминах ООП) и манипулировать ими (что называется посылкой *сообщений*, или *запросом*; вы производите запрос, и объект решает, что с ним делать). Члены (элементы) каждого класса обладают сходством: у каждого счета имеется баланс, каждый кассир принимает депозиты и т. п. В то же время все члены отличаются внутренним состоянием: у каждого счета баланс индивидуален, каждый кассир имеет человеческое имя. Поэтому все кассиры, заказчики, счета, переводы и прочее могут быть представлены уникальными сущностями внутри компьютерной программы. Это и есть суть объекта, и каждый объект принадлежит к определенному классу, который определяет его характеристики и поведение.

Таким образом, хотя мы реально создаем в объектных языках новые типы данных, фактически все эти языки используют ключевое слово «класс». Когда видите слово «тип», думайте «класс», и наоборот².

Поскольку класс определяет набор объектов с идентичными характеристиками (элементы данных) и поведением (функциональность), класс на самом деле является типом данных, потому что, например, число с плавающей запятой тоже имеет ряд характеристик и особенности поведения. Разница состоит в том, что программист определяет класс для представления некоторого аспекта задачи, вместо использования уже существующего типа, представляющего единицу хранения данных в машине. Вы

¹ На самом деле это слишком сильное утверждение, поскольку объекты могут существовать на разных компьютерах и адресных пространствах, а также храниться на диске. В таких случаях для идентификации объекта приходится использовать не адрес памяти, а что-то другое.

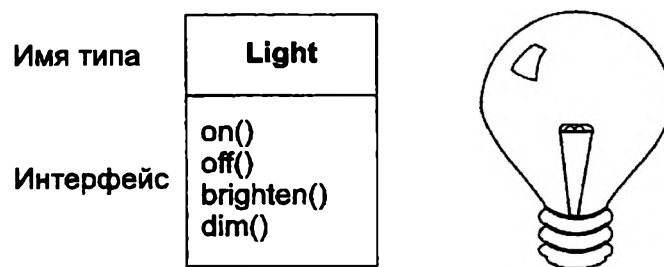
² Некоторые специалисты различают эти два понятия: они считают, что тип определяет интерфейс, а класс — конкретную реализацию этого интерфейса.

расширяете язык программирования, добавляя новые типы данных, соответствующие вашим потребностям. В системе программирования новые классы обладают точно такими же правами, как и встроенные типы.

Объектно-ориентированный подход не ограничивается построением моделей. Согласитесь вы или нет, что любая программа — модель разрабатываемой вами системы, независимо от вашего мнения ООП-технологии упрощают решение широкого круга задач.

После определения нового класса вы можете создать любое количество объектов этого класса, а затем манипулировать ими так, как будто они представляют собой элементы решаемой задачи. На самом деле одной из основных трудностей в ООП является установление однозначного соответствия между объектами пространства задачи и объектами пространства решения.

Но как заставить объект выполнять нужные вам действия? Должен существовать механизм передачи запроса к объекту на выполнение некоторого действия — завершения транзакции, рисования на экране и т. д. Каждый объект умеет выполнять только определенный круг запросов. Запросы, которые вы можете посылать объекту, определяются его *интерфейсом*, причем интерфейс объекта определяется его типом. Простейшим примером может стать электрическая лампочка:



```
Light lt = new Light();
lt.on();
```

Интерфейс определяет, с *какими* запросами можно обращаться к определенному объекту. Однако где-то должен существовать и код, выполняющий запросы. Этот код наряду со скрытыми данными составляет *реализацию*. С точки зрения процедурного программирования происходящее не так уж сложно. Тип содержит метод для каждого возможного запроса, и при получении определенного запроса вызывается нужный метод. Процесс обычно объединяется в одно целое: и «отправка сообщения» (передача запроса) объекту, и его обработка объектом (выполнение кода).

В данном примере существует тип/класс с именем `Light` (лампа), конкретный объект типа `Light` с именем `lt`, и класс поддерживает различные запросы к объекту `Light`: выключить лампочку, включить, сделать ярче или притушить. Вы создаете объект `Light`, определяя «ссылку» на него (`lt`) и вызывая оператор `new` для создания нового экземпляра этого типа. Чтобы послать сообщение объекту, следует указать имя объекта и связать его с помощью точки с нужным запросом. С точки зрения пользователя заранее определенного класса, этого вполне достаточно для того, чтобы оперировать его объектами.

Диаграмма, показанная выше, следует формату *UML* (Unified Modeling Language). Каждый класс представлен прямоугольником, все описываемые *поля данных* помещены

в средней его части, а *методы* (функции объекта, которому вы посылаете сообщения) перечисляются в нижней части прямоугольника. Часто на диаграммах UML показываются только имя класса и открытые методы, а средняя часть отсутствует. Если же вас интересует только имя класса, то можете пропустить и нижнюю часть.

Объект предоставляет услуги

В тот момент, когда вы пытаетесь разработать или понять структуру программы, часто бывает полезно представить объекты в качестве «поставщиков услуг». Ваша программа оказывает услуги пользователю, и делает она это посредством услуг, предоставляемых другими объектами. Ваша цель — произвести (а еще лучше отыскать в библиотеках классов) тот набор объектов, который будет оптимальным для решения вашей задачи.

Для начала спросите себя: «Если бы я мог по волшебству вынимать объекты из шляпы, какие бы из них смогли решить мою задачу прямо сейчас?» Предположим, что вы разрабатываете бухгалтерскую программу. Можно представить себе набор объектов, предоставляющих стандартные окна для ввода бухгалтерской информации, еще один набор объектов, выполняющих бухгалтерские расчеты, объект, ведающий распечаткой чеков и счетов на всевозможных принтерах. Возможно, некоторые из таких объектов уже существуют, а для других объектов стоит выяснить, как они могли бы выглядеть. Какие услуги могли бы предоставлять *те* объекты и какие объекты понадобились бы *им* для выполнения своей работы? Если вы будете продолжать в том же духе, то рано или поздно скажете: «Этот объект достаточно прост, так что можно сесть и записать его», или «Наверняка такой объект уже существует». Это разумный способ разбиения задачи на отдельные объекты.

Представление объекта в качестве поставщика услуг обладает дополнительным преимуществом: оно помогает улучшить *связуемость* (cohesiveness) объекта. *Хорошая связуемость* — важнейшее качество программного продукта: она означает, что различные аспекты программного компонента (такого, как объект, хотя сказанное также может относиться к методу или к библиотеке объектов) хорошо «стыкуются» друг с другом. Одной из типичных ошибок, допускаемых при проектировании объекта, является перенасыщение его большим количеством свойств и возможностей. Например, при разработке модуля, ведающего распечаткой чеков, вы можете захотеть, чтобы он «знал» все о форматировании и печати. Если подумать, скорее всего, вы придете к выводу, что для одного объекта этого слишком много, и перейдете к трем или более объектам. Один объект будет представлять собой каталог всех возможных форм чеков и его можно будет запросить о том, как следует распечатать чек. Другой объект или набор объектов станут отвечать за обобщенный интерфейс печати, «знающий» все о различных типах принтеров (но ничего не «понимающий» в бухгалтерии — такой объект лучше купить, чем разрабатывать самому). Наконец, третий объект просто будет пользоваться услугами описанных объектов, для того чтобы выполнить задачу. Таким образом каждый объект представляет собой связанный набор предлагаемых им услуг. В хорошо спланированном объектно-ориентированном проекте каждый объект хорошо справляется с одной конкретной задачей, не пытаясь при этом сделать больше нужного. Как было показано, это не только позволяет определить, какие объекты

стоит приобрести (объект с интерфейсом печати), но также дает возможность получить в итоге объект, который затем можно использовать где-то еще (каталог чеков).

Представление объектов в качестве поставщиков услуг значительно упрощает задачу. Оно полезно не только во время разработки, но и когда кто-либо попытается понять ваш код или повторно использовать объект — тогда он сможет адекватно оценить объект по уровню предоставляемого сервиса, и это значительно упростит интеграцию последнего в другой проект.

Скрытая реализация

Программистов имеет смысл разделить на *создателей классов* (те, кто создает новые типы данных) и *программистов-клиентов*¹ (потребители классов, использующие типы данных в своих приложениях). Цель вторых — собрать как можно больше классов, чтобы заниматься быстрой разработкой программ. Цель создателя класса — построить класс, открывающий только то, что необходимо программисту-клиенту, и скрывающий все остальное. Почему? Программист-клиент не сможет получить доступ к скрытым частям, а значит, создатель классов оставляет за собой возможность произвольно их изменять, не опасаясь, что это кому-то повредит. Скрытая часть обычно и самая «хрупкая» часть объекта, которую легко может испортить неосторожный или несведущий программист-клиент, поэтому сокрытие реализации сокращает количество ошибок в программах.

В любых отношениях важно иметь какие-либо границы, не переступаемые никем из участников. Создавая библиотеку, вы устанавливаете отношения с программистом-клиентом. Он является таким же программистом, как и вы, но будет использовать вашу библиотеку для создания приложения (а может быть, библиотеки более высокого уровня). Если предоставить доступ ко всем членам класса кому угодно, программист-клиент сможет сделать с классом все, что ему заблагорассудится, и вы никак не сможете заставить его «играть по правилам». Даже если вам впоследствии понадобится ограничить доступ к определенным членам вашего класса, без механизма контроля доступа это осуществить невозможно. Все внутреннее строение класса открыто для всех желающих.

Таким образом, первой причиной для ограничения доступа является необходимость уберечь «хрупкие» детали от программиста-клиента — части внутренней «кухни», не являющиеся составляющими интерфейса, при помощи которого пользователи решают свои задачи. На самом деле это полезно и пользователям — они сразу увидят, что для них важно, а на что можно не обращать внимания.

Вторая причина появления ограничений доступа — стремление позволить разработчику библиотеки изменить внутренние механизмы класса, не беспокоясь о том, как это отразится на программисте-клиенте. Например, вы можете реализовать определенный класс «на скорую руку», чтобы ускорить разработку программы, а затем переписать его, чтобы повысить скорость работы. Если вы правильно разделили и защитили интерфейс и реализацию, сделать это будет совсем несложно.

¹ Этим термином я обязан своему другу Скотту Мейерсу.

Java использует три явных ключевых слова, характеризующих уровень доступа: `public`, `private` и `protected`. Эти *спецификаторы доступа* определяют, кто имеет право использовать следующие за ними определения. `public` означает, что последующие определения доступны всем. С другой стороны, слово `private` означает, что к следующему элементу не может обратиться никто, кроме создателя типа, внутри методов этого типа. `private` — «крепостная стена» между вами и программистом-клиентом. Попытка внешнего обращения к `private`-членам пресекается ошибкой компиляции. Спецификатор `protected` действует схоже с `private`, за одним исключением — производные классы имеют доступ к членам со спецификатором `protected`, но не имеют доступа к `private`-членам (наследование мы вскоре рассмотрим).

В Java также есть доступ по умолчанию, используемый при отсутствии какого-либо из перечисленных спецификаторов. Он также иногда называется *доступом в пределах пакета* (package access), поскольку классы могут использовать дружественные члены других классов из своего *пакета*, но за его пределами те же дружественные члены приобретают статус `private`.

Повторное использование реализации

Созданный и протестированный класс должен (в идеале) представлять собой полезный блок кода. Однако оказывается, что добиться этой цели гораздо труднее, чем многие полагают; для разработки повторно используемых объектов требуется опыт и понимание сути дела. Но как только у вас получится хорошая конструкция, она будет просто напрашиваться на внедрение в другие программы. Многократное использование кода — одно из самых впечатляющих преимуществ объектно-ориентированных языков.

Проще всего использовать класс повторно, непосредственно создавая его объект, но вы можете также поместить объект этого класса внутрь нового класса. Мы называем это *внедрением объекта*. Новый класс может содержать любое количество объектов других типов в любом сочетании, которое необходимо для достижения необходимой функциональности. Так как мы составляем новый класс из уже существующих классов, этот способ называется *композицией* (если композиция выполняется динамически, она обычно именуется *агрегированием*). Композицию часто называют связью типа «содержит» (has-a), как, например, в предложении «машина содержит двигатель».



(На UML-диаграммах композиция обозначается закрашенным ромбом. Я несколько упрощу этот формат: оставлю только простую линию, без ромба, чтобы обозначить связь¹.)

¹ Для большинства диаграмм этого вполне достаточно. Не обязательно уточнять, что именно используется в данном случае — композиция или агрегирование.

Композиция — очень гибкий инструмент. Объекты-члены вашего нового класса обычно объявляются закрытыми (`private`), что делает их недоступными для программистов-клиентов, использующих класс. Это позволяет вносить изменения в эти объекты-члены без модификации уже существующего клиентского кода. Вы можете также изменять эти члены во время исполнения программы, чтобы динамически управлять поведением вашей программы. Наследование, рассмотренное ниже, не имеет такой гибкости, так как компилятор накладывает определенные ограничения на классы, созданные с применением наследования.

Наследование играет важную роль в объектно-ориентированном программировании, поэтому на нем часто акцентируется повышенное внимание, и у новичка может возникнуть впечатление, что наследование должно применяться повсюду. А это чревато созданием громоздких, излишне сложных решений. Вместо этого при создании новых классов прежде всего следует оценить возможность композиции, так как она проще и гибче. Если вы возьмете на вооружение рекомендуемый подход, ваши программные конструкции станут гораздо яснее. А по мере накопления практического опыта понять, где следует применять наследование, не составит труда.

Наследование

Сама по себе идея объекта крайне удобна. Объект позволяет совмещать данные и функциональность на *концептуальном уровне*, то есть вы можете представить нужное понятие проблемной области прежде, чем начнете его конкретизировать применительно к диалекту машины. Эти концепции и образуют фундаментальные единицы языка программирования, описываемые с помощью ключевого слова `class`.

Но согласитесь, было бы неэффективно создавать какой-то класс, а потом проделывать всю работу заново для похожего класса. Гораздо рациональнее взять готовый класс, «клонировать» его, а затем внести добавления и обновления в полученном клоне. Это именно то, что вы получаете в результате *наследования*, с одним исключением — если изначальный класс (называемый также *базовым классом*, *суперклассом* или *родительским классом*) изменяется, то все изменения отражаются и на его «клоне» (называемом *производным классом*, *унаследованным классом*, *субклассом* или *дочерним классом*).



1

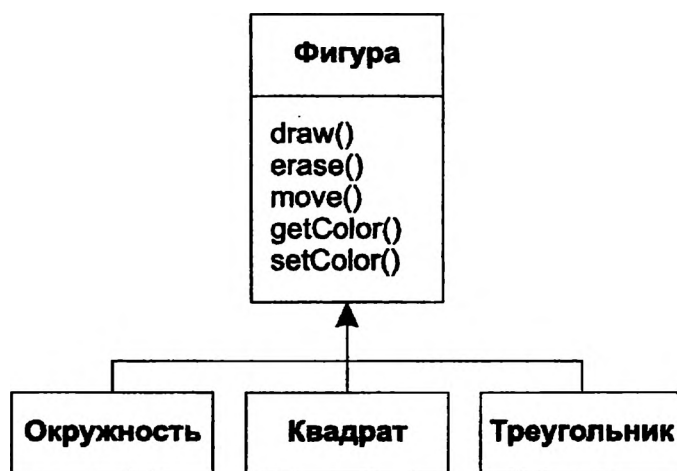
(Стрелка на UML-диаграмме направлена от производного класса к базовому классу. Как вы вскоре увидите, может быть и больше одного производного класса.)

Тип определяет не только свойства группы объектов; он также связан с другими типами. Два типа могут иметь общие черты и поведение, но различаться количеством

характеристик, а также в способности обработать большее число сообщений (или обработать их по-другому). Для выражения этой общности типов при наследовании используется понятие базовых и производных типов. Базовый тип содержит все характеристики и действия, общие для всех типов, производных от него. Вы создадите базовый тип, чтобы заложить основу своего представления о каких-то объектах в вашей системе. От базового типа порождаются другие типы, выражающие другие реализации этой сущности.

Например, машина по переработке мусора сортирует отходы. Базовым типом будет «мусор», и каждая частица мусора имеет вес, стоимость и т. п. и может быть раздроблена, расплавлена или разложена. Отталкиваясь от этого, наследуются более определенные виды мусора, имеющие дополнительные характеристики (бутылка имеет цвет) или поведения (алюминиевую банку можно смять, стальная банка притягивается магнитом). Вдобавок, некоторые характеристики поведения могут различаться (стоимость бумаги зависит от ее типа и состояния). Наследование позволяет составить иерархию типов, описывающую решаемую задачу в контексте ее типов.

Второй пример — классический пример с геометрическими фигурами. Базовым типом здесь является «фигура», и каждая фигура имеет размер, цвет, расположение и т. п. Каждую фигуру можно нарисовать, стереть, переместить, закрасить и т. д. Далее производятся (наследуются) конкретные разновидности фигур: окружность; прямоугольник, треугольник и т. п., каждая из которых имеет свои дополнительные характеристики и черты поведения. Например, для некоторых фигур поддерживается операция зеркального отображения. Отдельные черты поведения могут различаться, как в случае вычисления площади фигуры. Иерархия типов воплощает как схожие, так и различные свойства фигур.

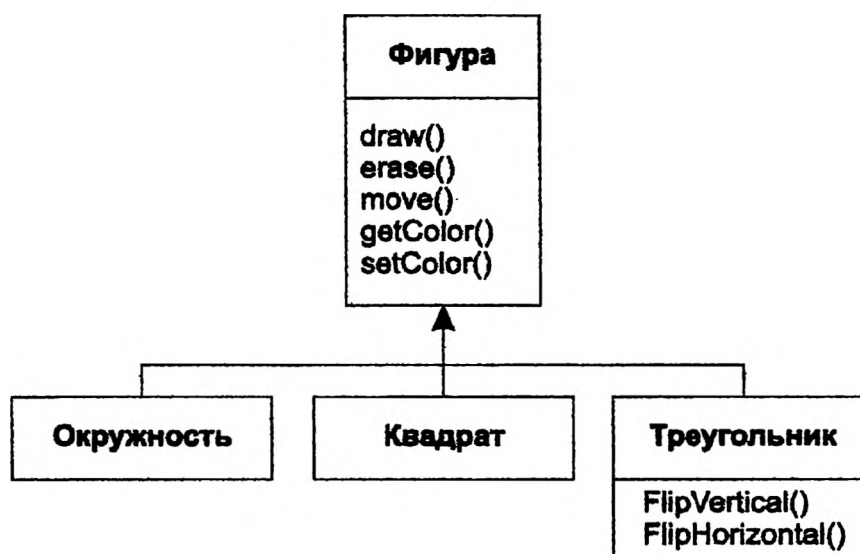


Приведение решения к понятиям, использованным в примере, чрезвычайно удобно, потому что вам не потребуется множество промежуточных моделей, связывающих описание решения с описанием задачи. При работе с объектами первичной моделью становится иерархия типов, так что вы переходите от описания системы реального мира прямо к описанию системы в программном коде. На самом деле, одна из трудностей в объектно-ориентированном планировании состоит в том, что уж очень просто вы проходите от начала задачи до конца решения. Разум, натренированный на сложные решения, часто заходит в тупик при использовании простых подходов.

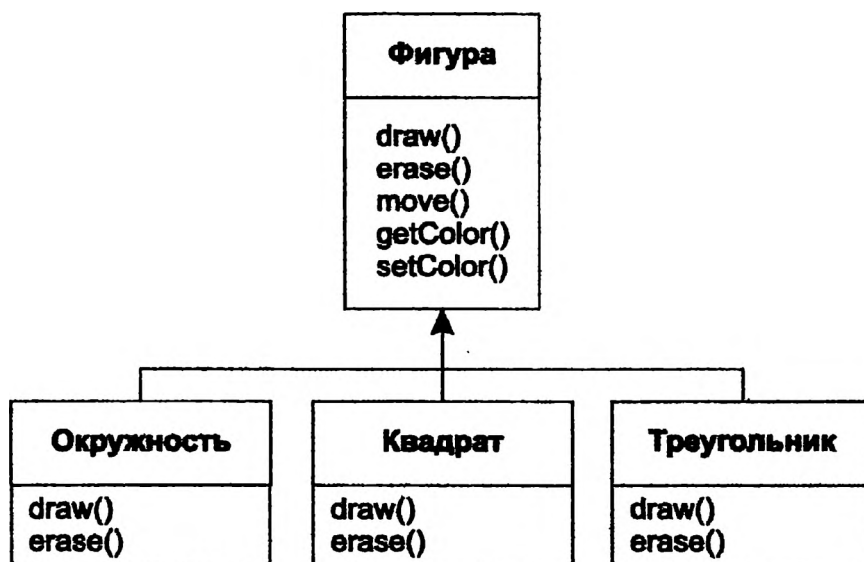
Используя наследование от существующего типа, вы создаете новый тип. Этот новый тип содержит не только все члены существующего типа (хотя члены со спецификатором `private` скрыты и недоступны), но, что еще важнее, повторяет интерфейс базового класса. Значит, все сообщения, которые вы могли посылать базовому классу, также можно посылать и производному классу. А так как мы различаем типы классов по совокупности сообщений, которые можем им посылать, это означает, что производный класс *является частным случаем базового класса*. В предыдущем примере «окружность есть фигура». Эквивалентность типов, достигаемая при наследовании, является одним из основополагающих условий понимания смысла объектно-ориентированного программирования.

Так как и базовый, и производный классы имеют одинаковый основной интерфейс, должна существовать и реализация для этого интерфейса. Другими словами, где-то должен быть код, выполняемый при получении объектом определенного сообщения. Если вы просто унаследовали класс и больше не предпринимали никаких действий, методы из интерфейса базового класса перейдут в производный класс без изменений. Это значит, что объекты производного класса не только однотипны, но и обладают одинаковым поведением, а при этом само наследование теряет смысл.

Существует два способа изменения нового класса по сравнению с базовым классом. Первый достаточно очевиден: в производный класс включаются новые методы. Они уже не являются частью интерфейса базового класса. Видимо, базовый класс не делал всего, что требовалось в данной задаче, и вы дополнили его новыми методами. Впрочем, такой простой и примитивный подход к наследованию иногда оказывается идеальным решением проблемы. Однако надо внимательно рассмотреть, действительно ли базовый класс нуждается в этих дополнительных методах. Процесс выявления закономерностей и пересмотра архитектуры является повседневным делом в объектно-ориентированном программировании.



Хотя наследование иногда наводит на мысль, что интерфейс будет дополнен новыми методами (особенно в Java, где наследование обозначается ключевым словом `extends`, то есть «расширять»), это совсем не обязательно. Второй, более важный способ модификации классов заключается в *изменении* поведения уже существующих методов базового класса. Это называется *переопределением* (или *замещением*) метода.

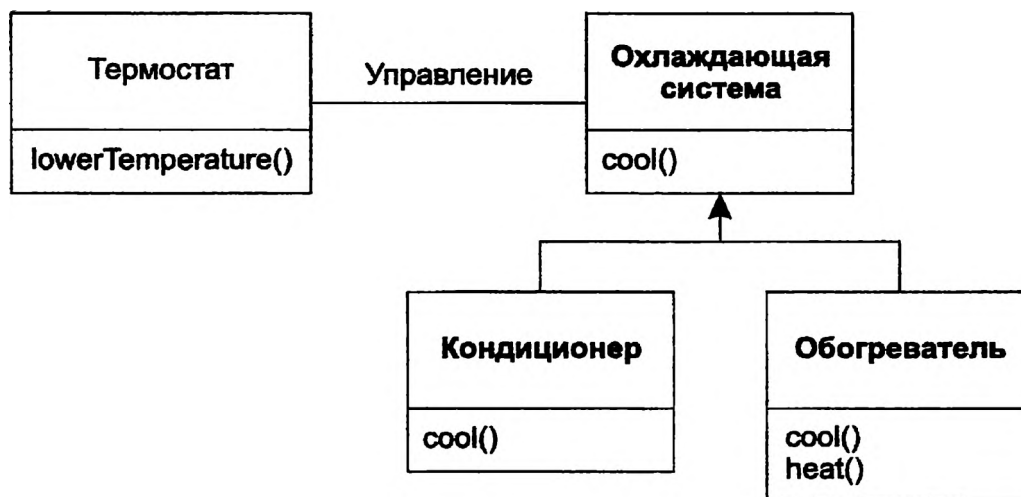


Для замещения метода нужно просто создать новое определение этого метода в производном классе. Вы как бы говорите: «Я использую тот же метод интерфейса, но хочу, чтобы он выполнял другие действия для моего нового типа».

Отношение «является» в сравнении с «похоже»

При использовании наследования встает очевидный вопрос: следует ли при наследовании переопределять *только* методы базового класса (и не добавлять новых методов, не существующих в базовом классе)? Это бы означало, что производный тип будет *точно* такого же типа, как и базовый класс, так как они имеют одинаковый интерфейс. В результате вы можете свободно заменять объекты базового класса объектами производных классов. Можно говорить о *полной замене*, и это часто называется *принципом замены*. В определенном смысле этот способ наследования идеален. Подобный способ взаимосвязи базового и производного классов часто называют связью «*является тем-то*», поскольку можно сказать «круг *является* фигурой». Чтобы определить, насколько уместным будет наследование, достаточно проверить, существует ли отношение «является» между классами, и насколько оно оправданно.

В иных случаях интерфейс производного класса дополняется новыми элементами, что приводит к его расширению. Новый тип все еще может применяться вместо базового, но теперь эта замена не идеальна, потому что она не позволяет использовать новые методы из базового типа. Подобная связь описывается выражением «похоже на» (это мой термин); новый тип содержит интерфейс старого типа, но также включает в себя и новые методы, и нельзя сказать, что эти типы абсолютно одинаковы. Для примера возьмем кондиционер. Предположим, что ваш дом снабжен всем необходимым оборудованием для контроля процесса охлаждения. Представим теперь, что кондиционер сломался и вы заменили его обогревателем, способным как нагревать, так и охлаждать. Обогреватель «*похож на*» кондиционер, но он способен и на большее. Так как система управления вашего дома способна контролировать только охлаждение, она ограничена в коммуникациях с охлаждающей частью нового объекта. Интерфейс нового объекта был расширен, а существующая система ничего не признает, кроме оригинального интерфейса.



Конечно, при виде этой иерархии становится ясно, что базовый класс «охлаждающая система» недостаточно гибок; его следует переименовать в «систему контроля температуры», так, чтобы он включал и нагрев — и после этого заработает принцип замены. Тем не менее эта диаграмма представляет пример того, что может произойти в реальности.

После знакомства с принципом замены может возникнуть впечатление, что этот подход (полная замена) — единственный способ разработки. Вообще говоря, если ваши иерархии типов так работают, это *действительно* хорошо. Но в некоторых ситуациях совершенно необходимо добавлять новые методы к интерфейсу производного класса. При внимательном анализе оба случая представляются достаточно очевидными.

Взаимозаменяемые объекты и полиморфизм

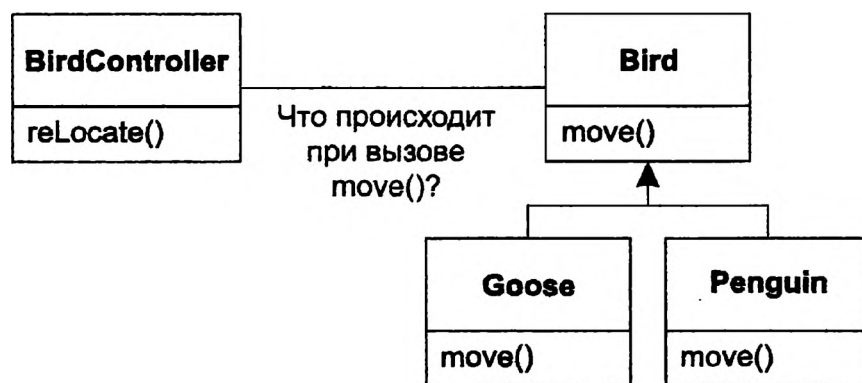
При использовании иерархий типов часто приходится обращаться с объектом определенного типа как с базовым типом. Это позволяет писать код, не зависящий от конкретных типов. Так, в примере с фигурами методы манипулируют просто фигурами, не обращая внимания на то, являются ли они окружностями, прямоугольниками, треугольниками или некоторыми еще даже не определенными фигурами. Все фигуры могут быть нарисованы, стерты и перемещены, а методы просто посылают сообщения объекту «фигура»; им безразлично, как объект обойдется с этим сообщением.

Подобный код не зависит от добавления новых типов, а добавление новых типов является наиболее распространенным способом расширения объектно-ориентированных программ для обработки новых ситуаций. Например, вы можете создать новый подкласс фигуры (пятиугольник), и это не приведет к изменению методов, работающих только с обобщенными фигурами. Возможность простого расширения программы введением новых производных типов очень важна, потому что она заметно улучшает архитектуру программы, в то же время снижая стоимость сопровождения программного обеспечения.

Однако при попытке обращения к объектам производных типов как к базовым типам (окружности как фигуре, велосипеду как средству передвижения, баклану как птице и т. п.) возникает одна проблема. Если метод собирается приказать обобщенной фигуре нарисовать себя, или средству передвижения следовать по определенному курсу, или птице полететь, компилятор не может точно знать, какая именно часть кода выполнится.

В этом все дело — когда посылается сообщение, программист и *не хочет* знать, какой код выполняется; метод прорисовки с одинаковым успехом может применяться и к окружности, и к прямоугольнику, и к треугольнику, а объект выполнит верный код, зависящий от его характерного типа.

Если вам не нужно знать, какой именно фрагмент кода выполняется, то когда вы добавляете новый подтип, код его реализации может измениться, но без изменений в том методе, из которого он был вызван. Если компилятор не обладает информацией, какой именно код следует выполнить, что же он делает? В следующем примере объект `BirdController` (управление птицей) может работать только с обобщенными объектами `Bird` (птица), не зная типа конкретного объекта. С точки зрения `BirdController` это удобно, поскольку для него не придется писать специальный код проверки типа используемого объекта `Bird` для обработки какого-то особого поведения. Как же все-таки происходит, что при вызове метода `move()` без указания точного типа `Bird` исполняется верное действие — объект `Goose` (гусь) бежит, летит или плавает, а объект `Penguin` (пингвин) бежит или плавает?



Ответ объясняется главной особенностью объектно-ориентированного программирования: компилятор не может вызывать такие функции традиционным способом. При вызовах функций, созданных не ООП-компилятором, используется *раннее связывание* — многие не знают этого термина просто потому, что не представляют себе другого варианта. При раннем связывании компилятор генерирует вызов функции с указанным именем, а компоновщик привязывает этот вызов к абсолютному адресу кода, который необходимо выполнить. В ООП программа не в состоянии определить адрес кода до времени исполнения, поэтому при отправке сообщения объекту должен срабатывать иной механизм.

Для решения этой задачи языки объектно-ориентированного программирования используют концепцию *позднего связывания*. Когда вы посылаете сообщение объекту, вызываемый код неизвестен вплоть до времени исполнения. Компилятор лишь убеждается в том, что метод существует, проверяет типы для его параметров и возвращаемого значения, но не имеет представления, какой именно код будет исполняться.

Для осуществления позднего связывания Java вместо абсолютного вызова использует специальные фрагменты кода. Этот код вычисляет адрес тела метода на основе информации, хранящейся в объекте (процесс очень подробно описан в главе 7). Таким образом, каждый объект может вести себя различно, в зависимости от содержимого этого кода. Когда вы посылаете сообщение, объект фактически сам решает, что же с ним делать.

В некоторых языках необходимо явно указать, что для метода должен использоваться гибкий механизм позднего связывания (в C++ для этого предусмотрено ключевое слово `virtual`). В этих языках методы по умолчанию компонируются *не* динамически. В Java позднее связывание производится по умолчанию, и вам не нужно помнить о необходимости добавления каких-либо ключевых слов для обеспечения полиморфизма.

Вспомним о примере с фигурами. Семейство классов (основанных на одинаковом интерфейсе) было показано на диаграмме чуть раньше в этой главе. Для демонстрации полиморфизма мы напишем фрагмент кода, который игнорирует характерные особенности типов и работает только с базовым классом. Этот код *отделен* от специфики типов, поэтому его проще писать и понимать. И если новый тип (например, шестиугольник) будет добавлен посредством наследования, то написанный вами код будет работать для нового типа фигуры так же хорошо, как прежде. Таким образом, программа становится *расширяемой*.

Допустим, вы написали на Java следующий метод (вскоре вы узнаете, как это делать):

```
void doSomething(Shape shape) {
    shape.erase(); // стереть
    //...
    shape.draw();  // нарисовать
}
```

Метод работает с обобщенной фигурой (`Shape`), то есть не зависит от конкретного типа объекта, который рисуется или стирается. Теперь мы используем вызов метода `doSomething()` в другой части программы:

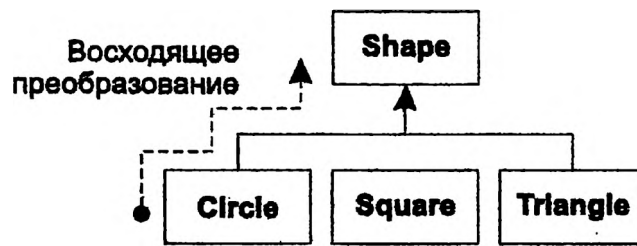
```
Circle circle = new Circle(); // окружность
Triangle triangle = new Triangle(); // треугольник
Line line = new Line(); // линия
doSomething(circle);
doSomething(triangle);
doSomething(line);
```

Вызовы метода `doSomething()` автоматически работают правильно, вне зависимости от фактического типа объекта. На самом деле это довольно важный факт. Рассмотрим строку:

```
doSomething(c);
```

Здесь происходит следующее: методу, ожидающему объект `Shape`, передается объект «окружность» (`Circle`). Так как окружность (`Circle`) одновременно является фигурой (`Shape`), то метод `doSomething()` и обращается с ней как с фигурой. Другими словами, любое сообщение, которое метод может послать `Shape`, также принимается и `Circle`. Это действие совершенно безопасно и настолько же логично.

Мы называем этот процесс обращения с производным типом как с базовым *восходящим преобразованием типов*. Слово *преобразование* означает, что объект трактуется как принадлежащий к другому типу, а *восходящее* оно потому, что на диаграммах наследования базовые классы обычно располагаются вверху, а производные классы располагаются внизу «веером». Значит, преобразование к базовому типу — это движение по диаграмме вверх, и поэтому оно «восходящее».



В объектно-ориентированной программе почти всегда где-то присутствует восходящее преобразование, потому что оно избавляет разработчика от необходимости знать точный тип объекта, с которым он работает. Посмотрите на тело метода `doSomething()`:

```
shape.erase();  
//...  
shape.draw();
```

Заметьте, что здесь не сказано: «Если ты объект `Circle`, делай это, а если ты объект `Square`, делай то-то и то-то». Такой код с отдельными действиями для каждого возможного типа `Shape` будет путаным, и его придется менять каждый раз при добавлении нового подтипа `Shape`. А так, вы просто говорите: «Ты фигура, и я знаю, что ты способна нарисовать и стереть себя, ну так и делай это, а о деталях позаботься сама».

В коде метода `doSomething()` интересно то, что все само собой получается правильно. При вызове `draw()` для объекта `Circle` выполняется другой код, а не тот, что отработывает при вызове `draw()` для объектов `Square` или `Line`, а когда `draw()` применяется для неизвестной фигуры `Shape`, правильное поведение обеспечивается использованием реального типа `Shape`. Это в высшей степени интересно, потому что, как было замечено чуть ранее, когда компилятор генерирует код `doSomething()`, он не знает точно, с какими типами он работает. Соответственно, можно было бы ожидать вызова версий методов `draw()` и `erase()` из базового класса `Shape`, а не их вариантов из конкретных классов `Circle`, `Square` или `Line`. И тем не менее все работает правильно благодаря полиморфизму. Компилятор и система исполнения берут на себя все подробности; все, что вам нужно знать — как это происходит... и что еще важнее — как создавать программы, используя такой подход. Когда вы посылаете сообщение объекту, объект выберет правильный вариант поведения даже при восходящем преобразовании.

Однокорневая иерархия

Вскоре после появления C++ стал активно обсуждаться вопрос — должны ли все классы обязательно наследовать от единого базового класса? В Java (как практически во всех других ООП-языках, *кроме* C++) на этот вопрос был дан положительный ответ. В основе всей иерархии типов лежит единый базовый класс `Object`. Оказалось, что однокорневая иерархия имеет множество преимуществ.

Все объекты в однокорневой иерархии имеют некий общий интерфейс, так что, по большому счету, все они могут рассматриваться как один основополагающий тип. В C++ был выбран другой вариант — общего предка в этом языке не существует. С точки зрения совместимости со старым кодом эта модель лучше соответствует традициям C, и можно подумать, что она менее ограничена. Но как только возникнет

необходимость в полноценном объектно-ориентированном программировании, вам придется создавать собственную иерархию классов, чтобы получить те же преимущества, что встроены в другие ООП-языки. Да и в любой новой библиотеке классов вам может встретиться какой-нибудь несовместимый интерфейс. Включение этих новых интерфейсов в архитектуру вашей программы потребует лишних усилий (и, возможно, множественного наследования). Стоит ли дополнительная «гибкость» C++ подобных издержек? Если вам это нужно (например, при больших вложениях в разработку кода C), то в проигрыше вы не останетесь. Если же разработка начинается «с нуля», подход Java выглядит более продуктивным.

Все объекты из однокорневой иерархии гарантированно обладают некоторой общей функциональностью. Вы знаете, что с любым объектом в системе можно провести определенные основные операции. Все объекты легко создаются в динамической «куче», а передача аргументов сильно упрощается.

Однокорневая иерархия позволяет гораздо проще реализовать *уборку мусора* — одно из важнейших усовершенствований Java по сравнению с C++. Так как информация о типе во время исполнения гарантированно присутствует в любом из объектов, в системе никогда не появится объект, тип которого не удастся определить. Это особенно важно при выполнении системных операций, таких как обработка исключений, и для обеспечения большей гибкости программирования.

Контейнеры

Часто бывает заранее неизвестно, сколько объектов потребуется для решения определенной задачи и как долго они будут существовать. Также непонятно, как хранить такие объекты. Сколько памяти следует выделить для хранения этих объектов? Неизвестно, так как эта информация станет доступна только во время работы программы.

Многие проблемы в объектно-ориентированном программировании решаются простым действием: вы создаете еще один тип объекта. Новый тип объекта, решающего эту конкретную задачу, содержит ссылки на другие объекты. Конечно, для решения этой задачи можно использовать и *массивы*, поддерживаемые в большинстве языков. Однако новый объект, обычно называемый *контейнером* (или же *коллекцией*, но в Java этот термин используется в другом смысле), будет расширяться по мере необходимости, чтобы вместить все, что вы в него положите. Поэтому вам не нужно будет знать загодя, сколько объектов будет храниться в контейнере. Просто создайте контейнер, а он уже позаботится о подробностях.

К счастью, хороший ООП-язык поставляется с набором готовых контейнеров. В C++ это часть стандартной библиотеки C++, иногда называемая *библиотекой стандартных шаблонов* (Standard Template Library, STL). Smalltalk поставляется с очень широким набором контейнеров. Java также содержит контейнеры в своей стандартной библиотеке. Для некоторых библиотек считается, что достаточно иметь один единый контейнер для всех нужд, но в других (например, в Java) предусмотрены различные контейнеры на все случаи жизни: несколько различных типов списков *List* (для хранения последовательностей элементов), карты *map* (известные также как *ассоциативные массивы*,

позволяют связывать объекты с другими объектами), а также множества `Set` (обеспечивающие уникальность значений для каждого типа). Контейнерные библиотеки также могут содержать очереди, деревья, стеки и т. п.

С позиций проектирования, все, что вам действительно необходимо, — это контейнер, способный решить вашу задачу. Если один вид контейнера отвечает всем потребностям, нет основания использовать другие виды. Существуют две причины, по которым вам приходится выбирать из имеющихся контейнеров. Во-первых, контейнеры предоставляют различные интерфейсы и возможности взаимодействия. Поведение и интерфейс стека отличаются от поведения и интерфейса очереди, которая ведет себя по-иному, чем множество или список. Один из этих контейнеров способен обеспечить более эффективное решение вашей задачи в сравнении с остальными. Во-вторых, разные контейнеры по-разному выполняют одинаковые операции. Лучший пример — это `ArrayList` и `LinkedList`. Оба представляют собой простые последовательности, которые могут иметь идентичные интерфейсы и черты поведения. Но некоторые операции значительно отличаются по времени исполнения. Скажем, время выборки произвольного элемента в `ArrayList` всегда остается неизменным вне зависимости от того, какой именно элемент выбирается. Однако в `LinkedList` невыгодно работать с произвольным доступом — чем дальше по списку находится элемент, тем большую задержку вызывает его поиск. С другой стороны, если потребуется вставить элемент в середину списка, `LinkedList` сделает это быстрее `ArrayList`. Эти и другие операции имеют разную эффективность, зависящую от внутренней структуры контейнера. На стадии планирования программы вы можете выбрать список `LinkedList`, а потом, в процессе оптимизации, переключиться на `ArrayList`. Благодаря абстрактному характеру интерфейса `List` такой переход потребует минимальных изменений в коде.

Параметризованные типы

До выхода Java SE5 в контейнерах могли храниться только данные `Object` — единственного универсального типа Java. Однокорневая иерархия означает, что любой объект может рассматриваться как `Object`, поэтому контейнер с элементами `Object` подойдет для хранения любых объектов¹.

При работе с таким контейнером вы просто помещаете в него ссылки на объекты, а позднее извлекаете их. Но если контейнер способен хранить только `Object`, то при помещении в него ссылки на другой объект происходит его преобразование к `Object`, то есть утрата его «индивидуальности». При выборке вы получаете ссылку на `Object`, а не ссылку на тип, который был помещен в контейнер. Как же преобразовать ее к конкретному типу объекта, помещенного в контейнер?

Задача решается тем же преобразованием типов, но на этот раз тип изменяется не по восходящей линии (от частного к общему), а по нисходящей (от общего к частному). Данный способ называется *нисходящим преобразованием*. В случае восходящего

Примитивные типы в контейнерах храниться не могут, но благодаря механизму *автоматической упаковки* Java SE5 это ограничение почти несущественно. Далее в книге эта тема будет рассмотрена более подробно.

преобразования известно, что окружность есть фигура, поэтому преобразование заведомо безопасно, но при обратном преобразовании невозможно заранее сказать, представляет ли экземпляр `Object` объект `Circle` или `Shape`, поэтому нисходящее преобразование безопасно только в том случае, если вам точно известен тип объекта.

Впрочем, опасность не столь уж велика — при нисходящем преобразовании к неверному типу произойдет ошибка времени исполнения, называемая *исключением* (см. далее). Но при извлечении ссылок на объекты из контейнера необходимо каким-то образом запоминать фактический тип их объектов, чтобы выполнить верное преобразование.

Нисходящее преобразование и проверки типа во время исполнения требуют дополнительного времени и лишних усилий от программиста. А может быть, можно каким-то образом создать контейнер, который знает тип хранимых объектов, и таким образом снимает необходимость преобразования типов и избавляет от потенциальных ошибок? *Параметризованные типы* представляют собой классы, которые компилятор может автоматически адаптировать для работы с определенными типами. Например, компилятор может настроить параметризованный контейнер так, чтобы тот мог сохранять и извлекать только объекты `Shape` и никакие другие.

Одним из важнейших изменений Java SE5 является поддержка параметризованных типов, которые в Java называются *обобщенными* типами (generics). Обобщенные типы легко узнать по угловым скобкам, в которые заключаются имена типов-параметров; например, контейнер `ArrayList`, предназначенный для хранения объектов `Shape`, создается следующим образом:

```
ArrayList<Shape> shapes = new ArrayList<Shape>();
```

Многие стандартные библиотечные компоненты также были изменены для использования обобщенных типов. Как вы вскоре увидите, обобщенные типы встречаются во многих примерах программ этой книги.

Создание и время жизни объектов

Один из важнейших аспектов работы с объектами — организация их создания и уничтожения. Для существования каждого объекта требуются некоторые ресурсы, прежде всего память. Когда объект становится не нужен, его следует уничтожить, чтобы занимаемые им ресурсы стали доступны другим. В простых ситуациях задача не кажется сложной: вы создаете объект, используете его, пока требуется, а затем уничтожаете. Однако на практике часто встречаются и более сложные ситуации.

Допустим, например, что вы разрабатываете систему для управления движением авиатранспорта. (Эта же модель пригодна и для управления движением тары на складе, или для системы видеопроката, или в питомнике для бездомных животных.) Сначала все кажется просто: создается контейнер для самолетов, затем строится новый самолет, который помещается в контейнер определенной зоны регулировки воздушного движения. Что касается освобождения ресурсов, соответствующий объект просто уничтожается при выходе самолета из зоны слежения.

Но возможно, существует и другая система регистрации самолетов, и эти данные не требуют такого пристального внимания, как главная функция управления. Может

быть, это записи о планах полетов всех малых самолетов, покидающих аэропорт. Так появляется второй контейнер для малых самолетов; каждый раз, когда в системе создается новый объект самолета, он также включается и во второй контейнер, если самолет является малым. Далее некий фоновый процесс работает с объектами в этом контейнере в моменты минимальной занятости.

Теперь задача усложняется: как узнать, когда нужно удалять объекты? Даже если вы закончили работу с объектом, возможно, с ним продолжает взаимодействовать другая часть системы. Этот же вопрос возникает и в ряде других ситуаций, и в программных системах, где необходимо явно удалять объекты после завершения работы с ними (например, в C++), он становится достаточно сложным.

Где хранятся данные объекта и как определяется время его жизни? В C++ на первое место ставится эффективность, поэтому программисту предоставляется выбор. Для достижения максимальной скорости исполнения место хранения и время жизни могут определяться во время написания программы. В этом случае объекты размещаются в стеке (такие переменные называются *автоматическими*) или в области статического хранилища. Таким образом, основным фактором является скорость создания и уничтожения объектов, и это может быть неопределимо в некоторых ситуациях. Однако при этом приходится жертвовать гибкостью, так как количество объектов, время их жизни и типы должны быть точно известны на стадии разработки программы. При решении задач более широкого профиля — разработки систем автоматизированного проектирования (CAD), складского учета или управления воздушным движением — это требование может оказаться слишком жестким.

Второй путь — динамическое создание объектов в области памяти, называемой кучей (heap). В таком случае количество объектов, их точные типы и время жизни остаются неизвестными до момента запуска программы. Все это определяется «на ходу» во время работы программы. Если вам понадобится новый объект, вы просто создаете его в куче тогда, когда потребуется. Так как управление кучей осуществляется динамически, во время исполнения программы на выделение памяти из кучи требуется гораздо больше времени, чем при выделении памяти в стеке. (Для выделения памяти в стеке достаточно всего одной машинной инструкции, сдвигающей указатель стека вниз, а освобождение осуществляется перемещением этого указателя вверх. Время, требуемое на выделение памяти в куче, зависит от структуры хранилища.)

При использовании динамического подхода подразумевается, что объекты большие и сложные, таким образом, дополнительные затраты времени на выделение и освобождение памяти не окажут заметного влияния на процесс их создания. Потом, дополнительная гибкость очень важна для решения основных задач программирования.

В Java используется исключительно второй подход¹. Каждый раз при создании объекта используется ключевое слово `new` для построения динамического экземпляра.

Впрочем, есть и другой фактор, а именно время жизни объекта. В языках, поддерживающих создание объектов в стеке, компилятор определяет продолжительность существования объекта и может автоматически уничтожить его. Однако при создании объекта в куче компилятор не имеет представления о сроках жизни объекта. В таких

Примитивные типы, о которых речь пойдет далее, являются особым случаем.

языках, как C++, операция уничтожения объекта должна явно выполняться в программе; если этого не сделать, возникает утечка памяти (обычная проблема в программах C++). В Java существует механизм, называемый *уборкой мусора*; он автоматически определяет, когда объект перестает использоваться, и уничтожает его. Уборщик мусора очень удобен, потому что избавляет программиста от лишних хлопот. Что еще важнее, уборщик мусора дает гораздо большую уверенность в том, что в вашу программу не закралась коварная проблема утечки памяти (которая загнала в угол не один проект на языке C++).

В Java уборщик мусора спроектирован так, чтобы он мог самостоятельно решать проблему освобождения памяти (это не касается других аспектов завершения жизни объекта). Уборщик мусора «знает», когда объект перестает использоваться, и использует свои знания для автоматического освобождения памяти. Этот факт (вместе с тем, что все объекты наследуются от единого базового класса `Object` и создаются только в куче) существенно упрощает программирование на Java по сравнению с программированием на C++. Разработчику приходится принимать меньше решений и преодолевать меньше препятствий.

Обработка исключений: борьба с ошибками

С первых дней существования языков программирования обработка ошибок была одним из самых каверзных вопросов. Разработать хороший механизм обработки ошибок очень трудно, поэтому многие языки попросту игнорируют эту проблему, оставляя ее разработчикам программных библиотек. Последние предоставляют половинчатые решения, которые работают во многих ситуациях, но которые часто можно попросту обойти (как правило, просто не обращая на них внимания). Главная проблема многих механизмов обработки исключений состоит в том, что они полагаются на добросовестное соблюдение программистом правил, выполнение которых не обеспечивается языком. Если программист проявит невнимательность — а это часто происходит при спешке в работе, — он может легко забыть об этих механизмах.

Механизм обработки исключений встраивает обработку ошибок прямо в язык программирования или даже в операционную систему. *Исключение* представляет собой объект, генерируемый на месте возникновения ошибки, который затем может быть «перехвачен» подходящим обработчиком исключений, предназначенным для ошибок определенного типа. Обработка исключений словно определяет параллельный путь выполнения программы, вступающий в силу, когда что-то идет не по плану. И так как она определяет отдельный путь исполнения, код обработки ошибок не смешивается с обычным кодом. Это упрощает написание программ, поскольку вам не приходится постоянно проверять возможные ошибки. Вдобавок исключение принципиально отличается от числового кода ошибки, возвращаемого методом, или флага, устанавливаемого в случае проблемной ситуации, — последние могут быть проигнорированы. Исключение же нельзя пропустить, оно обязательно будет где-то обработано. Наконец, исключения дают возможность восстановить нормальную работу программы после неверной операции. Вместо того чтобы просто завершить программу, можно исправить ситуацию и продолжить ее выполнение; тем самым повышается надежность программы.

Механизм обработки исключений Java выделяется среди остальных, потому что он был встроен в язык с самого начала, и разработчик обязан его использовать. Если он не напишет кода для подобающей обработки исключений, компилятор выдаст ошибку. Подобный последовательный подход иногда заметно упрощает обработку ошибок.

Стоит отметить, что обработка исключений не является особенностью объектно-ориентированного языка, хотя в этих языках исключение обычно представлено объектом. Такой механизм существовал и до возникновения объектно-ориентированного программирования.

Параллельное выполнение

Одной из фундаментальных концепций программирования является идея одновременного выполнения нескольких операций. Многие задачи требуют, чтобы программа прервала свою текущую работу, решила какую-то другую задачу, а затем вернулась в основной процесс. Проблема решалась разными способами. На первых порах программисты, знающие машинную архитектуру, писали процедуры обработки прерываний, то есть приостановка основного процесса выполнялась на аппаратном уровне. Такое решение работало неплохо, но оно было сложным и немобильным, что значительно усложняло перенос подобных программ на новые типы компьютеров.

Иногда прерывания действительно необходимы для выполнения операций задач, критичных по времени, но существует целый класс задач, где просто нужно разбить задачу на несколько отдельно выполняемых частей, так, чтобы программа быстрее реагировала на внешние воздействия. Эти отдельно выполняемые части программы называются потоками, а весь принцип получил название *многозадачности*, или *параллельных вычислений*. Часто встречающийся пример многозадачности — пользовательский интерфейс. В программе, разбитой на потоки, пользователь может нажать кнопку и получить быстрый ответ, не ожидая, пока программа завершит текущую операцию.

Обычно задачи всего лишь определяют схему распределения времени на однопроцессорном компьютере. Но если операционная система поддерживает многопроцессорную обработку, каждая задача может быть назначена на отдельный процессор; так достигается настоящий параллелизм. Одно из удобных свойств встроенной в язык многозадачности состоит в том, что программисту не нужно знать, один процессор в системе или несколько. Программа логически разделяется на потоки, и если машина имеет больше одного процессора, она исполняется быстрее, без каких-либо специальных настроек.

Все это создает впечатление, что потоки использовать очень легко. Но тут кроется подвох: совместно используемые ресурсы. Если несколько потоков пытаются одновременно получить доступ к одному ресурсу, возникнут проблемы. Например, два процесса не могут одновременно посылать информацию на принтер. Для предотвращения конфликта совместные ресурсы (такие, как принтер) должны блокироваться во время использования. Поток блокирует ресурс, завершает свою операцию, а затем снимает блокировку для того, чтобы кто-то еще смог получить доступ к ресурсу.

Поддержка параллельного выполнения встроена в язык Java, а с выходом Java SE5 к ней добавилась значительная поддержка на уровне библиотек.

Java и Интернет

Если Java представляет собой очередной язык программирования, возникает вопрос: чем же он так важен и почему преподносится как революционный шаг в разработке программ? С точки зрения традиционных задач программирования ответ очевиден не сразу. Хотя язык Java пригодится и при построении автономных приложений, самым важным его применением было и остается программирование для сети World Wide Web.

Что такое Web?

На первый взгляд Web выглядит довольно загадочно из-за обилия новомодных терминов вроде «серфинга», «присутствия» и «домашних страниц». Чтобы понять, что же это такое, полезно представить себе картину в целом, — но сначала необходимо разобраться во взаимодействии клиент–серверных систем, которые представляют собой одну из самых сложных задач компьютерных вычислений.

Вычисления «клиент–сервер»

Основная идея клиент–серверных систем состоит в том, что у вас существует централизованное хранилище информации — обычно в форме базы данных — и эта информация доставляется по запросам каких-либо групп людей или компьютеров. В системе «клиент–сервер» ключевая роль отводится централизованному хранилищу информации, которое обычно позволяет изменять данные так, что эти изменения будут быстро переданы пользователям информации. Все вместе: хранилище информации, программы, распределяющие информацию, и компьютер, на котором хранятся программы и данные, называется *сервером*. Программное обеспечение на машине пользователя, которое устанавливает связь с сервером, получает информацию, обрабатывает ее и затем отображает соответствующим образом, называется *клиентом*.

Таким образом, основная концепция клиент–серверных вычислений не так уж сложна. Проблемы возникают из-за того, что получить доступ к серверу пытаются сразу несколько клиентов одновременно. Обычно для решения привлекается система управления базой данных, и разработчик пытается «оптимизировать» структуру данных, распределяя их по таблицам. Дополнительно система часто дает возможность клиенту добавлять новую информацию на сервер. А это значит, что новая информация клиента должна быть защищена от потери во время сохранения в базе данных, а также от возможности ее перезаписи данными другого клиента. (Это называется обработкой транзакций.) При изменении клиентского программного обеспечения необходимо не только скомпилировать и протестировать его, но и установить на клиентских машинах, что может обойтись гораздо дороже, чем можно представить. Особенно сложно организовать поддержку множества различных операционных систем и компьютерных архитектур. Наконец, необходимо учитывать важнейший фактор производительности: к серверу одновременно могут поступать сотни запросов, и малейшая задержка грозит серьезными последствиями. Для уменьшения задержки программисты стараются распределить вычисления, зачастую даже проводя их на клиентской машине, а иногда и переводя на дополнительные серверные машины, используя так называемое *связующее программное обеспечение* (middleware). (Программы-посредники также упрощают сопровождение программ.)

Простая идея распространения информации между людьми имеет столько уровней сложности в своей реализации, что в целом ее решение кажется недостижимым. И все-таки она жизненно необходима: примерно половина всех задач программирования основана именно на ней. Она задействована в решении разнообразных проблем, от обслуживания заказов и операций по кредитным карточкам, до распространения всевозможных данных — научных, правительственных, котировок акций... Список можно продолжать до бесконечности. В прошлом для каждой новой задачи приходилось создавать отдельное решение. Эти решения непросто создавать, еще труднее ими пользоваться, и пользователю приходилось изучать новый интерфейс с каждой новой программой. Задача клиент—серверных вычислений нуждается в более широком подходе.

Web как гигантский сервер

Фактически Web представляет собой одну огромную систему «клиент—сервер». Впрочем, это еще не все: в единой сети одновременно сосуществуют все серверы и клиенты. Впрочем, этот факт вас не должен интересовать, поскольку обычно вы соединяетесь и взаимодействуете только с одним сервером (даже если его приходится разыскивать по всему миру).

На первых порах использовался простой однонаправленный обмен информацией. Вы делали запрос к серверу, он отсылал вам файл, который обрабатывала для вас ваша программа просмотра (то есть клиент). Но вскоре простого получения статических страниц с сервера стало недостаточно. Пользователи хотели использовать все возможности системы «клиент—сервер», отсылать информацию от клиента к серверу, чтобы, например, просматривать базу данных сервера, добавлять новую информацию на сервер или делать заказы (что требовало особых мер безопасности). Эти изменения мы постоянно наблюдаем в процессе развития Web.

Средства просмотра Web (браузеры) стали большим шагом вперед: они ввели понятие информации, которая одинаково отображается на любых типах компьютеров. Впрочем, первые браузеры были все же примитивны и быстро перестали соответствовать предъявляемым требованиям. Они оказались не особенно интерактивны и тормозили работу как серверов, так и Интернета в целом — при любом действии, требующем программирования, приходилось посылать информацию серверу и ждать, когда он ее обработает. Иногда приходилось ждать несколько минут только для того, чтобы узнать, что вы пропустили в запросе одну букву. Так как браузер представлял собой только средство просмотра, он не мог выполнить даже простейших программных задач. (С другой стороны, это гарантировало безопасность — пользователь был огражден от запуска программ, содержащих вирусы или ошибки.)

Для решения этих задач предпринимались разные подходы. Для начала были улучшены стандарты отображения графики, чтобы браузеры могли отображать анимацию и видео. Остальные задачи требовали появления возможности запуска программ на машине клиента, внутри браузера. Это было названо *программированием на стороне клиента*.

Программирование на стороне клиента

Изначально система взаимодействия «сервер—браузер» разрабатывалась для интерактивного содержимого, но поддержка этой интерактивности была полностью возложена на сервер. Сервер генерировал статические страницы для браузера клиента, который их

просто обрабатывал и показывал. Стандарт *HTML* поддерживает простейшие средства ввода данных: текстовые поля, переключатели, флажки, списки и раскрывающиеся списки, вместе с кнопками, которые могут выполнить только два действия: сброс данных формы и ее отправка серверу. Отправленная информация обрабатывается интерфейсом *CGI* (Common Gateway Interface), поддерживаемым всеми веб-серверами. Текст запроса указывает CGI, как именно следует поступить с данными. Чаще всего по запросу запускается программа из каталога *cgi-bin* на сервере. (В строке с адресом страницы в браузере, после отправки данных формы, иногда можно разглядеть в мешанине символов подстроку *cgi-bin*.) Такие программы можно написать почти на всех языках. Обычно используется Perl, так как он ориентирован на обработку текста, а также является интерпретируемым языком, соответственно, может быть использован на любом сервере, независимо от типа процессора или операционной системы. Впрочем, язык Python (мой любимый язык — зайдите на www.Python.org) постепенно отвоевывает у него «территорию» благодаря своей мощи и простоте.

Многие мощные веб-серверы сегодня функционируют целиком на основе CGI; в принципе, эта технология позволяет решать почти любые задачи. Однако сайты, построенные на CGI-программах, тяжело сопровождать, и на них существуют проблемы со скоростью отклика. Время отклика CGI-программы зависит от количества посылаемой информации, а также от загрузки сервера и сети. (Из-за всего упомянутого запуск CGI-программы может занять продолжительное время.) Первые проектировщики Web не предвидели, как быстро истощатся ресурсы системы при ее использовании в различных приложениях. Например, выводить графики в реальном времени в ней почти невозможно, так как при любом изменении ситуации необходимо построить новый GIF-файл и передать его клиенту. Без сомнения, у вас есть собственный горький опыт — например, полученный при простой посылке данных формы. Вы нажимаете кнопку для отправки информации; сервер запускает CGI-программу, которая обнаруживает ошибку, формирует HTML-страницу, сообщающую вам об этом, а затем отсылает эту страницу в вашу сторону; вам приходится набирать данные заново и повторять попытку. Это не только медленно, это попросту неэлегантно.

Проблема решается программированием на стороне клиента. Как правило, браузеры работают на мощных компьютерах, способных на решение широкого диапазона задач, а при стандартном подходе на базе HTML компьютер просто ожидает, когда ему подадут следующую страницу. При клиентском программировании браузеру поручается вся работа, которую он способен выполнить, а для пользователя это оборачивается более быстрой работой в сети и улучшенной интерактивностью.

Впрочем, обсуждение клиентского программирования мало чем отличается от дискуссий о программировании в целом. Условия все те же, но платформы разные: браузер напоминает сильно усеченную операционную систему. В любом случае приходится программировать, поэтому программирование на стороне клиента порождает головокружительное количество проблем и решений. В завершение этого раздела приводится обзор некоторых проблем и подходов, свойственных программированию на стороне клиента.

Модули расширения

Одним из важнейших направлений в клиентском программировании стала разработка модулей расширения (*plug-ins*). Этот подход позволяет программисту добавить

к браузеру новые функции, загрузив небольшую программу, которая встраивается в браузер. Фактически с этого момента браузер обзаводится новой функциональностью. (Модуль расширения загружается только один раз.) Подключаемые модули позволили оснастить браузеры рядом быстрых и мощных нововведений, но написание такого модуля — совсем непростая задача, и вряд ли каждый раз при создании какого-то нового сайта вы захотите создавать расширения. Ценность модулей расширения для клиентского программирования состоит в том, что они позволяют опытному программисту дополнить браузер новыми возможностями, не спрашивая разрешения у его создателя. Таким образом, модули расширения предоставляют «черный ход» для интеграции новых языков программирования на стороне клиента (хотя и не все языки реализованы в таких модулях).

Языки сценариев

Разработка модулей расширения привела к появлению множества языков для написания сценариев. Используя язык сценария, вы встраиваете клиентскую программу прямо в HTML-страницу, а модуль, обрабатывающий данный язык, автоматически активизируется при ее просмотре. Языки сценария обычно довольно просты для изучения; в сущности, сценарный код представляет собой текст, входящий в состав HTML-страницы, поэтому он загружается очень быстро, как часть одного запроса к серверу во время получения страницы. Расплачиваться за это приходится тем, что любой в силах просмотреть (и украсть) ваш код. Впрочем, вряд ли вы будете писать что-либо заслуживающее подражания и утонченное на языках сценариев, поэтому проблема копирования кода не так уж страшна.

Языком сценариев, который поддерживается практически любым браузером без установки дополнительных модулей, является JavaScript (имеющий весьма мало общего с Java; имя было выбрано для того, чтобы «урвать» кусочек успеха Java на рынке). К сожалению, исходные реализации JavaScript в разных браузерах довольно сильно отличались друг от друга, и даже между разными версиями одного браузера. Стандартизация JavaScript в форме *ECMAScript* была полезна, но потребовалось время, чтобы ее поддержка появилась во всех браузерах (вдобавок компания Microsoft активно продвигала собственный язык VBScript, отдаленно напоминавший JavaScript). В общем случае разработчику приходится ограничиваться минимумом возможностей JavaScript, чтобы код гарантированно работал во всех браузерах. Что касается обработки ошибок и отладки кода JavaScript, то занятие это в лучшем случае непростое. Лишь недавно разработчикам удалось создать действительно сложную систему, написанную на JavaScript (компания Google, служба GMail), и это потребовало высочайшего напряжения сил и опыта.

Это показывает, что языки сценариев, используемые в браузерах, были предназначены для решения круга определенных задач, в основном для создания более насыщенного и интерактивного графического пользовательского интерфейса (GUI). Однако язык сценариев может быть использован для решения 80 процентов задач клиентского программирования. Ваша задача может как раз входить в эти 80 процентов. Поскольку языки сценариев позволяют легко и быстро создавать программный код, вам стоит сначала рассмотреть именно такой язык, перед тем как переходить к более сложным технологическим решениям вроде Java.

Java

Если языки сценариев берут на себя 80 процентов задач клиентского программирования, кому же тогда «по зубам» остальные 20 процентов? Для них наиболее популярным решением сегодня является Java. Это не только мощный язык программирования, разработанный с учетом вопросов безопасности, платформенной совместимости и интернационализации, но также постоянно совершенствуемый инструмент, дополняемый новыми возможностями и библиотеками, которые элегантно вписываются в решение традиционно сложных задач программирования: многозадачности, доступа к базам данных, сетевого программирования и распределенных вычислений. Клиентское программирование на Java сводится к разработке *апплетов*, а также к использованию пакета *Java Web Start*.

Апплет — мини-программа, которая может исполняться только внутри браузера. Апплеты автоматически загружаются в составе веб-страницы (так же, как загружается, например, графика). Когда апплет активизируется, он выполняет программу. Это одно из преимуществ апплета — он позволяет автоматически распространять программы для клиентов с сервера именно тогда, когда пользователю понадобятся эти программы, и не раньше. Пользователь получает самую свежую версию клиентской программы, без всяких проблем и трудностей, связанных с переустановкой. В соответствии с идеологией Java, программист создает только одну программу, которая автоматически работает на всех компьютерах, где имеются браузеры со встроенным интерпретатором Java. (Это верно практически для всех компьютеров.) Так как Java является полноценным языком программирования, как можно большая часть работы должна выполняться на стороне клиента перед обращением к серверу (или после него). Например, вам не понадобится пересылать запрос по Интернету, чтобы узнать, что в полученных данных или каких-то параметрах была ошибка, а компьютер клиента сможет быстро начертить какой-либо график, не ожидая, пока это сделает сервер и отошлет обратно файл с изображением. Такая схема не только обеспечивает мгновенный выигрыш в скорости и отзывчивости, но также снижает загрузку основного сетевого транспорта и серверов, предотвращая замедление работы с Интернетом в целом.

Альтернативы

Честно говоря, апплеты Java не оправдали начальных восторгов. При первом появлении Java все относились к апплетам с большим энтузиазмом, потому что они делали возможным серьезное программирование на стороне клиента, повышали скорость отклика и снижали загрузку канала для интернет-приложений. Апплетам предрекали большое будущее.

И действительно, в Web можно встретить ряд очень интересных апплетов. И все же массовый переход на апплеты так и не состоялся. Вероятно, главная проблема заключалась в том, что загрузка 10-мегабайтного пакета для установки среды Java Runtime Environment (JRE) слишком пугала рядового пользователя. Тот факт, что компания Microsoft не стала включать JRE в поставку Internet Explorer, окончательно решил судьбу апплетов. Как бы то ни было, апплеты Java так и не получили широкого применения.

Впрочем, апплеты и приложения *Java Web Start* в некоторых ситуациях приносят большую пользу. Если конфигурация компьютеров конечных пользователей находится под контролем (например, в организациях), применение этих технологий для

распространения и обновления клиентских приложений вполне оправдано; оно экономит немало времени, труда и денег (особенно при частых обновлениях).

В главе 22 будет рассмотрена перспективная новая технология Macromedia *Flex*, позволяющая создавать аналоги апплетов на базе Flash. Так как модуль Flash Player доступен более чем в 98 процентах всех браузеров (на платформах Windows, Linux и Mac), его можно считать общепринятым стандартом. Операции установки и обновления Flash Player выполняются быстро и просто. Язык ActionScript основан на ECMAScript, так что выглядит он относительно знакомо, но при этом Flex позволяет программировать, не отвлекаясь на специфику браузеров, а следовательно, данная технология намного привлекательнее JavaScript. В области программирования на стороне клиента эта альтернатива вполне заслуживает внимания.

.NET и C#

Некоторое время основным соперником Java-апплетов считались компоненты ActiveX от компании Microsoft, хотя они и требовали для своей работы наличия на машине клиента Windows. Теперь Microsoft противопоставила Java полноценных конкурентов: это платформа .NET и язык программирования C#. Платформа .NET представляет собой примерно то же самое, что и *виртуальная машина Java* (JVM) и библиотеки Java, а язык C# имеет явное сходство с языком Java. Вне всяких сомнений, это лучшее, что создала компания Microsoft в области языков и сред программирования. Конечно, разработчики из Microsoft имели некоторое преимущество; они видели, что в Java удалось, а что нет, и могли отталкиваться от этих фактов, но результат получился вполне достойным. Впервые с момента своего появления у Java появился реальный соперник. Разработчикам из Sun пришлось как следует взглянуть на C#, выяснить, по каким причинам программисты могут захотеть перейти на этот язык, и приложить максимум усилий для серьезного улучшения Java в Java SE5.

В данный момент основные сомнения вызывает вопрос о том, разрешит ли Microsoft *полностью* переносить .NET на другие платформы. В Microsoft утверждают, что никакой проблемы в этом нет и проект Mono (www.go-mono.com) предоставляет частичную реализацию .NET для Linux. Впрочем, раз реализация эта неполная, то пока Microsoft не решит выкинуть из нее какую-либо часть, делать ставку на .NET как на межплатформенную технологию еще рано.

Интернет и интрасети

Web представляет решение наиболее общего характера для клиент—серверных задач, так что имеет смысл использовать ту же технологию для решения задач в частных случаях; в особенности это касается классического клиент—серверного взаимодействия *внутри* компании. При традиционном подходе «клиент—сервер» возникают проблемы с различиями в типах клиентских компьютеров, к ним добавляется трудность установки *новых* программ для клиентов; обе проблемы решаются веб-браузерами и программированием на стороне клиента. Когда технология Web используется для формирования информационной сети внутри компании, ее называют интрасетью. Интрасети предоставляют гораздо большую безопасность в сравнении с Интернетом, потому что вы можете физически контролировать доступ к серверам вашей компании. Что касается обучения, человеку, понимающему концепцию браузера, гораздо легче разобраться в разных страницах и апплетах, так что время освоения новых систем сокращается.

Проблема безопасности подводит нас к одному из направлений, которое автоматически возникает в клиентском программировании. Если ваша программа выполняется в Интернете, то вы не знаете, на какой платформе ей предстоит работать. Приходится проявлять особую осторожность, чтобы избежать распространения некорректного кода. Здесь нужны межплатформенные и безопасные решения, наподобие Java или языка сценариев.

В интрасетях действуют другие ограничения. Довольно часто все машины сети работают на платформе Intel/Windows. В интрасети вы отвечаете за качество своего кода и можете устранять ошибки по мере их обнаружения. Вдобавок у вас уже может накопиться коллекция решений, которые проверены на прочность в более традиционных клиент—серверных системах, в то время как новые программы придется вручную устанавливать на машину клиента при каждом обновлении. Время, затрачиваемое на обновления, является самым веским доводом в пользу браузерных технологий, где обновления осуществляются невидимо и автоматически (то же позволяет сделать Java Web Start). Если вы участвуете в обслуживании интрасети, благоразумнее всего использовать тот путь, который позволит привлечь уже имеющиеся наработки, не переписывая программы на новых языках.

Сталкиваясь с огромным количеством решений задач клиентского программирования, способным поставить в тупик любого проектировщика, лучше всего оценить их с позиций соотношения «затраты/прибыли». Рассмотрите ограничения вашей задачи и попробуйте представить кратчайший способ ее решения. Так как клиентское программирование все же остается программированием, всегда актуальны технологии разработки, обещающие наиболее быстрое решение. Такая активная позиция даст вам возможность подготовиться к неизбежным проблемам разработки программ.

Программирование на стороне сервера

Наше обсуждение обошло стороной тему серверного программирования, которое, как считают многие, является самой сильной стороной Java. Что происходит, когда вы посылаете запрос серверу? Чаще всего запрос сводится к простому требованию «отправьте мне этот файл». Браузер затем обрабатывает файл подходящим образом: как HTML-страницу, как изображение, как Java-апплет, как сценарий и т. п.

Более сложный запрос к серверу обычно связан с обращением к базе данных. В самом распространенном случае делается запрос на сложный поиск в базе данных, результаты которого сервер затем преобразует в HTML-страницу и посылает вам. (Конечно, если клиент способен производить какие-то действия с помощью Java или языка сценариев, данные могут быть обработаны и у него, что будет быстрее и снизит загрузку сервера.) А может быть, вам понадобится зарегистрироваться в базе данных при присоединении к какой-то группе или оформить заказ, что потребует изменений в базе данных. Подобные запросы должны обрабатываться неким кодом на сервере; в целом это и называется серверным программированием. Традиционно программирование на сервере осуществлялось на Perl, Python, C++ или другом языке, позволяющем создавать программы CGI, но появляются и более интересные варианты. К их числу относятся и основанные на Java веб-серверы, позволяющие заниматься серверным программированием на Java с помощью так называемых *сервлетов*. Сервлеты и их детища, JSPs, составляют две

основные причины для перехода компаний по разработке веб-содержимого на Java, в главном из-за того, что они решают проблемы несовместимости различных браузеров. Тема программирования на стороне сервера рассматривается в материалах курса *Thinking in Enterprise Java* на сайте www.MindView.net.

Несмотря на все разговоры о Java как языке интернет-программирования, Java в действительности является полноценным языком программирования, способным решить практически все задачи, решаемые на других языках. Преимущества Java не ограничиваются хорошей переносимостью: это и пригодность к решению задач программирования, и устойчивость к ошибкам, и большая стандартная библиотека, и многочисленные разработки сторонних фирм — как существующие, так и постоянно появляющиеся.

Резюме

Вы знаете, как выглядят процедурные программы: определения данных и вызовы функций. Чтобы выяснить предназначение такой программы, необходимо приложить усилие, просматривая функции и создавая в уме общую картину. Именно из-за этого создание таких программ требует использования промежуточных средств — сами по себе они больше ориентированы на компьютер, а не на решаемую задачу.

Так как ООП добавляет много новых понятий к тем, что уже имеются в процедурных языках, естественно будет предположить, что код Java будет гораздо сложнее, чем аналогичный метод на процедурном языке. Но здесь вас ждет приятный сюрприз: хорошо написанную программу на Java обычно гораздо легче понять, чем ее процедурный аналог. Все, что вы видите — это определения объектов, представляющих понятия пространства решения (а не понятия компьютерной реализации), и сообщения, посылаемые этим объектам, которые представляют действия в этом пространстве. Одно из преимуществ ООП как раз и состоит в том, что хорошо спроектированную программу можно понять, просто проглядывая исходные тексты. К тому же обычно приходится писать гораздо меньше кода, поскольку многие задачи с легкостью решаются уже существующими библиотеками классов.

ООП и язык Java подходят не для всех. Очень важно сначала выяснить свои потребности, чтобы решить, удовлетворит ли вас переход на Java или лучше остановить свой выбор на другой системе программирования (в том числе и на той, что вы сейчас используете). Если вы знаете, что в обозримом будущем столкнетесь с весьма специфическими потребностями или в вашей работе будут действовать ограничения, с которыми Java не справляется, лучше рассмотреть другие возможности. (В особенности я рекомендую присмотреться к языку Python, www.Python.org.) Выбирая Java, необходимо понимать, какие еще доступны варианты и почему вы выбрали именно этот путь.

2

Все является объектом

Если бы мы говорили на другом языке, то и мир воспринимали бы по-другому.

Людвиг Витгенштейн (1889–1951)

Хотя язык Java основан на C++, он является более «чистокровным» объектно-ориентированным языком.

Как C++, так и Java относятся к семейству смешанных языков, но для создателей Java эта неоднородность была не так важна, если сравнивать с C++. Смешанный язык позволяет использовать несколько стилей программирования; причиной смешанной природы C++ стало желание сохранить совместимость с языком C. Так как язык C++ является надстройкой над языком C, он включает много нежелательных характеристик своего предшественника, что приводит к излишнему усложнению некоторых аспектов этого языка.

Язык программирования Java подразумевает, что вы занимаетесь только объектно-ориентированным программированием. А это значит, что прежде чем начать с ним работать, нужно «переключиться» на менталитет объектно-ориентированного мира (если вы уже этого не сделали). Выгода от этого начального усилия — возможность программировать на языке, который по простоте изучения и использования превосходит все остальные языки ООП. В этой главе мы рассмотрим основные компоненты Java-программы и узнаем, что в Java (почти) все является объектом.

Для работы с объектами используются ссылки

Каждый язык программирования имеет свои средства для работы с данными в памяти. Иногда программисту приходится быть постоянно в курсе, какая именно манипуляция производится в программе. С чем вы работаете — с самим объектом или же с каким-то видом его косвенного представления (указатель в C или в C++), требующим особого синтаксиса?

Все эти различия упрощены в Java. Вы обращаетесь с любыми данными как с объектом, и поэтому повсюду используется единый последовательный синтаксис. Хотя вы *обращаетесь* со всеми данными как с объектом, идентификатор, которым вы манипулируете, на самом деле представляет собой *ссылку* на объект¹. Представьте себе телевизор (объект) с пультом дистанционного управления (ссылка). Во время владения этой ссылкой у вас имеется связь с телевизором, но при переключении канала или уменьшения громкости вы распоряжаетесь ссылкой, которая, в свою очередь, манипулирует объектом. А если вам захочется перейти в другое место комнаты, все еще управляя телевизором, вы берете с собой «ссылку», а не сам телевизор.

Также пульт может существовать сам по себе, без телевизора. Таким образом, сам факт наличия ссылки еще не значит наличия присоединенного к ней объекта. Например, для хранения слова или предложения создается ссылка `String`:

```
String s;
```

Однако здесь определяется *только* ссылка, но не объект. Если вы решите послать сообщение `s`, произойдет ошибка, потому что ссылка `s` на самом деле ни к чему не присоединена (телевизора нет). Значит, безопаснее всегда инициализировать ссылку при ее создании:

```
String s = "asdf";
```

В примере используется специальная возможность Java: инициализация строк текстом в кавычках. Вы будете использовать более общий способ инициализации объектов.

Все объекты должны создаваться явно

Когда вы определяете ссылку, желательно присоединить ее к новому объекту. Обычно это делается при помощи ключевого слова `new`. Фактически оно означает: «Создайте мне новый объект». В предыдущем примере можно написать:

```
String s = new String("asdf");
```

¹ Этот вопрос очень важен. Некоторые программисты скажут: «понятно, это указатель», но это предполагает соответствующую реализацию. Также ссылки Java по синтаксису более похожи на ссылки C++, чем на его указатели. В первом издании книги я решил ввести новый термин «дескриптор» (`handle`), потому что между ссылками Java и C++ существуют серьезные отличия. Я основывался на опыте C++ и не хотел сбивать с толку программистов на этом языке, так как большей частью именно они будут изучать Java. Во втором издании я решил прибегнуть к более традиционному термину «ссылка», предполагая, что это поможет быстрее освоить новые особенности языка, в котором и без моих новых терминов много необычного. Однако есть люди, возражающие даже против термина «ссылка». Я прочитал в одной книге, что «совершенно неверно говорить, что Java поддерживает передачу объектов по ссылке», потому что идентификаторы объектов Java *на самом деле* (согласно автору) являются ссылками на объекты. И (он продолжает) все *фактически* передается по значению. Так что передача идет не по ссылке, а «ссылка на объект передается по значению». Можно поспорить с тем, насколько точны столь запутанные рассуждения, но я полагаю, что мое объяснение упрощает понимание концепции и ничему не вредит (блюстителю чистоты языка могут сказать, что это неправда, но я всегда могу возразить, что речь идет всего лишь о подходящей абстракции).

Это не только значит «предоставьте мне новый объект `String`», но также указывает, *как* создать строку посредством передачи начального набора символов.

Конечно, кроме `String`, в Java имеется множество готовых типов. Но еще важнее то, что вы можете создавать свои собственные типы. Вообще говоря, именно создание новых типов станет вашим основным занятием при программировании на Java, и именно его мы будем рассматривать в книге.

Где хранятся данные

Полезно отчетливо представлять, что происходит во время работы программы, и в частности, как данные размещаются в памяти. Существует пять разных мест для хранения данных.

1. **Регистры.** Это самое быстрое хранилище, потому что данные хранятся прямо внутри процессора. Однако количество регистров жестко ограничено, поэтому регистры используются компилятором по мере необходимости. У вас нет прямого доступа к регистрам, вы не найдете и малейших следов их поддержки в языке. (С другой стороны, языки C и C++ позволяют порекомендовать компилятору хранить данные в регистрах.)
2. **Стек.** Эта область хранения данных находится в общей оперативной памяти (RAM), но процессор предоставляет прямой доступ к ней с использованием *указателя стека*. Указатель стека перемещается вниз для выделения памяти или вверх для ее освобождения. Это чрезвычайно быстрый и эффективный способ размещения данных, по скорости уступающий только регистрам. Во время обработки программы компилятор Java должен знать жизненный цикл данных, размещаемых в стеке. Это ограничение уменьшает гибкость ваших программ, поэтому, хотя некоторые данные Java хранятся в стеке (особенно ссылки на объекты), сами объекты Java не помещаются в стек.
3. **Куча.** Пул памяти общего назначения (находится также в RAM), в котором размещаются все объекты Java. Преимущество кучи состоит в том, что компилятору не обязательно знать, как долго просуществуют находящиеся там объекты. Таким образом, работа с кучей дает значительное преимущество в гибкости. Когда вам нужно создать объект, вы пишете код с использованием `new`, и память выделяется из кучи во время выполнения программы. Конечно, за гибкость приходится расплачиваться: выделение памяти из кучи занимает больше времени, чем в стеке (даже если бы вы *могли* явно создавать объекты в стеке, как в C++).
4. **Постоянное хранилище.** Значения констант часто встраиваются прямо в код программы, так как они неизменны. Иногда такие данные могут размещаться в постоянной памяти (ROM), если речь идет о «встроенных» системах¹.
5. **Внешнее хранилище.** Если данные хранятся вне программы, они могут существовать и тогда, когда она не выполняется. Два основных примера: *потокковые объекты*

¹ В качестве примера можно привести пулы строк. Все строковые литералы и константы со строковыми значениями автоматически помещаются в специальную статическую область памяти.

(streamed objects), в которых объекты представлены в виде потока байтов, обычно используются для передачи на другие машины, и *долгоживущие* (persistent) *объекты*, которые запоминаются на диске и сохраняют свое состояние даже после окончания работы программы. Особенностью этих видов хранения данных является возможность перевода объектов в нечто, что может быть сохранено на другом носителе информации, а потом восстановлено в виде обычного объекта, хранящегося в оперативной памяти. В Java организована поддержка *легковесного* (lightweight) *сохранения состояния*, а такие механизмы, как JDBC и Hibernate, предоставляют более совершенную поддержку сохранения и выборки информации об объектах из баз данных.

Особый случай: примитивные типы

Одна из групп типов, часто применяемых при программировании, требует особого обращения. Их можно назвать «примитивными» типами (табл. 2.1). Дело в том, что создание объекта с помощью new — особенно маленькой, простой переменной — недостаточно эффективно, так как new помещает объекты в кучу. В таких случаях Java следует примеру языков C и C++. То есть вместо создания переменной с помощью new создается «автоматическая» переменная, *не являющаяся ссылкой*. Переменная напрямую хранит значение и располагается в стеке, так что операции с ней гораздо производительнее.

В Java размеры всех примитивных типов жестко фиксированы. Они не меняются с переходом на иную машинную архитектуру, как это происходит во многих других языках. Незыблемость размера — одна из причин улучшенной переносимости Java-программ.

Таблица 2.1. Примитивные типы

Примитивный тип	Размер, битов	Минимум	Максимум	Тип упаковки
boolean (логические значения)	—	—	—	Boolean
char (символьные значения)	16	Unicode 0	Unicode 2 ¹⁶ –1	Character
byte (байт)	8	–128	+127	Byte
short (короткое целое)	16	–2 ¹⁵	+2 ¹⁵ –1	Short
int (целое)	32	–2 ³¹	+2 ³¹ –1	Integer
long (длинное целое)	64	–2 ⁶³	+2 ⁶³ –1	Long
float (число с плавающей запятой)	32	IEEE754	IEEE754	Float
double (число с повышенной точностью)	64	IEEE754	IEEE754	Double
Void («пустое» значение)	—	—	—	Void

Все числовые значения являются знаковыми, так что не ищите слова unsigned.

Размер типа boolean явно не определяется; указывается лишь то, что этот тип может принимать значения true и false.

Классы-«обертки» позволяют создать в куче не-примитивный объект для представления примитивного типа. Например:

```
char c = 'x';
Character ch = new Character(c);
```

Также можно использовать такой синтаксис:

```
Character ch = new Character('x');
```

Механизм *автоматической упаковки* Java SE5 автоматически преобразует примитивный тип в объектную «обертку»:

```
Character ch = 'x';
```

и обратно:

```
char c = ch;
```

Причины создания подобных конструкций будут объяснены в последующих главах.

Числа повышенной точности

В Java существует два класса для проведения арифметических операций повышенной точности: `BigInteger` и `BigDecimal`. Хотя эти классы примерно подходят под определение классов-«обертков», ни один из них не имеет аналога среди примитивных типов.

Оба класса содержат методы, производящие операции, аналогичные тем, что проводятся над примитивными типами. Иначе говоря, с классами `BigInteger` и `BigDecimal` можно делать то же, что с `int` или `float`, просто для этого используются вызовы методов, а не встроенные операции. Также из-за использования увеличенного объема данных операции занимают больше времени. Приходится жертвовать скоростью ради точности.

Класс `BigInteger` поддерживает целые числа произвольной точности. Это значит, что вы можете использовать целочисленные значения любой величины без потери данных во время операций.

Класс `BigDecimal` представляет числа с фиксированной запятой произвольной точности; например, они могут применяться для финансовых вычислений.

За подробностями о конструкторах и методах этих классов обращайтесь к документации JDK.

Массивы в Java

Фактически все языки программирования поддерживают массивы. Использование массивов в C и C++ небезопасно, потому что массивы в этих языках представляют собой обычные блоки памяти. Если программа попытается получить доступ к массиву за пределами его блока памяти или использовать память без предварительной инициализации (типичные ошибки при программировании), последствия могут быть непредсказуемы.

Одной из основных целей Java является безопасность, поэтому многие проблемы, досаждавшие программистам на C и C++, не существуют в Java. Массив в Java гарантированно инициализируется, к нему невозможен доступ за пределами его границ. Проверка границ массива обходится относительно дорого, как и проверка индекса во время выполнения, но предполагается, что повышение безопасности и подъем производительности стоят того (к тому же Java иногда может оптимизировать эти операции).

При объявлении массива объектов на самом деле создается массив ссылок, и каждая из этих ссылок автоматически инициализируется специальным значением, представленным ключевым словом `null`. Оно означает, что ссылка на самом деле не указывает на объект. Вам необходимо присоединять объект к каждой ссылке перед тем, как ее использовать, или при попытке обращения по ссылке `null` во время исполнения программы произойдет ошибка. Таким образом, типичные ошибки при работе с массивами в Java предотвращаются заблаговременно.

Также можно создавать массивы простейших типов. И снова компилятор гарантирует инициализацию — выделенная для нового массива память заполняется нулями.

Массивы будут подробнее описаны в последующих главах.

Объекты никогда не приходится удалять

В большинстве языков программирования концепция жизненного цикла переменной требует относительно заметных усилий со стороны программиста. Сколько «живет» переменная? Если ее необходимо удалить, когда это следует делать? Путаница со сроками существования переменных может привести ко многим ошибкам, и этот раздел показывает, насколько Java упрощает решение затронутого вопроса, выполняя всю работу по удалению за вас.

Ограничение области действия

В большинстве процедурных языков существует понятие *области действия* (scope). Область действия определяет как видимость, так и срок жизни имен, определенных внутри нее. В C, C++ и Java область действия определяется положением фигурных скобок `{ }`. Например:

```
{
    int x = 12;
    // доступно только x
    {
        int q = 96;
        // доступны как x, так и q
    }
    // доступно только x
    // q находится "за пределами видимости"
}
```

Переменная, определенная внутри области действия, доступна только в пределах этой области.

Весь текст после символов `//` и до конца строки является комментарием.

Отступы упрощают чтение программы на Java. Так как Java относится к языкам со свободным форматом, дополнительные пробелы, табуляция и переводы строк не влияют на результирующую программу.

Учтите, что следующая конструкция *не разрешена*, хотя в C и C++ она возможна:

```
{
    int x = 12;
    {
        int x = 96; // неверно
    }
}
```

Компилятор объявит, что переменная `x` уже была определена. Таким образом, возможность языков `C` и `C++` «замещать» переменные из внешней области действия не поддерживается. Создатели `Java` посчитали, что она приводит к излишнему усложнению программ.

Область действия объектов

Объекты `Java` имеют другое время жизни в сравнении с примитивами. Объект, созданный оператором `Java new`, будет доступен вплоть до конца области действия. Если вы напишете:

```
{
    String s = new String("a string");
} // конец области действия
```

то ссылка `s` исчезнет в конце области действия. Однако объект `String`, на который указывала `s`, все еще будет занимать память. В показанном фрагменте кода невозможно получить доступ к объекту, потому что единственная ссылка вышла за пределы видимости. В следующих главах вы узнаете, как передаются ссылки на объекты и как их можно копировать во время работы программы.

Благодаря тому что объекты, созданные `new`, существуют ровно столько, сколько вам нужно, в `Java` исчезает целый пласт проблем, присущих `C++`. В `C++` приходится не только следить за тем, чтобы объекты продолжали существовать на протяжении своего жизненного цикла, но и удалять объекты после завершения работы с ними.

Возникает интересный вопрос. Если в `Java` объекты остаются в памяти, что же мешает им постепенно занять всю память и остановить выполнение программы? Именно это произошло бы в данном случае в `C++`. Однако в `Java` существует *уборщик мусора* (garbage collector), который наблюдает за объектами, созданными оператором `new`, и определяет, на какие из них больше нет ссылок. Тогда он освобождает память этих объектов, которая становится доступной для дальнейшего использования. Таким образом, вам никогда не придется «очищать» память вручную. Вы просто создаете объекты, и как только надобность в них отпадает, эти объекты исчезают сами по себе. При таком подходе исчезает целый класс проблем программирования: так называемые «утечки памяти», когда программист забывает освободить занятую память.

Создание новых типов данных

Если все является объектом, что определяет строение и поведение класса объектов? Другими словами, как устанавливается *тип* объекта? Наверное, для этой цели можно было бы использовать ключевое слово `type` («тип»); это было бы вполне разумно. Впрочем, с давних времен повелось, что большинство объектно-ориентированных

языков использовали ключевое слово `class` в смысле «Я собираюсь описать новый тип объектов». За ключевым словом `class` следует имя нового типа. Например:

```
class ATypeName { /* Тело класса */ }
```

Эта конструкция вводит новый тип. Тело класса состоит из комментария (символы `/*` и `*/` и все, что находится между ними, — см. далее в этой главе), и пользы от него немного, но вы можете теперь создавать объект этого типа ключевым словом `new`:

```
ATypeName a = new ATypeName();
```

Впрочем, объекту нельзя «приказать» что-то сделать (то есть послать ему необходимые сообщения) до тех пор, пока для него не будут определены методы.

Поля и методы

При определении класса (строго говоря, вся ваша работа на Java сводится к определению классов, созданию объектов этих классов и отправке сообщений этим объектам) в него можно включить две разновидности элементов: *поля* (fields) (иногда называемые переменными класса) и *методы* (methods) (еще называемые функциями класса). Поле представляет собой объект любого типа, с которым можно работать по ссылке, или объект примитивного типа. Если используется ссылка, ее необходимо инициализировать, чтобы связать с реальным объектом (ключевым словом `new`, как было показано ранее).

Каждый объект использует собственный блок памяти для своих полей данных; совместное использование обычных полей разными объектами класса невозможно. Пример класса с полями:

```
class DataOnly {  
    int i;  
    double d;  
    boolean b;  
}
```

Такой класс ничего не *делает*, кроме хранения данных, но вы можете создать объект этого класса:

```
DataOnly data = new DataOnly();
```

Полям класса можно присваивать значения, но для начала необходимо узнать, как обращаться к членам объекта. Для этого сначала указывается имя ссылки на объект, затем следует точка, а далее имя члена, принадлежащего объекту:

ссылка.член

Например:

```
data.i = 47;  
data.d = 1.1;  
data.b = false;
```

Также ваш объект может содержать другие объекты, данные которых вы хотели бы изменить. Для этого просто продолжите «цепочку из точек». К примеру:

```
myPlane.leftTank.capacity = 100;
```

Класс `DataOnly` не способен ни на что, кроме хранения данных, так как в нем отсутствуют методы. Чтобы понять, как они работают, необходимо разобраться, что такое *аргументы* и *возвращаемые значения*. Вскоре мы вернемся к этой теме.

Значения по умолчанию для полей примитивных типов

Если поле данных относится к примитивному типу, ему гарантированно присваивается значение по умолчанию, даже если оно не было инициализировано явно (табл. 2.2).

Таблица 2.2. Значения по умолчанию для полей примитивных типов

Примитивный тип	Значение по умолчанию
<code>boolean</code>	<code>false</code>
<code>char</code>	<code>'\u0000'</code> (null)
<code>byte</code>	<code>(byte)0</code>
<code>short</code>	<code>(short)0</code>
<code>int</code>	<code>0</code>
<code>long</code>	<code>0L</code>
<code>float</code>	<code>0.0f</code>
<code>double</code>	<code>0.0d</code>

Значения по умолчанию гарантируются Java только в том случае, если переменная используется *как член класса*. Тем самым обеспечивается обязательная инициализация элементарных типов (чего не делается в C++), которая уменьшает вероятность ошибок. Однако значение по умолчанию может быть неверным или даже недопустимым для вашей программы. Переменные всегда лучше инициализировать явно.

Такая гарантия не относится к *локальным переменным*, которые не являются полями класса. Допустим, в определении метода встречается объявление переменной:

```
int x;
```

Переменной `x` будет присвоено случайное значение (как в C и C++); она не будет автоматически инициализирована нулем. Вы отвечаете за присвоение правильного значения перед использованием `x`. Если же вы забудете это сделать, в Java существует очевидное преимущество в сравнении с C++: компилятор выдает ошибку, в которой указано, что переменная не была инициализирована. (Многие компиляторы C++ предупреждают о таких переменных, но в Java это считается ошибкой.)

Методы, аргументы и возвращаемые значения

Во многих языках (таких, как C и C++) для обозначения именованной подпрограммы употребляется термин *функция*. В Java чаще предпочитают термин *метод*, как бы подразумевающий «способ что-то сделать». Если вам хочется, вы можете продолжать пользоваться термином «функция». Разница только в написании, но в дальнейшем в книге будет употребляться преимущественно термин «метод».

Методы в Java определяют сообщения, принимаемые объектом. Основные части метода — имя, аргументы, возвращаемый тип и тело. Вот примерная форма:

```
возвращаемыйТип имяМетода( /* список аргументов */ ) {  
    /* тело метода */  
}
```

Возвращаемый тип — это тип объекта, «выдаваемого» методом после его вызова. Список аргументов определяет типы и имена для информации, которую вы хотите передать в метод. Имя метода и его список аргументов (объединяемые термином *сигнатура*) обеспечивают однозначную идентификацию метода.

Методы в Java создаются только как части класса. Метод может вызываться только для объекта¹, и этот объект должен обладать возможностью произвести такой вызов. Если вы попытаетесь вызвать для объекта несуществующий метод, то получите ошибку компиляции. Вызов метода осуществляется следующим образом: сначала записывается имя объекта, за ним точка, за ней следуют имя метода и его список аргументов:

```
имяОбъекта.имяМетода(arg1, arg2, arg3)
```

Например, представьте, что у вас есть метод `f()`, вызываемый без аргументов, который возвращает значение типа `int`. Если у вас имеется в наличии объект `a`, для которого может быть вызван метод `f()`, в вашей власти использовать следующую конструкцию:

```
int x = a.f();
```

Тип возвращаемого значения должен быть совместим с типом `x`.

Такое действие вызова метода часто называется *отправкой сообщения объекту*. В примере выше сообщением является вызов `f()`, а объектом `a`. Объектно-ориентированное программирование нередко характеризуется обобщающей формулой «отправка сообщений объектам».

Список аргументов

Список аргументов определяет, какая информация передается методу. Как легко догадаться, эта информация — как и все в Java — воплощается в форме объектов, поэтому в списке должны быть указаны как типы передаваемых объектов, так и их имена. Как и в любой другой ситуации в Java, где мы вроде бы работаем с объектами, на самом деле используются ссылки². Впрочем, тип ссылки должен соответствовать типу передаваемых данных. Если предполагается, что аргумент является строкой (то есть объектом `String`), вы должны передать именно строку или ожидайте сообщения об ошибке.

Рассмотрим метод, получающий в качестве аргумента строку (`String`). Следующее определение должно размещаться внутри определения класса, для которого создается метод:

¹ Статические методы, о которых вы узнаете немного позже, вызываются для класса, а не для объекта.

² За исключением уже упомянутых «специальных» типов данных: `boolean`, `byte`, `short`, `char`, `int`, `float`, `long`, `double`. Впрочем, в основном вы будете передавать объекты, а значит, ссылки на них.

```
int storage(String s) {
    return s.length() * 2;
}
```

Метод указывает, сколько байтов потребуется для хранения данных определенной строки. (Строки состоят из символов `char`, размер которых — 16 битов, или 2 байта; это сделано для поддержки набора символов Юникод.) Аргумент имеет тип `String` и называется `s`. Получив объект `s`, метод может работать с ним точно так же, как и с любым другим объектом (то есть отправлять ему сообщения). В данном случае вызывается метод `length()`, один из методов класса `String`; он возвращает количество символов в строке.

Также обратите внимание на ключевое слово `return`, выполняющее два действия. Во-первых, оно означает: «выйти из метода, все сделано». Во-вторых, если метод возвращает значение, это значение указывается сразу же за командой `return`. В нашем случае возвращаемое значение — это результат вычисления `s.length() * 2`.

Метод может возвращать любой тип, но если вы не хотите пользоваться этой возможностью, следует указать, что метод возвращает `void`. Ниже приведено несколько примеров:

```
boolean flag() { return true; }
float naturalLogBase() { return 2.718; }
void nothing() { return; }
void nothing2() {}
```

Когда выходным типом является `void`, ключевое слово `return` нужно лишь для завершения метода, поэтому при достижении конца метода его присутствие необязательно. Вы можете покинуть метод в любой момент, но если при этом указывается возвращаемый тип, отличный от `void`, то компилятор заставит вас (сообщениями об ошибках) вернуть подходящий тип независимо от того, в каком месте метода было прервано выполнение.

К этому моменту может сложиться впечатление, что программа — это просто набор объектов со своими методами, которые принимают другие объекты в качестве аргументов и отправляют им сообщения. По большому счету так оно и есть, но в следующей главе вы узнаете, как производить кропотливую низкоуровневую работу с принятием решений внутри метода. В этой главе достаточно рассмотрения на уровне отправки сообщений.

Создание программы на Java

Есть еще несколько вопросов, которые необходимо понять перед созданием первой программы на Java.

Видимость имен

Проблема управления именами существует в любом языке программирования. Если имя используется в одном из модулей программы и оно случайно совпало с именем в другом модуле у другого программиста, то как отличить одно имя от другого и предотвратить их конфликт? В С это определенно является проблемой, потому что

программа с трудом поддается контролю в условиях «моря» имен. Классы C++ (на которых основаны классы Java) скрывают функции внутри классов, поэтому их имена не пересекаются с именами функций других классов. Однако в C++ допускается использование глобальных данных и глобальных функций, соответственно, конфликты полностью не исключены. Для решения означенной проблемы в C++ введены *пространства имен* (namespaces), которые используют дополнительные ключевые слова.

В языке Java для решения этой проблемы было использовано свежее решение. Для создания уникальных имен библиотек разработчики Java предлагают использовать доменное имя, записанное в обратном порядке, так как эти имена всегда уникальны. Мое доменное имя — *MindView.net*, и утилиты моей программной библиотеки могли бы называться *net.mindview.utility.foibles*. За перевернутым доменным именем следует перечень каталогов, разделенных точками.

В версиях Java 1.0 и 1.1 доменные суффиксы *com*, *edu*, *org*, *net* по умолчанию записывались заглавными буквами, таким образом, имя библиотеки выглядело так: *NET.mindview.utility.foibles*. В процессе разработки Java 2 было обнаружено, что принятый подход создает проблемы, и с тех пор имя пакета записывается строчными буквами.

Такой механизм означает, что все ваши файлы автоматически располагаются в своих собственных пространствах имен и каждый класс в файле должен иметь уникальный идентификатор. Язык сам предотвращает конфликты имен.

Использование внешних компонентов

Когда вам понадобится использовать уже определенный класс в вашей программе, компилятор должен знать, как этот класс обнаружить. Конечно, класс может уже находиться в том же самом исходном файле, откуда он вызывается. В таком случае вы просто его используете — даже если определение класса следует где-то дальше в файле. (В Java не существует проблемы «опережающих ссылок».)

Но что, если класс находится в каком-то внешнем файле? Казалось бы, компилятор должен запросто найти его, но здесь существует проблема. Представьте, что вам необходим класс с неким именем, для которого имеется более одного определения (вероятно, отличающихся друг от друга). Или, что еще хуже, представьте, что вы пишете программу и при ее создании в библиотеку добавляется новый класс, конфликтующий с именем уже существующего класса.

Для решения проблемы вам необходимо устранить все возможные неоднозначности. Задача решается при помощи ключевого слова `import`, которое говорит компилятору Java, какие точно классы вам нужны. Слово `import` приказывает компилятору загрузить *пакет* (package), представляющий собой библиотеку классов. (В других языках библиотека может состоять как из классов, так и из функций и данных, но в Java весь код принадлежит классам.)

Большую часть времени вы будете работать с компонентами из стандартных библиотек Java, поставляющихся с компилятором. Для них не нужны длинные, обращенные доменные имена; например, запись следующего вида:

```
import java.util.ArrayList;
```

сообщает компилятору, что вы хотите использовать класс `ArrayList`. Впрочем, пакет `util` содержит множество классов; возможно, вы захотите использовать несколько из них, не объявляя их по отдельности. Чтобы избежать последовательного перечисления классов, используйте подстановочный символ `*`:

```
import java.util.*;
```

Как правило, целый набор классов импортируется именно таким образом (вместо импортирования каждого класса по отдельности).

Ключевое слово `static`

Обычно при создании класса вы описываете, как объекты этого класса ведут себя и как они выглядят. Объект появляется только после того, как он будет создан ключевым словом `new`, и только начиная с этого момента для него выделяется память и появляется возможность вызова методов.

Но есть две ситуации, в которых такой подход недостаточен. Первая — это когда некоторые данные должны храниться «в единственном числе» независимо от того, сколько было создано объектов класса. Вторая — когда вам потребуется метод, не привязанный ни к какому конкретному объекту класса (то есть метод, который можно вызвать даже при полном отсутствии объектов класса).

Такой эффект достигается использованием ключевого слова `static`, делающего элемент класса *статическим*. Данные или метод, объявленные как `static`, не привязаны к определенному экземпляру этого класса. Поэтому даже если в программе еще не создавался ни один объект класса, вы можете вызывать статический метод или получить доступ к статическим данным. С обычным объектом вам необходимо сначала создать его и использовать для вызова метода или доступа к информации, так как нестатические данные и методы должны точно знать объект, с которым работают.

Некоторые объектно-ориентированные языки используют термины *данные уровня класса* и *методы уровня класса*, подразумевая, что данные и методы существуют только на уровне класса в целом, а не для отдельных объектов этого класса. Иногда эти термины встречаются в литературе по Java.

Чтобы сделать данные или метод статическими, просто поместите ключевое слово `static` перед их определением. Например, следующий код создает статическое поле класса и инициализирует его:

```
class StaticTest {
    static int i = 47;
}
```

Теперь, даже при создании двух объектов `StaticTest`, для элемента `StaticTest.i` выделяется единственный блок памяти. Оба объекта совместно используют одно значение `i`. Пример:

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

В данном примере как `st1.i`, так и `st2.i` имеют одинаковые значения, равные 47, потому что они ссылаются на один блок памяти.

Существует два способа обратиться к статической переменной. Как было видно выше, на нее можно ссылаться по имени объекта, например `st2.i`. Также возможно обратиться к ней прямо через имя класса; для нестатических членов класса такая возможность отсутствует.

```
StaticTest.i++;
```

Оператор `++` увеличивает значение на единицу (инкремент). После выполнения предложения значения `st1.i` и `st2.i` будут равны 48.

Синтаксис с именем класса является предпочтительным, потому что он не только подчеркивает, что переменная является статической, но и в некоторых случаях предоставляет компилятору больше возможностей для оптимизации.

Та же логика верна и для статических методов. Вы можете обратиться к такому методу или через объект, как это делается для всех методов, или в специальном синтаксисе *ИмяКласса.метод()*. Статические методы определяются по аналогии со статическими данными:

```
class Incrementable {  
    static void increment( ) { StaticTest.i++; }  
}
```

Нетрудно заметить, что метод `increment()` класса `Incrementable` увеличивает значение статического поля `i`. Метод можно вызвать стандартно, через объект:

```
Incrementable sf = new Incrementable();  
sf.increment();
```

Или, поскольку метод `increment()` является статическим, можно вызвать его с прямым указанием класса:

```
Incrementable.increment();
```

Если применительно к полям ключевое слово `static` радикально меняет способ определения данных (статические данные существуют на уровне класса, в то время как нестатические данные существуют на уровне объектов), но в отношении методов изменения не столь принципиальны. Одним из важных применений `static` является определение методов, которые могут вызываться без объектов. В частности, это абсолютно необходимо для метода `main()`, который представляет собой точку входа в приложение.

Ваша первая программа на Java

Наконец, мы подошли к первой законченной программе. Она запускается, выводит на экран строку, а затем текущую дату, используя стандартный класс `Date` из стандартной библиотеки `Java`.

```
// HelloDate.java  
import java.util.*;
```

продолжение ➤

```
public class HelloDate {
    public static void main(String[] args) {
        System.out.println("Привет, сегодня: ");
        System.out.println(new Date());
    }
}
```

В начале каждого файла с программой должны находиться необходимые директивы `import`, в которых перечисляются все дополнительные классы, необходимые вашей программе. Обратите внимание на слово «дополнительные» — существует целая библиотека классов, присоединяющаяся автоматически к каждому файлу Java: `java.lang`. Запустите ваш браузер и просмотрите документацию фирмы Sun. (Если вы не загрузили документацию JDK с сайта <http://java.sun.com> или не получили иным способом, обязательно это сделайте¹.) Учтите, что документация не входит в комплект JDK; ее необходимо загрузить отдельно. Взглянув на список пакетов, вы найдете в нем различные библиотеки классов, поставляемые с Java. Выберите `java.lang`. Здесь вы увидите список всех классов, составляющих эту библиотеку. Так как пакет `java.lang` автоматически включается в каждую программу на Java, эти классы всегда доступны для использования. Класа `Date` в нем нет, а это значит, что для его использования придется импортировать другую библиотеку. Если вы не знаете, в какой библиотеке находится нужный класс, или если вам понадобится увидеть все классы, выберите `Tree` (дерево классов) в документации. В нем можно обнаружить любой из доступных классов Java. Функция поиска текста в браузере поможет найти класс `Date`. Результат поиска показывает, что класс называется `java.util.Date`, то есть находится в библиотеке `util`, и для получения доступа к классу `Date` необходимо будет использовать директиву `import` для загрузки пакета `java.util.*`.

Если вы вернетесь к началу, выберете пакет `java.lang`, а затем класс `System`, то увидите, что он имеет несколько полей. При выборе поля `out` обнаруживается, что оно представляет собой статический объект `PrintStream`. Так как поле описано с ключевым словом `static`, вам не понадобится создавать объекты ключевым словом `new`. Действия, которые можно выполнять с объектом `out`, определяются его типом: `PrintStream`. Для удобства в описание этого типа включена гиперссылка, и если щелкнуть на ней, вы обнаружите список всех доступных методов. Этих методов довольно много, и они будут позже рассмотрены. Сейчас нас интересует только метод `println()`, вызов которого фактически означает: «вывести то, что передано методу, на консоль и перейти на новую строку». Таким образом, в любую программу на Java можно включить вызов вида `System.out.println("Какой-то текст")`, чтобы вывести сообщение на консоль.

Имя класса совпадает с именем файла. Когда вы создаете отдельную программу, подобную этой, один из классов, описанных в файле, должен иметь совпадающее с ним название. (Если это условие нарушено, компилятор сообщит об ошибке.) Одноименный класс должен содержать метод с именем `main()` со следующей сигнатурой и возвращаемым типом:

```
public static void main(String[] args) {
```

¹ Документация JDK и компилятор Java не включены в состав компакт-диска, поставляемого с этой книгой, потому что они регулярно обновляются. Загрузив их самостоятельно, вы получите самые свежие версии.

Ключевое слово `public` обозначает, что метод доступен для внешнего мира (об этом подробно рассказывает глава 5). Аргументом метода `main()` является массив строк. В данной программе массив `args` не используется, но компилятор Java настаивает на его присутствии, так как массив содержит параметры, переданные программе в командной строке.

Строка, в которой распечатывается дата, довольно интересна:

```
System.out.println(new Date());
```

Аргумент представляет собой объект `Date`, который создается лишь затем, чтобы передать свое значение (автоматически преобразуемое в `String`) методу `println()`. Как только команда будет выполнена, объект `Date` становится ненужным, уборщик мусора заметит это, и в конце концов сам удалит его. Нам не нужно беспокоиться об его удалении.

Обратившись к документации JDK с сайта <http://java.sun.com>, вы увидите, что класс `System` содержит много других методов для получения интересных результатов (одной из сильных сторон Java является обширный набор стандартных библиотек). Пример:

```
//: object/ShowProperties.java
```

```
public class ShowProperties {
    public static void main(String[] args) {
        System.getProperties().list(System.out);
        System.out.println(System.getProperty("user.name"));
        System.out.println(
            System.getProperty("java.library.path"));
    }
} ///:~
```

Первая строка `main()` выводит все «свойства» системы, в которой запускается программа (то есть информацию окружения). Метод `list()` направляет результаты в свой аргумент `System.out`. Позднее вы увидите, что он также может направить результаты в другое место, например в файл. Также можно запросить конкретное свойство — в данном случае имя пользователя и `java.library.path`. (Смысл необычных комментариев в начале и конце листинга будет объяснен позднее.)

Компиляция и выполнение

Чтобы скомпилировать и выполнить эту программу, а также все остальные программы в книге, вам понадобится среда разработки Java. Существует множество различных сред разработок от сторонних производителей, но в этой книге мы предполагаем, что вы избрали бесплатную среду JDK (Java Developer's Kit) от фирмы Sun. Если же вы используете другие системы разработки программ¹, вам придется просмотреть их документацию, чтобы узнать, как компилировать и запускать программы.

¹ Часто используется компилятор IBM `jikes`, так как он работает намного быстрее компилятора `javac` от Sun (впрочем, при построении групп файлов с использованием `Ant` различия не столь заметны). Также существуют проекты с открытыми исходными текстами, направленные на создание компиляторов, сред времени исполнения и библиотек Java.

Подключитесь к Интернету и посетите сайт <http://java.sun.com>. Там вы найдете информацию и необходимые ссылки, чтобы загрузить и установить JDK для вашей платформы.

Как только вы установите JDK и правильно укажете пути запуска, в результате чего система сможет найти утилиты `javac` и `java`, загрузите и распакуйте исходные тексты программ для этой книги (их можно загрузить с сайта www.MindView.net). Там вы обнаружите каталоги (папки) для каждой главы книги. Перейдите в папку `objects` и выполните команду:

```
javac HelloDate.java
```

Команда не должна выводить каких-либо сообщений. Если вы получили сообщение об ошибке, значит, вы неверно установили JDK и вам нужно разобраться со своими проблемами.

И наоборот, если все прошло успешно, выполните следующую команду:

```
java HelloDate
```

и вы увидите сообщение¹ и число как результат работы программы.

Эта последовательность действий позволяет откомпилировать и выполнить любую программу-пример из этой книги. Однако также вы увидите, что каждая папка содержит файл `build.xml` с командами для инструмента `ant` по автоматической сборке файлов для данной главы. После установки `ant` с сайта <http://jakarta.apache.org/ant> можно будет просто набрать команду `ant` в командной строке, чтобы скомпилировать и запустить программу из любого примера. Если `ant` на вашем компьютере еще не установлен, команды `javac` и `java` придется вводить вручную.

¹ Существует один фактор, который следует учитывать при работе с JVM на платформе MS Windows. Для вывода сообщений на консоль используется кодировка символов DOS (cp866). Так как для Windows по умолчанию принята кодировка Windows-1251, то очень часто бывает так, что русскоязычные сообщения не удастся прочитать с экрана, они будут казаться нероглифами. Для исправления ситуации можно перенаправлять поток вывода следующим способом: `java HelloDate > result.txt`, тогда вывод программы окажется в файле `result.txt` (годится любое другое имя) и его можно будет прочитать. Этот подход применим к любой программе. Или же просто используйте одну из множества программ-«знакогенераторов» (например, `keyrus`), работая с экраном MS-DOS. Тогда вам не потребуются дополнительные действия по перенаправлению. Плюс станет возможной работа под отладчиком JDB. Третий вариант, более сложный, но обеспечивающий вам независимость от машины, заключается во встраивании перекодирования в свою программу посредством методов `setOut` и `setErr` (обходит байт-ориентированность потока `PrintStream`). Российские программисты давно (а отсчет идет с 1997 года) приспособились к этой ситуации. Одно из решений, позволяющее печатать на консоль в правильной кодировке, можно найти на сайте www.javaportal.ru (статья «Русские буквы и не только...»). (Нужно загрузить класс <http://www.javaportal.ru/java/artides/ruschars/CodepagePrintStream.java>, скомпилировать его и описать в переменной окружения. Данный путь лучше отложить до ознакомления с соответствующей темой (глава 12).) — *Примеч. ред.*

Комментарии и встроенная документация

В Java приняты два вида комментариев. Первый — традиционные комментарии в стиле C, также унаследованные языком C++. Такие комментарии начинаются с комбинации `/*`, и распространяются иногда на множество строк, после чего заканчиваются символами `*/`. Заметьте, что многие программисты начинают каждую новую строку таких комментариев символом `*`, соответственно, часто можно увидеть следующее:

```
/* Это комментарий,  
 * распространяющийся на  
 * несколько строк  
 */
```

Впрочем, все символы между `/*` и `*/` игнорируются, и с таким же успехом можно использовать запись

```
/* Это комментарий, распространяющийся  
на несколько строк */
```

Второй вид комментария пришел из языка C++. Однострочный комментарий начинается с комбинации `//` и продолжается до конца строки. Такой стиль очень удобен и прост и поэтому широко используется на практике. Вам не приходится искать на клавиатуре сначала символ `/`, а затем `*` (вместо этого вы дважды нажимаете одну и ту же клавишу) и не нужно закрывать комментарий. Поэтому часто можно увидеть такие примеры:

```
// это комментарий в одну строку
```

Документация в комментариях

Пожалуй, основные проблемы с документированием кода связаны с его сопровождением. Если код и его документация существуют отдельно, корректировать описание программы при каждом ее изменении становится задачей не из легких. Решение выглядит очень просто: совместить код и документацию. Проще всего объединить их в одном файле. Но для полноты картины понадобится специальный синтаксис комментариев, чтобы помечать документацию, и инструмент, который извлекал бы эти комментарии и оформлял их в подходящем виде. Именно это было сделано в Java.

Инструмент для извлечения комментариев называется `javadoc`, он является частью пакета JDK. Некоторые возможности компилятора Java используются в нем для поиска пометок в комментариях, включенных в ваши программы. Он не только извлекает помеченную информацию, но также узнает имя класса или метода, к которому относится данный фрагмент документации. Таким образом, с минимумом затраченных усилий можно создать вполне приличную сопроводительную документацию для вашей программы.

Результатом работы программы `javadoc` является HTML-файл, который можно просмотреть в браузере. Таким образом, утилита `javadoc` позволяет создавать и поддерживать единый файл с исходным текстом и автоматически строить полезную документацию. В результате получается простой и практичный стандарт по созданию документации, поэтому мы можем ожидать (и даже требовать) наличия документации для всех библиотек Java.

Вдобавок, вы можете дополнить `javadoc` своими собственными расширениями, называемыми *доклетами* (doclets), в которых можно проводить специальные операции над обрабатываемыми данными (например, выводить их в другом формате). Доклеты представлены в приложении по адресу <http://MindView.net/Books/BetterJava>.

Далее следует лишь краткое введение и обзор основных возможностей `javadoc`. Более подробное описание можно найти в документации JDK. Распаковав документацию, загляните в папку `tooldocs` (или перейдите по ссылке `tooldocs`).

Синтаксис

Все команды `javadoc` находятся только внутри комментариев `/**`. Комментарии, как обычно, завершаются последовательностью `*/`. Существует два основных способа работы с `javadoc`: встраивание HTML-текста или использование разметки документации (тегов). *Автономные теги документации* — это команды, которые начинаются символом `@` и размещаются с новой строки комментария. (Начальный символ `*` игнорируется.) *Встроенные теги документации* могут располагаться в любом месте комментария `javadoc`, также начинаются со знака `@`, но должны заключаться в фигурные скобки.

Существуют три вида документации в комментариях для разных элементов кода: класса, переменной и метода. Комментарий к классу записывается прямо перед его определением; комментарий к переменной размещается непосредственно перед ее определением, а комментарии к методу тоже записываются прямо перед его определением. Простой пример:

```
//: object/Documentation1.java
/** Комментарий к классу */
public class Documentation1 {
    /** Комментарий к переменной */
    public int i;
    /** Комментарий к методу */
    public void f() {}
} ///:~
```

Заметьте, что `javadoc` обрабатывает документацию в комментариях только для членов класса с уровнем доступа `public` и `protected`. Комментарии для членов `private` и членов с доступом в пределах пакета игнорируются, и документация по ним не строится. (Впрочем, флаг `-private` включает обработку и этих членов.) Это вполне логично, поскольку только `public`- и `protected`-члены доступны вне файла, и именно они интересуют программиста-клиента.

Результатом работы программы является HTML-файл в том же формате, что и остальная документация для Java, так что пользователям будет привычно и удобно просматривать и вашу документацию. Попробуйте набрать текст предыдущего примера, «пропустите» его через `javadoc` и просмотрите полученный HTML-файл, чтобы увидеть результат.

Встроенный HTML

`Javadoc` вставляет команды HTML в итоговый документ. Это позволяет полностью использовать все возможности HTML; впрочем, данная возможность прежде всего ориентирована на форматирование кода:

```
//: object/Documentation2.java
/**
 * <pre>
 * System.out.println(new Date());
 * </pre>
 */
///:~
```

Вы можете использовать HTML точно так же, как в обычных страницах, чтобы привести описание к нужному формату:

```
//: object/Documentation3.java
/**
 * Можно <em>даже</em> вставить список:
 * <ol>
 * <li> Пункт первый
 * <li> Пункт второй
 * <li> Пункт третий
 * </ol>
 */
///:~
```

Javadoc игнорирует звездочки в начале строк, а также начальные пробелы. Текст переформатируется таким образом, чтобы он соответствовал виду стандартной документации. Не используйте заголовки вида <h1> или <h2> во встроенном HTML, потому что javadoc вставляет свои собственные заголовки, и ваши могут с ними «пересечься».

Встроенный HTML-код поддерживается всеми типами документации в комментариях — для классов, переменных или методов.

11.

Примеры тегов

Далее описаны некоторые из тегов javadoc, используемых при документировании программы. Прежде чем применять javadoc для создания чего-либо серьезного, посмотрите руководство по нему в документации пакета JDK, чтобы получить полную информацию о его использовании.

@see: ссылка на другие классы

Тег позволяет ссылаться на документацию к другим классам. Там, где были записаны теги @see, Javadoc создает HTML-ссылки на другие документы. Основные формы использования тега:

```
@see имя класса
@see полное-имя-класса
@see полное-имя-класса#имя-метода
```

Каждая из этих форм включает в генерируемую документацию замечание See Also («см. также») со ссылкой на указанные классы. Javadoc не проверяет передаваемые ему гиперссылки.

{@link пакет.класс#член_класса метка}

Тег очень похож на @see, не считая того, что он может использоваться как встроенный, а вместо стандартного текста See Also в ссылке размещается текст, указанный в поле метка.

{@docRoot}

Позволяет получить относительный путь к корневой папке, в которой находится документация. Полезен при явном задании ссылок на страницы из дерева документации.

{@inheritDoc}

Наследует документацию базового класса, ближайшего к документируемому классу, в текущий файл с документацией.

@version

Имеет следующую форму:

`@version информация-о-версии`

Поле информация-о-версии содержит ту информацию, которую вы сочли нужным включить. Когда в командной строке `javadoс` указывается опция `-version`, в созданной документации специально отводится место, заполняемое информацией о версиях.

@author

Записывается в виде:

`@author информация-об-авторе`

Предполагается, что поле информация-об-авторе представляет собой имя автора, хотя в него также можно включить адрес электронной почты и любую другую информацию. Когда в командной строке `javadoс` указывается опция `-author`, в созданной документации сохраняется информация об авторе.

Для создания списка авторов можно записать сразу несколько таких тегов, но они должны размещаться последовательно. Вся информация об авторах объединяется в один раздел в сгенерированном коде HTML.

@since

Тег позволяет задать версию кода, с которой началось использование некоторой возможности. В частности, он присутствует в HTML-документации по Java, где служит для указания версии JDK.

@param

Полезен при документировании методов. Форма использования:

`@param имя-параметра описание`

где `имя-параметра` — это идентификатор в списке параметров метода, а `описание` — текст описания, который можно продолжить на несколько строк. Описание считается завершенным, когда встретится новый тег. Можно записывать любое количество тегов `@param`, по одному для каждого параметра метода.

@return

Форма использования:

`@return описание`

где `описание` объясняет, что именно возвращает метод. Описание может состоять из нескольких строк.

@throws

Исключения будут рассматриваться в главе 9. В двух словах это объекты, которые можно «инициировать» (throw) в методе, если его выполнение потерпит неудачу. Хотя при вызове метода создается всегда один объект исключения, определенный метод может вырабатывать произвольное количество исключений, и все они требуют описания. Соответственно, форма тега исключения такова:

@throws полное-имя-класса описание

где полное-имя-класса дает уникальное имя класса исключения, который где-то определен, а описание (которое может продолжаться в последующих строках) объясняет, почему данный метод способен создавать это исключение при своем вызове.

@deprecated

Тег используется для пометки устаревших возможностей, замещенных новыми и улучшенными. Он сообщает о том, что определенные средства программы не следует использовать, так как в будущем они, скорее всего, будут убраны. При использовании метода с пометкой @deprecated компилятор выдает предупреждение. В Java SE5 тег @deprecated был заменен *аннотацией* @Deprecated (см. далее).

Пример документации

Вернемся к нашей первой программе на Java, но на этот раз добавим в нее комментарии со встроенной документацией:

```
//: object/HelloDate.java
import java.util.*;

/** Первая программа-пример книги.
 * Выводит строку и текущее число.
 * @author Брюс Эккель
 * @author www.MindView.net
 * @version 4.0
 */
public class HelloDate {
    /** Точка входа в класс и приложение
     * @param args Массив строковых аргументов
     * @throws exceptions Исключения не выдаются
     */
    public static void main(String[] args) {
        System.out.println("Привет, сегодня: ");
        System.out.println(new Date());
    }
} /* Output: (55% match)
Привет, сегодня:
Wed Oct 05 14:39:36 MDT 2005
*///:~
```

В первой строке файла используется мой фирменный прием включения специального маркера `//:` в комментарий, как признак того, что в этой строке комментарий содержит имя файла с исходным текстом. Здесь указывается путь к файлу (object означает эту главу) с последующим именем файла. Последняя строка также завершается комментарием (`///:~`), обозначающим конец исходного текста программы. Он помогает

автоматически извлекать из текста книги программы для проверки компилятором и выполнения.

Тег `/* Output:` обозначает начало выходных данных, сгенерированных данным файлом. В этой форме их можно автоматически проверить на точность. В данном случае значение (55% match) сообщает системе тестирования, что результаты будут заметно отличаться при разных запусках программы. В большинстве примеров книги результаты приводятся в комментариях такого вида, чтобы вы могли проверить их на правильность.

Стиль оформления программ

Согласно правилам стиля, описанным в руководстве *Code Conventions for the Java Programming Language*¹, имена классов должны записываться с прописной буквы. Если имя состоит из нескольких слов, они объединяются (то есть символы подчеркивания не используются для разделения) и каждое слово в имени начинается с большой буквы:

```
class AllTheColorsOfTheRainbow { // ...
```

Практически для всего остального: методов, полей и ссылок на объекты — используется такой же способ записи, *за одним исключением* — первая буква идентификатора записывается строчной. Например:

```
class AllTheColorsOfTheRainbow {
    int anIntegerRepresentingColors;
    void changeTheHueOfTheColor(int newHue) {
        // ...
    }
    // ...
}
```

Помните, что пользователю ваших классов и методов придется вводить все эти длинные имена, так что будьте милосердны.

В исходных текстах Java, которые можно увидеть в библиотеках фирмы Sun, также используется схема размещения открывающих и закрывающих фигурных скобок, которая встречается в примерах данной книги.

Резюме

В этой главе я постарался привести информацию о программировании на Java, достаточную для написания самой простой программы. Также был представлен обзор языка и некоторых его основных свойств. Однако примеры до сих пор имели форму «сначала это, потом это, а после что-то еще». В следующих двух главах будут представлены основные операторы, используемые при программировании на Java, а также способы передачи управления в вашей программе.

¹ Находится по адресу java.sun.com/docs/codeconv/index.html. Для экономии места в данной книге и на слайдах для семинаров я следовал не всем рекомендациям.

Упражнения

В других главах упражнения будут распределяться по тексту главы. Поскольку в этой главе мы только рассмотрели принципы написания простейших программ, упражнения собраны в конце.

В круглых скобках после номера указывается сложность упражнения по шкале от 1 до 10.

Решения некоторых упражнений приведены в электронном документе *The Thinking in Java Annotated Solution Guide*, который можно приобрести на сайте www.MindView.net.

1. (2) Создайте класс с полями `int` и `char`, которые не инициализируются в программе. Выведите их значения, чтобы убедиться в том, что Java выполняет инициализацию по умолчанию.
2. (1) На основании примера `HelloDate.java` в этой главе напишите программу «Привет, мир», которая просто выводит это сообщение. Программа будет содержать только один метод (тот, который выполняется при запуске программы — `main()`). Не забудьте объявить его статическим (`static`) и включить список аргументов, даже если вы не будете его использовать. Скомпилируйте программу с помощью `javac` и запустите на исполнение из `java`. Если вы используете не JDK, а другую среду разработки программ, выясните, как в ней компилируются и запускаются программы.
3. (1) Найдите фрагмент кода с классом `ATypeName` и сделайте из него программу, пригодную для компиляции и запуска.
4. (1) Сделайте то же для кода с использованием класса `DataOnly`.
5. (1) Измените упражнение 4 так, чтобы данным класса `DataOnly` присваивались значения, а затем распечатайте их в методе `main()`.
6. (2) Напишите программу, включающую метод `storage()`, приведенный ранее в этой главе.
7. (1) Превратите фрагменты кода с классом `Incrementable` в работающую программу.
8. (3) Напишите программу, которая демонстрирует, что независимо от количества созданных объектов класс содержит только один экземпляр поля `static`.
9. (2) Напишите программу, демонстрирующую автоматическую упаковку примитивных типов.
10. (2) Напишите программу, которая выводит три параметра командной строки. Для получения аргументов вам потребуется обращение к элементам массива строк (`String`).
11. (1) Преобразуйте пример с классом `AllTheColorsOfTheRainbow` в работающую программу.
12. (2) Найдите вторую версию программы `HelloDate.java`, представляющую пример простой документации в комментариях. Выполните команду `javadoc` для этого файла и просмотрите результаты в браузере.

- 13.** (1) Запустите программы `Documentation1.java`, `Documentation2.java` и `Documentation3.java` в `javadoc`. Просмотрите результаты в браузере.
- 14.** (1) Добавьте список HTML к документации, создаваемой в упражнении 13.
- 15.** (1) Возьмите программу из упражнения 2 и добавьте к ней документацию в комментариях. Извлеките эту документацию в HTML-файл с помощью `javadoc` и просмотрите полученную страницу в браузере.
- 16.** Найдите в главе 5 пример `Overloading.java` и добавьте в него комментарии `javadoc`. Преобразуйте их в страницу HTML и просмотрите ее в браузере.

3 Операторы

На нижнем уровне операции с данными в Java осуществляются посредством операторов.

Язык Java создавался на основе C++, поэтому большинство этих операторов и конструкций знакомы программистам на C и C++. Также в Java были добавлены некоторые улучшения и упрощения.

Если вы знакомы с синтаксисом C или C++, бегло просмотрите эту и следующую главу, останавливаясь на тех местах, в которых Java отличается от этих языков. Если чтение дается вам с трудом, попробуйте обратиться к мультимедийному семинару *Thinking in C*, свободно загружаемому с сайта www.MindView.net. Он содержит аудиолекции, слайды, упражнения и решения, специально разработанные для быстрого ознакомления с синтаксисом C, необходимым для успешного овладения языком Java.

Простые команды печати

В предыдущей главе была представлена команда печати Java:

```
System.out.println("Какая длинная команда...");
```

Вероятно, вы заметили, что команда не только получается слишком длинной, но и плохо читается. Во многих языках до и после Java используется более простой подход к выполнению столь распространенной операции.

В главе 6 представлена концепция *статического импорта*, появившаяся в Java SE5, а также создается крошечная библиотека, упрощающая написание команд печати. Тем не менее для использования библиотеки не обязательно знать все подробности. Программу из предыдущей главы можно переписать в следующем виде:

```
//: operators/HelloDate.java
import java.util.*;
import static net.mindview.util.Print.*;

public class HelloDate {
```

продолжение ➤

```

public static void main(String[] args) {
    print("Привет, сегодня: ");
    print(new Date());
}
} /* Output: (55% match)
Привет, сегодня:
Wed Oct 05 14:39:36 MDT 2005
*///:~

```

Результат смотрится гораздо приятнее. Обратите внимание на ключевое слово `static` во второй команде `import`.

Чтобы использовать эту библиотеку, необходимо загрузить архив с примерами кода. Распакуйте его и включите корневой каталог дерева в переменную окружения `CLASSPATH` вашего компьютера. Хотя использование `net.mindview.util.Print` упрощает программный код, оно оправданно не везде. Если программа содержит небольшое количество команд печати, я отказываюсь от `import` и записываю полный вызов `System.out.println()`.

1. (1) Напишите программу, в которой используются как «короткие», так и «длинные» команды печати.

Операторы Java

Оператор получает один или несколько аргументов и создает на их основе новое значение. Форма передачи аргументов несколько иная, чем при вызове метода, но эффект тот же самый. Сложение (+), вычитание и унарный минус (–), умножение (*), деление (/) и присваивание (=) работают одинаково фактически во всех языках программирования.

Все операторы работают с операндами и выдают какой-то результат. Вдобавок некоторые операторы могут изменить значение операнда. Это называется *побочным эффектом*. Как правило, операторы, изменяющие значение своих операндов, используются именно ради побочного эффекта, но вы должны помнить, что полученное значение может быть использовано в программе и обычным образом, независимо от побочных эффектов.

Почти все операторы работают только с примитивами. Исключениями являются `=`, `= =` и `!=`, которые могут быть применены к объектам (и нередко создают изрядную путаницу). Вдобавок класс `String` поддерживает операции `+` и `+=`.

Приоритет

Приоритет операций определяет порядок вычисления выражений с несколькими операторами. В Java существуют конкретные правила для определения очередности вычислений. Легче всего запомнить, что деление и умножение выполняются раньше сложения и вычитания. Программисты часто забывают правила предшествования, поэтому для явного задания порядка вычислений следует использовать круглые скобки. Например, взгляните на команды (1) и (2):

```

//: operators/Precedence.java

public class Precedence {

```

```
public static void main(String[] args) {  
    int x = 1, y = 2, z = 3;           // (1)  
    int b = x + (y - 2)/(2 + z);      // (2)  
    System.out.println("a = " + a + " b = " + b;  
}  
} /* Output  
a = 5 b = 1  
*///:~
```

Команды похожи друг на друга, но из результатов хорошо видно, что они имеют разный смысл в зависимости от присутствия круглых скобок.

Обратите внимание на оператор `+` в команде `System.out.println`. В данном контексте `+` означает конкатенацию строк, а не суммирование. Когда компилятор встречает объект `String`, за которым следует `+`, и объект, отличный от `String`, он пытается преобразовать последний объект в `String`. Как видно из выходных данных, для `a` и `b` тип `int` был успешно преобразован в `String`.

Присваивание

Присваивание выполняется оператором `=`. Трактуются он так: «взять значение из правой части выражения (часто называемое просто *значением*) и скопировать его в левую часть (часто называемую *именующим выражением*)». Значением может быть любая константа, переменная или выражение, но в качестве именующего выражения обязательно должна использоваться именованная переменная (то есть для хранения значения должна выделяться физическая память). Например, вы можете присвоить постоянное значение переменной:

```
a = 4
```

но нельзя присвоить что-либо константе — она не может использоваться в качестве именующего выражения (например, запись `4 = a` недопустима).

Для примитивов присваивание выполняется тривиально. Так как примитивный тип хранит данные, а не ссылку на объект, то присваивание сводится к простому копированию данных из одного места в другое. Например, если команда `a = b` выполняется для примитивных типов, то содержимое `b` просто копируется в `a`. Естественно, последующие изменения `a` никак не отражаются на `b`. Для программиста именно такое поведение выглядит наиболее логично.

При присваивании объектов все меняется. При выполнении операций с объектом вы в действительности работаете со ссылкой, поэтому присваивание «одного объекта другому» в действительности означает копирование ссылки из одного места в другое. Это значит, что при выполнении команды `c = d` для объектов в конечном итоге `c` и `d` указывают на один объект, которому изначально соответствовала только ссылка `d`. Сказанное демонстрирует следующий пример:

```
//: operators/Assignment.java  
// Присваивание объектов имеет ряд хитростей.  
import static net.mindview.util.Print.*;  
  
class Tank {  
    int level;
```

```

}
public class Assignment {
    public static void main(String[] args) {
        Tank t1 = new Tank();
        Tank t2 = new Tank();
        t1.level = 9;
        t2.level = 47;
        print("1: t1.level: " + t1.level +
            ", t2.level: " + t2.level);
        t1 = t2;
        print("2: t1.level: " + t1.level +
            ", t2.level: " + t2.level);
        t1.level = 27;
        print("3: t1.level: " + t1.level +
            ", t2.level: " + t2.level);
    }
} /* Output:
1: t1.level: 9, t2.level: 47
2: t1.level: 47, t2.level: 47
3: t1.level: 27, t2.level: 27
*///:~

```

Класс `Tank` предельно прост, и два его экземпляра (`t1` и `t2`) создаются внутри метода `main()`. Переменной `level` для каждого экземпляра придаются различные значения, а затем ссылка `t2` присваивается `t1`, в результате чего `t1` изменяется. Во многих языках программирования можно было ожидать, что `t1` и `t2` будут независимы все время, но из-за присваивания ссылок изменение объекта `t1` отражается на объекте `t2`! Это происходит из-за того, что `t1` и `t2` содержат одинаковые ссылки, указывающие на один объект. (Исходная ссылка, которая содержалась в `t1` и указывала на объект со значением 9, была перезаписана во время присваивания и фактически потеряна; ее объект будет вскоре удален уборщиком мусора.)

Этот феномен совмещения имен часто называют *синонимией* (aliasing), и именно она является основным способом работы с объектами в Java. Но что делать, если совмещение имен нежелательно? Тогда можно пропустить присваивание и записать:

```
t1.level = t2.level;
```

При этом программа сохранит два разных объекта, а не «выбросит» один из них, «привязав» ссылки `t1` и `t2` к единственному объекту. Вскоре вы поймете, что прямая работа с полями данных внутри объектов противоречит принципам объектно-ориентированной разработке. Впрочем, это непростой вопрос, так что пока вам достаточно запомнить, что присваивание объектов может таить в себе немало сюрпризов.

2. (1) Создайте класс с полем типа `float`. Используйте его для демонстрации совмещения имен.

Совмещение имен во время вызова методов

Совмещение имен также может происходить при передаче объекта методу:

```

//: operators/PassObject.java
// Передача объектов методам может работать
// не так, как вы привыкли.
import static net.mindview.util.Print.*;

```

```

class Letter {
    char c;
}

public class PassObject {
    static void f(Letter y) {
        y.c = 'z';
    }
    public static void main(String[] args) {
        Letter x = new Letter();
        x.c = 'a';
        print("1: x.c: " + x.c);
        f(x);
        print("2: x.c: " + x.c);
    }
} /* Output
1: x.c: a
2: x.c: z
*/ ///:~

```

Во многих языках программирования метод `f()` создал бы копию своего параметра `Letter y` внутри своей области действия. Но из-за передачи ссылки строка:

```
y.c = 'z';
```

на самом деле изменяет объект за пределами метода `f()`.

Совмещение имен и решение этой проблемы — сложная тема, которая подробно рассматривается в одном из электронных приложений к книге. Будьте очень внимательными в таких случаях во избежание ловушек.

3. (1) Создайте класс с полем типа `float`. Используйте его для демонстрации совмещения имен при вызове методов.

Математические операторы

Основные математические операторы остаются неизменными почти во всех языках программирования: сложение (+), вычитание (-), деление (/), умножение (*) и остаток от деления нацело (%). При делении нацело выполняется усечение, а не округление результата.

В Java также используется укороченная форма записи для того, чтобы одновременно произвести операцию и присваивание. Она обозначается оператором с последующим знаком равенства и работает одинаково для всех операторов языка (когда в этом есть смысл). Например, чтобы прибавить 4 к переменной `x` и присвоить результат `x`, используйте команду `x += 4`.

Следующий пример демонстрирует использование математических операторов:

```

//: operators/MathOps.java
// Демонстрация математических операций.
import java.util.*;
import static net.mindview.util.Print.*;

public class MathOps {

```

продолжение ➤

```

public static void main(String[] args) {
    // Создание и раскрутка генератора случайных чисел
    Random rand = new Random(47);
    int i, j, k;
    // Выбор значения от 1 до 100:
    j = rand.nextInt(100) + 1;
    print("j : " + j);
    k = rand.nextInt(100) + 1;
    print("k : " + k);
    i = j + k;
    print("j + k : " + i);
    i = j - k;
    print("j - k : " + i);
    i = k / j;
    print("k / j : " + i);
    i = k * j;
    print("k * j : " + i);
    i = k % j;
    print("k % j : " + i);
    j %= k;
    print("j %/ k : " + j);
    // Тесты для вещественных чисел
    float u,v,w; // также можно использовать double
    v = rand.nextFloat();
    print("v : " + v);
    w = rand.nextFloat();
    print("w : " + w);
    u = v + w;
    print("v + w : " + u);
    u = v - w;
    print("v - w : " + u);
    u = v * w;
    print("v * w : " + u);
    u = v / w;
    print("v / w : " + u);
    // следующее также относится к типам
    // char, byte, short, int, long и double:
    u += v;
    print("u += v : " + u);
    u -= v;
    print("u -= v : " + u);
    u *= v;
    print("u *= v : " + u);
    u /= v;
    print("u /= v : " + u);
}
} /* Output:
j : 59
k : 56
j + k : 115
j - k : 3
k / j : 0
k * j : 3304
k % j : 56
j %= k : 3
v : 0.5309454
w : 0.0534122
v + w : 0.5843576

```

```
v - w : 0.47753322
v * w : 0.028358962
v / w : 9.940527
u += v : 10.471473
u -= v : 9.940527
u *= v : 5.2778773
u /= v : 9.940527
*///:~
```

Для получения случайных чисел создается объект `Random`. Если он создается без параметров, Java использует текущее время для раскрутки генератора, чтобы при каждом запуске программы выдавались разные числа. Однако в примерах, приведенных в книге, результаты должны быть по возможности стабильными, чтобы их можно было проверить внешними средствами. Если при создании объекта `Random` указывается начальное число, то при каждом выполнении программы будет генерироваться одна и та же серия значений¹. Чтобы результаты стали более разнообразными, удалите начальное число из примеров.

Программа генерирует различные типы случайных чисел, вызывая соответствующие методы объекта `Random`: `nextInt()` и `nextFloat()` (также можно использовать `nextLong()` и `nextDouble()`). Аргумент `nextInt()` задает верхнюю границу генерируемых чисел. Нижняя граница равна 0, но для предотвращения возможного деления на 0 результат смещается на 1.

4. (2) Напишите программу, которая вычисляет скорость для постоянных значений расстояния и времени.

Унарные операторы плюс и минус

Унарные минус (-) и плюс (+) внешне не отличаются от аналогичных бинарных операторов. Компилятор выбирает нужный оператор в соответствии с контекстом использования. Например, команда

```
x = -a;
```

имеет очевидный смысл. Компилятор без труда разберется, что значит

```
x = a * -b;
```

но читающий код может запутаться, так что яснее будет написать так:

```
x = a * (-b);
```

Унарный минус меняет знак числа на противоположный. Унарный плюс существует «для симметрии», хотя и не производит никаких действий.

Автоувеличение и автоуменьшение

В Java, как и C, существует множество различных сокращений. Сокращения могут упростить написание кода, а также упростить или усложнить его чтение.

¹ В колледже, в котором я учился, число 47 считалось «волшебным» — тогда это и вошло в привычку.

Два наиболее полезных сокращения — это операторы увеличения (инкремента) и уменьшения (декремента) (также часто называемые операторами автоматического приращения и уменьшения). Оператор декремента записывается в виде `--` и означает «уменьшить на единицу». Оператор инкремента обозначается символами `++` и позволяет «увеличить на единицу». Например, если переменная `a` является целым числом, то выражение `++a` будет эквивалентно (`a = a + 1`). Операторы инкремента и декремента не только изменяют переменную, но и устанавливают ей в качестве результата новое значение.

Каждый из этих операторов существует в двух версиях — *префиксной* и *постфиксной*. Префиксный инкремент значит, что оператор `++` записывается перед переменной или выражением, а при постфиксном инкременте оператор следует после переменной или выражения. Аналогично, при префиксном декременте оператор `--` указывается перед переменной или выражением, а при постфиксном — после переменной или выражения. Для префиксного инкремента и декремента (то есть `++a` и `--a`) сначала выполняется операция, а затем выдается результат. Для постфиксной записи (`a++` и `a--`) сначала выдается значение и лишь затем выполняется операция. Например:

```
//: operators/AutoInc.java
import static net.mindview.util.Print.*;

public class AutoInc {
    public static void main(String[] args) {
        int i = 1;
        print("i : " + i);
        print("++i : " + ++i); // Префиксный инкремент
        print("i++ : " + i++); // Постфиксный инкремент
        print("i : " + i);
        print("--i : " + --i); // Префиксный декремент
        print("i-- : " + i--); // Постфиксный декремент
        print("i : " + i);
    }
} /* Output:
i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1
*///:~
```

Вы видите, что при использовании префиксной формы результат получается после выполнения операции, тогда как с постфиксной формой он доступен до выполнения операции. Это единственные операторы (кроме операторов присваивания), которые имеют побочный эффект. (Иначе говоря, они изменяют свой операнд вместо простого использования его значения.)

Оператор инкремента объясняет происхождение названия языка C++; подразумевается «шаг вперед по сравнению с C». В одной из первых речей, посвященных Java, Билл Джой (один из его создателей) сказал, что «Java = C++--» («Си плюс плюс минус минус»). Он имел в виду, что Java — это C++, из которого убрано все, что затрудняет программирование, и поэтому язык стал гораздо проще. Продвигаясь вперед, вы увидите, что отдельные аспекты языка, конечно, проще, и все же Java не *настолько* проще C++.

Операторы сравнения

Операторы сравнения выдают логический (`boolean`) результат. Они проверяют, в каком отношении находятся значения их операндов. Если условие проверки истинно, оператор выдает `true`, а если ложно — `false`. К операторам сравнения относятся следующие: «меньше чем» (`<`), «больше чем» (`>`), «меньше чем или равно» (`<=`), «больше чем или равно» (`>=`), «равно» (`==`) и «не равно» (`!=`). «Равно» и «не равно» работают для всех примитивных типов данных, однако остальные сравнения не применимы к типу `boolean`.

Проверка объектов на равенство

Операторы отношений `==` и `!=` также работают с любыми объектами, но их смысл нередко сбивает с толку начинающих программистов на Java. Пример:

```
//: operators/Equivalence.java
```

```
public class Equivalence {
    public static void main(String[] args) {
        Integer n1 = new Integer(47);
        Integer n2 = new Integer(47);
        System.out.println(n1 == n2);
        System.out.println(n1 != n2);
    }
} /* Output:
false
true
*///:~
```

Выражение `System.out.println(n1 == n2)` выведет результат логического сравнения, содержащегося в скобках. Казалось бы, в первом случае результат должен быть истинным (`true`), а во втором — ложным (`false`), так как оба объекта типа `Integer` имеют одинаковые значения. Но в то время как содержимое объектов одинаково, ссылки на них разные, а операторы `!=` и `==` сравнивают именно ссылки. Поэтому результатом первого выражения будет `false`, а второго — `true`. Естественно, такие результаты поначалу ошеломляют.

А если понадобится сравнить действительное содержимое объектов? Придется использовать специальный метод `equals()`, поддерживаемый всеми объектами (но не примитивами, для которых более чем достаточно операторов `==` и `!=`). Вот как это делается:

```
//: operators/EqualsMethod.java
```

```
public class EqualsMethod {
    public static void main(String[] args) {
        Integer n1 = new Integer(47);
        Integer n2 = new Integer(47);
        System.out.println(n1.equals(n2));
    }
} /* Output:
true
*///:~
```

На этот раз результат окажется «истиной» (`true`), как и предполагалось. Но все не так просто, как кажется. Если вы создадите свой собственный класс вроде такого:

```

//: operators/EqualsMethod2.java
// Метод equals() по умолчанию не сравнивает содержимое

class Value {
    int i;
}

public class EqualsMethod2 {
    public static void main(String[] args) {
        Value v1 = new Value();
        Value v2 = new Value();
        v1.i = v2.i = 100;
        System.out.println(v1.equals(v2));
    }
} /* Output:
false
*///:~

```

мы вернемся к тому, с чего начинали: результатом будет `false`. Дело в том, что метод `equals()` по умолчанию сравнивает ссылки. Следовательно, пока вы не *переопределите* этот метод в вашем новом классе, то и не получите желаемого результата. К сожалению, переопределение будет рассматриваться только в главе 8, а пока осторожность и общее понимание принципа работы `equals()` позволит избежать некоторых неприятностей.

Большинство классов библиотек Java реализуют метод `equals()` по-своему, сравнивая содержимое объектов, а не ссылки на них.

5. (2) Создайте класс `Dog`, содержащий два поля типа `String`: `name` и `says`. В методе `main()` создайте два объекта `Dog` с разными именами (`spot` и `scruffy`) и сообщениями. Выведите значения обоих полей для каждого из объектов.
6. (3) В упражнении 5 создайте новую ссылку на `Dog` и присвойте ее объекту `spot`. Сравните ссылки оператором `==` и методом `equals()`.

Логические операторы

Логические операторы И (`&&`), ИЛИ (`||`) и НЕ (`!`) производят логические значения `true` и `false`, основанные на логических отношениях своих аргументов. В следующем примере используются как операторы сравнения, так логические операторы:

```

//: operators/Bool.java
// Операторы сравнений и логические операторы.
import java.util.*;
import static net.mindview.util.Print.*;

public class Bool {
    public static void main(String[] args) {
        Random rand = new Random(47);
        int i = rand.nextInt(100);
        int j = rand.nextInt(100);
        print("i = " + i);
        print("j = " + j);
        print("i > j is " + (i > j));
        print("i < j is " + (i < j));
    }
}

```

```
print("i >= j is " + (i >= j));
print("i <= j is " + (i <= j));
print("i == j is " + (i == j));
print("i != j is " + (i != j));
// В Java целое число (int) не может
// интерпретироваться как логический тип (boolean)
///! print("i && j is " + (i && j));
///! print("i || j is " + (i || j));
///! print("!i is " + !i);
print("(i < 10) && (j < 10) is "
      + ((i < 10) && (j < 10)) );
print("(i < 10) || (j < 10) is "
      + ((i < 10) || (j < 10)) );
}
} /* Output:
i = 58
j = 55
i > j is true
i < j is false
i >= j is true
i <= j is false
i == j is false
i != j is true
(i < 10) && (j < 10) is false
(i < 10) || (j < 10) is false
*///:~
```

Операции И, ИЛИ и НЕ применяются только к логическим (`boolean`) значениям. Нельзя использовать в логических выражениях не-`boolean`-типы в качестве булевых, как это разрешается в С и С++. Неудачные попытки такого рода видны в строках, помеченных особым комментарием `///!` (этот синтаксис позволяет автоматически удалять комментарии для удобства тестирования). Последующие выражения вырабатывают логические результаты, используя операторы сравнений, после чего к полученным значениям применяются логические операции.

Заметьте, что значение `boolean` автоматически переделывается в подходящее строковое представление там, где предполагается использование строкового типа `String`.

Определение `int` в этой программе можно заменить любым примитивным типом, за исключением `boolean`. Впрочем, будьте осторожны с вещественными числами, поскольку их сравнение проводится с крайне высокой точностью. Число, хотя бы чуть-чуть отличающееся от другого, уже считается неравным ему. Число, на тысячную долю большее нуля, уже не является нулем.

7. (3) Напишите программу, моделирующую бросок монетки.

Ускоренное вычисление

При работе с логическими операторами можно столкнуться с феноменом, называемым «ускоренным вычислением». Это значит, что выражение вычисляется только до тех пор, пока не станет очевидно, что оно принимает значение «истина» или «ложь». В результате некоторые части логического выражения могут быть проигнорированы в процессе сравнения. Следующий пример демонстрирует ускоренное вычисление:

```

//: operators/ShortCircuit.java
// Демонстрация ускоренного вычисления
// при использовании логических операторов.
import static net.mindview.util.Print.*;

public class ShortCircuit {
    static boolean test1(int val) {
        print("test1(" + val + ")");
        print("результат: " + (val < 1));
        return val < 1;
    }
    static boolean test2(int val) {
        print("test2(" + val + ")");
        print("результат: " + (val < 2));
        return val < 2;
    }
    static boolean test3(int val) {
        print("test3(" + val + ")");
        print("результат: " + (val < 3));
        return val < 3;
    }
    public static void main(String[] args) {
        boolean b = test1(0) && test2(2) && test3(2);
        print ("выражение: " + b);
    }
} /* Output:
test1(0)
результат: true
test2(2)
результат: false
выражение: false
*///:~

```

Каждый из методов `test()` проводит сравнение своего аргумента и возвращает либо `true`, либо `false`. Также они выводят информацию о факте своего вызова. Эти методы используются в выражении:

```
test1(0) && test2(2) && test3(2)
```

Естественно было бы ожидать, что все три метода должны выполняться, но результат программы показывает другое. Первый метод возвращает результат `true`, поэтому вычисление выражения продолжается. Однако второй метод выдает результат `false`. Так как это автоматически означает, что все выражение будет равно `false`, зачем продолжать вычисления? Только лишняя трата времени. Именно это и стало причиной введения в язык ускоренного вычисления; отказ от лишних вычислений обеспечивает потенциальный выигрыш в производительности.

Литералы

Обычно, когда вы записываете в программе какое-либо значение, компилятор точно знает, к какому типу оно относится. Однако в некоторых ситуациях однозначно определить тип не удастся. В таких случаях следует помочь компилятору определить точный тип, добавив дополнительную информацию в виде определенных символьных

обозначений, связанных с типами данных. Эти обозначения используются в следующей программе:

```
//: operators/Literals.java
import static net.mindview.util.Print.*;

public class Literals {
    public static void main(String[] args) {
        int i1 = 0x2f; // Шестнадцатеричное (нижний регистр)
        print("i1: " + Integer.toBinaryString(i1));
        int i2 = 0X2F; // Шестнадцатеричное (верхний регистр)
        print("i2: " + Integer.toBinaryString(i2));
        int i3 = 0177; // Восьмеричное (начинается с нуля)
        print("i3: " + Integer.toBinaryString(i3));
        char c = 0xffff; // макс. шестнадцатеричное знач. char
        print("c: " + Integer.toBinaryString(c));
        byte b = 0x7f; // макс. шестнадцатеричное знач. byte
        print("b: " + Integer.toBinaryString(b));
        short s = 0x7fff; // макс. шестнадцатеричное знач. short
        print("s: " + Integer.toBinaryString(s));
        long n1 = 200L; // Суффикс, обозначающий long
        long n2 = 200l; // Суффикс, обозначающий long (можно запутаться)
        long n3 = 200;
        float f1 = 1;
        float f2 = 1F; // Суффикс, обозначающий float
        float f3 = 1f; // Суффикс, обозначающий float
        double d1 = 1d; // Суффикс, обозначающий double
        double d2 = 1D; // Суффикс, обозначающий double
    }
} /* Output:
i1: 101111
i2: 101111
i3: 1111111
c: 1111111111111111
b: 1111111
s: 1111111111111111
*///:~
```

Последний символ обозначает тип записанного литерала. Прописная или строчная буква L определяет тип long (впрочем, строчная l может создать проблемы, потому что она похожа на цифру 1); прописная или строчная F соответствует типу float, а заглавная или строчная D подразумевает тип double.

Шестнадцатеричное представление (основание 16) работает со всеми встроенными типами данных и обозначается префиксом 0x или 0X с последующим числовым значением из цифр 0-9 и букв a-f, прописных или строчных. Если при определении переменной задается значение, превосходящее максимально для нее возможное (независимо от числовой формы), компилятор сообщит вам об ошибке. В программе указаны максимальные значения для типов char, byte и short. При выходе за эти границы компилятор автоматически сделает значение типом int и сообщит вам, что для присваивания понадобится сужающее приведение.

Восьмеричное представление (по основанию 8) обозначается начальным нулем в записи числа, состоящего из цифр от 0 до 7.

Литеральная запись двоичных чисел в Java, C и C++ не поддерживается. Впрочем, при работе с шестнадцатеричными и восьмеричными числами часто требуется

получить двоичное представление результата. Задача легко решается методами `static toBinaryString()` классов `Integer` и `Long`. Учтите, что при передаче меньших типов методу `Integer.toBinaryString()` тип автоматически преобразуется в `int`.

8. (2) Покажите, что шестнадцатеричная и восьмеричная записи могут использоваться с типом `long`. Для вывода результатов используйте метод `Long.toBinaryString()`.

Экспоненциальная запись

Экспоненциальные значения записываются, по-моему, очень неудачно:

```
//: operators/Exponents.java
// "e" означает "10 в степени".

public class Exponents {
    public static void main(String[] args) {
        // Прописная и строчная буквы 'e' эквивалентны:
        float expFloat = 1.39e-43f;
        expFloat = 1.39E-43f;
        System.out.println(expFloat);
        double expDouble = 47e47d; // Суффикс 'd' не обязателен
        double expDouble2 = 47e47; // Автоматически определяется double
        System.out.println(expDouble);
    }
} /* Output:
1.39E-43
4.7E48
*///:~
```

В науке и технических расчетах символом *e* обозначается основание натурального логарифма, равное примерно 2,718. (Более точное значение этой величины можно получить из свойства `Math.E`.) Оно используется в экспоненциальных выражениях, таких как $1,39 \times e^{-47}$, что фактически значит $1,39 \times 2,718^{-47}$. Однако во время изобретения языка FORTRAN было решено, что *e* будет обозначать «десять в степени», что достаточно странно, поскольку FORTRAN разрабатывался для науки и техники и можно было предположить, что его создатели обратят внимание на подобную неоднозначность¹. Так или иначе, этот обычай был перенят в C, C++, а затем перешел в Java. Таким об-

¹ Джон Кирхем пишет: «Я начал программировать в 1960 году на FORTRAN II, используя компьютер IBM 1620. В то время, в 60-е и 70-е годы, FORTRAN использовал только заглавные буквы. Возможно, это произошло потому, что большинство старых устройств ввода были теле-тайпами, работавшими с 5-битовым кодом Бодо, который не поддерживал строчные буквы. Буква E в экспоненциальной записи также была заглавной и не смешивалась с основанием натурального логарифма e, которое всегда записывается маленькой буквой. Символ E просто выражал экспоненциальный характер, то есть обозначал основание системы счисления — обычно таким было 10. В те годы программисты широко использовали восьмеричную систему. И хотя я и не замечал такого, но если бы я увидел восьмеричное число в экспоненциальной форме, я бы предположил, что имеется в виду основание 8. Первый раз я встретился с использованием маленькой e в экспоненциальной записи в конце 1970-х годов, и это было очень неудобно. Проблемы появились потом, когда строчные буквы по инерции перешли в FORTRAN. У нас существовали все нужные функции для действий с натуральными логарифмами, но все они записывались прописными буквами».

разом, если вы привыкли видеть в e основание натурального логарифма, вам придется каждый раз делать преобразование в уме: если вы увидели в Java выражение `1.39e-43f`, на самом деле оно значит $1,39 \times 10^{-43}$.

Если компилятор может определить тип автоматически, наличие завершающего суффикса типа не обязательно. В записи:

```
long n3 = 200;
```

не существует никаких неясностей, и поэтому использование символа `L` после значения `200` было бы излишним. Однако в записи:

```
float f4 = 1e-43f; // десять в степени
```

компилятор обычно трактует экспоненциальные числа как `double`. Без завершающего символа `f` он сообщит вам об ошибке и необходимости использования приведения для преобразования `double` к типу `float`.

9. (1) Выведите наибольшее и наименьшее число в экспоненциальной записи для типов `float` и `double`.

Поразрядные операторы

Поразрядные операторы манипулируют отдельными битами в целочисленных примитивных типах данных. Результат определяется действиями булевой алгебры с соответствующими битами двух операндов.

Эти битовые операторы происходят от низкоуровневой направленности языка C, где часто приходится напрямую работать с оборудованием и устанавливать биты в аппаратных регистрах. Java изначально разрабатывался для управления телевизионными приставками, поэтому эта низкоуровневая ориентация все еще была нужна. Впрочем, вам вряд ли придется часто использовать эти операторы.

Поразрядный оператор И (`&`) заносит 1 в выходной бит, если оба входных бита были равны 1; в противном случае результат равен 0. Поразрядный оператор ИЛИ (`|`) заносит 1 в выходной бит, если хотя бы один из битов операндов был равен 1; результат равен 0 только в том случае, если оба бита операндов были нулевыми. Оператор ИСКЛЮЧАЮЩЕЕ ИЛИ (`XOR`, `^`) имеет результатом единицу тогда, когда один или другой из входных битов был единицей, но не оба вместе. Поразрядный оператор НЕ (`~`, также называемый оператором *двоичного дополнения*) является унарным оператором, то есть имеет только один операнд. Поразрядное НЕ производит бит, «противоположный» исходному — если входящий бит является нулем, то в результирующем бите окажется единица, если входящий бит — единица, получится ноль.

Поразрядные операторы и логические операторы записываются с помощью одних и тех же символов, поэтому полезно запомнить мнемоническое правило: так как биты «маленькие», в поразрядных операторах используется всего один символ.

Поразрядные операторы могут комбинироваться со знаком равенства `=`, чтобы совместить операцию с присваиванием: `&=`, `|=` и `^=` являются допустимыми сочетаниями. (Так как `~` является унарным оператором, он не может использоваться вместе со знаком `=`.)

Тип `boolean` трактуется как однобитовый, поэтому операции с ним выглядят по-другому. Вы вправе выполнить поразрядные И, ИЛИ и ИСКЛЮЧАЮЩЕЕ ИЛИ, но НЕ использовать запрещено (видимо, чтобы предотвратить путаницу с логическим НЕ). Для типа `boolean` поразрядные операторы производят тот же эффект, что и логические, за одним исключением — они не поддерживают ускоренного вычисления. Кроме того, в число поразрядных операторов для `boolean` входит оператор ИСКЛЮЧАЮЩЕЕ ИЛИ, отсутствующий в списке логических операторов. Для булевых типов не разрешается использование операторов сдвига, описанных в следующем разделе.

10. (3) Напишите программу с двумя константами: обе константы состоят из чередующихся нулей и единиц, но у одной нулю равен младший бит, а у другой старший (подсказка: константы проще всего определить в шестнадцатеричном виде). Объедините эти две константы всеми возможными поразрядными операторами. Для вывода результатов используйте метод `Integer.toBinaryString()`.

Операторы сдвига

Операторы сдвига также манипулируют битами и используются только с примитивными целочисленными типами. Оператор сдвига влево (`<<`) сдвигает влево операнд, находящийся слева от оператора, на количество битов, указанных после оператора. Оператор сдвига вправо (`>>`) сдвигает вправо операнд, находящийся слева от оператора, на количество битов, указанных после оператора. При сдвиге вправо используется *заполнение знаком*: при положительном значении новые биты заполняются нулями, а при отрицательном — единицами. В Java также поддерживается беззнаковый сдвиг вправо `>>>`, использующий *заполнение нулями*: независимо от знака старшие биты заполняются нулями. Такой оператор не имеет аналогов в C и C++.

Если сдвигаемое значение относится к типу `char`, `byte` или `short`, эти типы приводятся к `int` перед выполнением сдвига, и результат также получится `int`. При этом используется только пять младших битов с «правой» стороны. Таким образом, нельзя сдвинуть битов больше, чем вообще существует для целого числа `int`. Если вы проводите операции с числами `long`, то получите результаты типа `long`. При этом будет задействовано только шесть младших битов с «правой» стороны, что предотвращает использование излишнего числа битов.

Сдвиги можно совмещать со знаком равенства (`<<=`, или `>>=`, или `>>>=`). Именуемое выражение заменяется им же, но с проведенными над ним операциями сдвига. Однако при этом возникает проблема с оператором беззнакового правого сдвига, совмещенного с присваиванием. При использовании его с типом `byte` или `short` вы не получите правильных результатов. Вместо этого они сначала будут преобразованы к типу `int` и сдвинуты вправо, а затем обрезаны при возвращении к исходному типу, и результатом станет `-1`. Следующий пример демонстрирует это:

```
//: operators/URShift.java
// Проверка беззнакового сдвига вправо.
import static net.mindview.util.Print.*;

public class URShift {
    public static void main(String[] args) {
        int i = -1;
```


продолжение Φ

```

    printBinaryInt("i & j", i & j);
    printBinaryInt("i | j", i | j);
    printBinaryInt("i ^ j", i ^ j);
    printBinaryInt("i << 5", i << 5);
    printBinaryInt("i >> 5", i >> 5);
    printBinaryInt("(~i) >> 5", (~i) >> 5);
    printBinaryInt("i >>> 5", i >>> 5);
    printBinaryInt("(~i) >>> 5", (~i) >>> 5);

    long l = rand.nextLong();
    long m = rand.nextLong();
    printBinaryLong("-1L", -1L);
    printBinaryLong("+1L", +1L);
    long ll = 9223372036854775807L;
    printBinaryLong("макс. положит.", ll);
    long llN = -9223372036854775808L;
    printBinaryLong("макс. отрицат.", llN);
    printBinaryLong("1", 1);
    printBinaryLong("~1", ~1);
    printBinaryLong("-1", -1);
    printBinaryLong("m", m);
    printBinaryLong("l & m", l & m);
    printBinaryLong("l | m", l | m);
    printBinaryLong("l ^ m", l ^ m);
    printBinaryLong("l << 5", l << 5);
    printBinaryLong("l >> 5", l >> 5);
    printBinaryLong("(~l) >> 5", (~l) >> 5);
    printBinaryLong("l >>> 5", l >>> 5);
    printBinaryLong("(~l) >>> 5", (~l) >>> 5);
}
static void printBinaryInt(String s, int i) {
    print(s + ", int: " + i + ", двоичное: \n" +
        Integer.toBinaryString(i));
}
static void printBinaryLong(String s, long l) {
    print(s + ", long: " + l + ", двоичное:\n" +
        Long.toBinaryString(l));
}
} /* Output:
-1, int: -1, двоичное:
11111111111111111111111111111111
+1, int: 1, двоичное:
1
макс. положит., int: 2147483647, двоичное:
11111111111111111111111111111111
макс. отрицат., int: -2147483648, двоичное:
10000000000000000000000000000000
i, int: -1172028779, двоичное:
10111010001001000100001010010101
~i, int: 1172028778, двоичное:
1000101110110111011110101101010
-i, int: 1172028779, двоичное:
1000101110110111011110101101011
j, int: 1717241110, двоичное:
1100110010110110000010100010110
i & j, int: 570425364, двоичное:
1000100000000000000000000010100
i | j, int: -25213033, двоичное:

```

```

11111110011111110100011110010111
i ^ j, int: -595638397, двоичное:
11011100011111110100011110000011
i << 5, int: 1149784736, двоичное:
1000100100010000101001010100000
i >> 5, int: -36625900, двоичное:
1111101110100010010001000010100
(~i) >> 5, int: 36625899, двоичное:
10001011101101110111101011
i >>> 5, int: 97591828, двоичное:
101110100010010001000010100
(~i) >>> 5, int: 36625899, двоичное:
10001011101101110111101011
...
*///:~

```

Два метода в конце, `printBinaryInt()` и `printBinaryLong()`, получают в качестве параметров соответственно числа `int` и `long` и выводят их в двоичном формате вместе с сопроводительным текстом. Вместе с демонстрацией поразрядных операций для типов `int` и `long` этот пример также выводит минимальное/максимальное значение, `+1` и `-1` для этих типов, чтобы вы лучше понимали, как они выглядят в двоичном представлении. Заметьте, что старший бит обозначает знак: `0` соответствует положительному и `1` — отрицательному числам. Результат работы для типа `int` приведен в конце листинга.

Двоичное представление чисел еще называют *знаковым двоичным дополнением*.

- 11.** (3) Начните с числа, содержащего двоичную `1` в старшем бите (подсказка: воспользуйтесь шестнадцатеричной константой). Используя знаковый оператор сдвига вправо, сдвигайте знак до крайней правой позиции, с выводом всех промежуточных результатов методом `Integer.toBinaryString()`.
- 12.** (3) Начните с числа, состоящего из двоичных единиц. Сдвиньте его влево, а затем используйте беззнаковый оператор сдвига вправо по всем двоичным позициям, с выводом всех промежуточных результатов методом `Integer.toBinaryString()`.
- 13.** (1) Напишите метод для вывода `char` в двоичном представлении. Продемонстрируйте его работу на нескольких разных символах.

Тернарный оператор

Тернарный оператор необычен тем, что он использует три операнда. И все же это действительно оператор, так как он производит значение, в отличие от обычной конструкции выбора `if-else`, описанной в следующем разделе. Выражение записывается в такой форме:

```
логическое-условие ? выражение0 : выражение1
```

Если логическое-условие истинно (`true`), то затем вычисляется `выражение0`, и именно его результат становится результатом выполнения всего оператора. Если же логическое-условие ложно (`false`), то вычисляется `выражение1` и его значение становится результатом работы оператора.

Конечно, здесь можно было бы использовать стандартную конструкцию `if-else` (описанную чуть позже), но тернарный оператор гораздо компактнее. Хотя `C` (где этот

оператор впервые появился) претендует на звание лаконичного языка, и тернарный оператор вводился отчасти для достижения этой цели, будьте благоразумны и не используйте его всюду и постоянно — он может ухудшить читаемость программы.

Тернарный оператор отличается от `if-else` тем, что он генерирует определенный результат. Следующий пример сравнивает эти две конструкции:

```
//: operators/TernaryIfElse.java
import static net.mindview.util.Print.*;

public class TernaryIfElse {
    static int ternary(int i) {
        return i < 10 ? i * 100 : i * 10;
    }
    static int standardIfElse(int i) {
        if(i < 10)
            return i * 100;
        else
            return i * 10;
    }
    public static void main(String[] args) {
        print(ternary(9));
        print(ternary(10));
        print(standardIfElse(9));
        print(standardIfElse(10));
    }
} /* Output:
900
100
900
100
*///:~
```

Вы видите, что код `ternary()` более компактен по сравнению с записью без применения тернарного оператора в `standardIfElse()`. С другой стороны, код `standardIfElse()` более понятен, а объем не настолько уж увеличивается. Хорошенько подумайте, стоит ли использовать тернарный оператор — обычно его применение ограничивается присваиванием переменной одного из двух значений.

Операторы `+` и `+=` для `String`

В Java существует особый случай использования оператора: операторы `+` и `+=` могут применяться для конкатенации (объединения) строк, и вы уже это видели. Такое действие для этих операторов выглядит вполне естественно, хотя оно и не соответствует традиционным принципам их использования.

При создании C++ в язык была добавлена возможность *перегрузки операторов*, позволяющей программистам C++ изменять и расширять смысл почти любого оператора. К сожалению, перегрузка операторов в сочетании с некоторыми ограничениями C++ создала немало проблем при проектировании классов. Хотя реализацию перегрузки операторов в Java можно было осуществить проще, чем в C++ (это доказывает язык C#, где *существует* простой механизм перегрузки), эту возможность все же посчитали излишне сложной, и поэтому программистам на Java не дано реализовать свои собственные перегруженные операторы, как это делают программисты на C++.

Использование `+` и `+=` для строк (`String`) имеет интересные особенности. Если выражение начинается строкой, то все последующие операнды также должны быть строками (помните, что компилятор превращает символы в кавычках в объект `String`).

```
//: operators/StringOperators.java
import static net.mindview.util.Print.*;

public class StringOperators {
    public static void main(String[] args) {
        int x = 0, y = 1, z = 2;
        String s = "x, y, z ";
        print(s + x + y + z);
        print(x + " " + s); // Преобразует x в String
        s += "(summed) = "; // Оператор конкатенации
        print(s + (x + y + z));
        print("" + x); // Сокращение для Integer.toString()
    }
} /* Output:
x, y, z 012
0 x, y, z
x, y, z (summed) = 3
0
*///:~
```

Обратите внимание: первая команда `print` выводит `012` вместо `3` (как было бы при простом суммировании целых чисел). Это объясняется тем, что компилятор Java преобразует `x`, `y` и `z` в их строковые представления и выполняет конкатенацию этих строк. Вторая команда `print` преобразует начальную переменную к типу `String`, так что преобразование не зависит от того, что стоит на первом месте. Далее мы видим пример использования оператора `+=` для присоединения строки к `s` и использование круглых скобок для управления порядком вычисления выражения, так что значения `int` суммируются перед выводом.

Обратите внимание на последний пример в `main()`: иногда конструкция из пустой строки, `+` и примитива используется для выполнения преобразования без более громоздкого явного вызова метода (`Integer.toString()` в данном случае).

Типичные ошибки при использовании операторов

Многие программисты склонны второпях записывать выражение без скобок, даже когда они не уверены в последовательности вычисления выражения. Это верно и для Java.

Еще одна распространенная ошибка в C и C++ выглядит следующим образом:

```
while(x = y) {
    // ...
}
```

Программист хотел выполнить сравнение (`==`), а не присваивание. В C и C++ результат этого выражения всегда будет истинным, если только `y` не окажется нулем; вероятно, возникнет бесконечный цикл. В языке Java результат такого выражения не будет являться логическим типом (`boolean`), а компилятор ожидает в этом выражении именно `boolean` и не разрешает использовать целочисленный тип `int`, поэтому вовремя сообщит вам об ошибке времени компиляции, предупредив проблему еще перед запуском

программы. Поэтому подобная ошибка в Java никогда не происходит. (Программа откомпилируется только в одном случае: если *x* и *y* одновременно являются типами `boolean`, и тогда выражение `x = y` будет допустимо, что может привести к ошибке.)

Похожая проблема возникает в C и C++ при использовании поразрядных операторов И и ИЛИ вместо их логических аналогов. Поразрядные И и ИЛИ записываются одним символом (`&` и `|`), в то время как логические И и ИЛИ требуют в написании двух символов (`&&` и `||`). Так же как и в случае с операторами `=` и `==`, легко ошибиться и набрать один символ вместо двух. В Java компилятор предотвращает такие ошибки, так как он не позволяет использовать тип данных в неподходящем контексте.

Операторы приведения

Слово *приведение* используется в смысле «приведение к другому типу». В определенных ситуациях Java самостоятельно преобразует данные к другим типам. Например, если вещественной переменной присваивается целое значение, компилятор автоматически выполняет соответствующее преобразование (`int` преобразуется во `float`). Приведение позволяет сделать замену типа более очевидной или выполнить ее принудительно в случаях, где это не происходит в обычном порядке.

Чтобы выполнить приведение явно, запишите необходимый тип данных (включая все модификаторы) в круглых скобках слева от преобразуемого значения. Пример:

```
//: operators/Casting.java
```

```
public class Casting {
    public static void main(String[] args) {
        int i = 200;
        long lng = (long)i;
        lng = i; // "Расширение", явное преобразование не обязательно
        long lng2 = (long)200;
        lng2 = 200;
        // "Сужающее" преобразование
        i = (int)lng2; // Преобразование необходимо
    }
} ///:~
```

Как видите, приведение может выполняться и для чисел, и для переменных. Впрочем, в указанных примерах приведение является излишним, поскольку компилятор при необходимости автоматически преобразует целое `int` к типу `long`. Однако это не мешает вам выполнять необязательные приведения — например, чтобы подчеркнуть какое-то обстоятельство или просто для того, чтобы сделать программу более понятной. В других ситуациях приведение может быть необходимо для нормальной компиляции программы.

В C и C++ приведение могло стать источником ошибок и неоднозначности. В Java приведение безопасно, за одним исключением: при выполнении так называемого *сужающего приведения* (от типа данных, способного хранить больше информации, к менее содержательному типу данных), то есть при опасности потери данных. В таком случае компилятор заставляет вас выполнить явное приведение; фактически он говорит: «Это может быть опасно, но если вы уверены в своей правоте, опишите действие явно». В случае с *расширяющим приведением* явное описание не понадобится, так как

новый тип данных способен хранить больше информации, чем прежний, и поэтому потеря данных исключена.

В Java разрешается приводить любой простейший тип данных к любому другому простейшему типу, но это не относится к типу `boolean`, который вообще не подлежит приведению. Классы также не поддерживают произвольное приведение. Чтобы преобразовать один класс в другой, требуются специальные методы. (Как будет показано позднее, объекты можно преобразовывать в рамках *семейства* типов; объект *Дуб* можно преобразовать в *Дерево*, и наоборот, но не к постороннему типу вроде *Камня*.)

Округление и усечение

При выполнении сужающих преобразований необходимо обращать внимание на усечение и округление данных. Например, как должен действовать компилятор Java при преобразовании вещественного числа в целое? Скажем, если значение 29,7 приводится к типу `int`, что получится — 29 или 30? Ответ на этот вопрос может дать следующий пример:

```
//: operators/CastingNumbers.java
// Что происходит при приведении типов
// float или double к целочисленным значениям?
import static net.mindview.util.Print.*;

public class CastingNumbers {
    public static void main(String[] args) {
        double above = 0.7, below = 0.4;
        float fabove = 0.7f, fbelow = 0.4f;
        print("(int)above: " + (int)above);
        print("(int)below: " + (int)below);
        print("(int)fabove: " + (int)fabove);
        print("(int)fbelow: " + (int)fbelow);
    }
} /* Output:
(int)above: 0
(int)below: 0
(int)fabove: 0
(int)fbelow: 0
*///:~
```

Итак, приведение от типов с повышенной точностью `double` и `float` к целочисленным значениям всегда осуществляется с усечением целой части. Если вы предпочитаете, чтобы результат округлялся, используйте методы `round()` из `java.lang.Math`:

```
//: operators/RoundingNumbers.java
// Rounding floats and doubles.
import static net.mindview.util.Print.*;

public class RoundingNumbers {
    public static void main(String[] args) {
        double above = 0.7, below = 0.4;
        float fabove = 0.7f, fbelow = 0.4f;
        print("Math.round(above): " + Math.round(above));
        print("Math.round(below): " + Math.round(below));
        print("Math.round(fabove): " + Math.round(fabove));
        print("Math.round(fbelow): " + Math.round(fbelow));
    }
}
```

```

} /* Output:
Math.round(above): 1
Math.round(below): 0
Math.round(fabove): 1
Math.round(fbelow): 0
*///:~

```

Так как `round()` является частью `java.lang`, дополнительное импортирование не требуется.

Повышение

При проведении любых математических и поразрядных операций примитивные типы данных, меньшие `int` (то есть `char`, `byte` и `short`), приводятся к типу `int` перед проведением операций, и получаемый результат имеет тип `int`. Поэтому, если вам снова понадобится присвоить его меньшему типу, придется использовать приведение (с возможной потерей информации). В основном самый емкий тип данных, присутствующий в выражении, и определяет величину результата этого выражения; так, при перемножении `float` и `double` результатом станет `double`, а при сложении `long` и `int` вы получите в результате `long`.

В Java отсутствует sizeof

В C и C++ оператор `sizeof()` выдает количество байтов, выделенных для хранения данных. Главная причина для использования `sizeof()` — переносимость программы. Различным типам данных может отводиться различное количество памяти на разных компьютерах, поэтому для программиста важно определить размер этих типов перед проведением операций, зависящих от этих величин. Например, один компьютер выделяет под целые числа 32 бита, а другой — всего лишь 16 бит. В результате на первой машине программа может хранить в целочисленном представлении числа из большего диапазона. Конечно, аппаратная совместимость создает немало хлопот для программистов на C и C++.

В Java оператор `sizeof()` не нужен, так как все типы данных имеют одинаковые размеры на всех машинах. Вам не нужно заботиться о переносимости на низком уровне — она встроена в язык.

Сводка операторов

Следующий пример показывает, какие примитивные типы данных используются с теми или иными операторами. Вообще-то это один и тот же пример, повторенный много раз, но для разных типов данных. Файл должен компилироваться без ошибок, поскольку все строки, содержащие неверные операции, предварены символами `//!`.

```

//: operators/AllOps.java
// Проверяет все операторы со всеми
// примитивными типами данных, чтобы показать,
// какие операции допускаются компилятором Java.

public class AllOps {
    // для получения результатов тестов типа boolean:

```



```
void f(boolean b) {}
void boolTest(boolean x, boolean y) {
    // Арифметические операции:
    //! x = x * y;
    //! x = x / y;
    //! x = x % y;
    //! x = x + y;
    //! x = x - y;
    //! x++;
    //! x--;
    //! x = +y;
    //! x = -y;
    // Операции сравнения и логические операции:
    //! f(x > y);
    //! f(x >= y);
    //! f(x < y);
    //! f(x <= y);
    f(x == y);
    f(x != y);
    f(!y);
    x = x && y;
    x = x || y;
    // Поразрядные операторы:
    //! x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Совмещенное присваивание:
    //! x += y;
    //! x -= y;
    //! x *= y;
    //! x /= y;
    //! x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Приведение:
    //! char c = (char)x;
    //! byte B = (byte)x;
    //! short s = (short)x;
    //! int i = (int)x;
    //! long l = (long)x;
    //! float f = (float)x;
    //! double d = (double)x;
}
void charTest(char x, char y) {
    // Арифметические операции:
    x = (char)(x * y);
    x = (char)(x / y);
    x = (char)(x % y);
    x = (char)(x + y);
    x = (char)(x - y);
}
```

```

x++;
x--;
x = (char)+y;
x = (char)-y;
// Операции сравнения и логические операции:
f(x > y);
f(x >= y);
f(x < y);
f(x <= y);
f(x == y);
f(x != y);
//! f(!x);
//! f(x && y);
//! f(x || y);
// Поразрядные операции:
x = (char)~y;
x = (char)(x & y);
x = (char)(x | y);
x = (char)(x ^ y);
x = (char)(x << 1);
x = (char)(x >> 1);
x = (char)(x >>> 1);
// Совмещенное присваивание:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <<= 1;
x >>= 1;
x >>>= 1;
x &= y;
x ^= y;
x |= y;
// Приведение:
//! boolean b = (boolean)x;
byte B = (byte)x;
short s = (short)x;
int i = (int)x;
long l = (long)x;
float f = (float)x;
double d = (double)x;
}
void byteTest(byte x, byte y) {
    // Арифметические операции:
    x = (byte)(x* y);
    x = (byte)(x / y);
    x = (byte)(x % y);
    x = (byte)(x + y);
    x = (byte)(x - y);
    x++;
    x--;
    x = (byte)+ y;
    x = (byte)- y;
    // Операции сравнения и логические операции:
    f(x > y);
    f(x >= y);
    f(x < y);

```

```

f(x <= y);
f(x == y);
f(x != y);
//! f(!x);
//! f(x && y);
//! f(x || y);
// Поразрядные операции:
x = (byte)~y;
x = (byte)(x & y);
x = (byte)(x | y);
x = (byte)(x ^ y);
x = (byte)(x << 1);
x = (byte)(x >> 1);
x = (byte)(x >>> 1);
// Совмещенное присваивание:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <<= 1;
x >>= 1;
x >>>= 1;
x &= y;
x ^= y;
x |= y;
// Приведение:
//! boolean b = (boolean)x;
char c = (char)x;
short s = (short)x;
int i = (int)x;
long l = (long)x;
float f = (float)x;
double d = (double)x;
}
void shortTest(short x, short y) {
    // Арифметические операции:
    x = (short)(x * y);
    x = (short)(x / y);
    x = (short)(x % y);
    x = (short)(x + y);
    x = (short)(x - y);
    x++;
    x--;
    x = (short)+y;
    x = (short)-y;
    // Операции сравнения и логические операции:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Поразрядные операции:
    x = (short)~y;

```

```

x = (short)(x & y);
x = (short)(x | y);
x = (short)(x ^ y);
x = (short)(x << 1);
x = (short)(x >> 1);
x = (short)(x >>> 1);
// Совмещенное присваивание:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <<= 1;
x >>= 1;
x >>>= 1;
x &= y;
x ^= y;
x |= y;
// Преобразование:
//! boolean b = (boolean)x;
char c = (char)x;
byte B = (byte)x;
int i = (int)x;
long l = (long)x;
float f = (float)x;
double d = (double)x;
}
void intTest(int x, int y) {
    // Арифметические операции:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Операции сравнения и логические операции:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Поразрядные операции:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
    // Совмещенное присваивание:
    x += y;

```

```
x -= y;
x *= y;
x /= y;
x %= y;
x <<= 1;
x >>= 1;
x >>>= 1;
x &= y;
x ^= y;
x |= y;
// Приведение:
//! boolean b = (boolean)x;
char c = (char)x;
byte B = (byte)x;
short s = (short)x;
long l = (long)x;
float f = (float)x;
double d = (double)x;
}
void longTest(long x, long y) {
    // Арифметические операции:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Операции сравнения и логические операции:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Поразрядные операции:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
    // Совмещенное присваивание:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
```

```

x ^= y;
x |= y;
// Приведение:
//! boolean b = (boolean)x;
char c = (char)x;
byte B = (byte)x;
short s = (short)x;
int i = (int)x;
float f = (float)x;
double d = (double)x;
}
void floatTest(float x, float y) {
    // Арифметические операции:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Операции сравнения и логические операции:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Поразрядные операции:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Совмещенное присваивание:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    //! x &= y;
    //! x ^= y;
    //! x |= y;
    // Приведение:
    //! boolean b = (boolean)x;
    char c = (char)x;
    byte B = (byte)x;
    short s = (short)x;
    int i = (int)x;
}

```

```

    long l = (long)x;
    double d = (double)x;
}
void doubleTest(double x, double y) {
    // Арифметические операции:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Операции сравнения и логические операции:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    /// f(!x);
    /// f(x && y);
    /// f(x || y);
    // Поразрядные операции:
    /// x = ~y;
    /// x = x & y;
    /// x = x | y;
    /// x = x ^ y;
    /// x = x << 1;
    /// x = x >> 1;
    /// x = x >>> 1;
    // Совмещенное присваивание:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    /// x <<= 1;
    /// x >>= 1;
    /// x >>>= 1;
    /// x &= y;
    /// x ^= y;
    /// x |= y;
    // Приведение:
    /// boolean b = (boolean)x;
    char c = (char)x;
    byte B = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
}
} ///:~

```

Заметьте, что действия с типом `boolean` довольно ограничены. Ему можно присвоить значение `true` или `false`, проверить на истинность или ложность, но нельзя добавлять логические переменные к другим типам или производить с ними любые иные операции.

В случае с типами `char`, `byte` и `short` можно заметить эффект повышения при использовании арифметических операторов. Любая арифметическая операция с этими типами дает результат типа `int`, который затем нужно явно приводить к изначальному типу (сужающее приведение, при котором возможна потеря информации). При использовании значений типа `int` приведение осуществлять не придется, потому что все значения уже имеют этот тип. Однако не заблуждайтесь относительно безопасности происходящего. При перемножении двух достаточно больших целых чисел `int` произойдет переполнение. Следующий пример демонстрирует сказанное:

```
//: operators/Overflow.java
// Сюрприз! В Java возможно переполнение.

public class Overflow {
    public static void main(String[] args) {
        int big = Integer.MAX_VALUE;
        System.out.println("большое = " + big);
        int bigger = big * 4;
        System.out.println("еще больше = " + bigger);
    }
} /* Output:
большое = 2147483647
еще больше = -4
*///:~
```

Компилятор не выдает никаких ошибок или предупреждений, и во время исполнения не возникнет исключений. Язык Java хорош, но хорош *не настолько*.

Совмещенное присваивание *не требует* приведения для типов `char`, `byte` и `short`, хотя для них и производится повышение, как и в случае с арифметическими операциями. С другой стороны, отсутствие приведения в таких случаях, несомненно, упрощает программу.

Можно легко заметить, что, за исключением типа `boolean`, любой примитивный тип может быть преобразован к другому примитиву. Как упоминалось ранее, необходимо остерегаться сужающего приведения при преобразованиях к меньшему типу, так как при этом возникает риск потери информации.

14. (3) Напишите метод, который получает два аргумента `String`, выполняет с ними все операции логических сравнений и выводит результаты. Для операций `==` и `!=` также выполните проверку `equals()`. Вызовите свой метод из `main()` для нескольких разных объектов `String`.

Резюме

Читатели с опытом работы на любом языке семейства C могли убедиться, что операторы Java почти ничем не отличаются от классических. Если же материал этой главы показался трудным, обращайтесь к мультимедийной презентации «Thinking in C» (www.MindView.net).

4

Управляющие конструкции

Подобно любому живому существу, программа должна управлять своим миром и принимать решения во время исполнения. В языке Java для принятия решений используются управляющие конструкции.

В Java задействованы все управляющие конструкции языка C, поэтому читателям с опытом программирования на языках C или C++ основная часть материала будет знакома. Почти во всех процедурных языках поддерживаются стандартные команды управления, и во многих языках они совпадают. В Java к их числу относятся ключевые слова `if-else`, `while`, `do-while`, `for`, `return`, `break`, а также команда выбора `switch`. Однако в Java не поддерживается часто критикуемый оператор `goto` (который, впрочем, все же является самым компактным решением в некоторых ситуациях). Безусловные переходы «в стиле» `goto` возможны, но гораздо более ограничены по сравнению с классическими переходами `goto`.

true и false

Все конструкции с условием вычисляют истинность или ложность условного выражения, чтобы определить способ выполнения. Пример условного выражения — `A == B`. Оператор сравнения `==` проверяет, равно ли значение `A` значению `B`. Результат проверки может быть истинным (`true`) или ложным (`false`). Любой из описанных в этой главе операторов сравнения может применяться в условном выражении. Заметьте, что Java не разрешает использовать числа в качестве логических значений, хотя это позволено в C и C++ (где не-ноль считается «истинным», а ноль — «ложным»). Если вам потребуется использовать числовой тип там, где требуется `boolean` (скажем, в условии `if(a)`), то сначала придется его преобразовать к логическому типу оператором сравнения в условном выражении — например, `if(a != 0)`.

if-else

Команда `if-else` является, наверное, наиболее распространенным способом передачи управления в программе. Присутствие ключевого слова `else` не обязательно, поэтому конструкция `if` существует в двух формах:

```
if(логическое выражение)
    команда
```

или

```
if(логическое выражение)
    команда
else
    команда
```

Условие должно дать результат типа `boolean`. В секции *команда* располагается либо простая команда, завершенная точкой с запятой, или составная конструкция из команд, заключенная в фигурные скобки.

В качестве примера применения `if-else` представлен метод `test()`, который выдает информацию об отношениях между двумя числами — «больше», «меньше» или «равно»:

```
//: control/IfElse.java
import static net.mindview.util.Print.*;

public class IfElse {
    static int result = 0;
    static void test(int testval, int target) {
        if(testval > target)
            result = +1;
        else if(testval < target)
            result = -1;
        else
            result = 0; // равные числа
    }
    public static void main(String[] args) {
        test(10, 5);
        print(result);
        test(5, 10);
        print(result);
        test(5, 5);
        print(result);
    }
} /* Output:
1
-1
0
*///:~
```

Внутри метода `test()` встречается конструкция `else if`; это не новое ключевое слово, а `else`, за которым следует начало другой команды `if`.

Java, как C и C++, относится к языкам со свободным форматом. Тем не менее в командах управления рекомендуется делать отступы, благодаря чему читателю программы будет легче понять, где начинается и заканчивается управляющая конструкция.

Циклы

Конструкции `while`, `do-while` и `for` управляют циклами и иногда называются *циклическими командами*. Команда повторяется до тех пор, пока управляющее логическое выражение не станет ложным. Форма цикла `while` следующая:

while(логическое выражение)
команда

логическое выражение вычисляется перед началом цикла, а затем каждый раз перед выполнением очередного повторения оператора.

Следующий простой пример генерирует случайные числа до тех пор, пока не будет выполнено определенное условие:

```
//: control/WhileTest.java
// Пример использования цикла while

public class WhileTest {
    static boolean condition() {
        boolean result = Math.random() < 0.99;
        System.out.print(result + ", ");
        return result;
    }
    public static void main(String[] args) {
        while(condition())
            System.out.println("Inside 'while'");
        System.out.println("Exited 'while'");
    }
} /* (Выполните, чтобы посмотреть результат) *///:~
```

В методе `condition()` используется статический метод `random()` из библиотеки `Math`, который генерирует значение `double`, находящееся между 0 и 1 (включая 0, но не 1). Значение `result` определяется оператором `<`, который создает результат `boolean`. При выводе значения `boolean` автоматически выводится строка «true» или «false». Условие `while` означает: «повторять, пока `condition()` возвращает true».

do-while

Форма конструкции `do-while` такова:

```
do
    команда
while(логическое выражение);
```

Единственное отличие цикла `do-while` от `while` состоит в том, что цикл `do-while` выполняется по крайней мере единожды, даже если условие изначально ложно. В цикле `while`, если условие изначально ложно, тело цикла никогда не отрабатывает. На практике конструкция `do-while` употребляется реже, чем `while`.

for

Пожалуй, конструкции `for` составляют наиболее распространенную разновидность циклов. Цикл `for` проводит инициализацию перед первым шагом цикла. Затем выполняется проверка условия цикла, и в конце каждой итерации осуществляется некое «приращение» (обычно изменение управляющей переменной). Цикл `for` записывается следующим образом:

```
for(инициализация; логическое выражение; шаг)
    команда
```

Любое из трех выражений цикла (инициализация, логическое выражение или шаг) можно пропустить. Перед выполнением каждого шага цикла проверяется условие цикла; если оно окажется ложным, выполнение продолжается с инструкции, следующей за конструкцией `for`. В конце каждой итерации выполняется секция шаг.

Цикл `for` обычно используется для «счетных» задач:

```
//: control/ListCharacters.java
// Пример использования цикла "for": перебор
// всех ASCII-символов нижнего регистра

public class ListCharacters {
    public static void main(String[] args) {
        for(char c = 0; c < 128; c++)
            if(Character.isLowerCase(c))
                System.out.println("значение: " + (int)c +
                                   " символ: " + c);
    }
} /* Output:
значение: 97 символ: a
значение: 98 символ: b"
значение: 99 символ: c"
значение: 100 символ: d"
значение: 101 символ: e"
значение: 102 символ: f"
значение: 103 символ: g"
значение: 104 символ: h"
значение: 105 символ: i
значение: 106 символ: j
...
*///:~
```

Обратите внимание, что переменная `c` определяется в точке ее использования в управляющем выражении цикла `for`, а не в начале блока, обозначенного фигурными скобками. Область действия для `c` — все выражения, принадлежащие циклу.

В программе также использует класс-«обертка» `java.lang.Character`, который позволяет не только представить простейший тип `char` в виде объекта, но и содержит ряд дополнительных возможностей. В нашем примере используется статический метод этого класса `isLowerCase()`, который проверяет, является ли некоторая буква строчной.

Традиционные процедурные языки (такие, как `C`) требовали, чтобы все переменные определялись в начале блока цикла, чтобы компилятор при создании блока мог выделить память под эти переменные. В `Java` и `C++` переменные разрешено объявлять в том месте блока цикла, где это необходимо. Это позволяет программировать в более удобном стиле и упрощает понимание кода.

- 15.** (1) Напишите программу, которая выводит числа от 1 до 100.
- 16.** (2) Напишите программу, которая генерирует 25 случайных значений типа `int`. Для каждого значения команда `if-else` сообщает, в каком отношении оно находится с другим случайно сгенерированным числом (больше, меньше, равно).
- 17.** (1) Измените упражнение 2 так, чтобы код выполнялся в «бесконечном» цикле `while`. Программа должна работать до тех пор, пока ее выполнение не будет прервано с клавиатуры (как правило, нажатием клавиш `Ctrl+C`).

18. (3) Напишите программу, использующую два вложенных цикла `for` и оператор остатка (%) для поиска и вывода простых чисел (то есть целых чисел, не делящихся нацело ни на какое другое число, кроме себя и 1).
19. (4) Повторите упражнение 1 из предыдущей главы, используя тернарный оператор и поразрядную проверку для вывода нулей и единиц (вместо вызова метода `Integer.toString()`).

Оператор-запятая

Ранее в этой главе уже упоминалось о том, что *оператор «запятая»* (но не *запятая-разделитель*, которая разграничивает определения и аргументы функций) может использоваться в Java только в управляющем выражении цикла `for`. И в секции инициализации цикла, и в его управляющем выражении можно записать несколько команд, разделенных запятыми; они будут обработаны последовательно.

Оператор «запятая» позволяет определить несколько переменных в цикле `for`, но все эти переменные должны принадлежать к одному типу:

```
//: control/CommaOperator.java
```

```
public class CommaOperator {
    public static void main(String[] args) {
        for(int i = 1, j = i + 10; i < 5; i++, j = i * 2) {
            System.out.println("i = " + i + " j = " + j);
        }
    }
} /* Output:
i = 1 j = 11
i = 2 j = 4
i = 3 j = 6
i = 4 j = 8
*///:~
```

Определение `int` в заголовке `for` относится как к `i`, так и к `j`. Инициализационная часть может содержать любое количество определений переменных *одного типа*. Определение переменных в управляющих выражениях возможно только в цикле `for`. На другие команды выбора или циклов этот подход не распространяется.

Как видите, в частях инициализация и шаг команды рассматриваются в последовательном порядке.

Синтаксис foreach

В Java SE5 появилась новая, более компактная форма `for` для перебора элементов массивов и контейнеров (см. главы 16 и 17). Эта упрощенная форма, называемая *синтаксисом `foreach`*, не требует ручного изменения переменной `int` для перебора последовательности объектов — цикл автоматически представляет очередной элемент.

Следующая программа создает массив `float`, после чего перебирает все его элементы:

```

//: control/ForEachFloat.java
import java.util.*;

public class ForEachFloat {
    public static void main(String[] args) {
        Random rand = new Random(47);
        float f[] = new float[10];
        for(int i = 0; i < 10; i++)
            f[i] = rand.nextFloat();
        for(float x : f)
            System.out.println(x);
    }
} /* Output:
0.72711575
0.39982635
0.5309454
0.0534122
0.16020656
0.57799757
0.18847865
0.4170137
0.51660204
0.73734957
*///:~

```

Массив заполняется уже знакомым циклом `for`, потому что для его заполнения должны использоваться индексы. Упрощенный синтаксис используется в следующей команде:

```
for(float x : f)
```

Эта конструкция определяет переменную `x` типа `float`, после чего последовательно присваивает ей элементы `f`.

Любой метод, возвращающий массив, может использоваться с `foreach`¹. Например, класс `String` содержит метод `toCharArray()`, возвращающий массив `char`; следовательно, перебор символов строки может осуществляться так:

```

//: control/ForEachString.java

public class ForEachString {
    public static void main(String[] args) {
        for(char c : "An African Swallow".toCharArray() )
            System.out.print(c + " ");
    }
} /* Output:
A n   A f r i c a n   S w a l l o w
*///:~

```

Как будет показано далее, «синтаксис `foreach`» также работает для любого объекта, поддерживающего интерфейс `Iterable`.

Многие команды `for` основаны на переборе серии целочисленных значений:

```
for (int i = 0; i < 100; i++)
```

¹ Далее этот термин будет использоваться для обозначения данной разновидности циклов, хотя такого ключевого слова в Java нет. — *Примеч. ред.*

В таких случаях «синтаксис `foreach`» работать не будет, если только вы предварительно не создадите массив `int`. Для упрощения этой задачи я включил в библиотеку `net.mindview.util.Range` метод `range()`, который автоматически генерирует соответствующий массив (и используется для статического импортирования):

```
//: control/ForEachInt.java
import static net.mindview.util.Range.*;
import static net.mindview.util.Print.*;

public class ForEachInt {
    public static void main(String[] args) {
        for(int i : range(10)) // 0..9
            printnb(i + " ");
        print();
        for(int i : range(5, 10)) // 5..9
            printnb(i + " ");
        print();
        for(int i : range(5, 20, 3)) // 5..20 step 3
            printnb(i + " ");
        print();
    }
} /* Output:
0 1 2 3 4 5 6 7 8 9
5 6 7 8 9
5 8 11 14 17
*///:~
```

Метод `range()` *перегружен*, то есть одно имя метода может использоваться с разными списками аргументов (вскоре перегрузка будет рассмотрена более подробно). Первая перегруженная форма `range()` просто начинает с нуля и генерирует значения до верхней границы диапазона (не включая ее). Вторая форма начинает с первого значения и проходит до значения, на единицу меньшую второго. Наконец, третья форма использует величину приращения. Как вы вскоре увидите, `range()` является очень простой формой *генераторов*, которые будут рассматриваться позже.

Обратите внимание на использование `printnb()` вместо `print()`. Метод `printnb()` не выводит символ новой строки, что позволяет выводить строку по частям.

Синтаксис `foreach` не только экономит объем вводимого кода. Что еще важнее, он значительно упрощает чтение программы и указывает, *что* вы пытаетесь сделать (перебрать все элементы массива), вместо подробного описания того, *как* вы собираетесь это сделать («Я создаю индекс для перебора всех элементов массива»). Я буду использовать синтаксис `foreach` везде, где это возможно.

return

Следующая группа ключевых слов обеспечивает *безусловные переходы*, то есть передачу управления без проверки каких-либо условий. К их числу относятся команды `return`, `break` и `continue`, а также конструкция перехода по метке, аналогичная `goto` в других языках.

У ключевого слова `return` имеются два предназначения: оно указывает, какое значение возвращается методом (если только он не возвращает тип `void`), а также используется

для немедленного выхода из метода. Метод `test()` из предыдущего примера можно переписать так, чтобы он воспользовался новыми возможностями:

```
//: control/IfElse2.java
import static net.mindview.util.Print.*;

public class IfElse2 {
    static int test(int testval, int target) {
        if(testval > target)
            return +1;
        else if(testval < target)
            return -1;
        else
            return 0; // Одинаковые значения
    }
    public static void main(String[] args) {
        print(test(10, 5));
        print(test(5, 10));
        print(test(5, 5));
    }
} /* Output:
1
-1
0
*///:~
```

В данном случае секция `else` не нужна, поскольку работа метода не продолжается после выполнения инструкции `return`.

Если метод, возвращающий `void`, не содержит команды `return`, такая команда неявно выполняется в конце метода. Тем не менее, если метод возвращает любой тип, кроме `void`, проследите за тем, чтобы каждая логическая ветвь возвращала конкретное значение.

- 20.** (2) Измените метод `test()` так, чтобы он получал два дополнительных аргумента `begin` и `end`, а значение `testval` проверялось на принадлежность к диапазону `[begin, end]` (с включением границ).

break и continue

В теле любого из циклов можно управлять потоком программы при помощи специальных ключевых слов `break` и `continue`. Команда `break` завершает цикл, при этом оставшиеся операторы цикла не выполняются. Команда `continue` останавливает выполнение текущей итерации цикла и переходит к началу цикла, чтобы начать выполнение нового шага.

Следующая программа показывает пример использования команд `break` и `continue` внутри циклов `for` и `while`:

```
//: control/BreakAndContinue.java
// Применение ключевых слов break и continue
import static net.mindview.util.Range.*;
```



```

public class BreakAndContinue {
    public static void main(String[] args) {
        for(int i = 0; i < 100; i++) {
            if(i == 74) break;          // Выход из цикла
            if(i % 9 != 0) continue;    // Следующая итерация
            System.out.print(i + " ");
        }
        System.out.println();
        // Использование foreach:
        for(int i : range(100)) {
            if(i == 74) break;          // Выход из цикла
            if(i % 9 != 0) continue;    // Следующая итерация
            System.out.print(i + " ");
        }
        System.out.println();
        int i = 0;
        // "Бесконечный цикл":
        while(true) {
            i++;
            int j = i * 27;
            if(j == 1269) break;        // Выход из цикла
            if(i % 10 != 0) continue;   // Возврат в начало цикла
            System.out.print(i + " ");
        }
    }
}
/* Output:
0 9 18 27 36 45 54 63 72
0 9 18 27 36 45 54 63 72
10 20 30 40
*///:~

```

В цикле `for` переменная `i` никогда не достигает значения 100 — команда `break` прерывает цикл, когда значение переменной становится равным 74. Обычно `break` используется только тогда, когда вы точно знаете, что условие выхода из цикла действительно достигнуто. Команда `continue` переводит исполнение в начало цикла (и таким образом увеличивает значение `i`), когда `i` не делится без остатка на 9. Если деление производится без остатка, значение выводится на экран.

Второй цикл `for` демонстрирует использование синтаксиса `foreach` с тем же результатом.

Последняя часть программы демонстрирует «бесконечный цикл», который, в теории, должен исполняться вечно. Однако в теле цикла вызывается команда `break`, которая и завершает цикл. Команда `continue` переводит исполнение к началу цикла, и при этом остаток цикла не выполняется. (Таким образом, вывод на экран в последнем цикле происходит только в том случае, если значение `i` делится на 10 без остатка.) Значение 0 выводится, так как `0 % 9` дает в результате 0.

Вторая форма бесконечного цикла — `for(;;)`. Компилятор реализует конструкции `while(true)` и `for(;;)` одинаково, так что выбор является делом вкуса.

- 21.** (1) Измените упражнение 1 так, чтобы выход из программы осуществлялся ключевым словом `break` при значении 99. Попробуйте использовать ключевое слово `return`.

Нехорошая команда goto

Ключевое слово `goto` появилось одновременно с языками программирования. Действительно, безусловный переход заложил основы принятия решений в языке ассемблера: «если условие А — перейти туда, а иначе — перейти сюда». Если вам доводилось читать код на ассемблере, который генерируют фактически все компиляторы, наверняка вы замечали многочисленные переходы, управляющие выполнением программы (компилятор Java производит свой собственный «ассемблерный» код, но последний выполняется виртуальной машиной Java, а не аппаратным процессором).

Команда `goto` реализует безусловный переход на уровне исходного текста программы, и именно это обстоятельство принесло ему дурную славу. Если программа постоянно «прыгает» из одного места в другое, нет ли способа реорганизовать ее код так, чтобы управление программой перестало быть таким «прыгучим»? Команда `goto` впала в настоящую немилость с опубликованием знаменитой статьи Эдгара Дейкстры «Команда GOTO вредна» (*Goto considered harmful*¹), и с тех пор порицание команды `goto` стало чуть ли не спортом, а защитники репутации многострадального оператора разбежались по укронным углам.

Как всегда в ситуациях такого рода, существует «золотая середина». Проблема состоит не в использовании `goto` вообще, но в злоупотреблении — все же иногда именно команда `goto` позволяет лучше всего организовать управление программой.

Хотя слово `goto` зарезервировано в языке Java, оно там не используется; Java не имеет команды `goto`. Однако существует механизм, чем-то похожий на безусловный переход и осуществляемый командами `break` и `continue`. Скорее, это способ прервать итерацию цикла, а не передать управление в другую точку программы. Причина его обсуждения вместе с `goto` состоит в том, что он использует тот же механизм — метки.

Метка представляет собой идентификатор с последующим двоеточием:

```
label1:
```

Единственное место, где в Java метка может оказаться полезной, — прямо перед телом цикла. Причем никаких дополнительных команд между меткой и телом цикла быть не должно. Причина помещения метки перед телом цикла может быть лишь одна — вложение внутри цикла другого цикла или конструкции выбора. Обычные версии `break` и `continue` прерывают только текущий цикл, в то время как их версии с метками способны досрочно завершать циклы и передавать выполнение в точку, адресуемую меткой:

```
label1:
внешний-цикл {
    внутренний-цикл {
        //...
        break;           // 1
        //...
        continue;       // 2
        //...
        continue label1; // 3
    }
}
```

¹ Оригинал статьи *Go To Statement considered harmful* имеет постоянный адрес в Интернете: <http://www.acm.org/classics/oct95>. — Примеч. ред.

```

    //...
    break label1; // 4
  }
}

```

В первом случае (1) команда `break` прерывает выполнение внутреннего цикла, и управление переходит к внешнему циклу. Во втором случае (2) оператор `continue` передает управление к началу внутреннего цикла. Но в третьем варианте (3) команда `continue label1` влечет выход из внутреннего *и внешнего* циклов и возврат к метке `label1`. Далее выполнение цикла фактически продолжается, но с внешнего цикла. В четвертом случае (4) команда `break label1` также вызывает переход к метке `label1`, но на этот раз повторный вход в итерацию не происходит. Это действие останавливает выполнение обоих циклов.

Пример использования цикла `for` с метками:

```

//: control/LabeledFor.java
// Цикл for с метками
import static net.mindview.util.Print.*;

public class LabeledFor {
    public static void main(String[] args) {
        int i = 0;
        outer:          // Другие команды недопустимы
        for(;; true ;) { // infinite loop
            inner:       // Другие команды недопустимы
            for(;; i < 10; i++) {
                print("i = " + i);
                if(i == 2) {
                    print("continue");
                    continue;
                }
                if(i == 3) {
                    print("break");
                    i++; // В противном случае значение i
                        // не увеличивается
                    break;
                }
                if(i == 7) {
                    print("continue outer");
                    i++; // В противном случае значение i
                        // не увеличивается
                    continue outer;
                }
                if(i == 8) {
                    print("break outer");
                    break outer;
                }
            }
            for(int k = 0; k < 5; k++) {
                if (k == 3) {
                    print("continue inner");
                    continue inner;
                }
            }
        }
    }
}

```

```

        // Использовать break или continue
        // с метками здесь не разрешается
    }
} /* Output:
i = 0
continue inner
i = 1
continue inner
i = 2
continue
i = 3
break
i = 4
continue inner
i = 5
continue inner
i = 6
continue inner
i = 7
continue outer
i = 8
break outer
*///:~

```

Заметьте, что оператор `break` завершает цикл `for`, вследствие этого выражение с инкрементом не выполняется до завершения очередного шага. Поэтому из-за пропуска операции инкремента в цикле переменная непосредственно увеличивается на единицу, когда `i == 3`. При выполнении условия `i == 7` команда `continue outer` переводит выполнение на начало цикла; инкремент опять пропускается, поэтому и в этом случае переменная увеличивается явно.

Без команды `break outer` программе не удалось бы покинуть внешний цикл из внутреннего цикла, так как команда `break` сама по себе завершает выполнение только текущего цикла (это справедливо и для `continue`).

Конечно, если завершение цикла также приводит к завершению работы метода, можно просто применить команду `return`.

Теперь рассмотрим пример, в котором используются команды `break` и `continue` с метками в цикле `while`:

```

//: control/LabeledWhile.java
// Цикл while с метками
import static net.mindview.util.Print.*;

public class LabeledWhile {
    public static void main(String[] args) {
        int i = 0;
        outer:
        while(true) {
            print("Внешний цикл while");
            while(true) {
                i++;
                print("i = " + i);
                if(i == 1) {
                    print("continue");
                    continue;
                }
            }
        }
    }
}

```

```

    }
    if(i == 3) {
        print("continue outer");
        continue outer;
    }
    if(i == 5) {
        print("break");
        break;
    }
    if(i == 7) {
        print("break outer");
        break outer;
    }
}
}
}
} /* Output:
Внешний цикл while
i = 1
continue
i = 2
i = 3
continue outer
Внешний цикл while
i = 4
i = 5
break
Внешний цикл while
i = 6
i = 7
break outer
*///:~

```

Те же правила верны и для цикла `while`:

- ☐ Обычная команда `continue` переводит исполнение к началу текущего внутреннего цикла, программа продолжает работу.
- ☐ Команда `continue` с меткой вызывает переход к метке и повторный вход в цикл, следующий прямо за этой меткой.
- ☐ Команда `break` завершает выполнение текущего цикла.
- ☐ Команда `break` с меткой завершает выполнение внутреннего цикла и цикла, который находится после указанной метки.

Важно помнить, что *единственная* причина для использования меток в Java — наличие вложенных циклов и необходимость выхода по `break` и продолжения по `continue` не только для внутренних, но и для внешних циклов.

В статье Дейкстры особенно критикуются метки, а не сам оператор `goto`. Дейкстра отмечает, что, как правило, количество ошибок в программе растет с увеличением количества меток в этой программе. Метки затрудняют анализ программного кода. Заметьте, что метки Java не страдают этими пороками, потому что их место расположения ограничено и они не могут использоваться для беспорядочной передачи управления. В данном случае от ограничения возможностей функциональность языка только выигрывает.

switch

Команду `switch` часто называют *командой выбора*. С помощью конструкции `switch` осуществляется выбор из нескольких альтернатив, в зависимости от значения целочисленного выражения. Форма команды выглядит так:

```
switch(целочисленное-выражение) {
    case целое-значение1 : команда; break;
    case целое-значение2 : команда; break;
    case целое-значение3 : команда; break;
    case целое-значение4 : команда; break;
    case целое-значение5 : команда; break;
    // ...
    default: оператор;
}
```

целочисленное-выражение — выражение, в результате вычисления которого получается целое число. Команда `switch` сравнивает результат целочисленного-выражения с каждым последующим целым-значением. Если обнаруживается совпадение, исполняется соответствующая команда (простая или составная). Если же совпадения не находится, исполняется команда после ключевого слова `default`.

Нетрудно заметить, что каждая секция `case` заканчивается командой `break`, которая передает управление к концу команды `switch`. Такой синтаксис построения конструкции `switch` считается стандартным, но команда `break` не является строго обязательной. Если она отсутствует, при выходе из секции будет выполняться код следующих секций `case`, пока в программе не встретится очередная команда `break`. Необходимость в подобном поведении возникает довольно редко, но опытному программисту оно может пригодиться. Заметьте, что последняя секция `default` не содержит команды `break`; выполнение продолжается в конце конструкции `switch`, то есть там, где оно оказалось бы после вызова `break`. Впрочем, вы можете использовать `break` и в предложении `default`, без практической пользы, просто ради «единства стиля».

Команда `switch` обеспечивает компактный синтаксис реализации множественного выбора (то есть выбора из нескольких путей выполнения программы), но для нее необходимо управляющее выражение, результатом которого является целочисленное значение, такое как `int` или `char`. Если, например, критерием выбора является строка или вещественное число, то команда `switch` не подойдет. Придется использовать серию команд `if-else`. В конце следующей главы будет показано, что одно из нововведений Java SE5 — *перечисления* — помогает снять эти ограничения, так как перечисления удобно использовать со `switch`.

Следующий пример случайным образом генерирует английские буквы. Программа определяет, гласные они или согласные:

```
//: control/VowelsAndConsonants.java
// Демонстрация конструкции switch.
import java.util.*;
import static net.mindview.util.Print.*;

public class VowelsAndConsonants {
    public static void main(String[] args) {
        Random rand = new Random(47);
```

```

for(int i = 0; i < 100; i++) {
    int c = rand.nextInt(26) + 'a';
    println((char)c + ", " + c + ": ");
    switch(c) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u': print("гласная");
                break;
        case 'y':
        case 'w': print("Условно гласная");
                break;
        default: print("согласная");
    }
}
} /* Output:
y, 121: Условно гласная
n, 110: согласная
z, 122: согласная
b, 98: согласная
r, 114: согласная
n, 110: согласная
y, 121: Условно гласная
g, 103: согласная
c, 99: согласная
f, 102: согласная
o, 111: гласная
w, 119: Условно гласная
z, 122: согласная
...
*///:~

```

Так как метод `Random.nextInt(26)` генерирует значение между 0 и 26, для получения символа нижнего регистра остается прибавить смещение `'a'`. Символы в апострофах в секциях `case` также представляют собой целочисленные значения, используемые для сравнения.

Обратите внимание на «стопки» секций `case`, обеспечивающие возможность множественного сравнения для одной части кода. Будьте начеку и не забывайте добавлять команду `break` после каждой секции `case`, иначе программа просто перейдет к выполнению следующей секции `case`.

В команде

```
int c = rand.nextInt(26) + 'a';
```

метод `rand.nextInt()` выдает случайное число `int` от 0 до 25, к которому затем прибавляется значение `'a'`. Это означает, что символ `a` автоматически преобразуется к типу `int` для выполнения сложения.

Чтобы вывести `c` в символьном виде, его необходимо преобразовать к типу `char`; в противном случае значение будет выведено в числовом виде.

- 22.** (2) Создайте команду `switch`, которая выводит сообщение в каждой секции `case`. Разместите ее в цикле `for`, проверяющем все допустимые значения `case`. Каждая

секция `case` должна завершаться командой `break`. Затем удалите команды `break` и посмотрите, что произойдет.

- 23.** (4) *Числами Фибоначчи* называется числовая последовательность 1, 1, 2, 3, 5, 8, 13, 21, 34 и т. д., в которой каждое число, начиная с третьего, является суммой двух предыдущих. Напишите метод, который получает целочисленный аргумент и выводит указанное количество чисел Фибоначчи. Например, при запуске командой `java Fibonacci 5` (где `Fibonacci` — имя класса) должна выводиться последовательность 1, 1, 2, 3, 5.
- 24.** *Вампирами* называются числа, состоящие из четного количества цифр и полученные перемножением пары чисел, каждое из которых содержит половину цифр результата. Цифры берутся из исходного числа в произвольном порядке, завершающие нули недопустимы. Примеры:
- 1) $1261 = 21 * 60$;
 - 2) $1827 = 21 * 87$;
 - 3) $2187 = 27 * 81$.

Напишите программу, которая находит всех «вампиров», состоящих из 4 цифр. (Задача предложена Дэном Форханом.)

Резюме

В этой главе завершается описание основных конструкций, присутствующих почти во всех языках программирования: вычислений, приоритета операторов, приведения типов, условных конструкций и циклов. Теперь можно сделать следующий шаг на пути к миру объектно-ориентированного программирования. Следующая глава ответит на важные вопросы об инициализации объектов и завершении их жизненного цикла, после чего мы перейдем к важнейшей концепции сокрытия реализации.

5

Инициализация и завершение

В ходе компьютерной революции выяснилось, что основной причиной чрезмерных затрат в программировании является «небезопасное» программирование.

Основные проблемы с безопасностью относятся к *инициализации* и *завершению*. Очень многие ошибки при программировании на языке C обусловлены неверной инициализацией переменных. Это особенно часто происходит при работе с библиотеками, когда пользователи не знают, как нужно инициализировать компонент библиотеки, или забывают это сделать. Завершение — очень актуальная проблема; слишком легко забыть об элементе, когда вы закончили с ним работу и его дальнейшая судьба вас не волнует. В этом случае ресурсы, занимаемые элементом, не освобождаются, и в программе может возникнуть нехватка ресурсов (прежде всего памяти).

В C++ появилось понятие *конструктора* — специального метода, который вызывается при создании нового объекта. Конструкторы используются и в Java; к тому же в Java есть уборщик мусора, который автоматически освобождает ресурсы, когда объект перестает использоваться. В этой главе рассматриваются вопросы инициализации и завершения, а также их поддержка в Java.

Конструктор гарантирует инициализацию

Конечно, можно создать особый метод, назвать его `initialize()` и включить во все ваши классы. Имя метода подсказывает пользователю, что он должен вызвать этот метод, прежде чем работать с объектом. К сожалению, это означает, что пользователь должен постоянно помнить о необходимости вызова данного метода. В Java разработчик класса может в обязательном порядке выполнить инициализацию каждого объекта при помощи специального метода, называемого *конструктором*. Если у класса имеется конструктор, Java автоматически вызывает его при создании объекта, перед тем как пользователи смогут обратиться к этому объекту. Таким образом, инициализация объекта гарантирована.

Как должен называться конструктор? Здесь есть две тонкости. Во-первых, любое имя, которое вы используете, может быть задействовано при определении членов класса; так

возникает потенциальный конфликт имен. Во-вторых, за вызов конструктора отвечает компилятор, поэтому он всегда должен знать, какой именно метод следует вызвать. Реализация конструктора в C++ кажется наиболее простым и логичным решением, поэтому оно использовано и в Java: имя конструктора совпадает с именем класса. Смысл такого решения очевиден — именно такой метод способен автоматически вызываться при инициализации.

Взглянем на описание простого класса с конструктором:

```
//: initialization/SimpleConstructor.java
// Демонстрация простого конструктора.
import com.bruceeckel.simpletest.*;

class Rock {
    Rock() { // Это и есть конструктор
        System.out.print("Rock ");
    }
}

public class SimpleConstructor {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
            new Rock();
    }
} /* Output:
Rock Rock Rock Rock Rock Rock Rock Rock Rock Rock
*///:~
```

Теперь при создании объекта:

```
new Rock( );
```

выделяется память и вызывается конструктор. Тем самым гарантируется, что объект будет инициализирован, прежде чем программа сможет работать с ним.

Заметьте, что стиль программирования, при котором имена методов начинаются со строчной буквы, к конструкторам не относится, поскольку имя конструктора должно *точно* совпадать с именем класса.

Конструктор, не получающий аргументов, называется *конструктором по умолчанию*. В документации Java обычно используется термин «конструктор без аргументов» (no-arg constructor), но термин «конструктор по умолчанию» употребляется уже много лет, и я предпочитаю использовать его. Подобно любому методу, у конструктора могут быть аргументы, для того чтобы позволить вам указать, *как* создать объект. Предыдущий пример легко изменить так, чтобы конструктору при вызове передавался аргумент:

```
//: initialization/SimpleConstructor2.java
// Конструкторы могут получать аргументы

class Rock2 {
    Rock2(int i) {
        System.out.println("Rock " + i + " ");
    }
}

public class SimpleConstructor2 {
```

```
public static void main(String[] args) {
    for(int i = 0; i < 8; i++)
        new Rock2(i);
}
} /* Output:
Rock 0 Rock 1 Rock 2 Rock 3 Rock 4 Rock 5 Rock 6 Rock 7
*///:~
```

В аргументах конструктора передаются параметры для инициализации объекта. Например, если у класса `Tree` (дерево) имеется конструктор, который получает в качестве аргумента целое число, обозначающее высоту дерева, то объекты `Tree` будут создаваться следующим образом:

```
Tree t = new Tree(12); // 12-метровое дерево
```

Если `Tree(int)` является единственным конструктором класса, то компилятор не позволит создавать объекты `Tree` каким-либо другим способом.

Конструкторы устраняют большой пласт проблем и упрощают чтение кода. В предыдущем фрагменте кода не встречаются явные вызовы метода, подобного `initialize()`, который концептуально отделен от создания. В Java создание и инициализация являются неразделимыми понятиями — одно без другого невозможно.

Конструктор — не совсем обычный метод, так как у него отсутствует возвращаемое значение. Это ощутимо отличается даже от случая с возвратом значения `void`, когда метод ничего не возвращает, но при этом все же можно заставить его вернуть что-нибудь другое. Конструкторы не возвращают никогда и ничего (оператор `new` возвращает ссылку на вновь созданный объект, но сами конструкторы не имеют выходного значения). Если бы у них существовало возвращаемое значение и программа могла бы его выбрать, то компилятору пришлось бы как-то объяснять, что же делать с этим значением.

1. (1) Создайте класс с неинициализированной ссылкой на `String`. Покажите, что Java инициализирует ссылку значением `null`.
2. (2) Создайте класс с полем `String`, инициализируемым в точке определения, и другим полем, инициализируемым конструктором. Чем отличаются эти два подхода?

Перегрузка методов

Одним из важнейших аспектов любого языка программирования является использование имен. Создавая объект, вы фактически присваиваете имя области памяти. Метод — имя для действия. Использование имен при описании системы упрощает ее понимание и модификацию. Работа программиста сродни работе писателя; в обоих случаях задача состоит в том, чтобы донести свою мысль до читателя.

Проблемы возникают при перенесении нюансов человеческого языка в языки программирования. Часто одно и то же слово имеет несколько разных значений — оно *перегружено*. Это полезно, особенно в отношении простых различий. Вы говорите «вымыть посуду», «вымыть машину» и «вымыть собаку». Было бы глупо вместо этого говорить «посудоМыть посуду», «машиноМыть машину» и «собакоМыть собаку» только для того, чтобы слушатель не утруждал себя выявлением разницы между

этим действиями. Большинство человеческих языков несет избыточность, и даже при пропуске некоторых слов уяснить смысл не так сложно. Уникальные имена не обязательны — сказанное можно понять из контекста.

Большинство языков программирования (и в особенности С) требовали использования уникальных имен для каждого метода (которые в этих языках назывались *функциями*). Иначе говоря, программа не могла содержать функцию `print()` для распечатки целых чисел и одноименную функцию для вывода вещественных чисел — каждая функция должна была иметь уникальное имя.

В Java (и в C++) также существует другой фактор, который заставляет использовать перегрузку имен методов: наличие конструкторов. Так как имя конструктора предопределено именем класса, оно может быть только единственным. Но что, если вы захотите создавать объекты разными способами? Допустим, вы создаете класс с двумя вариантами инициализации — либо стандартно, либо на основании из некоторого файла. В этом случае необходимость двух конструкторов очевидна: конструктор по умолчанию и конструктор, получающий в качестве аргумента строку с именем файла. Оба они являются полноценными конструкторами и поэтому должны называться одинаково — именем класса. Здесь *перегрузка методов* (overloading) однозначно необходима, чтобы мы могли использовать методы с одинаковыми именами, но с разными аргументами¹. И хотя перегрузка методов обязательна только для конструкторов, она удобна в принципе и может быть применена к любому методу.

Следующая программа демонстрирует пример перегрузки как конструктора, так и обычного метода:

```
//: initialization/Overloading.java
// Демонстрация перегрузки конструкторов наряду
// с перегрузкой обычных методов.
import static net.mindview.util.Print.*;

class Tree {
    int height;
    Tree() {
        print("Сажаем росток");
        height = 0;
    }
    Tree(int initialHeight) {
        height = initialHeight;
        print("Создание нового дерева высотой " +
            height + " м.");
    }
    void info() {
        print("Дерево высотой " + height + " м.");
    }
    void info(String s) {
        print(s + ": Дерево высотой " + height + " м.");
    }
}
```

¹ Перегрузку (overloading), то есть использование одного идентификатора для ссылки на разные элементы в одной области действия, следует отличать от замещения (overriding) — иной реализации метода в подклассе, первоначально определившего метод класса. — *Примеч. ред.*

```
public class Overloading {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            Tree t = new Tree(i);
            t.info();
            t.info("Перегруженный метод");
        }
        // Перегруженный конструктор:
        new Tree();
    }
} /* Output:
```

Создание нового дерева высотой 0 м.

Дерево высотой 0 м.

Перегруженный метод: Дерево высотой 0 м.

Создание нового дерева высотой 1 м.

Дерево высотой 1 м.

Перегруженный метод: Дерево высотой 1 м.

Создание нового дерева высотой 2 м.

Дерево высотой 2 м.

Перегруженный метод: Дерево высотой 2 м.

Создание нового дерева высотой 3 м.

Дерево высотой 3 м.

Перегруженный метод: Дерево высотой 3 м.

Создание нового дерева высотой 4 м.

Дерево высотой 4 м.

Перегруженный метод: Дерево высотой 4 м.

Сажаем росток

*///:~

Объект `Tree` (дерево) может быть создан или в форме ростка (без аргументов), или в виде «взрослого растения» с некоторой высотой. Для этого в классе определяются два конструктора; один используется по умолчанию, а другой получает аргумент с высотой дерева.

Возможно, вы захотите вызывать метод `info()` несколькими способами. Например, вызов с аргументом-строкой `info(String)` используется при необходимости вывода дополнительной информации, а вызов без аргументов `info()` — когда дополнений к сообщению метода не требуется. Было бы странно давать два разных имени методам, когда их схожесть столь очевидна. К счастью, перегрузка методов позволяет использовать одно и то же имя для обоих методов.

Различение перегруженных методов

Если у методов одинаковые имена, как Java узнает, какой именно из них вызывается? Ответ прост: каждый перегруженный метод должен иметь уникальный список типов аргументов.

Если немного подумать, такой подход оказывается вполне логичным. Как еще различить два одноименных метода, если не по типу аргументов?

Даже разного порядка аргументов достаточно для того, чтобы методы считались разными (хотя описанный далее подход почти не используется, так как он усложняет сопровождение программного кода):

```

//: initialization/OverloadingOrder.java
// Перегрузка, основанная на порядке
// следования аргументов.
import static net.mindview.util.Print.*;

public class OverloadingOrder {
    static void f(String s, int i) {
        print("String: " + s + ", int: " + i);
    }
    static void f(int i, String s) {
        print("int: " + i + ", String: " + s);
    }
    public static void main(String[] args) {
        f("Сначала строка", 11);
        f(99, "Сначала число");
    }
} /* Output:
String: Сначала строка, int: 11
int: 99, String: Сначала число
*///:~

```

Два метода `f()` имеют одинаковые аргументы с разным порядком следования, и это различие позволяет идентифицировать метод.

Перегрузка с примитивами

Простейший тип может быть автоматически приведен от меньшего типа к большему, и это в состоянии привести немалую путаницу в перегрузку. Следующий пример показывает, что происходит при передаче примитивного типа перегруженному методу:

```

//: initialization/PrimitiveOverloading.java
// Повышение примитивных типов и перегрузка.
import static net.mindview.util.Print.*;

public class PrimitiveOverloading {
    void f1(char x) { printnb("f1(char)"); }
    void f1(byte x) { printnb("f1(byte)"); }
    void f1(short x) { printnb("f1(short)"); }
    void f1(int x) { printnb("f1(int)"); }
    void f1(long x) { printnb("f1(long)"); }
    void f1(float x) { printnb("f1(float)"); }
    void f1(double x) { printnb("f1(double)"); }

    void f2(byte x) { printnb("f2(byte)"); }
    void f2(short x) { printnb("f2(short)"); }
    void f2(int x) { printnb("f2(int)"); }
    void f2(long x) { printnb("f2(long)"); }
    void f2(float x) { printnb("f2(float)"); }
    void f2(double x) { printnb("f2(double)"); }

    void f3(short x) { printnb("f3(short)"); }
    void f3(int x) { printnb("f3(int)"); }
    void f3(long x) { printnb("f3(long)"); }
    void f3(float x) { printnb("f3(float)"); }
    void f3(double x) { printnb("f3(double)"); }

    void f4(int x) { printnb("f4(int)"); }
}

```

```
void f4(long x) { printlnb("f4(long)"); }
void f4(float x) { printlnb("f4(float)"); }
void f4(double x) { printlnb("f4(double)"); }

void f5(long x) { printlnb("f5(long)"); }
void f5(float x) { printlnb("f5(float)"); }
void f5(double x) { printlnb("f5(double)"); }

void f6(float x) { printlnb("f6(float)"); }
void f6(double x) { printlnb("f6(double)"); }

void f7(double x) { printlnb("f7(double)"); }

void testConstVal() {
    printlnb("5: ");
    f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5);print();
}
void testChar() {
    char x = 'x';
    printlnb("char: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);print();
}
void testByte() {
    byte x = 0;
    System.out.println("параметр типа byte:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testShort() {
    short x = 0;
    printlnb("short: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);print();
}
void testInt() {
    int x = 0;
    printlnb("int: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);print();
}
void testLong() {
    long x = 0;
    printlnb("long:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);print();
}
void testFloat() {
    float x = 0;
    System.out.println("float:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);print();
}
void testDouble() {
    double x = 0;
    printlnb("double:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);print();
}
public static void main(String[] args) {
    PrimitiveOverloading p =
        new PrimitiveOverloading();
    p.testConstVal();
    p.testChar();
    p.testByte();
}
```

```

    p.testShort();
    p.testInt();
    p.testLong();
    p.testFloat();
    p.testDouble();
}
} /* Output:
5: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
char: f1(char) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
byte: f1(byte) f2(byte) f3(short) f4(int) f5(long) f6(float) f7(double)
short: f1(short) f2(short) f3(short) f4(int) f5(long) f6(float) f7(double)
int: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
long: f1(long) f2(long) f3(long) f4(long) f5(long) f6(float) f7(double)
float: f1(float) f2(float) f3(float) f4(float) f5(float) f6(float) f7(double)
double: f1(double) f2(double) f3(double) f4(double) f5(double) f6(double) f7(double)
*///:~

```

Из результатов работы программы видно, что константа 5 трактуется как `int`, поэтому если есть перегруженный метод, принимающий аргумент типа `int`, то он и используется. Во всех остальных случаях, если имеется тип данных, «меньший», чем требуется для существующего метода, то этот тип данных повышается соответствующим образом. Только тип `char` ведет себя несколько иначе, по той причине, что если метода с параметром `char` нет, этот тип приводится сразу к типу `int`, а не к промежуточным типам `byte` или `short`.

Что же произойдет, если ваш аргумент «*больше*», чем аргумент, требующийся в перегруженном методе? Ответ можно найти в модификации рассмотренной программы:

```

//: initialization/Demotion.java
// Понижение примитивов и перегрузка.
import static net.mindview.util.Print.*;

public class Demotion {
    void f1(char x) { System.out.println("f1(char)"); }
    void f1(byte x) { System.out.println("f1(byte)"); }
    void f1(short x) { System.out.println("f1(short)"); }
    void f1(int x) { System.out.println("f1(int)"); }
    void f1(long x) { System.out.println("f1(long)"); }
    void f1(float x) { System.out.println("f1(float)"); }
    void f1(double x) { System.out.println("f1(double)"); }

    void f2(char x) { System.out.println("f2(char)"); }
    void f2(byte x) { System.out.println("f2(byte)"); }
    void f2(short x) { System.out.println("f2(short)"); }
    void f2(int x) { System.out.println("f2(int)"); }
    void f2(long x) { System.out.println("f2(long)"); }
    void f2(float x) { System.out.println("f2(float)"); }

    void f3(char x) { System.out.println("f3(char)"); }
    void f3(byte x) { System.out.println("f3(byte)"); }
    void f3(short x) { System.out.println("f3(short)"); }
    void f3(int x) { System.out.println("f3(int)"); }
    void f3(long x) { System.out.println("f3(long)"); }

    void f4(char x) { System.out.println("f4(char)"); }
    void f4(byte x) { System.out.println("f4(byte)"); }
    void f4(short x) { System.out.println("f4(short)"); }
    void f4(int x) { System.out.println("f4(int)"); }
}

```



```
void f5(char x) { System.out.println("f5(char)"); }
void f5(byte x) { System.out.println("f5(byte)"); }
void f5(short x) { System.out.println("f5(short)"); }

void f6(char x) { System.out.println("f6(char)"); }
void f6(byte x) { System.out.println("f6(byte)"); }

void f7(char x) { System.out.println("f7(char)"); }

void testDouble() {
    double x = 0;
    print("параметр типа double:");
    f1(x); f2((float)x); f3((long)x); f4((int)x);
    f5((short)x); f6((byte)x); f7((char)x);
}
public static void main(String[] args) {
    Demotion p = new Demotion();
    p.testDouble();
}
} /* Output:
параметр типа double:,
f1(double),
f2(float),
f3(long),
f4(int),
f5(short),
f6(byte),
f7(char)
*///:~
```

Здесь методы требуют сужения типов данных. Если ваш аргумент «шире», необходимо явно *привести* его к нужному типу. В противном случае компилятор выведет сообщение об ошибке.

Перегрузка по возвращаемым значениям

Логично спросить, почему при перегрузке используются только имена классов и списки аргументов? Почему не идентифицировать методы по их возвращаемым значениям? Следующие два метода имеют одинаковые имена и аргументы, но их легко различить:

```
void f() {}
int f() { return 1; }
```

Такой подход прекрасно сработает в ситуации, где компилятор может однозначно выбрать нужную версию метода по контексту, например `int x = f()`. Однако возвращаемое значение при вызове метода может быть проигнорировано; это часто называется *вызовом метода для получения побочного эффекта*, так как метод вызывается не для естественного результата, а для каких-то других целей. Допустим, метод вызывается следующим способом:

```
f();
```

Как здесь Java определит, какая из версий метода `f()` должна выполняться? И поймет ли читатель программы, что происходит при этом вызове? Именно из-за подобных проблем перегруженные методы не разрешается различать по возвращаемым значениям.

Конструкторы по умолчанию

Как упоминалось ранее, конструктором по умолчанию называется конструктор без аргументов, применяемый для создания «типового объекта». Если созданный вами класс не имеет конструктора, компилятор автоматически добавит конструктор по умолчанию. Например:

```
//: initialization/DefaultConstructor.java

class Bird {}

public class DefaultConstructor {
    public static void main(String[] args) {
        Bird b = new Bird(); // по умолчанию!
    }
} ///:~
```

Строка

```
new Bird();
```

создает новый объект и вызывает конструктор по умолчанию, хотя последний и не был явно определен в классе. Без него не существовало бы метода для построения объекта класса из данного примера. Но если вы уже определили некоторый конструктор (или несколько конструкторов, с аргументами или без), компилятор *не будет* генерировать конструктор по умолчанию:

```
//: initialization/NoSynthesis.java

class Bird2 {
    Bird2(int i) {}
    Bird2(double d) {}
}

public class NoSynthesis {
    public static void main(String[] args) {
        //! Bird2 b = new Bird2(); // Нет конструктора по умолчанию!
        Bird2 b2 = new Bird2(1);
        Bird2 b3 = new Bird2(1.0);
    }
} ///:~
```

Теперь при попытке выполнения `new Bird2()` компилятор заявит, что не может найти конструктор, подходящий по описанию. Получается так: если определения конструкторов отсутствуют, компилятор скажет: «Хотя бы *один* конструктор необходим, позвольте создать его за вас». Если же вы записываете конструктор явно, компилятор говорит: «Вы написали конструктор, а следовательно, знаете, что вам нужно; и если вы создали конструктор по умолчанию, значит, он вам и не нужен».

3. (1) Создайте класс с конструктором по умолчанию (без параметров), который выводит на экран сообщение. Создайте объект этого класса.
4. Добавьте к классу из упражнения 3 перегруженный конструктор, принимающий в качестве параметра строку (`String`) и распечатывающий ее вместе с сообщением.

5. Создайте класс `Dog` (собака) с перегруженным методом `bark()` (лай). Метод должен быть перегружен для разных примитивных типов данных с целью вывода сообщения о лае, завывании, поскуливании и т. п. в зависимости от версии перегруженного метода. Напишите метод `main()`, вызывающий все версии.
6. Измените предыдущее упражнение так, чтобы два перегруженных метода принимали два аргумента (разных типов) и отличались только порядком их следования в списке аргументов. Проверьте, работает ли это.
7. Создайте класс без конструктора. Создайте объект этого класса в методе `main()`, чтобы удостовериться, что конструктор по умолчанию синтезируется автоматически.

Ключевое слово `this`

Если у вас есть два объекта одинакового типа с именами `a` и `b`, вы, возможно, интересуетесь, каким образом производится вызов метода `peel()` для обоих объектов:

```
//: initialization/BananaPeel.java

class Banana { void peel(int i) { /* ... */ } }

public class BananaPeel {
    public static void main(String[] args) {
        Banana a = new Banana(),
            b = new Banana();
        a.peel(1);
        b.peel(2);
    }
} ///:~
```

Если существует только один метод с именем `peel()`, как этот метод узнает, для какого объекта он вызывается — `a` или `b`?

Чтобы программа могла записываться в объектно-ориентированном стиле, основанном на «отправке сообщений объектам», компилятор выполняет для вас некоторую тайную работу. При вызове метода `peel()` передается скрытый первый аргумент — не что иное, как ссылка на используемый объект. Таким образом, вызовы указанного метода на самом деле можно представить как:

```
Banana.peel(a, 1);
Banana.peel(b, 2);
```

Передача дополнительного аргумента относится к внутреннему синтаксису. При попытке явно воспользоваться ею компилятор выдает сообщение об ошибке, но вы примерно представляете суть происходящего.

Предположим, во время выполнения метода вы хотели бы получить ссылку на текущий объект. Так как эта ссылка передается компилятором *скрытно*, идентификатора для нее не существует. Но для решения этой задачи существует ключевое слово — `this`. Ключевое слово `this` может использоваться только внутри не-статического метода и предоставляет ссылку на объект, для которого был вызван метод. Обращаться с ней можно точно так же, как и с любой другой ссылкой на объект. Помните, что при вызове метода вашего класса из другого метода этого класса использовать `this` не нужно;

просто укажите имя метода. Текущая ссылка `this` будет автоматически использована в другом методе. Таким образом, продолжая сказанное:

```
//: initialization/Apricot.java
public class Apricot {
    void pick() { /* ... */ }
    void pit() { pick(); /* ... */ }
} ///:~
```

Внутри метода `pit()` *можно* использовать запись `this.pick()`, но в этом нет необходимости¹. Компилятор сделает это автоматически. Ключевое слово `this` употребляется только в особых случаях, когда вам необходимо явно сослаться на текущий объект. Например, оно часто применяется для возврата ссылки на текущий объект в команде `return`:

```
//: initialization/Leaf.java
// Простой пример использования ключевого слова "this".
```

```
public class Leaf {
    int i = 0;
    Leaf increment() {
        i++;
        return this;
    }
    void print() {
        System.out.println("i = " + i);
    }
    public static void main(String[] args) {
        Leaf x = new Leaf();
        x.increment().increment().increment().print();
    }
} /* Output:
i = 3
*///:~
```

Так как метод `increment()` возвращает ссылку на текущий объект посредством ключевого слова `this`, над одним и тем же объектом легко можно провести множество операций.

Ключевое слово `this` также может пригодиться для передачи текущего объекта другому методу:

```
//: initialization/PassingThis.java

class Person {
    public void eat(Apple apple) {
        Apple peeled = apple.getPeeled();
        System.out.println("Yummy");
    }
}
```

¹ Некоторые демонстративно пишут `this` перед каждым методом и полем класса, объясняя это тем, что «так яснее и доходчивее». Не делайте этого. Мы используем языки высокого уровня по одной причине: они выполняют работу за нас. Если вы станете писать `this` там, где это не обязательно, то запутаете и разозлите любого человека, читающего ваш код, поскольку в большинстве программ ссылки `this` в таком контексте не используются. Последовательный и понятный стиль программирования экономит и время, и деньги.

```

class Peeler {
    static Apple peel(Apple apple) {
        // ... Снимаем кожуру
        return apple; // Очищенное яблоко
    }
}

class Apple {
    Apple getPeeled() { return Peeler.peel(this); }
}

public class PassingThis {
    public static void main(String[] args) {
        new Person().eat(new Apple());
    }
} /* Output:
Yummy
*///:~

```

Класс `Apple` вызывает `Peeler.peel()` — вспомогательный метод, который по какой-то причине должен быть оформлен как внешний по отношению к `Apple` (может быть, он должен обслуживать несколько разных классов, и вы хотите избежать дублирования кода). Для передачи текущего объекта внешнему методу используется ключевое слово `this`.

- 8.** (1) Создайте класс с двумя методами. В первом методе дважды вызовите второй метод: один раз без ключевого слова `this`, а во второй с `this` — просто для того, чтобы убедиться в работоспособности этого синтаксиса; не используйте этот способ вызова на практике.

Вызов конструкторов из конструкторов

Если вы пишете для класса несколько конструкторов, иногда бывает удобно вызвать один конструктор из другого, чтобы избежать дублирования кода. Такая операция проводится с использованием ключевого слова `this`.

Обычно при употреблении `this` подразумевается «этот объект» или «текущий объект», и само слово является ссылкой на текущий объект. В конструкторе ключевое слово `this` имеет другой смысл: при использовании его со списком аргументов вызывается конструктор, соответствующий данному списку. Таким образом, появляется возможность прямого вызова других конструкторов:

```

//: initialization/Flower.java
// Вызов конструкторов с использованием "this"
import static net.mindview.util.Print.*;

public class Flower {
    int petalCount = 0;
    String s = "initial value";
    Flower(int petals) {
        petalCount = petals;
        print("Конструктор с параметром int, petalCount= "
            + petalCount);
    }
    Flower(String ss) {
        print("Конструктор с параметром String, s = " + ss);
    }
}

```

продолжение ➤

```

    s = ss;
}
Flower(String s, int petals) {
    this(petals);
}
//! this(s); // Вызов другого конструктора запрещен!
    this.s = s; // Другое использование "this"
    print("Аргументы String и int");
}
Flower() {
    this("hi", 47);
    print("конструктор по умолчанию (без аргументов)");
}
void printPetalCount() {
//! this(11); // Разрешается только в конструкторах!
    print("petalCount = " + petalCount + " s = " + s);
}
public static void main(String[] args) {
    Flower x = new Flower();
    x.printPetalCount();
}
} /* Output:
Конструктор с параметром int, petalCount= 47
Аргументы String и int
Конструктор по умолчанию (без аргументов)
petalCount = 47 s = hi
*///:~

```

Конструктор `Flower(String s, int petals)` показывает, что при вызове одного конструктора через `this` вызывать второй запрещается. Вдобавок вызов другого конструктора должен быть первой выполняемой операцией, иначе компилятор выдаст сообщение об ошибке.

Пример демонстрирует еще один способ использования `this`. Так как имена аргумента `s` и поля данных класса `s` совпадают, возникает неоднозначность. Разрешить это затруднение можно при помощи конструкции `this.s`, однозначно определяющей поле данных класса. Вы еще не раз встретите такой подход в различных Java-программах, да и в этой книге он практикуется довольно часто.

Метод `printPetalCount()` показывает, что компилятор не разрешает вызывать конструктор из обычного метода; это разрешено только в конструкторах.

9. (1) Подготовьте класс с двумя (перегруженными) конструкторами. Используя ключевое слово `this`, вызовите второй конструктор из первого.

Значение ключевого слова `static`

Ключевое слово `this` поможет лучше понять, что же фактически означает объявление статического (`static`) метода. У таких методов не существует ссылки `this`. Вы не в состоянии вызывать нестатические методы из статических¹ (хотя обратное позволено), и статические методы можно вызывать для имени класса, без каких-либо объектов.

¹ Впрочем, это можно сделать, передав ссылку на объект в статический метод. Тогда по переданной ссылке (которая заменяет `this`) вы сможете вызывать обычные, нестатические, методы и получать доступ к обычным полям. Но для получения такого эффекта проще создать обычный, нестатический, метод.

Статические методы отчасти напоминают глобальные функции языка С, но с некоторыми исключениями: глобальные функции не разрешены в Java, и создание статического метода внутри класса дает ему право на доступ к другим статическим методам и полям.

Некоторые люди утверждают, что статические методы со своей семантикой глобальной функции противоречат объектно-ориентированной парадигме; в случае использования статического метода вы не посылаете сообщение объекту, поскольку отсутствует ссылка `this`. Возможно, что это справедливый упрек, и если вы обнаружите, что используете *слишком много* статических методов, то стоит пересмотреть вашу стратегию разработки программ. Однако ключевое слово `static` полезно на практике, и в некоторых ситуациях они определенно необходимы. Споры же о «чистоте ООП» лучше оставить теоретикам.

Очистка: финализация и уборка мусора

Программисты помнят и знают о важности инициализации, но часто забывают о значимости «приборки». Да и зачем, например, «прибирать» после использования обычной переменной `int`? Но при использовании программных библиотек «просто забыть» об объекте после завершения его работы не всегда безопасно. Конечно, в Java существует уборщик мусора, освобождающий память от ненужных объектов. Но представим себе необычную ситуацию. Предположим, что объект выделяет «специальную» память без использования оператора `new`. Уборщик мусора умеет освобождать память, выделенную `new`, но ему неизвестно, как следует очищать специфическую память объекта. Для таких ситуаций в Java предусмотрен метод `finalize()`, который вы можете определить в вашем классе. Вот как он *должен* работать: когда уборщик мусора готов освободить память, использованную вашим объектом, он для начала вызывает метод `finalize()` и только после этого освобождает занимаемую объектом память. Таким образом, метод `finalize()` позволяет выполнить завершающие действия *во время работы уборщика мусора*.

Все это может создать немало проблем для программистов, особенно для программистов на языке C++, так как они могут спутать метод `finalize()` с *деструктором* языка C++ — функцией, *всегда* вызываемой перед разрушением объекта. Но здесь очень важно понять разницу между Java и C++, поскольку в C++ *объекты уничтожаются всегда* (в правильно написанной программе), в то время как в Java объекты удаляются уборщиком мусора не во всех случаях.

ВНИМАНИЕ

1. Ваши объекты могут быть и не переданы уборщику мусора.
 2. Уборка мусора не является уничтожением.
-

Запомните эту формулу, и многих проблем удастся избежать. Она означает, что если перед тем, как объект станет ненужным, необходимо выполнить некоторое завершающее действие, то это действие вам придется выполнить собственноручно. В Java нет понятия деструктора или сходного с ним, поэтому придется написать обычный метод

для проведения завершающих действий. Предположим, например, что в процессе создания объект рисуется на экране. Если вы вручную не сотрете изображение с экрана, его за вас никто удалять не станет. Поместите действия по стиранию изображения в метод `finalize()`; тогда при удалении объекта уборщиком мусора оно будет выполнено, и рисунок исчезнет, иначе изображение останется.

Может случиться так, что память объекта никогда не будет освобождена, потому что программа даже не приблизится к точке критического расхода ресурсов. Если программа завершает свою работу и уборщик мусора не удалил ни одного объекта и не освободил занимаемую память, то эта память будет возвращена операционной системе *после* завершения работы программы. Это хорошо, так как уборка мусора сопровождается весомыми издержками, и если уборщик не используется, то, соответственно, эти издержки не проявляются.

Для чего нужен метод `finalize()`?

Итак, если метод `finalize()` не стоит использовать для проведения стандартных операций завершения, то для чего же он нужен?

ВНИМАНИЕ

3. Процесс уборки мусора относится только к памяти.

Единственная причина существования уборщика мусора — освобождение памяти, которая перестала использоваться вашей программой. Поэтому все действия, так или иначе связанные с уборкой мусора, особенно те, что записаны в методе `finalize()`, должны относиться к управлению и освобождению памяти.

Но значит ли это, что если ваш объект содержит другие объекты, то в `finalize()` они должны явно удаляться? Нет — уборщик мусора займется освобождением памяти и удалением объектов вне зависимости от способа их создания. Получается, что использование метода `finalize()` ограничено особыми случаями, в которых ваш объект размещается в памяти необычным способом, не связанным с прямым созданием экземпляра. Но если в Java все является объектом, как же тогда такие особые случаи происходят?

Похоже, что поддержка метода `finalize()` была введена в язык, чтобы сделать возможными операции с памятью в стиле C, с привлечением нестандартных механизмов выделения памяти. Это может произойти в основном при использовании методов, предоставляющих способ вызова не-Java-кода из программы на Java. C и C++ пока являются единственными поддерживаемыми языками, но так как для них таких ограничений нет, в действительности программа Java может вызвать любую процедуру или функцию на любом языке. Во внешнем коде можно выделить память вызовом функций C, относящихся к семейству `malloc()`. Если не воспользоваться затем функцией `free()`, произойдет «утечка» памяти. Конечно, функция `free()` тоже принадлежит к C и C++, поэтому придется в методе `finalize()` провести вызов еще одного «внешнего» метода.

После прочтения этого абзаца у вас, скорее всего, сложилось мнение, что метод `finalize()` используется нечасто¹. И правда, это не то место, где следует проводить рутинные операции очистки. Но где же тогда эти обычные операции будут уместны?

Очистка — ваш долг

Для очистки объекта его пользователю нужно вызвать соответствующий метод в той точке, где эти завершающие действия по откреплению и должны осуществляться. Звучит просто, но немного противоречит традиционным представлениям о деструкторах C++. В этом языке все объекты *должны* уничтожаться. Если объект C++ создается локально (то есть в стеке, что невозможно в Java), то удаление и вызов деструктора происходят у закрывающей фигурной скобки, ограничивающей область действия такого объекта. Если же объект создается оператором `new` (как в Java), то деструктор вызывается при выполнении программистом оператора C++ `delete` (не имеющего аналога в Java). А когда программист на C++ забывает вызвать оператор `delete`, деструктор не вызывается и происходит «утечка» памяти, к тому же остальные части объекта не проходят необходимой очистки. Такого рода ошибки очень сложно найти и устранить, и они являются веским доводом в пользу перехода с C++ на Java.

Java не позволяет создавать локальные объекты — все объекты должны быть результатом действия оператора `new`. Но в Java отсутствует аналог оператора `delete`, вызываемого для разрушения объекта, так как уборщик мусора и без того выполнит освобождение памяти. Значит, в несколько упрощенном изложении можно утверждать, что деструктор в Java отсутствует из-за присутствия уборщика мусора. Но в процессе чтения книги вы еще не раз убедитесь, что наличие уборщика мусора не устраняет необходимости в деструкторах или их аналогах. (И никогда не стоит вызывать метод `finalize()` непосредственно, так как этот подход не решает проблему.) Если же потребуются провести какие-то завершающие действия, отличные от освобождения памяти, *все же* придется явно вызвать подходящий метод, выполняющий функцию деструктора C++, но это уже не так удобно, как встроенный деструктор.

Помните, что ни уборка мусора, ни финализация не гарантированы. Если виртуальная машина Java (Java Virtual Machine, JVM) далека от критической точки расходования ресурсов, она не станет тратить время на освобождение памяти с использованием уборки мусора.

Условие «готовности»

В общем, вы не должны полагаться на вызов метода `finalize()` — создавайте отдельные «функции очистки» и вызывайте их явно. Скорее всего, `finalize()` пригодится только в особых ситуациях нестандартного освобождения памяти, с которыми большинство программистов никогда не сталкивается. Тем не менее существует очень интересное

¹ Джошуа Блош в своей книге (в разделе «избегайте финализаторов») высказывается еще решительнее: «Финализаторы непредсказуемы, зачастую опасны и чаще всего не нужны». *Effective Java*, с. 20 (издательство Addison-Wesley, 2011).

применение метода `finalize()`, не зависящее от того, вызывается он каждый раз или нет. Это проверка *условия готовности*¹ объекта.

В той точке, где объект становится ненужным, — там, где он готов к проведению очистки, — этот объект должен находиться в состоянии, когда освобождение закрепленной за ним памяти безопасно. Например, если объект представляет открытый файл, то он должен быть соответствующим образом закрыт, перед тем как его «приберет» уборщик мусора. Если какая-то часть объекта не будет готова к уничтожению, результатом станет ошибка в программе, которую затем очень сложно обнаружить. Ценность `finalize()` в том и состоит, что он позволяет вам обнаружить такие ошибки, даже если и не всегда вызывается. Единожды проведенная финализация явным образом укажет на ошибку, а это все, что вам нужно.

Простой пример использования данного подхода:

```
//: initialization/TerminationCondition.java
// Использование finalize() для выявления объекта,
// не осуществившего необходимой финализации.

class Book {
    boolean checkedOut = false;
    Book(boolean checkOut) {
        checkedOut = checkOut;
    }
    void checkIn() {
        checkedOut = false;
    }
    public void finalize() {
        if(checkedOut)
            System.out.println("Ошибка: checkedOut");
        // Обычно это делается так:
        // Super.finalize(); // Вызов версии базового класса
    }
}

public class TerminationCondition {
    public static void main(String[] args) {
        Book novel = new Book(true);
        // Правильная очистка:
        novel.checkIn();
        // Теряем ссылку, забыли про очистку:
        new Book(true);
        // Принудительная уборка мусора и финализация:
        System.gc();
    }
} /* Output:
Ошибка: checkedOut
* ///:~
```

«Условие готовности» состоит в том, что все объекты `Book` должны быть «сняты с учета» перед предоставлением их в распоряжение уборщика мусора, но в методе `main()` программист ошибся и не отметил один из объектов `Book`. Если бы в методе `finalize()`

¹ Этот термин предложил Билл Веннерс (www.artima.com) во время семинара, который мы проводили с ним вместе.

не было проверки на условие «готовности», такую оплошность было бы очень сложно обнаружить.

Заметьте, что для проведения принудительной финализации был использован метод `System.gc()`. Но даже если бы его не было, с высокой степенью вероятности можно сказать, что «утерянный» объект `Book` рано или поздно будет обнаружен в процессе исполнения программы (в этом случае предполагается, что программе будет выделено столько памяти, сколько нужно, чтобы уборщик мусора приступил к своим обязанностям).

Обычно следует считать, что версия `finalize()` базового класса делает что-то важное, и вызывать ее в синтаксисе `super`, как показано в `Book.finalize()`. В данном примере вызов закомментирован, потому что он требует обработки исключений, а эта тема нами еще не рассматривалась.

10. (2) Создайте класс с методом `finalize()`, который выводит сообщение. В методе `main()` создайте объект вашего класса. Объясните поведение программы.
11. (4) Измените предыдущее упражнение так, чтобы метод `finalize()` обязательно был исполнен.
12. Включите в класс с именем `Tank` (емкость), который можно наполнить и опустошить. Условие «готовности» требует, чтобы он был пуст перед очисткой. Напишите метод `finalize()`, проверяющий это условие. В методе `main()` протестируйте возможные случаи использования вашего класса.

Как работает уборщик мусора

Если ранее вы работали на языке программирования, в котором выделение места для объектов в куче было связано с большими издержками, то вы можете предположить, что и в Java механизм выделения памяти из кучи для всех данных (за исключением примитивов) также обходится слишком дорого. Однако в действительности использование уборщика мусора дает немалый эффект по *ускорению* создания объектов. Сначала это может звучать немного странно — освобождение памяти сказывается на ее выделении, — но именно так работают некоторые JVM, и это значит, что резервирование места для объектов в куче Java не уступает по скорости выделению пространства в стеке в других языках.

Представьте кучу языка C++ в виде лужайки, где каждый объект «застолбил» свой собственный участок. Позднее площадка освобождается для повторного использования. В некоторых виртуальных машинах Java куча выглядит совсем иначе; она скорее похожа на ленту конвейера, которая передвигается вперед при создании нового объекта. А это значит, что скорость выделения хранилища для объекта оказывается весьма высокой. «Указатель кучи» просто передвигается вперед в «невозделанную» территорию, и по эффективности этот процесс близок к выделению памяти в стеке C++. (Конечно, учет выделенного пространства сопряжен с небольшими издержками, но их никоим образом нельзя сравнить с затратами, возникающими при поиске свободного блока в памяти.)

Конечно, использование кучи в режиме «ленты конвейера» не может продолжаться бесконечно, и рано или поздно память станет сильно фрагментирована (что значительно

снижает производительность), а затем и вовсе исчерпается. Как раз здесь в действие вступает уборщик мусора; во время своей работы он компактно размещает объекты кучи, как бы смещая «указатель кучи» ближе к началу «ленты», тем самым предотвращая фрагментацию памяти. Уборщик мусора реструктуризует внутреннее расположение объектов в памяти и позволяет получить высокоскоростную модель кучи для резервирования памяти.

Чтобы понять, как работает уборка мусора в Java, полезно знать, как устроены реализации уборщиков мусора (УМ) в других системах. Простой, но медленный механизм УМ называется *подсчетом ссылок*. С каждым объектом хранится счетчик ссылок на него, и всякий раз при присоединении новой ссылки к объекту этот счетчик увеличивается. Каждый раз при выходе ссылки из области действия или установке ее значения в null счетчик ссылок уменьшается. Таким образом, подсчет ссылок создает небольшие, но постоянные издержки во время работы вашей программы. Уборщик мусора перебирает объект за объектом списка; обнаружив объект с нулевым счетчиком, он освобождает ресурсы, занимаемые этим объектом. Но существует одна проблема — если объекты содержат циклические ссылки друг на друга, их счетчики ссылок не обнуляются, хотя на самом деле объекты уже являются «мусором». Обнаружение таких «циклических» групп является серьезной работой и отнимает у уборщика мусора достаточно времени. Подсчет ссылок часто используется для объяснения принципов процесса уборки мусора, но, судя по всему, он не используется ни в одной из виртуальных машин Java.

В более быстрых схемах уборка мусора не зависит от подсчета ссылок. Вместо этого она опирается на идею, что любой существующий объект прослеживается до ссылки, находящейся в стеке или в статической памяти. Цепочка проверки проходит через несколько уровней объектов. Таким образом, если начать со стека и статического хранилища, мы обязательно доберемся до всех используемых объектов. Для каждой найденной ссылки надо взять объект, на который она указывает, и отследить все ссылки этого объекта; при этом выявляются другие объекты, на которые они указывают, и так далее, пока не будет проверена вся инфраструктура ссылок, берущая начало в стеке и статической памяти. Каждый объект, обнаруженный в ходе поиска, все еще используется в системе. Заметьте, что проблемы циклических ссылок не существует — такие ссылки просто не обнаруживаются и поэтому становятся добычей уборщика мусора автоматически.

В описанном здесь подходе работает *адаптивный* механизм уборки мусора, при котором JVM обращается с найденными используемыми объектами согласно определенному варианту действий. Один из таких вариантов называется *остановить-и-копировать*. Смысл термина понятен: работа программы временно приостанавливается (эта схема не поддерживает уборку мусора в фоновом режиме). Затем все найденные «живые» (используемые) объекты копируются из одной кучи в другую, а «мусор» остается в первой. При копировании объектов в новую кучу они размещаются в виде компактной непрерывной цепочки, высвобождая пространство в куче (и позволяя удовлетворять заказ на новое хранилище простым перемещением указателя).

Конечно, когда объект перемещается из одного места в другое, все ссылки, указывающие на него, должны быть изменены. Ссылки в стеке или в статическом хранилище переопределяются сразу, но могут быть и другие ссылки на этот объект, которые

исправляются позже, во время очередного «прохода». Исправление происходит по мере нахождения ссылок.

Существует два фактора, из-за которых «копирующие уборщики» обладают низкой эффективностью. Во-первых, в системе существует две кучи, и вы «перелопачиваете» память то туда, то сюда, между двумя отдельными кучами, при этом половина памяти тратится впустую. Некоторые JVM пытаются решить эту проблему, выделяя память для кучи небольшими порциями по мере необходимости, а затем просто копируя одну порцию в другую.

Второй вопрос — копирование. Как только программа перейдет в фазу стабильной работы, она обычно либо становится «безотходной», либо производит совсем немного «мусора». Несмотря на это, копирующий уборщик все равно не перестанет копировать память из одного места в другое, что расточительно. Некоторые JVM определяют, что новых «отходов» не появляется, и переключаются на другую схему («адаптивная» часть). Эта схема называется *пометить-и-убрать* (удалить), и именно на ней работали ранние версии виртуальных машин фирмы Sun. Для повсеместного использования вариант «пометить-и-убрать» чересчур медлителен, но когда известно, что нового «мусора» мало или вообще нет, он выполняется быстро.

Схема «пометить-и-убрать» использует ту же логику — проверка начинается со стека и статического хранилища, после чего постепенно обнаруживаются все ссылки на «живые» объекты. Однако каждый раз при нахождении объект помечается флагом, но еще продолжает существование. «Уборка» происходит только после завершения процесса проверки и пометки. Все «мертвые» объекты при этом удаляются. Но копирования не происходит, и если уборщик решит «упаковать» фрагментированную кучу, то делается это перемещением объектов внутри нее.

Идея «остановиться-и-копировать» несовместима с фоновым процессом уборки мусора; в начале уборки программа останавливается. В литературе фирмы Sun можно найти немало заявлений о том, что уборка мусора является фоновым процессом с низким приоритетом, но оказывается, что реализации в таком виде (по крайней мере, в первых реализациях виртуальной машины Sun) в действительности не существует. Вместо этого уборщик мусора от Sun начинал выполнение только при нехватке памяти. Схема «пометить-и-убрать» также требует остановки программы.

Как упоминалось ранее, в описываемой здесь виртуальной машине память выделяется большими блоками. При создании большого объекта ему выделяется собственный блок. Строгая реализация схемы «остановиться-и-копировать» требует, чтобы каждый используемый объект из исходной кучи копировался в новую кучу перед освобождением памяти старой кучи, что сопряжено с большими перемещениями памяти. При работе с блоками памяти УМ использует незанятые блоки для копирования по мере их накопления. У каждого блока имеется *счетчик поколений*, следящий за использованием блока. В обычной ситуации «упаковываются» только те блоки, которые были созданы после последней уборки мусора; для всех остальных блоков значение счетчика увеличивается при создании внешних ссылок. Такой подход годится для стандартной ситуации — создания множества временных объектов с коротким сроком жизни. Периодически производится полная очистка — большие блоки не копируются (только наращиваются их счетчики), но блоки с маленькими объектами копируются

и «упаковываются». Виртуальная машина постоянно следит за эффективностью уборки мусора, и если она становится неэффективной, потому что в программе остались только долгоживущие объекты, переключается на схему «пометить-и-убрать». Аналогично JVM следит за успешностью схемы «пометить-и-убрать», и когда куча становится излишне фрагментированной, УМ переключается обратно к схеме «остановиться-и-копировать». Это и есть *адаптивный* механизм.

Существуют и другие способы ускорения работы в JVM. Наиболее важные — это действия загрузчика и то, что называется компиляцией «на лету» (Just-In-Time, JIT). Компилятор JIT частично или полностью конвертирует программу в «родной» машинный код, благодаря чему последний не нуждается в обработке виртуальной машиной и может выполняться гораздо быстрее. При загрузке класса (обычно это происходит при первом создании объекта этого класса) система находит файл .class, и байт-код из этого файла переносится в память. В этот момент можно просто провести компиляцию JIT для кода класса, но такой подход имеет два недостатка: во-первых, это займет чуть больше времени, что вместе с жизненным циклом программы может серьезно отразиться на производительности. Во-вторых, увеличивается размер исполняемого файла (байт-код занимает гораздо меньше места в сравнении с расширенным кодом JIT), что может привести к подкачке памяти, и это тоже замедлит программу. Альтернативная схема *отложенного вычисления* подразумевает, что код JIT компилируется только тогда, когда это станет необходимо. Иначе говоря, код, который никогда не исполняется, не компилируется JIT. Новая технология Java HotSpot, встроенная в последние версии JDK, делает это похожим образом с применением последовательной оптимизации кода при каждом его выполнении. Таким образом, чем чаще выполняется код, тем быстрее он работает.

Инициализация членов класса

Java иногда нарушает гарантии инициализации переменных перед их использованием. В случае с переменными, определенными локально, в методе эта гарантия предоставляется в форме сообщения об ошибке. Скажем, при попытке использования фрагмента

```
void f() {
    int i;
    i++; // Ошибка - переменная i не инициализирована
}
```

вы получите сообщение об ошибке, указывающее на то, что переменная *i* не была инициализирована. Конечно, компилятор мог бы присваивать таким переменным значения по умолчанию, но данная ситуация больше похожа на ошибку программиста, и подобный подход лишь скрыл бы ее. Заставляя программиста присвоить переменной значение по умолчанию, вы с большой вероятностью предотвращаете ошибку в программе.

Если примитивный тип является полем класса, то и способ обращения с ним несколько иной. Как было показано в главе 2, каждому примитивному полю класса гарантированно присваивается значение по умолчанию. Следующая программа подтверждает этот факт и выводит значения:

```
//: initialization/InitialValues.java
// Вывод начальных значений, присваиваемых по умолчанию.
import static net.mindview.util.print.*;
```

```

public class InitialValues {
    boolean t;
    char c;
    byte b;
    short s;
    int i;
    long l;
    float f;
    double d;
    InitialValues reference;
    void printInitialValues() {
        print("Тип данных      Начальное значение");
        print("boolean        " + t);
        print("char            [" + c + "]");
        print("byte            " + b);
        print("short           " + s);
        print("int             " + i);
        print("long            " + l);
        print("float           " + f);
        print("double          " + d);
        print("reference       " + reference);
    }
    public static void main(String[] args) {
        InitialValues iv = new InitialValues();
        iv.printInitialValues();
        /* Тут возможен следующий вариант:
        new InitialValues().printInitialValues();
        */
    }
} /* Output:
Тип данных      Начальное значение
boolean         false
char            [ ]
byte            0
short           0
int             0
long            0
float           0.0
double          0.0
reference       null
*///:~

```

Присмотритесь — даже если значения явно не указываются, они автоматически инициализируются. (Символьной переменной `char` присваивается нуль-символ, который отображается в виде пробела.) По крайней мере, нет опасности случайного использования неинициализированной переменной.

Если ссылка на объект, определяемая внутри класса, не связывается с новым объектом, то ей автоматически присваивается специальное значение `null` (ключевое слово Java).

Явная инициализация

Что делать, если вам понадобится придать переменной начальное значение? Проще все сделать это прямым присваиванием этой переменной значения в точке ее объявления

в классе. (Заметьте, что в C++ такое действие запрещено, хотя его постоянно пытаются выполнить новички.) В следующем примере полям уже знакомого класса `InitialValues` присвоены начальные значения:

```
//: initialization/InitialValues2.java
// Явное определение начальных значений переменных
```

```
public class InitialValues2 {
    boolean bool = true;
    char ch = 'x';
    byte b = 47;
    short s = 0xff;
    int i = 999;
    long lng = 1;
    float f = 3.14f;
    double d = 3.14159;
} ///:~
```

Аналогичным образом можно инициализировать и не-примитивные типы. Если `Depth` является классом, вы можете добавить переменную и инициализировать ее следующим образом:

```
//: initialization/Measurement.java
class Depth {}
```

```
public class Measurement {
    Depth d = new Depth();
    // ...
} ///:~
```

Если вы попытаетесь использовать ссылку `d`, которой не задано начальное значение, произойдет ошибка времени исполнения, называемая *исключением* (исключения подробно описываются в главе 10).

Начальное значение даже может задаваться вызовом метода:

```
//: initialization/MethodInit.java
public class MethodInit {
    int i = f();
    int f() { return 11; }
} ///:~
```

Конечно, метод может получать аргументы, но в качестве последних не должны использоваться неинициализированные члены класса. Например, так правильно:

```
//: initialization/MethodInit2.java
public class MethodInit2 {
    int i = f();
    int j = g(i);
    int f() { return 11; }
    int g(int n) { return n * 10; }
} ///:~
```

а так нет:

```
//: initialization/MethodInit3.java
public class MethodInit3 {
```



```
///! int j = g(i); // Недопустимая опережающая ссылка
int i = f();
int f() { return 11; }
int g(int n) { return n * 10; }
} ///:~
```

Это одно из мест, где компилятор на полном основании выражает недовольство преждевременной ссылкой, поскольку ошибка связана с порядком инициализации, а не с компиляцией программы.

Описанный подход инициализации очень прост и прямолинеен. У него есть ограничение — все объекты типа `InitialValues` получают одни и те же начальные значения. Иногда вам нужно именно это, но в других ситуациях необходима большая гибкость.

Инициализация конструктором

Для проведения инициализации можно использовать конструктор. Это придает большую гибкость процессу программирования, так как появляется возможность вызова методов и выполнения действия по инициализации прямо во время работы программы. Впрочем, при этом необходимо учитывать еще одно обстоятельство: оно не исключает автоматической инициализации, происходящей перед выполнением конструктора. Например, в следующем фрагменте

```
//: initialization/Counter.java
public class Counter {
    int i;
    Counter() { i = 7; }
    // ...
} ///:~
```

переменной `i` сначала будет присвоено значение 0, а затем уже 7. Это верно для всех примитивных типов и ссылок на объекты, включая те, которым задаются явные значения в точке определения. По этим причинам компилятор не пытается заставить вас инициализировать элементы в конструкторе, или в ином определенном месте, или перед их использованием — инициализация и так гарантирована.

Порядок инициализации

Внутри класса очередность инициализации определяется порядком следования переменных, объявленных в этом классе. Определения переменных могут быть разбросаны по разным определениям методов, но в любом случае переменные инициализируются перед вызовом любого метода — даже конструктора. Например:

```
//: initialization/OrderOfInitialization.java
// Демонстрирует порядок инициализации.
import static net.mindview.util.Print.*;

// При вызове конструктора для создания объекта
// Window выводится сообщение:
class Window {
    Window(int marker) { print("Window(" + marker + ")"); }
}
```

```

class House {
    Window w1 = new Window(1); // Перед конструктором
    House() {
        // Показывает, что выполняется конструктор:
        print("House()");
        w3 = new Window(33); // Повторная инициализация w3
    }
    Window w2 = new Window(2); // После конструктора
    void f() { print("f()"); }
    Window w3 = new Window(3); // В конце
}

public class OrderOfInitialization {
    public static void main(String[] args) {
        House h = new House();
        h.f(); // Показывает, что объект сконструирован
    }
} /* Output:
Window(1)
Window(2)
Window(3)
House()
Window(33)
f()
*///:~

```

В классе `House` определения объектов `Window` намеренно разбросаны, чтобы доказать, что все они инициализируются перед выполнением конструктора или каким-то другим действием. Вдобавок ссылка `w3` заново проходит инициализацию в конструкторе.

Из результатов программы видно, что ссылка `w3` инициализируется дважды: перед вызовом конструктора и во время него. (Первый объект теряется, и со временем его уничтожит уборщик мусора.) Поначалу это может показаться неэффективным, но такой подход гарантирует верную инициализацию — что бы произошло, если бы в классе был определен перегруженный конструктор, который *не инициализировал* бы ссылку `w3`, а она при этом не получала бы значения по умолчанию?

Инициализация статических данных

Данные статических полей всегда существуют в единственном экземпляре, независимо от количества созданных объектов. Ключевое слово `static` не может применяться к локальным переменным, только к полям. Если статическое поле относится к примитивному типу, при отсутствии явной инициализации ему присваивается значение по умолчанию. Если это ссылка на объект, то ей присваивается значение `null`.

Если вы хотите провести инициализацию в месте определения, она выглядит точно так же, как и у нестатических членов класса.

Следующий пример помогает понять, *когда* инициализируется статическая память:

```

//: initialization/StaticInitialization.java
// Указание значений по умолчанию в определении класса.
import static net.mindview.util.Print.*;

class Bowl {

```

```
Bowl(int marker) {
    print("Bowl(" + marker + ")");
}
void f1(int marker) {
    print("f1(" + marker + ")");
}
}

class Table {
    static Bowl bowl1 = new Bowl(1);
    Table() {
        print("Table()");
        bowl2.f1(1);
    }
    void f2(int marker) {
        print("f2(" + marker + ")");
    }
    static Bowl bowl2 = new Bowl(2);
}

class Cupboard {
    Bowl bowl3 = new Bowl(3);
    static Bowl bowl4 = new Bowl(4);
    Cupboard() {
        print("Cupboard()");    bowl4.f1(2);
    }
    void f3(int marker) {
        print("f3(" + marker + ")");
    }
    static Bowl bowl5 = new Bowl(5);
}

public class StaticInitialization {
    public static void main(String[] args) {
        print("Создание нового объекта Cupboard в main()");
        new Cupboard();
        print("Создание нового объекта Cupboard в main()");
        new Cupboard();
        table.f2(1);
        cupboard.f3(1);
    }
    static Table table = new Table();
    static Cupboard cupboard = new Cupboard();
} /* Output:
Bowl(1)
Bowl(2)
Table()
f1(1)
Bowl(4)
Bowl(5)
Bowl(3)
Cupboard()
f1(2)
Создание нового объекта Cupboard в main()
Bowl(3)
Cupboard()
f1(2)
Создание нового объекта Cupboard в main()

```

```

Bow1(3)
Cupboard()
f1(2)
f2(1)
f3(1)
*///:~

```

Класс `Bow1` позволяет проследить за процессом создания классов, классы `Table` и `Cupboard` содержат определения статических объектов `Bow1`. Заметьте, что в классе `Cupboard` нестатическая переменная `Bow1 bow13` создается до статических определений.

Из выходных данных программы видно, что статическая инициализация происходит только в случае необходимости. Если вы не создаете объектов `Table` и никогда не обращаетесь к `Table.bow11` или `Table.bow12`, то соответственно не будет и объектов `static Bow1 bow11` и `static Bow1 bow12`. Они инициализируются только при создании *первого* объекта `Table` (или при первом обращении к статическим данным). После этого статические объекты заново не инициализируются.

Сначала инициализируются статические члены, если они еще не были проинициализированы, и только затем нестатические объекты. Доказательство справедливости этого утверждения легко найти в результате работы программы. Для выполнения `main()` (а это статический метод!) загружается класс `StaticInitialization`; затем инициализируются статические поля `table` и `cupboard`, вследствие чего загружаются *эти* классы. И так как все они содержат статические объекты `Bow1`, загружается класс `Bow1`. Таким образом, все классы программы загружаются до начала `main()`. Впрочем, эта ситуация нетипична, поскольку в рядовой программе не все поля объявляются как статические, как в данном примере.

Неплохо теперь обобщить знания о процессе создания объекта. Для примера возьмем класс с именем `Dog`:

- ❑ Хотя ключевое слово `static` и не используется явно, конструктор в действительности является статическим методом. При создании первого объекта типа `Dog` или при первом вызове статического метода/обращения к статическому полю класса `Dog` интерпретатор Java должен найти класс `Dog.class`. Поиск осуществляется в стандартных каталогах, перечисленных в переменной окружения `CLASSPATH`.
- ❑ После загрузки файла `Dog.class` (с созданием особого объекта `Class`, о котором мы узнаем попозже) производится инициализация статических элементов. Таким образом, инициализация статических членов проводится только один раз, при первой загрузке объекта `Class`.
- ❑ При создании нового объекта конструкцией `new Dog()` для начала выделяется блок памяти, достаточный для хранения объекта `Dog` в куче.
- ❑ Выделенная память заполняется нулями, при этом все примитивные поля объекта `Dog` автоматически инициализируются значениями по умолчанию (ноль для чисел, его эквиваленты для типов `boolean` и `char`, `null` для ссылок).
- ❑ Выполняются все действия по инициализации, происходящие в точке определения полей класса.
- ❑ Выполняются конструкторы. Как вы узнаете из главы 7, на этом этапе выполняется довольно большая часть работы, особенно при использовании наследования.

Явная инициализация статических членов

Язык Java позволяет сгруппировать несколько действий по инициализации объектов `static` в специальной конструкции, называемой *статическим блоком*. Выглядит это примерно так:

```
//: initialization/Spoon.java
public class Spoon {
    static int i;
    static {
        i = 47;
    }
} ///:~
```

Похоже на определение метода, но на самом деле мы видим лишь ключевое слово `static` с последующим блоком кода. Этот код, как и остальная инициализация `static`, выполняется только один раз: при первом создании объекта этого класса *или* при первом обращении к статическим членам этого класса (даже если ни один объект класса не создается). Например:

```
//: initialization/ExplicitStatic.java
// Явная инициализация с использованием конструкции "static".
import static net.mindview.util.Print.*;

class Cup {
    Cup(int marker) {
        print("Cup(" + marker + ")");
    }
    void f(int marker) {
        print("f(" + marker + ")");
    }
}

class Cups {
    static Cup cup1;
    static Cup cup2;
    static {
        cup1 = new Cup(1);
        cup2 = new Cup(2);
    }
    Cups() {
        print("Cups()");
    }
}

public class ExplicitStatic {
    public static void main(String[] args) {
        print("Inside main()");
        Cups.cup1.f(99); // (1)
    }
    // static Cups cups1 = new Cups(); // (2)
    // static Cups cups2 = new Cups(); // (2)
} /* Output:
Inside main()
Cup(1)
Cup(2)
f(99)
*///:~
```

Статические инициализаторы класса `Cups` выполняются либо при обращении к статическому объекту `c1` в строке с пометкой (1), либо если строка (1) закомментирована — в строках (2) после снятия комментариев. Если же и строка (1), и строки (2) закомментированы, `static`-инициализация класса `Cups` никогда не выполнится. Также неважно, будет ли исполнена одна или обе строки (2) программы — `static`-инициализация все равно выполняется только один раз.

13. (1) Проверьте истинность утверждений из предыдущего абзаца.
14. (1) Создайте класс с полем `static String`, инициализируемым в точке определения, и другим полем, инициализируемым в блоке `static`. Добавьте статический метод, который выводит значения полей и демонстрирует, что оба поля инициализируются перед использованием.

Инициализация нестатических данных экземпляра

В Java имеется сходный синтаксис для инициализации нестатических переменных для каждого объекта. Вот пример:

```
//: initialization/Mugs.java
// "Инициализация экземпляра" в Java.
import static net.mindview.util.Print.*;

class Mug {
    Mug(int marker) {
        print("Mug(" + marker + ")");
    }
    void f(int marker) {
        print("f(" + marker + ")");
    }
}

public class Mugs {
    Mug mug1;
    Mug mug2;
    {
        mug1 = new Mug(1);
        mug2 = new Mug(2);
        print("mug1 & mug2 инициализированы");
    }
    Mugs() {
        print("Mugs()");
    }
    Mugs(int i) {
        print("Mugs(int)");
    }
    public static void main(String[] args) {
        print("В методе main()");
        new Mugs();
        print("new Mugs() завершено");
        new Mugs(1);
        print("new Mugs(1) завершено");
    }
} /* Output:
В методе main()
Mug(1)
```

```

Mug(2)
mug1 & mug2 инициализированы
Mugs()
new Mugs() завершено
Mug(1)
Mug(2)
mug1 & mug2 инициализированы
Mugs(int)
new Mugs(1) завершено
*///:~

```

Секция инициализации экземпляра:

```

{
    mug1 = new Mug(1);
    mug2 = new Mug(2);
    print("mug1 & mug2 инициализированы");
}

```

выглядит в точности так же, как и конструкция `static`-инициализации, разве что ключевое слово `static` отсутствует. Такой синтаксис необходим для поддержки инициализации *анонимных внутренних классов* (см. главу 9), но он также гарантирует, что некоторые операции будут выполнены независимо от того, какой именно конструктор был вызван в программе. Из результатов видно, что секция инициализации экземпляра выполняется раньше любых конструкторов.

- 15.** (1) Создайте класс, производный от `String`, инициализируемый в секции инициализации экземпляров.

Инициализация массивов

Массив представляет собой последовательность объектов или примитивов, относящихся к одному типу и обозначаемую одним идентификатором. Массивы определяются и используются с помощью *оператора индексирования* `[]`. Чтобы определить ссылку на массив в программе, вы просто указываете вслед за типом пустые квадратные скобки:

```
int[] a1;
```

Квадратные скобки также могут размещаться после идентификатора, эффект будет точно таким же:

```
int a1[];
```

Это соответствует ожиданиям программистов на `C` и `C++`, привыкших к такому синтаксису. Впрочем, первый стиль, пожалуй, выглядит более логично — он сразу дает понять, что имеется в виду «массив значений типа `int`». Он и будет использоваться в книге.

Компилятор не позволяет указать точный размер массива. Вспомните, что говорилось ранее о ссылках. Все, что у вас сейчас есть, — это ссылка на массив, для которого еще не было выделено памяти. Чтобы резервировать память для массива, необходимо записать некоторое выражение инициализации. Для массивов такое выражение может находиться в любом месте программы, но существует и особая разновидность выражений инициализации, используемая только в точке объявления массива. Эта

специальная инициализация выглядит как набор значений в фигурных скобках. Выделение памяти (эквивалентное действию оператора `new`) в этом случае проводится компилятором. Например:

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

Но зачем тогда вообще нужно определять ссылку на массив без самого массива?

```
int[] a2;
```

Во-первых, в Java можно присвоить один массив другому, записав следующее:

```
a2 = a1;
```

В данном случае вы на самом деле копируете ссылку, как показано в примере:

```
//: initialization/ArrayOfPrimitives.java
// Массивы простейших типов.
import static net.mindview.util.Print.*;

public class ArrayOfPrimitives {
    public static void main(String[] args) {
        int[] a1 = { 1, 2, 3, 4, 5 };
        int[] a2;
        a2 = a1;
        for(int i = 0; i < a2.length; i++)
            a2[i] = a2[i] + 1;
        for(int i = 0; i < a1.length; i++)
            print("a1[" + i + "] = " + a1[i]);
    }
} /* Output:
a1[0] = 2
a1[1] = 3
a1[2] = 4
a1[3] = 5
a1[4] = 6
*///:~
```

Массив `a1` инициализируется набором значений, в то время как массив `a2` — нет; присваивание по ссылке `a2` присваивается позже — в данном случае присваивается другой массив.

Все массивы (как массивы примитивов, так и массивы объектов) содержат поле, которое можно прочитать (но не изменить) для получения количества элементов в массиве. Это поле называется `length`. Так как в массивах Java, C и C++ нумерация элементов начинается с нуля, последнему элементу массива соответствует индекс `length-1`. При выходе за границы массива C и C++ не препятствуют «прогулкам в памяти» программы, что часто приводит к печальным последствиям. Но Java защищает вас от таких проблем — при выходе за рамки массива происходит ошибка времени исполнения (исключение, тема главы 10)¹.

¹ Конечно, проверка каждого массива на соблюдение границ требует времени и дополнительного кода и отключить ее невозможно. Это может снизить быстродействие программы, у которой в критичных (по времени) местах активно используются массивы. Но проектировщики Java решили, что для безопасности Интернета и продуктивности программиста такие издержки себя оправдывают.

А если во время написания программы вы не знаете, сколько элементов вам понадобится в новом массиве? Тогда просто используйте `new` для создания его элементов. В следующем примере `new` работает, хотя в программе создается массив примитивных типов (оператор `new` неприменим для создания примитивов вне массива):

```
//: initialization/ArrayNew.java
// Создание массивов оператором new.
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayNew {
    public static void main(String[] args) {
        int[] a;
        Random rand = new Random(47);
        a = new int[rand.nextInt(20)];
        print("Длина a = " + a.length);
        print(Arrays.toString(a));
    }
} /* Output:
Длина a = 18
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
*///:~
```

Размер массива выбирается случайным образом, с использованием метода `Random.nextInt()`, генерирующего число от нуля до переданного в качестве аргумента значения. Так как размер массива случаен, очевидно, что создание массива происходит во время исполнения программы. Вдобавок, результат работы программы позволяет убедиться в том, что элементы массивов простейших типов автоматически инициализируются «пустыми» значениями. (Для чисел и символов это ноль, а для логического типа `boolean` — `false`.)

Метод `Arrays.toString()`, входящий в стандартную библиотеку `java.util`, выдает печатную версию одномерного массива.

Конечно, в данном примере массив можно определить и инициализировать в одной строке:

```
int[] a = new int[rand.nextInt(20)];
```

Если возможно, рекомендуется использовать именно такую форму записи.

При создании массива непримитивных объектов вы фактически создаете массив ссылок. Для примера возьмем класс-«обертку» `Integer`, который является именно классом, а не примитивом:

```
//: initialization/ArrayClassObj.java
// Создание массива непримитивных объектов
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayClassObj {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Integer[] a = new Integer[rand.nextInt(20)];
        print("длина a = " + a.length);
        for(int i = 0; i < a.length; i++)
```

продолжение ➤

```

    a[i] = rand.nextInt(500); // Автоматическая упаковка
    print(Arrays.toString(a));
}
} /* Output: (пример)
длина a = 18
[55, 193, 361, 461, 429, 368, 200, 22, 207, 288, 128, 51, 89, 309, 278, 498, 361, 20]
*///:~

```

Здесь даже после вызова `new` для создания массива

```
Integer[] a = new Integer[rand.nextInt(20)];
```

мы имеем лишь массив из ссылок — до тех пор, пока каждая ссылка не будет инициализирована новым объектом `Integer` (в данном случае это делается посредством автоупаковки):

```
a[i] = rand.nextInt(500);
```

Если вы забудете создать объект, то получите исключение во время выполнения программы при попытке чтения несуществующего элемента массива.

Массивы объектов также можно инициализировать списком в фигурных скобках. Существует две формы синтаксиса:

```

//: initialization/ArrayInit.java
// Инициализация массивов.
import java.util.*;

public class ArrayInit {
    public static void main(String[] args) {
        Integer[] a = {
            new Integer(1),
            new Integer(2),
            3, // Autoboxing
        };
        Integer[] b = new Integer[]{
            new Integer(1),
            new Integer(2),
            3, // Автоматическая упаковка
        };
        System.out.println(Arrays.toString(a));
        System.out.println(Arrays.toString(b));
    }
} /* Output:
[1, 2, 3]
[1, 2, 3]
*///:~

```

В обоих случаях завершающая запятая в списке инициализаторов не обязательна (она всего лишь упрощает ведение длинных списков).

Первая форма полезна, но она более ограничена, поскольку может использоваться только в точке определения массива. Вторая форма может использоваться везде, даже внутри вызова метода. Например, можно создать массив объектов `String` для передачи альтернативных аргументов командной строки методу `main()` другого класса:

```

//: initialization/DynamicArray.java
// Инициализация массива.

```

```
public class DynamicArray {
    public static void main(String[] args) {
        Other.main(new String[]{ "fiddle", "de", "dum" });
    }
}

class Other {
    public static void main(String[] args) {
        for(String s : args)
            System.out.print(s + " ");
    }
} /* Output:
fiddle de dum
*///:~
```

Массив, создаваемый для аргумента `Other.main()`, создается в точке вызова метода, поэтому вы можете предоставить альтернативные аргументы на момент вызова.

16. (1) Создайте массив объектов `String`. Присвойте объект `String` каждому элементу. Выведите содержимое массива в цикле `for`.
17. (2) Создайте класс с конструктором, получающим аргумент `String`. Выведите значение аргумента во время конструирования. Создайте массив ссылок на этот класс, но не создавайте объекты, которыми заполняется массив. Запустите программу и посмотрите, будут ли выводиться сообщения при вызове конструкторов.
18. (1) Завершите предыдущее упражнение — создайте объекты, которыми заполняется массив ссылок.

Списки аргументов переменной длины

Синтаксис второй формы предоставляет удобный синтаксис создания и вызова методов с эффектом, напоминающим *списки аргументов переменной длины* языка C. Такой список способен содержать не известное заранее количество аргументов неизвестного типа. Так как абсолютно все классы являются производными от общего корневого класса `Object`, можно создать метод, принимающий в качестве аргумента массив `Object`, и вызывать его следующим образом:

```
//: initialization/VarArgs.java
// Использование синтаксиса массивов
// для получения переменного списка параметров.

class A { int i; }

public class VarArgs {
    static void printArray(Object[] args) {
        for(Object obj : args)
            System.out.print(obj + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        printArray(new Object[]{
            new Integer(47), new Float(3.14), new Double(11.11)
        });
        printArray(new Object[]{"раз", "два", "три" });
    }
}
```

продолжение ➤

```

    printArray(new Object[]{new A(), new A(), new A()});
}
} /* Output: (Sample)
47 3.14 11.11
раз два три
A@1a46e30 A@3e25a5 A@19821f
*///:~

```

Видно, что метод `print()` принимает массив объектов типа `Object`, перебирает его элементы и выводит их. Классы из стандартной библиотеки Java при печати выводят осмысленную информацию, однако объекты классов в данном примере выводят имя класса, затем символ `@` и несколько шестнадцатеричных цифр. Таким образом, по умолчанию класс выводит имя и адрес объекта (если только вы не переопределите в классе метод `toString()` — см. далее).

До выхода Java SE5 переменные списки аргументов реализовывались именно так. В Java SE5 эта долгожданная возможность наконец-то была добавлена в язык — теперь для определения переменного списка аргументов может использоваться многоточие, как видно в определении метода `printArray`:

```

//: initialization/NewVarArgs.java
// Создание списков аргументов переменной длины
// с использованием синтаксиса массивов.

public class NewVarArgs {
    static void printArray(Object... args) {
        for(Object obj : args)
            System.out.print(obj + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        // Можно передать отдельные элементы:
        printArray(new Integer(47), new Float(3.14),
            new Double(11.11));
        printArray(47, 3.14F, 11.11);
        printArray("раз", "два", "три");
        printArray(new A(), new A(), new A());
        // Или массив:
        printArray((Object[])new Integer[]{ 1, 2, 3, 4 });
        printArray(); // Пустой список тоже возможен
    }
} /* Output: (75% match)
47 3.14 11.11
47 3.14 11.11
раз два три
A@1bab50a A@c3c749 A@150bd4d
1 2 3 4
*///:~

```

Списки аргументов переменной длины избавляют от необходимости явной записи синтаксиса массивов — компилятор делает это автоматически. Вы по-прежнему получаете массив, поэтому `print()` может использовать цикл `foreach` для перебора элементов. Тем не менее происходящее не сводится к автоматическому преобразованию списка элементов в массив. Обратите внимание на предпоследнюю строку программы, в которой массив `Integer` (созданный посредством автоматической упаковки) приводится

к массиву `Object` (для устранения предупреждений от компилятора) и передается `printArray()`. Естественно, компилятор видит, что он уже получил массив, и не выполняет лишних преобразований. Таким образом, если у вас имеется набор элементов, его можно передать в виде списка, а если у вас имеется массив — он тоже будет принят как список аргументов переменной длины.

Последняя строка программы показывает, что список аргументов переменной длины может содержать ноль аргументов. Это может быть полезно при передаче необязательных завершающих аргументов:

```
//: initialization/OptionalTrailingArguments.java
```

```
public class OptionalTrailingArguments {
    static void f(int required, String... trailing) {
        System.out.print("обязательно: " + required + " ");
        for(String s : trailing)
            System.out.print(s + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        f(1, "один");
        f(2, "два", "три");
        f(0);
    }
} /* Output:
обязательно: 1 один
обязательно: 2 два три
обязательно: 0
*///:~
```

Также этот пример показывает, как использовать списки аргументов переменной длины с типом, отличным от `Object`. В данном случае все списки аргументов переменной длины содержат объекты `String`. Допускается использование аргументов любого типа, включая примитивные типы. Следующий пример также показывает, что список аргументов переменной длины превращается в массив, а при отсутствии элементов он становится массивом нулевой длины:

```
//: initialization/VarargType.java
```

```
public class VarargType {
    static void f(Character... args) {
        System.out.print(args.getClass());
        System.out.println(" длина " + args.length);
    }
    static void g(int... args) {
        System.out.print(args.getClass());
        System.out.println(" длина " + args.length);
    }
    public static void main(String[] args) {
        f('a');
        f();
        g(1);
        g();
        System.out.println("int[]: " + new int[0].getClass());
    }
}
```

```

} /* Output:
class [Ljava.lang.Character; длина 1
class [Ljava.lang.Character; длина 0
class [I длина 1
class [I длина 0
int[]: class [I
*///:~

```

Метод `getClass()` является частью `Object`; он будет подробно рассмотрен в главе 14. Этот метод возвращает описание класса объекта, при выводе которого отображается закодированная строка, представляющая тип класса. Начальный символ «[» означает, что это массив с элементами типа, указанного далее. Обозначение «I» соответствует примитивному типу `int`; для дополнительной проверки я создал в последней строке массив `int` и вывел его тип. Проверка подтверждает, что списки аргументов переменной длины не используют автоматическую упаковку, а действительно содержат примитивные типы.

Впрочем, списки аргументов переменной длины могут нормально сочетаться с автоматической упаковкой. Пример:

```

//: initialization/AutoboxingVarargs.java

```

```

public class AutoboxingVarargs {
    public static void f(Integer... args) {
        for(Integer i : args)
            System.out.print(i + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        f(new Integer(1), new Integer(2));
        f(4, 5, 6, 7, 8, 9);
        f(10, new Integer(11), 12);
    }
} /* Output:
1 2
4 5 6 7 8 9
10 11 12
*///:~

```

Обратите внимание на возможность смешения типов в одном списке аргументов, а также на избирательное преобразование аргументов `int` в `Integer` в процессе автоматической упаковки.

Списки аргументов переменной длины усложняют перегрузку, хотя поначалу все выглядит безопасно:

```

//: initialization/OverloadingVarargs.java

```

```

public class OverloadingVarargs {
    static void f(Character... args) {
        System.out.print("first");
        for(Character c : args)
            System.out.print(" " + c);
        System.out.println();
    }
    static void f(Integer... args) {

```

```

    System.out.print("second");
    for(Integer i : args)
        System.out.print(" " + i);
    System.out.println();
}
static void f(Long... args) {
    System.out.println("third");
}
public static void main(String[] args) {
    f('a', 'b', 'c');
    f(1);
    f(2, 1);
    f(0);
    f(0L);
    /// f(); // Не компилируется из-за неоднозначности
}
} /* Output:
first a b c
second 1
second 2 1
second 0
third
*///:~

```

В каждом случае компилятор использует автоматическую упаковку для поиска соответствия и вызывает наиболее точно подходящий перегруженный метод. Но при вызове `f()` без аргументов компилятор не может определить, какой из методов следует вызвать.

Можно попытаться решить проблему, добавляя в один из методов аргумент «постоянной длины»:

```

//: initialization/OverloadingVarargs2.java
// {CompileTimeError} (не компилируется)

public class OverloadingVarargs2 {
    static void f(float i, Character... args) {
        System.out.println("first");
    }
    static void f(Character... args) {
        System.out.print("second");
    }
    public static void main(String[] args) {
        f(1, 'a');
        f('a', 'b');
    }
} ///:~

```

Тег комментария `{CompileTimeError}` исключает файл из построения программой Ant. При попытке откомпилировать его вручную будет выведено сообщение об ошибке:

ссылка на `f` неоднозначна, подходит метод `f(float, java.lang.Character...)` из `OverloadingVarargs2` и метод `f(java.lang.Character...)` из `OverloadingVarargs2`

Но если добавить аргумент «постоянной длины» в *оба* метода, решение заработает:

```

//: initialization/OverloadingVarargs3.java
:
public class OverloadingVarargs3 {

```

продолжение ➤

```

static void f(float i, Character... args) {
    System.out.println("first");
}
static void f(char c, Character... args) {
    System.out.println("second");
}
public static void main(String[] args) {
    f(1, 'a');
    f('a', 'b');
}
} /* Output:
first
second
*///:~

```

Как правило, список аргументов переменной длины следует использовать только в одной версии перегруженного метода (или не использовать его вообще).

19. (2) Напишите метод, получающий список аргументов переменной длины с массивом `String`. Убедитесь в том, что этому методу может передаваться как список объектов `String`, разделенных запятыми, так и `String[]`.
20. (1) Напишите метод `main()`, использующий список аргументов переменной длины вместо обычного синтаксиса. Выведите все элементы полученного массива `args`. Протестируйте программу с разным количеством аргументов командной строки.

Перечисления

Одним из незначительных (на первый взгляд!) нововведений Java SE5 стало ключевое слово `enum`, значительно упрощающее работу программиста при группировке и использовании *перечислимых типов*. В прошлом для этого приходилось создавать набор целочисленных констант, но для таких значений не существует естественных ограничений в рамках указанного набора; таким образом, работа с ними связана с большим риском и сложностями. Перечислимые типы имеют достаточно стандартное применение, поэтому их поддержка всегда присутствовала в C, C++ и ряде других языков. До выхода Java SE5 программист Java, желающий смоделировать функциональность перечислимых типов, должен был много знать и действовать осторожно. Теперь перечисления также поддерживаются в Java, причем они обладают большими возможностями, чем их аналоги из C/C++. Рассмотрим простой пример:

```

//: initialization/Spiciness.java
public enum Spiciness {
    NOT, MILD, MEDIUM, HOT, FLAMING
} ///:~

```

Приведенный фрагмент создает перечислимый тип с именем `Spiciness`, состоящий из пяти именованных значений. Так как экземпляры перечислимых типов являются константами, по правилам они записываются исключительно прописными буквами (если имя состоит из нескольких слов, эти слова разделяются пробелами).

Чтобы использовать перечисление, создайте ссылку на перечислимый тип и присвойте ее переменной:


```
//: initialization/SimpleEnumUse.java
public class SimpleEnumUse {
    public static void main(String[] args) {
        Spiciness howHot = Spiciness.MEDIUM;
        System.out.println(howHot);
    }
} /* Output:
MEDIUM
*///:~
```

При создании перечисления компилятор автоматически наделяет его рядом полезных возможностей. Например, он создает метод `toString()` для простого вывода имени экземпляра перечислимого типа (именно так был получен результат команды `print` в приведенном примере). Компилятор также создает метод `ordinal()` для обозначения порядка объявления констант и статический метод `values()`, возвращающий массив констант `enum` в порядке их объявления:

```
//: initialization/EnumOrder.java
public class EnumOrder {
    public static void main(String[] args) {
        for(Spiciness s : Spiciness.values())
            System.out.println(s + ", ordinal " + s.ordinal());
    }
} /* Output:
NOT, ordinal 0
MILD, ordinal 1
MEDIUM, ordinal 2
HOT, ordinal 3
FLAMING, ordinal 4
*///:~
```

Может показаться, что перечисления составляют новый тип данных, но на самом деле перечисления являются классами и обладают своими методами, так что во многих отношениях с перечислением можно работать как с любым другим классом.

Особенно удобен механизм работы с перечислениями в командах `switch`:

```
//: initialization/Burrito.java

public class Burrito {
    Spiciness degree;
    public Burrito(Spiciness degree) { this.degree = degree; }
    public void describe() {
        System.out.print("This burrito is ");
        switch(degree) {
            case NOT:      System.out.println("not spicy at all.");
                           break;
            case MILD:
            case MEDIUM:  System.out.println("a little hot.");
                           break;
            case HOT:
            case FLAMING:
            default:      System.out.println("maybe too hot.");
        }
    }
    public static void main(String[] args) {
        Burrito
```

```

    plain = new Burrito(Spiciness.NOT),
    greenChile = new Burrito(Spiciness.MEDIUM),
    jalapeno = new Burrito(Spiciness.HOT);
plain.describe();
greenChile.describe();
jalapeno.describe();
}
} /* Output:
This burrito is not spicy at all.
This burrito is a little hot.
This burrito is maybe too hot.
*///:~

```

Команда `switch` проверяет варианты из ограниченного набора, поэтому она идеально сочетается с перечислениями. Обратите внимание на то, насколько четче имена из перечисления объясняют, что должна делать программа в том или ином случае.

Как правило, вы просто используете перечисление как дополнительный способ создания типа данных и работаете с полученными результатами. Собственно, перечисления для этого и создавались, поэтому слишком долго думать о них не нужно. Чтобы безопасно реализовать аналогичную функциональность до введения синтаксиса `enum` в Java SE5 программисту приходилось основательно потрудиться.

Приведенного материала достаточно для понимания и использования перечислений, но мы еще вернемся к ним в книге — перечислениям будет посвящена отдельная глава (см. главу 19).

21. (1) Создайте перечисление с названиями шести типов бумажных денег. Переберите результат `values()` с выводом каждого значения и его `ordinal()`.
22. (2) Напишите команду `switch` для перечисления из предыдущего примера. Для каждого случая выведите расширенное описание конкретной валюты.

Резюме

Такой сложный механизм инициализации, как конструктор, показывает, насколько важное внимание в языке уделяется инициализации. Когда Бьерн Страуструп разрабатывал C++, в первую очередь он обратил внимание на то, что низкая продуктивность C связана с плохо продуманной инициализацией, которой была обусловлена значительная доля ошибок. Аналогичные проблемы возникают и при некорректной финализации. Так как конструкторы позволяют *гарантировать* соответствующие инициализацию и завершающие действия по очистке (компилятор не позволит создать объект без вызова конструктора), тем самым обеспечивается полная управляемость и защищенность программы.

В языке C++ уничтожение объектов играет очень важную роль, потому что объекты, созданные оператором `new`, должны быть соответствующим образом разрушены. В Java память автоматически освобождается уборщиком мусора, и аналоги деструкторов обычно не нужны. В таких случаях уборщик мусора Java значительно упрощает процесс программирования и к тому же добавляет так необходимую безопасность при освобождении ресурсов. Некоторые уборщики мусора могут проводить завершающие

действия даже с такими ресурсами, как графические и файловые дескрипторы. Однако уборщики мусора добавляют издержки во время выполнения программы, которые пока трудно реально оценить из-за сложившейся исторически медлительности интерпретаторов Java. И хотя в последнее время язык Java намного улучшил свою производительность, проблема его «задумчивости» все-таки наложила свой отпечаток на возможность решения языком некоторого класса задач.

Так как для всех объектов гарантированно используются конструкторы, на последние возлагаются дополнительные обязанности, не описанные в этой главе. В частности, гарантия конструирования действует и при создании новых классов с использованием *композиции* или *наследования*, и для их поддержки требуются некоторые дополнения к синтаксису языка. Композиция и наследование, а также их влияние на конструкторы рассматриваются в следующих главах.

6

Управление доступом

Смысл управления доступом (или сокрытия реализации) заключается в постепенном совершенствовании кода.

Каждый хороший писатель — включая и тех, которые пишут программный код, — знает, что работа становится действительно качественной только после того, как она будет переписана, причем порой неоднократно. Если ненадолго оставить распечатку в ящике стола, а потом вернуться к ней, можно увидеть более эффективный способ решения той же задачи. Это одна из основных причин для использования *рефакторинга* — переработки работоспособного кода, направленной на то, чтобы сделать его более удобочитаемым и понятным, а следовательно, упростить его сопровождение¹.

Однако на пути изменения и совершенствования кода лежит одно препятствие. Часто у кода существуют пользователи (*программисты-клиенты*), которые зависят от неизменности некоторого аспекта вашего кода. Итак, вы хотите изменить код, а они хотят, чтобы код оставался неизменным. По этой причине важнейшим фактором объектно-ориентированной разработки является отделение переменной составляющей от постоянной.

Это особенно важно для библиотек. Пользователь библиотеки зависит от используемой части вашего кода и не желает переписывать свой код с выходом новой версии. С другой стороны, создатель библиотеки должен обладать достаточной свободой для проведения изменений и улучшений, но при этом изменения не должны нарушить работоспособность клиентского кода.

Желанная цель может быть достигнута определенными договоренностями. Например, программист библиотеки соглашается не удалять уже существующие методы класса,

¹ См. «Refactoring: Improving the Design of Existing Code», by Martin Fowler, et al. (Addison-Wesley, 1999). Некоторые специалисты возражают против рефакторинга на том основании, что от кода не требуется ничего, кроме его работоспособности, и рефакторинг такого кода является напрасной тратой времени. Проблема в том, что основные затраты времени и средств при работе над проектом связаны не с исходным написанием кода, а с его сопровождением. Более понятный код оборачивается значительной экономией средств.

потому что это может нарушить структуру кода программиста-клиента. В то же время обратная проблема гораздо острее. Например, как создатель библиотеки узнает, какие из полей данных используются программистом-клиентом? Это же относится и к методам, являющимся только частью реализации класса, то есть не предназначенным для прямого использования программистом-клиентом. А если создателю библиотеки понадобится удалить старую реализацию и заменить ее новой? Изменение любого из полей класса может нарушить работу кода программиста-клиента. Выходит, у создателя библиотеки «связаны руки», и он вообще ничего не вправе менять.

Для решения проблемы в Java определены *спецификаторы доступа* (access specifiers), при помощи которых создатель библиотеки указывает, что доступно программисту-клиенту, а что нет. Уровни доступа (от полного до минимального) задаются следующими ключевыми словами: `public`, `protected`, доступ в пределах пакета (не имеет ключевого слова) и `private`. Из предыдущего абзаца может возникнуть впечатление, что создателю библиотеки лучше всего хранить все как можно «секретнее», а открывать только те методы, которые, по вашему мнению, должен использовать программист-клиент. И это абсолютно верно, хотя и выглядит непривычно для людей, чьи программы на других языках (в особенности это касается C) «привыкли» к отсутствию ограничений. К концу этой главы вы наглядно убедитесь в полезности механизма контроля доступа в Java.

Однако концепция библиотеки компонентов и контроля над доступом к этим компонентам — это еще не все. Остается понять, как компоненты связываются в объединенную цельную библиотеку. В Java эта задача решается ключевым словом `package` (пакет), и спецификаторы доступа зависят от того, находятся ли классы в одном или в разных пакетах. Поэтому для начала мы разберемся, как компоненты библиотек размещаются в пакетах. После этого вы сможете в полной мере понять смысл спецификаторов доступа.

Пакет как библиотечный модуль

Пакет содержит группу классов, объединенных в одном *пространстве имен*.

Например, в стандартную поставку Java входит служебная библиотека, оформленная в виде пространства имен `java.util`. Один из классов `java.util` называется `ArrayList`. Чтобы использовать класс в программе, можно использовать его полное имя `java.util.ArrayList`:

```
//: access/FullQualification.java

public class FullQualification {
    public static void main(String[] args) {
        java.util.ArrayList list = new java.util.ArrayList();
    }
} ///:~
```

Впрочем, полные имена слишком громоздки, поэтому в программе удобнее использовать ключевое слово `import`. Если вы собираетесь использовать всего один класс, его можно указать прямо в директиве `import`:

```
//: access/SingleImport.java
import java.util.ArrayList;
```

продолжение ➤

```
public class SingleImport {
    public static void main(String[] args) {
        ArrayList list = new java.util.ArrayList();
    }
} ///:~
```

Теперь к классу `ArrayList` можно обращаться без указания полного имени, но другие классы пакета `java.util` останутся недоступными. Чтобы импортировать все классы, укажите `*` вместо имени класса, как это делается почти во всех примерах книги:

```
import java.util.*;
```

Механизм импортирования обеспечивает возможность управления пространствами имен. Имена членов классов изолируются друг от друга. Метод `f()` класса `A` не конфликтует с методом `f()` с таким же определением (списком аргументов) класса `B`. А как насчет имен классов? Предположим, что класс `Stack` создается на компьютере, где кем-то другим уже был определен класс с именем `Stack`. Потенциальные конфликты имен — основная причина, по которой так важно иметь возможность управлять пространствами имен в Java и создавать уникальные идентификаторы для всех классов.

До настоящего момента большинство примеров книги записывались в отдельных файлах и предназначались для локального использования, поэтому на имена пакетов можно было не обращать внимания. (В таком случае имена классов размещаются в «*пакете по умолчанию*».) Конечно, это тоже решение, и такой подход будет применяться далее где только возможно. Но если вы создаете библиотеку или программу, использующую другие программы Java на этой же машине, стоит подумать о предотвращении конфликтов имен.

Файл с исходным текстом на Java часто называют *компилируемым модулем*. Имя каждого компилируемого модуля должно завершаться суффиксом `.java`, а внутри него может находиться открытый (`public`) класс, имеющий то же имя, что и файл (с заглавной буквы, но без расширения `.java`). Любой компилируемый модуль может содержать не более *одного* открытого класса, иначе компилятор сообщит об ошибке. Остальные классы модуля, если они там есть, скрыты от окружающего мира — они не являются открытыми (`public`) и считаются «вспомогательными» по отношению к главному открытому классу.

Структура кода

В результате компиляции для *каждого* класса, определенного в файле `.java`, создается класс с тем же именем, но с расширением `.class`. Таким образом, при компиляции нескольких файлов `.java` может появиться целый ряд файлов с расширением `.class`. Если вы программировали на компилируемом языке, то, наверное, привыкли к тому, что компилятор генерирует промежуточные файлы (обычно с расширением `OBJ`), которые затем объединяются компоновщиком для получения исполняемого файла или библиотеки. Java работает не так. Рабочая программа представляет собой набор однородных файлов `.class`, которые объединяются в пакет и сжимаются в файл JAR (утилитой Java `jar`). Интерпретатор Java отвечает за поиск, загрузку и интерпретацию¹ этих файлов.

¹ Использовать Java-интерпретатор необязательно. Существует несколько компиляторов, создающих единый исполняемый файл.

Библиотека также является набором файлов с классами. В каждом файле имеется один `public`-класс с любым количеством классов, не имеющих спецификатора `public`. Если вы хотите объявить, что все эти компоненты (хранящиеся в отдельных файлах `.java` и `.class`) связаны друг с другом, воспользуйтесь ключевым словом `package`.

Директива `package` *должна* находиться в первой незакомментированной строке файла. Так, команда

```
package access;
```

означает, что данный компилируемый модуль входит в библиотеку с именем `access`. Иначе говоря, вы указываете, что открытый класс в этом компилируемом модуле принадлежит имени `access`, и если кто-то захочет использовать его, ему придется или полностью записать имя класса, или директиву `import` с `access` (конструкция, указанная выше). Заметьте, что по правилам Java имена пакетов записываются только строчными буквами.

Предположим, файл называется `MyClass.java`. Он может содержать один и только один открытый класс (`public`), причем последний должен называться `MyClass` (с учетом регистра символов):

```
//: access/mypackage/MyClass.java
package access.mypackage;
```

```
public class MyClass {
    // ...
} ///:~
```

Если теперь кто-то захочет использовать `MyClass` или любые другие открытые классы из пакета `access`, ему придется использовать ключевое слово `import`, чтобы имена из `access` стали доступными. Возможен и другой вариант — записать полное имя класса:

```
//: access/QualifiedMyClass.java

public class QualifiedMyClass {
    public static void main(String[] args) {
        access.mypackage.MyClass m =
            new access.mypackage.MyClass();
    }
} ///:~
```

С ключевым словом `import` решение выглядит гораздо аккуратнее:

```
//: access/ImportedMyClass.java
import access.mypackage.*;

public class ImportedMyClass {
    public static void main(String[] args) {
        MyClass m = new MyClass();
    }
} ///:~
```

Ключевые слова `package` и `import` позволяют разработчику библиотеки организовать логическое деление глобального пространства имен, предотвращающее конфликты имен независимо от того, сколько людей подключатся к Интернету и начнут писать свои классы на Java.

Создание уникальных имен пакетов

Вы можете заметить, что поскольку пакет на самом деле никогда не «упаковывается» в единый файл, он может состоять из множества файлов `.class`, что способно привести к беспорядку, может, даже хаосу. Для предотвращения проблемы логично было бы разместить все файлы `.class` конкретного пакета в одном каталоге, то есть воспользоваться иерархической структурой файловой системы. Это первый способ решения проблемы нагромождения файлов в Java; о втором вы узнаете при описании утилиты `jar`.

Размещение файлов пакета в отдельном каталоге решает две другие задачи: создание уникальных имен пакетов и обнаружение классов, потерянных в «дебрях» структуры каталогов. Как было упомянуто в главе 2, проблема решается «кодированием» пути файла `.class` в имени пакета. По общепринятой схеме первая часть имени пакета должна состоять из перевернутого доменного имени разработчика класса. Так как доменные имена Интернета уникальны, соблюдение этого правила обеспечит уникальность имен пакетов и предотвратит конфликты. (Только если ваше доменное имя не достанется кому-то другому, кто начнет писать программы на Java под тем же именем.) Конечно, если у вас нет собственного доменного имени, для создания уникальных имен пакетов придется придумать комбинацию с малой вероятностью повторения (скажем, имя и фамилия). Если же вы решите публиковать свои программы на Java, стоит немного потратиться на получение собственного доменного имени.

Вторая составляющая — преобразование имени пакета в каталог на диске компьютера. Если программе во время исполнения понадобится загрузить файл `.class` (что делается динамически, в точке, где программа создает объект определенного класса, или при запросе доступа к статическим членам класса), она может найти каталог, в котором располагается файл `.class`.

Интерпретатор Java действует по следующей схеме. Сначала он проверяет переменную окружения `CLASSPATH` (ее значение задается операционной системой, а иногда программой установки Java или инструментарием Java). `CLASSPATH` содержит список из одного или нескольких каталогов, используемых в качестве корневых при поиске файлов `.class`. Начиная с этих корневых каталогов интерпретатор берет имя пакета и заменяет точки на слэши для получения полного пути (таким образом, директива `package foo.bar.baz` преобразуется в `foo\bar\baz`, `foo/bar/baz` или что-то еще в зависимости от вашей операционной системы). Затем полученное имя присоединяется к различным элементам `CLASSPATH`. В указанных местах ведется поиск файлов `.class`, имена которых совпадают с именем создаваемого программой класса. (Поиск также ведется в стандартных каталогах, определяемых местонахождением интерпретатора Java.)

Чтобы понять, как работает эта схема, рассмотрим мое доменное имя: `MindView.net`. Обращая его, получаем уникальное глобальное имя для моих классов: `net.mindview`. (Расширения `com`, `edu`, `org` и другие в пакетах Java прежде записывались в верхнем регистре, но начиная с версии Java 2 имена пакетов записываются только строчными буквами.) Если потребуется создать библиотеку с именем `simple`, я получаю следующее имя пакета:

```
package net.mindview.simple;
```


Теперь полученное имя пакета можно использовать в качестве объединяющего пространства имен для следующих двух файлов:

```
//: net/mindview/simple/Vector.java
// Создание пакета.
package net.mindview.simple;

public class Vector {
    public Vector() {
        System.out.println("net.mindview.simple.Vector");
    }
} ///:~
```

Как упоминалось ранее, директива `package` должна находиться в первой строке исходного кода. Второй файл выглядит почти так же:

```
//: net/mindview/simple/List.java
// Создание пакета.
package net.mindview.simple;

public class List {
    public List() {
        System.out.println("net.mindview.simple.List");
    }
} ///:~
```

В моей системе оба файла находятся в следующем подкаталоге:

C:\DOC\JavaT\net\mindview\simple

(Обратите внимание: первый комментарий в каждом файле этой книги задает местонахождение файла в дереве исходного кода — эта информация используется средствами автоматического извлечения кода.)

Если вы посмотрите на файлы, то увидите имя пакета `net.mindview.simple`, но что с первой частью пути? О ней позаботится переменная окружения `CLASSPATH`, которая на моей машине выглядит следующим образом:

CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT

Как видите, `CLASSPATH` может содержать несколько альтернативных путей для поиска.

Однако для файлов JAR используется другой подход. Вы должны записать имя файла JAR в переменной `CLASSPATH`, не ограничиваясь указанием пути к месту его расположения. Таким образом, для файла JAR с именем `grape.jar` переменная окружения должна выглядеть так:

CLASSPATH=.;D:\JAVA\LIB;C:\flavors\grape.jar

После настройки `CLASSPATH` следующий файл можно разместить в любом каталоге:

```
//: access/LibTest.java
// Uses the library.
import net.mindview.simple.*;

public class LibTest {
```

```

public static void main(String[] args) {
    Vector v = new Vector();
    List l = new List();
}
} /* Output:
net.mindview.simple.Vector
net.mindview.simple.List
*///:~

```

Когда компилятор встречает директиву `import` для библиотеки `simple`, он начинает поиск в каталогах, перечисленных в переменной `CLASSPATH`, находит каталог `net/mindview/simple`, а затем переходит к поиску скомпилированных файлов с подходящими именами (`Vector.class` для класса `Vector` и `List.class` для класса `List`). Заметьте, что как классы, так и необходимые методы классов `Vector` и `List` должны быть объявлены со спецификатором `public`.

1. (1) Определите класс в пакете. Создайте экземпляр класса за пределами пакета.

Конфликты имен

Что происходит при импортировании конструкцией `*` двух библиотек, имеющих в своем составе идентичные имена? Предположим, программа содержит следующие директивы:

```

import net.mindview.simple.*;
import java.util.*;

```

Так как пакет `java.util.*` тоже содержит класс с именем `Vector`, это может привести к потенциальному конфликту. Но пока вы не начнете писать код, вызывающий конфликты, все будет в порядке — и это хорошо, поскольку иначе вам бы пришлось тратить лишние усилия на предотвращение конфликтов, которых на самом деле нет.

Конфликт *действительно* произойдет при попытке создать `Vector`:

```
Vector v = new Vector();
```

К какому из классов `Vector` относится эта команда? Этого не знают ни компилятор, ни читатель программы. Поэтому компилятор выдаст сообщение об ошибке и заставит явно указать нужное имя. Например, если мне понадобится стандартный класс Java с именем `Vector`, я должен явно указать этот факт:

```
java.util.Vector v = new java.util.Vector();
```

Данная команда (вместе с переменной окружения `CLASSPATH`) полностью описывает местоположение конкретного класса `Vector`, поэтому директива `import java.util.*` становится избыточной (по крайней мере, если вам не потребуются другие классы из этого пакета).

Также для предотвращения конфликтов можно воспользоваться директивой импортирования отдельного класса — при условии, что конфликтующие имена не будут использоваться в одной программе (в этом случае приходится использовать полную форму определения имен).

2. (2) Преобразуйте фрагменты из этого раздела в программу. Убедитесь в том, что конфликты имен действительно возникают.

Пользовательские библиотеки

Полученные знания позволяют вам создавать собственные библиотеки, сокращающие или полностью исключающие дублирование кода. Для примера можно взять уже знакомый псевдоним для метода `System.out.println()`, сокращающий количество вводимых символов. Его можно включить в класс `Print`:

```
//: net/mindview/util/Print.java
// Методы печати, которые могут использоваться
// без спецификаторов благодаря конструкции
// Java SE5 static import.
package net.mindview.util;
import java.io.*;

public class Print {
    // Печать с переводом строки:
    public static void print(Object obj) {
        System.out.println(obj);
    }
    // Перевод строки:
    public static void print(S) {
        System.out.println();
    }
    // Печать без перевода строки
    public static void printnb(Object obj) {
        System.out.print(obj);
    }
    // Новая конструкция Java SE5 printf() (из языка C):
    public static PrintStream
    printf(String format, Object... args) {
        return System.out.printf(format, args);
    }
} ///:~
```

Новые методы могут использоваться для вывода любых данных с новой строки (`print()`) или в текущей строке (`printnb()`).

Как нетрудно предположить, файл должен располагаться в одном из каталогов, указанных в переменной окружения `CLASSPATH`, по пути `net/mindview`. После компиляции методы `static print()` и `printnb()` могут использоваться где угодно, для чего в программу достаточно включить директиву `import static`:

```
//: access/PrintTest.java
// Использование статических методов печати из Print.java.
import static net.mindview.util.Print.*;

public class PrintTest {
    public static void main(String[] args) {
        print("Теперь это стало возможно!");
        print(100);
        print(100L);
        print(3.14159);
    }
} /* Output:
Теперь это стало возможно!
100
100
3.14159
```

Вторым компонентом этой библиотеки может быть набор методов `range()` (см. главу 4) с возможностью использования синтаксиса `foreach` для простых целочисленных последовательностей:

```
//: net/mindview/util/Range.java
// Array creation methods that can be used without
// qualifiers, using Java SE5 static imports:
package net.mindview.util;

public class Range {
    // Генерирование серии [0..n)
    public static int[] range(int n) {
        int[] result = new int[n];
        for(int i = 0; i < n; i++)
            result[i] = i;
        return result;
    }
    // Генерирование серии [start..end)
    public static int[] range(int start, int end) {
        int sz = end - start;
        int[] result = new int[sz];
        for(int i = 0; i < sz; i++)
            result[i] = start + i;
        return result;
    }
    // Генерирование серии [start..end) с приращением step
    public static int[] range(int start, int end, int step) {
        int sz = (end - start)/step;
        int[] result = new int[sz];
        for(int i = 0; i < sz; i++)
            result[i] = start + (i * step);
        return result;
    }
}
} ///:~
```

Теперь, когда бы вы ни придумали новый интересный инструмент, вы всегда можете добавить его в свою библиотеку. В книге мы еще будем дополнять `net.mindview.util` новыми компонентами.

Использование импортирования для изменения поведения

В Java не поддерживается механизм *условной компиляции* C, который позволяет изменить поведение программы при помощи ключей компилятора, без изменения кода. Вероятно, причина заключается в том, что в C условная компиляция чаще всего применяется для решения проблем межплатформенной совместимости: компилируемая часть кода выбирается в зависимости от целевой платформы. Так как язык Java проектировался как платформенно-универсальный, такая возможность в нем не нужна.

Впрочем, у условной компиляции есть и другие полезные применения. В частности, условная компиляция часто применяется при отладке: отладочный код активизируется в процессе отладки и отключается в окончательной версии. Однако того же результата можно добиться изменением импортируемого пакета. Этот прием может использоваться и в других ситуациях, в которых использовалась условная компиляция.

3. (2) Создайте два пакета `debug` и `debugoff`, содержащие одинаковые классы с методом `debug()`. Первая версия выводит на консоль свой аргумент типа `String`, вторая не делает ничего. Используйте директиву `static import` для импортирования класса в тестовую программу и продемонстрируйте эффект условной компиляции.

Предостережение при работе с пакетами

Помните, что создание пакета всегда неявно сопряжено с определением структуры каталогов. Пакет *обязан* находиться в одноименном каталоге, который, в свою очередь, определяется содержимым переменной `CLASSPATH`. Первые эксперименты с ключевым словом `package` могут оказаться неудачными, пока вы твердо не усвоите правило «имя пакета — его каталог». Иначе компилятор будет выводить множество сообщений о загадочных ошибках выполнения, о невозможности найти класс, который находится рядом в этом же каталоге. Если у вас возникают такие ошибки, попробуйте закомментировать директиву `package`; если все запустится, вы знаете, где искать причины.

Учтите, что исходный и откомпилированный код часто хранится в разных каталогах, но виртуальная машина Java все равно должна иметь возможность найти откомпилированный код через `CLASSPATH`.

Спецификаторы доступа Java

В Java спецификаторы доступа `public`, `protected` и `private` располагаются перед определением членов классов — как полей, так и методов. Каждый спецификатор доступа управляет только одним отдельным определением.

Если спецификатор доступа не указан, используется «пакетный» уровень доступа. Получается, что в любом случае действует та или иная категория доступа. В нескольких ближайших подразделах описаны разные уровни доступа.

Доступ в пределах пакета

Во всех рассмотренных ранее примерах спецификаторы доступа не указывались. Доступ по умолчанию не имеет ключевого слова, но часто его называют доступом в пределах пакета (*package access*, иногда «дружественным»). Это значит, что член класса доступен для всех остальных классов текущего пакета, но для классов за пределами пакета он воспринимается как приватный (`private`). Так как компилируемый модуль — файл — может принадлежать лишь одному пакету, все классы одного компилируемого модуля автоматически открыты друг для друга в границах пакета.

Доступ в пределах пакета позволяет группировать взаимосвязанные классы в одном пакете, чтобы они могли легко взаимодействовать друг с другом. Размещая классы в одном пакете, вы берете код пакета под полный контроль. Таким образом, только принадлежащий вам код будет обладать пакетным доступом к другому, принадлежащему вам же коду — и это вполне логично. Можно сказать, что доступ в пределах пакета и является основной причиной для группировки классов в пакетах. Во многих языках определения в классах организуются совершенно произвольным образом, но в Java придется

привыкать к более жесткой логике структуры. Вдобавок классы, которые не должны иметь доступ к классам текущего пакета, следует просто исключить из этого пакета.

Класс сам определяет, кому разрешен доступ к его членам. Код из другого пакета не может запросто обратиться к пакету и рассчитывать, что ему вдруг станут доступны все члены: `protected`, `private` и доступные в пакете. Получить доступ можно лишь несколькими «законными» способами:

- ❑ Объявить член класса открытым (`public`), то есть доступным для кого угодно и откуда угодно.
- ❑ Сделать член доступным в пакете, не указывая другие спецификаторы доступа, и разместить другие классы в этом же пакете.
- ❑ Как вы увидите в главе 7, где рассказывается о наследовании, производный класс может получить доступ к защищенным (`protected`) членам базового класса, вместе с открытыми членами `public` (но не к приватным членам `private`). Такой класс может пользоваться доступом в пределах пакета только в том случае, если второй класс принадлежит тому же пакету (впрочем, пока на наследование и доступ `protected` можно не обращать внимания).
- ❑ Предоставить «методы доступа», то есть методы для чтения и модификации значения. С точки зрения ООП этот подход является предпочтительным, и именно он используется в технологии JavaBeans (см. главу 22).

public

При использовании ключевого слова `public` вы фактически объявляете, что следующее за ним объявление члена класса доступно для всех, и прежде всего для клиентских программистов, использующих библиотеку. Предположим, вы определили пакет `dessert`, содержащий следующий компилируемый модуль:

```
//: access/dessert/Cookie.java
// Создание библиотеки.
package access.dessert;

public class Cookie {
    public Cookie() {
        System.out.println("Конструктор Cookie");
    }
    void bite() { System.out.println("bite"); }
} ///:~
```

Помните, что файл `Cookie.java` должен располагаться в подкаталоге `dessert` каталога с именем `access` (соответствующем данной главе книги), а последний должен быть включен в переменную `CLASSPATH`. Не стоит полагать, будто Java всегда начинает поиск с текущего каталога. Если вы не укажете символ `.` (точка) в переменной окружения `CLASSPATH` в качестве одного из путей поиска, то Java и не заглянет в текущий каталог.

Если теперь написать программу, использующую класс `Cookie`:

```
//: access/Dinner.java
// Использование библиотеки
import access.dessert.*;
```

```
public class Dinner {
    public static void main(String[] args) {
        Cookie x = new Cookie();
        //! x.bite(); // Обращение невозможно
    }
} /* Output: Конструктор Cookie
*///:~
```

то сможете создать объект `Cookie`, поскольку конструктор этого класса объявлен открытым (`public`), и сам класс также объявлен как `public`. (Понятие открытого класса мы позднее рассмотрим чуть подробнее.) Тем не менее метод `bite()` этого класса недоступен в файле `Dinner.java`, поскольку доступ к нему предоставляется только в пакете `dessert`. Так компилятор предотвращает неправильное использование методов.

Пакет по умолчанию

С другой стороны, следующий код работает, хотя на первый взгляд он вроде бы нарушает правила:

```
//: access/Cake.java
// Обращение к классу из другого компилируемого модуля

class Cake {
    public static void main(String[] args) {
        Pie x = new Pie();
        x.f();
    }
} /* Output:
Pie.f()
*///:~
```

Второй файл в том же каталоге:

```
//: access/Pie.java
// Другой класс

class Pie {
    void f() { System.out.println("Pie.f()"); }
} ///:~
```

Вроде бы эти два файла не имеют ничего общего, и все же в классе `Cake` можно создать объект `Pie` и вызвать его метод `f()`! (Чтобы файлы компилировались, переменная `CLASSPATH` должна содержать символ точки.) Естественно было бы предположить, что класс `Pie` и метод `f()` имеют доступ в пределах пакета и поэтому закрыты для `Cake`. Они действительно обладают доступом в пределах пакета — все верно. Однако их доступность в классе `Cake.java` объясняется тем, что они находятся в одном каталоге и не имеют явно заданного имени пакета. Java по умолчанию включает такие файлы в «пакет по умолчанию» для текущего каталога, поэтому они обладают доступом в пределах пакета к другим файлам в этом каталоге.

private

Ключевое слово `private` означает, что доступ к члену класса не предоставляется никому, кроме методов этого класса. Другие классы того же пакета также не могут обращаться к `private`-членам. На первый взгляд, вы вроде бы изолируете класс даже от самого себя.

С другой стороны, вполне вероятно, что пакет создается целой группой разработчиков; в этом случае `private` позволяет изменять члены класса, не опасаясь, что это отразится на другом классе данного пакета.

Предлагаемый по умолчанию доступ в пределах пакета часто оказывается достаточен для сокрытия данных; напомним, что такой член класса недоступен пользователю пакета. Это удобно, так как обычно используется именно такой уровень доступа (даже в том случае, когда вы просто забудете добавить спецификатор доступа). Таким образом, доступ `public` чаще всего используется тогда, когда вы хотите сделать какие-либо члены класса доступными для программиста-клиента. Может показаться, что спецификатор доступа `private` применяется редко и можно обойтись и без него. Однако разумное применение `private` очень важно, особенно в условиях многопоточного программирования (см. главу 21).

Пример использования `private`:

```
//: access/IceCream.java
// Демонстрация ключевого слова private.
```

```
class Sundae {
    private Sundae() {}
    static Sundae makeASundae() {
        return new Sundae();
    }
}

public class IceCream {
    public static void main(String[] args) {
        //! Sundae x = new Sundae();
        Sundae x = Sundae.makeASundae();
    }
} ///:~
```

Перед вами пример ситуации, в которой `private` может быть очень полезно: предположим, вы хотите контролировать процесс создания объекта, не разрешая посторонним вызывать конкретный конструктор (или любые конструкторы). В данном примере запрещается создавать объекты `Sundae` с помощью конструктора; вместо этого пользователь должен использовать метод `makeASundae()`.

Все «вспомогательные» методы классов стоит объявить как `private`, чтобы предотвратить их случайные вызовы в пакете; тем самым вы фактически запрещаете изменение поведения метода или его удаление.

То же относится и к `private`-полям внутри класса. Если только вы не собираетесь предоставить доступ пользователям к внутренней реализации (а это происходит гораздо реже, чем можно себе представить), объявляйте все поля своих классов со спецификатором `private`.

protected

Чтобы понять смысл спецификатора доступа `protected`, необходимо немного забежать вперед. Сразу скажу, что понимание этого раздела не обязательно до знакомства с наследованием (глава 7). И все же для получения цельного представления здесь приводится описание `protected` и примеры его использования.

Ключевое слово `protected` тесно связано с понятием *наследования*, при котором к уже существующему классу (называемому *базовым классом*) добавляются новые члены, причем исходная реализация остается неизменной. Также можно изменять поведение уже существующих членов класса. Для создания нового класса на базе существующего используется ключевое слово `extends`:

```
class Foo extends Bar {
```

Остальная часть реализации выглядит как обычно.

Если при создании нового пакета используется наследование от класса, находящегося в другом пакете, новый класс получает доступ только к открытым (`public`) членам из исходного пакета. (Конечно, при наследовании в пределах одного пакета можно получить доступ ко всем членам с пакетным уровнем доступа.) Иногда создателю базового класса необходимо предоставить производным классам доступ к конкретному методу, но закрыть его от всех остальных. Именно для этой задачи используется ключевое слово `protected`, определяющее *защищенные* члены класса. Спецификатор `protected` также предоставляет доступ в пределах пакета — то есть члены с этим спецификатором доступны для других классов из того же пакета.

Если вернуться к файлу `Cookie.java`, следующий класс не может вызвать метод `bite()` с доступом уровня пакета:

```
//: access/ChocolateChip.java
// Члены с пакетным доступом недоступны из другого пакета.
import access.dessert.*;

public class ChocolateChip extends Cookie {
    public ChocolateChip() {
        System.out.println("Конструктор ChocolateChip");
    }
    public void chomp() {
        //! bite(); // Метод bite недоступен
    }
    public static void main(String[] args) {
        ChocolateChip x = new ChocolateChip();
        x.chomp();
    }
} /* Output:
Конструктор Cookie
Конструктор ChocolateChip
*///:~
```

Одна из интересных особенностей наследования заключается в том, что если метод `bite()` существует в классе `Cookie`, он также существует в любом классе, производном от `Cookie`. Но поскольку `bite()` обладает пакетным доступом и находится во внешнем пакете, этот метод в данном случае недоступен. Конечно, можно объявить метод `bite()` открытым, но тогда его сможет вызвать кто угодно. Если привести класс `Cookie` к следующему виду:

```
//: access/cookie2/Cookie.java
package access.cookie2;

public class Cookie {
```

продолжение ➤

```

public Cookie() {
    System.out.println("Конструктор Cookie");
}
protected void bite() {
    System.out.println("bite");
}
} ///:~

```

метод `bite()` становится доступным для любого класса, производного от `Cookie`:

```

//: access/ChocolateChip2.java
import access.cookie2.*;

public class ChocolateChip2 extends Cookie {
    public ChocolateChip2() {
        System.out.println("ChocolateChip2 constructor");
    }
    public void chomp() { bite(); } // Защищенный метод
    public static void main(String[] args) {
        ChocolateChip2 x = new ChocolateChip2();
        x.chomp();
    }
} / Output:
Конструктор Cookie
Конструктор ChocolateChip2
bite
*///:~

```

Обратите внимание: хотя метод `bite()` также обладает доступом в пределах пакета, он не объявляется открытым.

4. (2) Покажите, что методы со спецификатором `protected` обладают доступом в пределах пакета, но не являются открытыми.
5. (2) Создайте класс с полями и методами, обладающими разными уровнями доступа: `public`, `private`, `protected`, доступом в пределах пакета. Создайте объект этого класса и посмотрите, какие сообщения выдает компилятор при попытке обращения к разным членам класса. Учтите, что классы, находящиеся в одном каталоге, входят в «пакет по умолчанию».
6. (1) Создайте класс с защищенными данными. Создайте в том же файле второй класс с методом, работающим с защищенными данными из первого класса.

Интерфейс и реализация

Контроль над доступом часто называют *сокрытием реализации*. Помещение данных и методов в классы в комбинации с сокрытием реализации часто называют *инкапсуляцией*¹. В результате появляется тип данных, обладающий характеристиками и поведением.

Доступ к типам данных ограничивается по двум причинам. Первая причина — программист-клиент должен знать, что он может использовать, а что не может. Вы вольны

¹ Впрочем, во многих книгах термином «инкапсуляция» обозначается простое сокрытие информации.

встроить в структуру реализации свои внутренние механизмы, не опасаясь того, что программисты-клиенты по случайности используют их в качестве части интерфейса.

Это подводит нас непосредственно ко второй причине — разделению интерфейса и реализации. Если в программе использована определенная структура, но программисты-клиенты не могут получить доступ к ее членам, кроме отправки сообщений `public`-интерфейсу, вы можете изменять все, что не объявлено как `public` (члены с доступом в пределах пакета, `protected` и `private`), не нарушая работоспособности изменений клиентского кода.

Для большей ясности при написании классов можно использовать такой стиль: сначала записываются открытые члены (`public`), затем следуют защищенные члены (`protected`), потом с доступом в пределах пакета и, наконец, закрытые члены (`private`). Преимущество такой схемы состоит в том, что при чтении исходного текста пользователь сначала видит то, что ему важно (открытые члены, доступ к которым можно получить отовсюду), а затем останавливается при переходе к закрытым членам, являющимся частью внутренней реализации:

```
//: access/OrganizedByAccess.java
```

```
public class OrganizedByAccess {  
    public void pub1() { /* ... */ }  
    public void pub2() { /* ... */ }  
    public void pub3() { /* ... */ }  
    private void priv1() { /* ... */ }  
    private void priv2() { /* ... */ }  
    private void priv3() { /* ... */ }  
    private int i;  
    // ...  
} ///:~
```

Такой подход лишь частично упрощает чтение кода, поскольку интерфейс и реализация все еще совмещены. Иначе говоря, вы все еще видите исходный код — реализацию — так, как он записан прямо в классе. Вдобавок документация в комментариях, создаваемая с помощью `javadoc`, снижает необходимость в чтении исходного текста программистом-клиентом.

Доступ к классам

В Java с помощью спецификаторов доступа возможно также указать, какие из классов *внутри* библиотеки будут доступны для ее пользователей. Если вы хотите, чтобы класс был открыт программисту-клиенту, то добавляете ключевое слово `public` для класса в целом. При этом вы управляете даже самой возможностью создания объектов данного класса программистом-клиентом.

Для управления доступом к классу спецификатор доступа записывается перед ключевым словом `class`:

```
public class Widget {
```

Если ваша библиотека называется, например, `access`, то любой программист-клиент сумеет обратиться извне к классу `Widget`:

```
import access.Widget;
```

или

```
import access.*;
```

Впрочем, при этом действуют некоторые ограничения.

- ❑ В каждом компилируемом модуле может существовать только один открытый (`public`) класс. Идея в том, что каждый компилируемый модуль содержит определенный открытый интерфейс и реализуется этим открытым классом. В модуле может содержаться произвольное количество вспомогательных классов с доступом в пределах пакета. Если в компилируемом модуле определяется более одного открытого класса, компилятор выдаст сообщение об ошибке.
- ❑ Имя открытого класса должно в точности совпадать с именем файла, в котором содержится компилируемый модуль, включая регистр символов. Поэтому для класса `Widget` имя файла должно быть `Widget.java`, но никак не `widget.java` или `WIDGET.java`. В противном случае вы снова получите сообщение об ошибке.
- ❑ Компилируемый модуль может вообще не содержать открытых классов (хотя это и не типично). В этом случае файлу можно присвоить любое имя по вашему усмотрению. С другой стороны, выбор произвольного имени создаст трудности у тех людей, которые будут читать и сопровождать ваш код.

Допустим, в пакете `access` имеется класс, который всего лишь выполняет некоторые служебные операции для класса `Widget` или для любого другого `public`-класса пакета. Конечно, вам не хочется возиться с созданием лишней документации для клиента; возможно, когда-нибудь вы просто измените структуру пакета, уберете этот вспомогательный класс и добавите новую реализацию. Но для этого нужно точно знать, что ни один программист-клиент не зависит от конкретной реализации библиотеки. Для этого вы просто опускаете ключевое слово `public` в определении класса; ведь в таком случае он ограничивается пакетным доступом, то есть может использоваться только в пределах своего пакета.

7. (1) Создайте библиотеку в соответствии с фрагментами кода, содержащими описания `access` и `Widget`. Создайте объект `Widget` в классе, не входящем в пакет `access`.

При создании класса с доступом в пределах пакета его поля все равно рекомендуется помечать как `private` (всегда нужно по максимуму перекрывать доступ к полям класса), но методам стоит давать тот же уровень доступа, что имеет и сам класс (в пределах пакета). Класс с пакетным доступом обычно используется только в своем пакете, и делать методы такого класса открытыми (`public`) стоит только при крайней необходимости, — а о таких случаях вам сообщит компилятор.

Заметьте, что класс нельзя объявить как `private` (что сделает класс недоступным для окружающих, использовать он сможет только «сам себя») или `protected`¹. Поэтому выбор при задании доступа к классу небольшой: в пределах пакета или открытый (`public`).

¹ На самом деле доступ `private` или `protected` могут иметь *внутренние классы*, но это особый случай (см. главу 8).

Если вы хотите перекрыть доступ к классу для всех, объявите все его конструкторы со спецификатором `private`, соответственно запрещая кому бы то ни было создание объектов этого класса. Только вы сами, в статическом методе своего класса, сможете создавать такие объекты. Пример:

```
//: access/Lunch.java
// Спецификаторы доступа для классов.
// Использование конструкторов, объявленных private,
// делает класс недоступным при создании объектов.

class Soup1 {
    private Soup1() {}
    // (1) Разрешаем создание объектов в статическом методе:
    public static Soup1 makeSoup() {
        return new Soup1();
    }
}

class Soup2 {
    private Soup2() {}
    // (2) Создаем один статический объект
    // и по требованию возвращаем ссылку на него.
    private static Soup2 ps1 = new Soup2();
    public static Soup2 access() {
        return ps1;
    }
    public void f() {}
}

// В файле может быть определен только один public-класс:
public class Lunch {
    void testPrivate() {
        // Запрещено, так как конструктор объявлен приватным:
        //! Soup1 soup = new Soup1();
    }
    void testStatic() {
        Soup1 soup = Soup1.makeSoup();
    }
    void testSingleton() {
        Soup2.access().f();
    }
} ///:~
```

До этого момента большинство методов возвращало или `void`, или один из примитивных типов, поэтому определение:

```
public static Soup1 makeSoup() {
    return new Soup1();
}
```

на первый взгляд смотрится немного странно. Слово `Soup1` перед именем метода (`makeSoup`) показывает, что возвращается методом. В предшествующих примерах обычно использовалось обозначение `void`, которое подразумевает, что метод не имеет возвращаемого значения. Однако метод также может возвращать ссылку на объект; в данном случае возвращается ссылка на объект класса `Soup1`.

Классы `Soup1` и `Soup2` наглядно показывают, как предотвратить прямое создание объектов класса, объявляя все его конструкторы со спецификатором `private`. Помните, что без явного определения хотя бы одного конструктора компилятор сгенерирует конструктор по умолчанию (конструктор без аргументов). Определяя конструктор по умолчанию в программе, вы запрещаете его автоматическое создание. Если конструктор объявлен со спецификатором `private`, никто не сможет создавать объекты данного класса. Но как же тогда использовать этот класс? Рассмотренный пример демонстрирует два способа. В классе `Soup1` определяется статический метод, который создает новый объект `Soup1` и возвращает ссылку на него. Это бывает полезно в ситуациях, где вам необходимо провести некоторые операции над объектом перед возвратом ссылки на него или при подсчете общего количества созданных объектов `Soup1` (например, для ограничения их максимального количества).

В классе `Soup2` использован другой подход — в программе всегда создается не более одного объекта этого класса. Объект `Soup2` создается как статическая приватная переменная, поэтому он всегда существует только в одном экземпляре и его невозможно получить без вызова открытого метода `access()`.

8. (4) По образцу примера `Lunch.java` создайте класс с именем `ConnectionManager`, который управляет фиксированным массивом объектов `Connection`. Программист-клиент не должен напрямую создавать объекты `Connection`, а может получать их только с помощью статического метода в классе `ConnectionManager`. Когда запас объектов у класса `ConnectionManager` будет исчерпан, он должен вернуть ссылку `null`. Протестируйте классы в методе `main()`.

9. (2) Поместите следующий файл в каталог `access/local` (предположительно заданный в переменной `CLASSPATH`):

```
// access/local/PackagedClass.java
package access.local;

class PackagedClass {
    public PackagedClass() {
        System.out.println("Создаем класс в пакете");
    }
}
```

Затем сохраните в каталоге, отличном от `access/local`, такой файл:

```
// access/foreign/Foreign.java
package access.foreign;
import access.local.*;

public class Foreign {
    public static void main (String[] args) {
        PackagedClass pc = new PackagedClass();
    }
}
```

Объясните, почему компилятор выдает сообщение об ошибке. Изменит ли что-нибудь помещение класса `Foreign` в пакет `access.local`?

Резюме

В любых отношениях важно установить ограничения, которые соблюдаются всеми сторонами. При создании библиотеки вы устанавливаете отношения с пользователем библиотеки (программистом-клиентом), который создает программы или библиотеки более высокого уровня с использованием ваших библиотек.

Если программисты-клиенты предоставлены сами себе и не ограничены никакими правилами, они могут делать все, что им заблагорассудится, с любыми членами класса — даже теми, доступ к которым вам хотелось бы ограничить. Все детали реализации класса открыты для окружающего мира.

В этой главе рассматривается процесс построения библиотек из классов; во-первых, механизм группировки классов внутри библиотеки и, во-вторых, механизм управления доступом к членам класса.

По оценкам проекты на языке C начинают «рассыпаться» примерно тогда, когда код достигает объема от 50 до 100 Кбайт, так как C имеет единое «пространство имен»; в системе возникают конфликты имен, создающие массу неудобств. В Java ключевое слово `package`, схема именования пакетов и ключевое слово `import` обеспечивают полный контроль над именами, так что конфликта имен можно легко избежать.

Существуют две причины для ограничения доступа к членам класса. Первая — предотвращение использования клиентами внутренней реализации класса, не входящей во внешний интерфейс. Объявление полей и методов со спецификатором `private` только помогает пользователям класса, так как они сразу видят, какие члены класса для них важны, а какие можно игнорировать. Все это упрощает понимание и использование класса.

Вторая, более важная причина для ограничения доступа — возможность изменения внутренней реализации класса, не затрагивающего программистов-клиентов. Например, сначала вы реализуете класс одним способом, а затем выясняется, что реструктуризация кода позволит повысить скорость работы. Отделение интерфейса от реализации позволит сделать это без нарушения работоспособности существующего пользовательского кода, в котором этот класс используется.

Открытый интерфейс класса — это то, что фактически *видит* его пользователь, поэтому очень важно «довести до ума» именно эту, самую важную, часть класса в процессе анализа и разработки. И даже при этом у вас остается относительная свобода действий. Даже если идеальный интерфейс не удалось построить с первого раза, вы можете *добавить* в него новые методы — без удаления уже существующих методов, которые могут использоваться программистами-клиентами.

Управление доступом в основном ориентировано на отношения (и передачу информации) между создателем библиотеки и ее внешними клиентами. Такая ситуация существует далеко не всегда. Например, если вы пишете весь код самостоятельно или работаете в составе небольшой группы, все объявления могут быть объединены в один пакет. В подобных случаях используется иная схема передачи информации, и жесткое управление доступом порой оказывается неэффективным — доступа в пределах пакета (по умолчанию) может быть вполне достаточно.

7

Повторное использование классов

Возможность повторного использования кода принадлежит к числу важнейших преимуществ Java. Впрочем, по-настоящему масштабные изменения отнюдь не сводятся к обычному копированию и правке кода.

Повторное использование на базе копирования кода характерно для процедурных языков, подобных C, но оно работало не очень хорошо. Решение этой проблемы в Java, как и многое другое, строится на концепции класса. Вместо того чтобы создавать новый класс «с чистого листа», вы берете за основу уже существующий класс, который кто-то уже создал и проверил на работоспособность.

Хитрость состоит в том, чтобы использовать классы без ущерба для существующего кода. В этой главе рассматриваются два пути реализации этой идеи. Первый довольно прямолинеен: объекты уже имеющихся классов просто создаются внутри вашего нового класса. Механизм построения нового класса из объектов существующих классов называется *композицией* (composition). Вы просто используете функциональность готового кода, а не его структуру.

Второй способ гораздо интереснее. Новый класс создается *как специализация* уже существующего класса. Взяв существующий класс за основу, вы добавляете к нему свой код без изменения существующего класса. Этот механизм называется *наследованием* (inheritance), и большую часть работы в нем совершает компилятор. Наследование является одним из «краеугольных камней» объектно-ориентированного программирования; некоторые из его дополнительных применений описаны в главе 8.

Синтаксис и поведение типов при использовании композиции и наследования нередко совпадают (что вполне логично, так как оба механизма предназначены для построения новых типов на базе уже существующих). В этой главе рассматриваются оба механизма повторного использования кода.

Синтаксис композиции

До этого момента мы уже довольно часто использовали композицию — ссылка на внедряемый объект просто включается в новый класс. Допустим, вам понадобился объект,

содержащий несколько объектов `String`, пару полей примитивного типа и объект еще одного класса. Для не-примитивных объектов в новый класс включаются ссылки, а примитивы определяются сразу:

```
//: reusing/SprinklerSystem.java
// Композиция для повторного использования кода.

class WaterSource {
    private String s;
    WaterSource() {
        System.out.println("WaterSource()");
        s = "сконструирован";
    }
    public String toString() { return s; }
}

public class SprinklerSystem {
    private String valve1, valve2, valve3, valve4;
    private WaterSource source = new WaterSource();
    private int i;
    private float f;
    public String toString() {
        return
            "valve1 = " + valve1 + " " +
            "valve2 = " + valve2 + " " +
            "valve3 = " + valve3 + " " +
            "valve4 = " + valve4 + "\n" +
            "i = " + i + " " + "f = " + f + " " +
            "source = " + source;
    }
    public static void main(String[] args) {
        SprinklerSystem sprinklers = new SprinklerSystem();
        System.out.println(sprinklers);
    }
} /* Output:
WaterSource()
valve1 = null valve2 = null valve3 = null valve4 = null
i = 0 f = 0.0 source = сконструирован
*///:~
```

В обоих классах определяется особый метод `toString()`. Позже вы узнаете, что каждый не-примитивный объект имеет метод `toString()`, который вызывается в специальных случаях, когда компилятор располагает объектом, а хочет получить его строковое представление в формате `String`. Поэтому в выражении из метода `SprinklerSystem.toString()`:

```
"source = " + source;
```

компилятор видит, что к строке `"source = "` «прибавляется» объект класса `WaterSource`. Компилятор не может это сделать, поскольку к строке можно «добавить» только такую же строку, поэтому он преобразует объект `source` в `String`, вызывая метод `toString()`. После этого компилятор уже в состоянии соединить две строки и передать результат в метод `System.out.println()` (или в статические методы `print()` и `printlnb()`, используемые в книге). Чтобы подобное поведение поддерживалось вашим классом, достаточно включить в него метод `toString()`.

Примитивные типы, определенные в качестве полей класса, автоматически инициализируются нулевыми значениями, как упоминалось в главе 2. Однако ссылки на объекты заполняются значениями `null`, и при попытке вызова метода по такой ссылке произойдет исключение. К счастью, ссылку `null` можно вывести без выдачи исключения.

Компилятор не создает объектов для ссылок «по умолчанию», и это логично, потому что во многих случаях это привело бы к лишним затратам ресурсов. Если вам понадобится проинициализировать ссылку, сделайте это самостоятельно:

- ❑ в точке определения объекта. Это значит, что объект всегда будет инициализироваться перед вызовом конструктора;
- ❑ в конструкторе данного класса;
- ❑ непосредственно перед использованием объекта. Этот способ часто называют *отложенной инициализацией*. Он может сэкономить вам ресурсы в ситуациях, где создавать объект каждый раз необязательно и накладно;
- ❑ с использованием *инициализации экземпляров*.

В следующем примере продемонстрированы все четыре способа.

```
//: reusing/Bath.java
// Инициализация в конструкторе с композицией.
import static net.mindview.util.Print.*;

class Soap {
    private String s;
    Soap() {
        print("Soap()");
        s = "Constructed";
    }
    public String toString() { return s; }
}

public class Bath {
    private String // Инициализация в точке определения:
        s1 = "Счастливый",
        s2 = "Счастливый",
        s3, s4;
    private Soap castille;
    private int i;
    private float toy;
    public Bath() {
        print("В конструкторе Bath()");
        s3 = "Радостный";
        toy = 3.14f;
        castille = new Soap();
    }
    // Инициализация экземпляра:
    { i = 47; }
    public String toString() {
        if(s4 == null) // Отложенная инициализация:
            s4 = "Радостный";
        return
            "s1 = " + s1 + "\n" +
            "s2 = " + s2 + "\n" +
```

```

        "s3 = " + s3 + "\n" +
        "s4 = " + s4 + "\n" +
        "i = " + i + "\n" +
        "toy = " + toy + "\n" +
        "castille = " + castille;
    }
    public static void main(String[] args) {
        Bath b = new Bath();
        print(b);
    }
} /* Output:
В конструкторе Bath()
Soap()
s1 = Счастливый
s2 = Счастливый
s3 = Радостный
s4 - Радостный
i = 47
toy = 3.14
castille = Сконструирован
*///:~

```

Заметьте, что в конструкторе класса `Bath` команда выполняется до проведения какой-либо инициализации. Если инициализация в точке определения не выполняется, нет никаких гарантий того, что она будет выполнена перед отправкой сообщения по ссылке объекта — кроме неизбежных исключений времени выполнения.

При вызове метода `toString()` в нем присваивается значение ссылке `s4`, чтобы все поля были должным образом инициализированы к моменту их использования.

1. (2) Создайте простой класс. Во втором классе определите ссылку на объект первого класса. Используйте отложенную инициализацию для создания экземпляров этого класса.

Синтаксис наследования

Наследование является неотъемлемой частью Java (и любого другого языка ООП). Фактически оно всегда используется при создании класса, потому что даже если класс не объявляется производным от другого класса, он автоматически становится производным от корневого класса `Java Object`.

Синтаксис композиции очевиден, но для наследования существует совершенно другая форма записи. При использовании наследования вы фактически говорите: «Этот новый класс похож на тот старый класс». В программе этот факт выражается перед фигурной скобкой, открывающей тело класса: сначала записывается ключевое слово `extends`, а затем имя *базового* (base) класса. Тем самым вы автоматически получаете доступ ко всем полям и методам базового класса. Пример:

```

//: reusing/Detergent.java
// Синтаксис наследования и его свойства.
import static net.mindview.util.Print.*;

class Cleanser {

```

продолжение ➤

```

private String s = "Cleanser";
public void append(String a) { s += a; }
public void dilute() { append(" dilute()"); }
public void apply() { append(" apply()"); }
public void scrub() { append(" scrub()"); }
public String toString() { return s; }
public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    print(x);
}
}

public class Detergent extends Cleanser {
    // Изменяем метод:
    public void scrub() {
        append(" Detergent.scrub()");
        super.scrub(); // Вызываем метод базового класса
    }
    // Добавляем новые методы к интерфейсу:
    public void foam() { append(" foam()"); }
    // Проверяем новый класс:
    public static void main(String[] args) {
        Detergent x = new Detergent();
        x.dilute();
        x.apply();
        x.scrub();
        x.foam();
        print(x);
        print("Проверяем базовый класс");
        Cleanser.main(args);
    }
} /* Output:
Cleanser dilute() apply() Detergent.scrub() scrub() foam()
Проверяем базовый класс
Cleanser dilute() apply() scrub()
*///:~

```

Пример демонстрирует сразу несколько особенностей наследования. Во-первых, в методе класса `Cleanser` `append()` новые строки присоединяются к строке `s` оператором `+=` — одним из операторов, специально «перегруженных» создателями Java для строк (`String`).

Во-вторых, как `Cleanser`, так и `Detergent` содержат метод `main()`. Вы можете определить метод `main()` в каждом из своих классов; это позволяет встроить тестовый код прямо в класс. Метод `main()` даже не обязательно удалять после завершения тестирования, его вполне можно оставить на будущее.

Даже если у вас в программе имеется множество классов, из командной строки исполняется только один (так как метод `main()` всегда объявляется как `public`, то неважно, объявлен ли класс, в котором он описан, как `public`). В нашем примере команда `java Detergent` вызывает метод `Detergent.main()`. Однако вы также можете использовать команду `java Cleanser` для вызова метода `Cleanser.main()`, хотя класс `Cleanser` не объявлен открытым. Даже если класс обладает доступом в пределах класса, открытый метод `main()` остается доступным.

Здесь метод `Detergent.main()` вызывает `Cleanser.main()` явно, передавая ему собственный массив аргументов командной строки (впрочем, для этого годится любой массив строк).

Важно, что все методы класса `Cleanser` объявлены открытыми. Помните, что при отсутствии спецификатора доступа член класса автоматически получает доступ «в пределах пакета», что позволяет обращаться к нему только из текущего пакета. Таким образом, *в пределах данного пакета* при отсутствии спецификатора доступа вызов этих методов разрешен кому угодно — например, это легко может сделать класс `Detergent`. Но если бы какой-то класс из другого пакета был объявлен производным от класса `Cleanser`, то он получил бы доступ только к его `public`-членам. С учетом возможности наследования все поля обычно помечаются как `private`, а все методы как `public`. (Производный класс также получает доступ к защищенным (`protected`) членам базового класса, но об этом позже.) Конечно, иногда вы будете отступать от этих правил, но, в любом случае, полезно их запомнить.

Класс `Cleanser` содержит ряд методов: `append()`, `dilute()`, `apply()`, `scrub()` и `toString()`. Так как класс `Detergent` *произведен* от класса `Cleanser` (с помощью ключевого слова `extends`), он автоматически получает все эти методы в своем интерфейсе, хотя они и не определяются явно в классе `Detergent`. Таким образом, наследование обеспечивает повторное использование класса.

Как показано на примере метода `scrub()`, разработчик может взять уже существующий метод базового класса и изменить его. Возможно, в этом случае потребуется вызвать метод базового класса из новой версии этого метода. Однако в методе `scrub()` вы не можете просто вызвать `scrub()` — это приведет к рекурсии, а нам нужно не это. Для решения проблемы в Java существует ключевое слово `super`, которое обозначает «суперкласс», то есть класс, производным от которого является текущий класс. Таким образом, выражение `super.scrub()` обращается к методу `scrub()` из базового класса.

При наследовании вы не ограничены использованием методов базового класса. В производный класс можно добавлять новые методы тем же способом, как и раньше, то есть просто определяя их. Метод `foam()` — наглядный пример такого подхода.

В методе `Detergent.main()` для объекта класса `Detergent` вызываются все методы, доступные как из класса `Cleanser`, так и из класса `Detergent` (имеется в виду метод `foam()`).

2. (2) Объявите новый класс, производный от `Detergent`. Переопределите метод `scrub()` и добавьте новый метод с именем `sterilize()`.

Инициализация базового класса

Так как в наследовании участвуют два класса, базовый и производный, не сразу понятно, какой же объект получится в результате. Внешне все выглядит так, словно новый класс имеет тот же интерфейс, что и базовый класс, плюс еще несколько дополнительных методов и полей. Однако наследование не просто копирует интерфейс базового класса. Когда вы создаете объект производного класса, внутри него содержится *подобъект* базового класса. Этот подобъект выглядит точно так же, как выглядел бы созданный обычным порядком объект базового класса. Поэтому извне представляется, будто бы в объекте производного класса «упакован» объект базового класса.

Конечно, очень важно, чтобы подобъект базового класса был правильно инициализирован, и гарантировать это можно только одним способом: выполнить инициализацию в конструкторе, вызывая при этом конструктор базового класса, у которого есть необходимые знания и привилегии для проведения инициализации базового класса. Java автоматически вставляет вызовы конструктора базового класса в конструктор производного класса. В следующем примере задействовано три уровня наследования:

```
//: reusing/Cartoon.java
// Вызовы конструкторов при проведении наследования.
import static net.mindview.util.Print.*;

class Art {
    Art() { print("Конструктор Art"); }
}

class Drawing extends Art {
    Drawing() { print("Конструктор Drawing"); }
}

public class Cartoon extends Drawing {
    public Cartoon() { print("Конструктор Cartoon"); }
    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
} /* Output:
Конструктор Art
Конструктор Drawing
Конструктор Cartoon
* ///:~
```

Как видите, конструирование начинается с «самого внутреннего» базового класса, поэтому базовый класс инициализируется еще до того, как он станет доступным для конструктора производного класса. Даже если конструктор класса `Cartoon` не определен, компилятор сгенерирует конструктор по умолчанию, в котором также вызывается конструктор базового класса.

3. (2) Докажите предыдущее утверждение.
4. (2) Докажите, что конструкторы базового класса: (а) всегда вызываются и (б) всегда вызываются перед исполнением конструкторов производного класса.
5. (1) Создайте два класса `A` и `B`, имеющие конструкторы по умолчанию (с пустым списком аргументов), которые выводят сообщение о вызове. Создайте новый класс `C`, производный от `A`; создайте в `C` поле типа `B`. Не создавайте конструктор `C`. Создайте объект класса `C` и проследите за происходящим.

Конструкторы с аргументами

В предыдущем примере использовались конструкторы по умолчанию, то есть конструкторы без аргументов. У компилятора не возникает проблем с вызовом таких конструкторов, так как вопросов о передаче аргументов не возникает. Если класс не имеет конструктора по умолчанию или вам понадобится вызвать конструктор базового класса с аргументами, этот вызов придется оформить явно, с указанием ключевого слова `super` и передачей аргументов:

```
//: reusing/Chess.java
// Наследование, конструкторы и аргументы.
import static net.mindview.util.Print.*;

class Game {
    Game(int i) {
        print("Конструктор Game");
    }
}

class BoardGame extends Game {
    BoardGame(int i) {
        super(i);
        print("Конструктор BoardGame");
    }
}

public class Chess extends BoardGame {
    Chess() {
        super(11);
        print("Конструктор Chess");
    }
    public static void main(String[] args) {
        Chess x = new Chess();
    }
} /* Output:
Конструктор Game
Конструктор BoardGame
Конструктор Chess
*///:~
```

Если не вызвать конструктор базового класса в `BoardGame()`, то компилятор «пожалуется» на то, что не может обнаружить конструктор в форме `Game()`. Вдобавок, вызов конструктора базового класса должен быть первой командой в конструкторе производного класса. (Если вы вдруг забудете об этом, компилятор вам тут же напомнит.)

6. (1) Используя пример `Chess.java`, докажите утверждения в предыдущем абзаце.
7. (1) Измените упражнение 5 так, чтобы классы `A` и `B` имели конструкторы с аргументами вместо конструкторов по умолчанию. Напишите конструктор для класса `C` и проведите всю необходимую инициализацию внутри него.
8. (1) Создайте базовый класс с единственным конструктором, не являющимся конструктором по умолчанию, и производный класс с конструктором как по умолчанию (без аргументов), так и с аргументами. В конструкторе производного класса вызовите конструктор базового класса.
9. (2) Создайте класс `Root`, содержащий экземпляры каждого из классов (также созданных вами) `Component1`, `Component2` и `Component3`. Объявите класс `Stem` производным от класса `Root`, так чтобы в нем также содержались экземпляры каждого из упомянутых классов `Component`. Каждый класс должен содержать конструктор по умолчанию, который выводит сообщение о своем вызове.
10. (1) Измените предыдущее упражнение так, чтобы во всех классах присутствовали лишь конструкторы с аргументами (не по умолчанию).

Делегирование

Третий вид отношений, не поддерживаемый в Java напрямую, называется *делегированием*. Он занимает промежуточное положение между наследованием и композицией: экземпляр существующего класса включается в создаваемый класс (как при композиции), но в то же время все методы встроенного объекта становятся доступными в новом классе (как при наследовании). Например, класс `SpaceShipControls` имитирует модуль управления космическим кораблем:

```
//: reusing/SpaceShipControls.java
```

```
public class SpaceShipControls {
    void up(int velocity) {}
    void down(int velocity) {}
    void left(int velocity) {}
    void right(int velocity) {}
    void forward(int velocity) {}
    void back(int velocity) {}
    void turboBoost() {}
} ///:~
```

Для построения космического корабля можно воспользоваться наследованием:

```
//: reusing/SpaceShip.java
```

```
public class SpaceShip extends SpaceShipControls {
    private String name;
    public SpaceShip(String name) { this.name = name; }
    public String toString() { return name; }
    public static void main(String[] args) {
        SpaceShip protector = new SpaceShip("NSEA Protector");
        protector.forward(100);
    }
} ///:~
```

Однако космический корабль не может рассматриваться как частный случай своего управляющего модуля — несмотря на то, что ему, к примеру, можно приказывать двигаться вперед (`forward()`). Точнее сказать, что `SpaceShip` *содержит* `SpaceShipControls`, и в то же время все методы последнего предоставляются классом `SpaceShip`. Проблема решается при помощи делегирования:

```
//: reusing/SpaceShipDelegation.java
```

```
public class SpaceShipDelegation {
    private String name;
    private SpaceShipControls controls =
        new SpaceShipControls();
    public SpaceShipDelegation(String name) {
        this.name = name;
    }
    // Делегированные методы:
    public void back(int velocity) {
        controls.back(velocity);
    }
    public void down(int velocity) {
```



```

    controls.down(velocity);
}
public void forward(int velocity) {
    controls.forward(velocity);
}
public void left(int velocity) {
    controls.left(velocity);
}
public void right(int velocity) {
    controls.right(velocity);
}
public void turboBoost() {
    controls.turboBoost();
}
public void up(int velocity) {
    controls.up(velocity);
}
public static void main(String[] args) {
    SpaceShipDelegation protector =
        new SpaceShipDelegation("NSEA Protector");
    protector.forward(100);
}
} ///:~

```

Как видите, вызовы методов переадресуются встроенному объекту `controls`, а интерфейс остается таким же, как и при наследовании. С другой стороны, делегирование позволяет лучше управлять происходящим, потому что вы можете ограничиться небольшим подмножеством методов встроенного объекта.

Хотя делегирование не поддерживается языком Java, его поддержка присутствует во многих средах разработки. Например, приведенный пример был автоматически сгенерирован в JetBrains Idea IDE.

- 11.** (3) Измените пример `Detergent.java` так, чтобы в нем использовалось делегирование.

Сочетание композиции и наследования

Композиция очень часто используется вместе с наследованием. Следующий пример демонстрирует процесс создания более сложного класса с объединением композиции и наследования, с выполнением необходимой инициализации в конструкторе:

```

//: reusing/PlaceSetting.java
// Совмещение композиции и наследования.
import static net.mindview.util.Print.*;

class Plate {
    Plate(int i) {
        print("Конструктор Plate");
    }
}

class DinnerPlate extends Plate {
    DinnerPlate(int i) {
        super(i);
    }
}

```

продолжение ➤

```

        print("Конструктор DinnerPlate");
    }
}

class Utensil {
    Utensil(int i) {
        print("Конструктор Utensil");
    }
}

class Spoon extends Utensil {
    Spoon(int i) {
        super(i);
        print("Конструктор Spoon");
    }
}

class Fork extends Utensil {
    Fork(int i) {
        super(i);
        System.out.println("Конструктор Fork");
    }
}

class Knife extends Utensil {
    Knife(int i) {
        super(i);
        print("Конструктор Knife");
    }
}

class Custom {
    Custom(int i) {
        print("Конструктор Custom");
    }
}

public class PlaceSetting extends Custom {
    private Spoon sp;
    private Fork frk;
    private Knife kn;
    private DinnerPlate pl;
    public PlaceSetting(int i) {
        super(i + 1);
        sp = new Spoon(i + 2);
        frk = new Fork(i + 3);
        kn = new Knife(i + 4);
        pl = new DinnerPlate(i + 5);
        print("Конструктор PlaceSetting");
    }
    public static void main(String[] args) {
        PlaceSetting x = new PlaceSetting(9);
    }
} /* Output:
Конструктор Custom
Конструктор Utensil
Конструктор Spoon
Конструктор Utensil

```

```
Конструктор Fork
Конструктор Utensil
Конструктор Knife
Конструктор Plate
Конструктор DinnerPlate
Конструктор PlaceSetting
*///:~
```

Несмотря на то что компилятор заставляет вас инициализировать базовые классы и требует, чтобы вы делали это прямо в начале конструктора, он не следит за инициализацией встроенных объектов, поэтому вы должны сами помнить об этом.

Просто удивительно, насколько четко разделяются классы в таком решении. Для повторного использования кода даже не нужен исходный код методов — достаточно простого импортирования пакета (это относится как к наследованию, так и к композиции).

Обеспечение правильного завершения

В Java отсутствует понятие *деструктора* из C++ — метода, автоматически вызываемого при уничтожении объекта. В Java программисты просто «забывают» об объектах, не уничтожая их самостоятельно, так как функции очистки памяти возложены на уборщика мусора.

Во многих случаях эта модель работает, но иногда класс выполняет некоторые операции, требующие завершающих действий. Как упоминалось в главе 5, вы не знаете, когда будет вызван уборщик мусора и произойдет ли это вообще. Поэтому, если в классе должны выполняться действия по очистке, вам придется написать для этого особый метод и сделать так, чтобы программисты-клиенты знали о необходимости вызова этого метода. Более того, как описано в главе 12, вам придется предусмотреть возможные исключения и выполнить завершающие действия в секции `finally`.

Представим пример системы автоматизированного проектирования, которая рисует на экране изображения:

```
//: reusing/CADSystem.java
// Обеспечение необходимого завершения
package reusing;
import static net.mindview.util.Print.*;

class Shape {
    Shape(int i) { print("Конструктор Shape"); }
    void dispose() { print("Завершение Shape"); }
}

class Circle extends Shape {
    Circle(int i) {
        super(i);
        print("Рисуем окружность Circle");
    }
    void dispose() {
        print("Стираем окружность Circle");
        super.dispose();
    }
}
```

```

class Triangle extends Shape {
    Triangle(int i) {
        super(i);
        print("Рисуем треугольник Triangle");
    }
    void dispose() {
        print("Стираем треугольник Triangle");
        super.dispose();
    }
}

class Line extends Shape {
    private int start, end;
    Line(int start, int end) {
        super(start);
        this.start = start;
        this.end = end;
        print("Рисуем линию Line: " + start + ", " + end);
    }
    void dispose() {
        print("Стираем линию Line: " + start + ", " + end);
        super.dispose();
    }
}

public class CADSystem extends Shape {
    private Circle c;
    private Triangle t;
    private Line[] lines = new Line[3];
    public CADSystem(int i) {
        super(i + 1);
        for(int j = 0; j < lines.length; j++)
            lines[j] = new Line(j, j*j);
        c = new Circle(1);
        t = new Triangle(1);
        print("Комбинированный конструктор");
    }
    void dispose() {
        print("CADSystem.dispose()");
        // Завершение осуществляется в порядке,
        // обратном порядку инициализации
        t.dispose();
        c.dispose();
        for(int i = lines.length - 1; i >= 0; i--)
            lines[i].dispose();
        super.dispose();
    }
    public static void main(String[] args) {
        CADSystem x = new CADSystem(47);
        try {
            // Код и обработка исключений...
        } finally {
            x.dispose();
        }
    }
}
/* Output:
Конструктор Shape
Конструктор Shape

```

```
Рисуем линию Line: 0, 0
Конструктор Shape
Рисуем линию Line: 1, 1
Конструктор Shape
Рисуем линию Line: 2, 4
Конструктор Shape
Рисуем окружность Circle
Конструктор Shape
Рисуем треугольник Triangle
Комбинированный конструктор
CADSystem.dispose()
Стираем треугольник Triangle
Завершение Shape
Стираем окружность Circle
Завершение Shape
Стираем линию Line: 2, 4
Завершение Shape
Стираем линию Line: 1, 1
Завершение Shape
Стираем линию Line: 0, 0
Завершение Shape
Завершение Shape
*///:~
```

Все в этой системе является некоторой разновидностью класса `Shape` (который, в свою очередь, неявно наследует от корневого класса `Object`). Каждый класс переопределяет метод `dispose()` класса `Shape`, вызывая при этом версию метода из базового класса с помощью ключевого слова `super`. Все конкретные классы, унаследованные от `Shape`, — `Circle`, `Triangle` и `Line`, имеют конструкторы, которые просто выводят сообщение, хотя во время жизни объекта любой метод может сделать что-то, требующее очистки. В каждом классе есть свой собственный метод `dispose()`, который восстанавливает ресурсы, не связанные с памятью, к исходному состоянию до создания объекта.

В методе `main()` вы можете заметить два новых ключевых слова, которые будут подробно рассмотрены в главе 12: `try` и `finally`. Ключевое слово `try` показывает, что следующий за ним блок (ограниченный фигурными скобками) является *защищенной секцией*. Код в секции `finally` выполняется всегда, независимо от того, как прошло выполнение блока `try`. (При обработке исключений можно выйти из блока `try` некоторыми необычными способами.) В данном примере секция `finally` означает: «Что бы ни произошло, в конце всегда вызывать метод `x.dispose()`».

Также обратите особое внимание на порядок вызова завершающих методов для базового класса и объектов-членов в том случае, если они зависят друг от друга. В основном нужно следовать тому же принципу, что использует компилятор C++ при вызове деструкторов: сначала провести завершающие действия для вашего класса в последовательности, обратной порядку их создания. (Обычно для этого требуется, чтобы элементы базовых классов продолжали существовать.) Затем вызываются завершающие методы из базовых классов, как и показано в программе.

Во многих случаях завершающие действия не создают проблем; достаточно дать уборщику мусора выполнить свою работу. Но уж если понадобилось провести их явно, сделайте это со всей возможной тщательностью и вниманием, так как в процессе уборки мусора трудно в чем-либо быть уверенным. Уборщик мусора вообще может не

вызываться, а если он начнет работать, то объекты будут уничтожаться в произвольном порядке. Лучше не полагаться на уборщик мусора в ситуациях, где дело не касается освобождения памяти. Если вы хотите провести завершающие действия, создайте для этой цели свой собственный метод и не полагайтесь на метод `finalize()`.

12. (3) Включите иерархию методов `dispose()` во все классы из упражнения 9.

Соккрытие имен

Если какой-либо из методов базового класса Java был перегружен несколько раз, переопределение имени этого метода в производном классе *не скроет* ни одну из базовых версий (в отличие от C++). Поэтому перегрузка работает вне зависимости от того, где был определен метод — на текущем уровне или в базовом классе:

```
//: reusing/Hide.java
// Перегрузка имени метода из базового класса
// в производном классе не скроет базовую версию метода.
import static net.mindview.util.Print.*;

class Homer {
    char doh(char c) {
        print("doh(char)");
        return 'd';
    }
    float doh(float f) {
        print("doh(float)");
        return 1.0f;
    }
}

class Milhouse {}

class Bart extends Homer {
    void doh(Milhouse m) {
        print("doh(Milhouse)");
    }
}

public class Hide {
    public static void main(String[] args) {
        Bart b = new Bart();
        b.doh(1);
        b.doh('x');
        b.doh(1.0f);
        b.doh(new Milhouse());
    }
} /* Output:
doh(float)
doh(char)
doh(float)
doh(Milhouse)
*///:~
```

Мы видим, что все перегруженные методы класса `Homer` доступны классу `Bart`, хотя класс `Bart` и добавляет новый перегруженный метод (в C++ такое действие спрятало

бы все методы базового класса). Как вы увидите в следующей главе, на практике при переопределении методов гораздо чаще используется точно такое же описание и список аргументов, как и в базовом классе. Иначе легко можно запутаться (и поэтому C++ запрещает это, чтобы предотвратить совершение возможной ошибки).

В Java SE5 появилась аннотация `@Override`; она не является ключевым словом, но может использоваться так, как если бы была им. Если вы собираетесь переопределить метод, используйте `@Override`, и компилятор выдаст сообщение об ошибке, если вместо переопределения будет случайно выполнена перегрузка:

```
//: reusing/Lisa.java
// {CompileTimeError} (Won't compile)

class Lisa extends Homer {
    @Override void doh(Milhouse m) {
        System.out.println("doh(Milhouse)");
    }
} ///:~
```

- 13.** (2) Создайте класс с троекратно перегруженным методом. Объявите новый класс производным от него, добавьте новую перегрузку метода и покажите, что все четыре метода доступны в производном классе.

Композиция в сравнении с наследованием

И композиция, и наследование позволяют вам помещать подобъекты внутрь вашего нового класса (при композиции это происходит явно, а в наследовании — опосредованно). Вы можете поинтересоваться, в чем между ними разница и когда следует выбирать одно, а когда другое.

Композиция в основном используется, когда в новом классе необходимо задействовать функциональность уже существующего класса, но не его интерфейс. То есть вы встраиваете объект, чтобы использовать его возможности в новом классе, а пользователь класса видит определенный вами интерфейс, но не замечает встроенных объектов. Для этого внедряемые объекты объявляются со спецификатором `private`.

Иногда требуется предоставить пользователю прямой доступ к композиции вашего класса, то есть сделать встроенный объект открытым (`public`). Встроенные объекты и сами используют сокрытие реализации, поэтому открытый доступ безопасен. Когда пользователь знает, что класс собирается из составных частей, ему значительно легче понять его интерфейс. Хорошим примером служит объект `Car` (машина):

```
//: reusing/Car.java
// Композиция с использованием открытых объектов

// двигатель
class Engine {
    public void start() {} // запустить
    public void rev() {}   // переключить
    public void stop() {}  // остановить
}
```

```

// колесо
class Wheel {
    public void inflate(int psi) {} // накачать
}

// окно
class Window {
    public void rollup() {} // поднять
    public void rolldown() {} // опустить
}

// дверь
class Door {
    public Window window = new Window(); // окно двери
    public void open() {} // открыть
    public void close() {} // закрыть
}

// машина
public class Car {
    public Engine engine = new Engine();
    public Wheel[] wheel = new Wheel[4];
    public Door
        left = new Door(),
        right = new Door(); // двухдверная машина
    public Car() {
        for(int i = 0; i < 4; i++)
            wheel[i] = new Wheel();
    }
    public static void main(String[] args) {
        Car car = new Car();
        car.left.window.rollup();
        car.wheel[0].inflate(72);
    }
} ///:~

```

Так как композиция объекта является частью проведенного анализа задачи (а не просто частью реализации класса), объявление членов класса открытыми (`public`) помогает программисту-клиенту понять, как использовать класс, и облегчает создателю класса написание кода. Однако нужно все-таки помнить, что описанный случай является специфическим и в основном поля класса следует объявлять как `private`.

При использовании наследования вы берете уже существующий класс и создаете его специализированную версию. В основном это значит, что класс общего назначения адаптируется для конкретной задачи. Если вы чуть-чуть подумаете, то поймете, что не имело бы смысла использовать композицию машины и средства передвижения — машина не содержит средства передвижения, она сама *есть* это средство. Взаимосвязь «является» выражается наследованием, а взаимосвязь «имеет» описывается композицией.

14. (1) В `Car.java` добавьте в класс `Engine` метод `service()` и вызовите его из `main()`.

protected

После знакомства с наследованием ключевое слово `protected` наконец-то обрело смысл. В идеале закрытых членов `private` должно было быть достаточно. В реальности

существуют ситуации, когда вам необходимо спрятать что-либо от окружающего мира, тем не менее оставив доступ для производных классов.

Ключевое слово `protected` — дань прагматизму. Оно означает: «Член класса является закрытым (`private`) для пользователя класса, но для всех, кто наследует от класса, и для соседей по пакету он доступен». (В Java `protected` автоматически предоставляет доступ в пределах пакета.)

Лучше всего, конечно, объявлять поля класса как `private` — всегда стоит оставить за собой право изменять лежащую в основе реализацию. Управляемый доступ наследникам класса предоставляется через методы `protected`:

```
//: reusing/Orc.java
// Ключевое слово protected.
import static net.mindview.util.Print.*;

class Villain {
    private String name;
    protected void set(String nm) { name = nm; }
    public Villain(String name) { this.name = name; }
    public String toString() {
        return "Я объект Villain и мое имя " + name;
    }
}

public class Orc extends Villain {
    private int orcNumber;
    public Orc(String name, int orcNumber) {
        super(name);
        this.orcNumber = orcNumber;
    }
    public void change(String name, int orcNumber) {
        set(name); // Доступно, так как объявлено protected
        this.orcNumber = orcNumber;
    }
    public String toString() {
        return "Orc " + orcNumber + ": " + super.toString();
    }
    public static void main(String[] args) {
        Orc orc = new Orc("Лимбургер", 12);
        print(orc);
        orc.change("Боб", 19);
        print(orc);
    }
} /* Output:
Orc 12: Я объект Villain и мое имя Лимбургер
Orc 19: Я объект Villain и мое имя Боб
*///:~
```

Как видите, метод `change()` имеет доступ к методу `set()`, поскольку тот объявлен как `protected`. Также обратите внимание, что метод `toString()` класса `Orc` определяется с использованием версии этого метода из базового класса.

15. (2) Создайте класс в пакете. Ваш класс должен содержать метод со спецификатором `protected`. Попробуйте вызвать метод `protected` за пределами пакета, и объясните, что происходит. Затем создайте класс, производный от вашего, и вызовите метод `protected` из другого метода вашего производного класса.

Восходящее преобразование типов

Самая важная особенность наследования заключается вовсе не в том, что оно предоставляет методы для нового класса, — наследование выражает отношения между новым и базовым классом. Ее можно выразить следующим образом: «Новый класс *является* разновидностью существующего класса».

Данная формулировка — не просто причудливый способ описания наследования, она напрямую поддерживается языком. В качестве примера рассмотрим базовый класс с именем `Instrument` для представления музыкальных инструментов и его производный класс `Wind`. Так как наследование означает, что все методы базового класса также доступны в производном классе, любое сообщение, которое вы в состоянии отправить базовому классу, можно отправить и производному классу. Если в классе `Instrument` имеется метод `play()`, то он будет присутствовать и в классе `Wind`. Таким образом, мы можем со всей определенностью утверждать, что объекты `Wind` также имеют тип `Instrument`. Следующий пример показывает, как компилятор поддерживает такое понятие:

```
//: reusing/Wind.java
// Наследование и восходящее преобразование.

class Instrument {
    public void play() {}
    static void tune(Instrument i) {
        // ...
        i.play();
    }
}

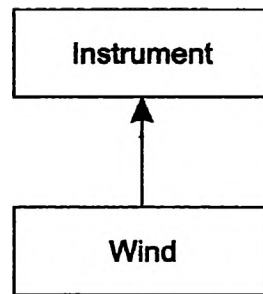
// Объекты Wind также являются объектами Instrument,
// поскольку они имеют тот же интерфейс:
public class Wind extends Instrument {
    public static void main(String[] args) {
        Wind flute = new Wind();
        Instrument.tune(flute); // Восходящее преобразование
    }
} ///:~
```

Наибольший интерес в этом примере представляет метод `tune()`, получающий ссылку на объект `Instrument`. Однако в методе `Wind.main()` методу `tune()` передается ссылка на объект `Wind`. С учетом всего, что говорилось о строгой проверке типов в Java, кажется странным, что метод спокойно принимает один тип вместо другого. Но стоит вспомнить, что объект `Wind` также является разновидностью объекта `Instrument` и не существует метода, который можно вызвать в методе `tune()` для объектов `Instrument`, но нельзя для объектов `Wind`. В методе `tune()` код работает для `Instrument` и любых объектов, производных от `Instrument`, а преобразование ссылки на объект `Wind` в ссылку на объект `Instrument` называется *восходящим преобразованием типов* (upcasting).

Почему «восходящее преобразование»?

Термин возник по историческим причинам: традиционно на диаграммах наследования корень иерархии изображался у верхнего края страницы, а диаграмма разрасталась

в нижнему краю страницы. (Конечно, вы можете рисовать свои диаграммы так, как сочтете нужным.) Для файла `Wind.java` диаграмма наследования выглядит так.



Преобразование от производного типа к базовому требует движения вверх по диаграмме, поэтому часто называется *восходящим* преобразованием. Восходящее преобразование всегда безопасно, так как это переход от конкретного типа к более общему типу. Иначе говоря, производный класс является надстройкой базового класса. Он может содержать больше методов, чем базовый класс, но *обязан* включать в себя все методы базового класса. Единственное, что может произойти с интерфейсом класса при восходящем преобразовании, — потеря методов, но никак не их приобретение. Именно поэтому компилятор всегда разрешает выполнять восходящее преобразование, не требуя явных преобразований или других специальных обозначений.

Преобразование также может выполняться и в обратном направлении — так называемое *нисходящее преобразование* (downcasting). Но при этом возникает проблема, которая рассматривается в главе 11.

Снова о композиции с наследованием

В объектно-ориентированном программировании разработчик обычно упаковывает данные вместе с методами в классе, а затем работает с объектами этого класса. Существующие классы также используются для создания новых классов посредством композиции. Наследование на практике применяется реже. Поэтому, хотя во время изучения ООП наследованию уделяется очень много внимания, это не значит, что его следует без разбора применять всюду, где можно. Наоборот, пользоваться им следует осмотрительно — только там, где полезность наследования не вызывает сомнений (одна из самых важных причин для применения наследования приводится в главе 8). Один из хороших критериев выбора между композицией и наследованием — спросить себя, собираетесь ли вы впоследствии проводить восходящее преобразование от производного класса к базовому классу.

6. (2) Создайте класс с именем `Amphibian`. Объявите производный от него класс с именем `Frog`. Разместите в базовом классе несколько методов. В методе `main()` создайте объект `Frog`, преобразуйте его в `Amphibian` и продемонстрируйте, что все методы работают.
7. Измените упражнение 16 так, чтобы класс `Frog` переопределял методы базового класса (предоставляя новые определения с той же сигнатурой метода). Посмотрите, что произойдет в методе `main()`.

Ключевое слово `final`

В Java смысл ключевого слова `final` зависит от контекста, но в основном оно означает: «*Это нельзя изменить*». Запрет на изменения может объясняться двумя причинами: архитектурой программы или эффективностью. Эти две причины основательно различаются, поэтому в программе возможно неверное употребление ключевого слова `final`.

В следующих разделах обсуждаются три возможных применения `final`: для данных, методов и классов.

Неизменные данные

Во многих языках программирования существует тот или иной способ сказать компилятору, что частица данных является «константой». Константы полезны в двух ситуациях:

- ❑ *константа времени компиляции*, которая никогда не меняется;
- ❑ значение, инициализируемое во время работы программы, которое нельзя изменять.

Компилятор подставляет значение константы времени компиляции во все выражения, где оно используется; таким образом предотвращаются некоторые издержки выполнения. В Java подобные константы должны относиться к примитивным типам, а для их определения используется ключевое слово `final`. Значение такой константы присваивается во время определения.

Поле, одновременно объявленное с ключевыми словами `static` и `final`, существует в памяти в единственном экземпляре и не может быть изменено.

При использовании слова `final` со ссылками на объекты его смысл не столь очевиден. Для примитивов `final` делает постоянным *значение*, но для ссылки на объект постоянной становится *ссылка*. После того как такая ссылка будет связана с объектом, она уже не сможет указывать на другой объект. Впрочем, сам объект при этом может изменяться; в Java нет механизмов, позволяющих сделать произвольный объект неизменным. (Впрочем, вы сами можете написать ваш класс так, чтобы его объекты фактически были константными.) Данное ограничение относится и к массивам, которые тоже являются объектами.

Следующий пример демонстрирует использование `final` для полей классов. По действующим правилам поля, объявленные одновременно с ключевыми словами `static` и `final` (то есть константы времени компиляции), записываются прописными буквами, а входящие в них слова разделяются символами подчеркивания.

```
//: reusing/FinalData.java
// Действие ключевого слова final для полей.
import java.util.*;
import static net.mindview.util.Print.*;

class Value {
    int i; // доступ в пределах пакета
    public Value(int i) { this.i = i; }
}
```

```

public class FinalData {
    private static Random rand = new Random(47);
    private String id;
    public FinalData(String id) { this.id = id; }
    // Могут быть константами времени компиляции:
    private final int valueOne = 9;
    private static final int VALUE_TWO = 99;
    // Типичная открытая константа:
    public static final int VALUE_THREE = 39;
    // Не может быть константой времени компиляции:
    private final int i4 = rand.nextInt(20);
    static final int INT_5 = rand.nextInt(20);
    private Value v1 = new Value(11);
    private final Value v2 = new Value(22);
    private static final Value VAL_3 = new Value(33);
    // Массивы:
    private final int[] a = { 1, 2, 3, 4, 5, 6 };
    public String toString() {
        return id + ": " + "i4 = " + i4 + ", INT_5 = " + INT_5;
    }
    public static void main(String[] args) {
        FinalData fd1 = new FinalData("fd1");
        /// fd1.valueOne++; // Ошибка: значение нельзя изменить
        fd1.v2.i++; // Объект не является неизменным!
        fd1.v1 = new Value(9); // ОК - не является неизменным
        for(int i = 0; i < fd1.a.length; i++)
            fd1.a[i]++; // Объект не является неизменным!
        /// fd1.v2 = new Value(0); // Ошибка: ссылку
        /// fd1.VAL_3 = new Value(1); // нельзя изменить
        /// fd1.a = new int[3];
        print(fd1);
        print("Создаем FinalData");
        FinalData fd2 = new FinalData("fd2");
        print(fd1);
        print(fd2);
    }
} /* Output:
fd1: i4 = 15, INT_5 = 18
Создаем FinalData
fd1: i4 = 15, INT_5 = 18
fd2: i4 = 13, INT_5 = 18
*///:~

```

Так как `valueOne` и `VALUE_TWO` являются примитивными типами со значениями, заданными на стадии компиляции, они оба могут использоваться в качестве констант времени компиляции, и принципиальных различий между ними нет. Константа `VALUE_THREE` демонстрирует общепринятый способ определения подобных полей: спецификатор `public` открывает к ней доступ за пределами пакета; ключевое слово `static` указывает, что она существует в единственном числе, а ключевое слово `final` указывает, что ее значение остается неизменным. Заметьте, что примитивы `final static` с неизменными начальными значениями (то есть константы времени компиляции) записываются целиком заглавными буквами, а слова разделяются подчеркиванием (эта схема записи констант позаимствована из языка C).

Само по себе присутствие `final` еще не означает, что значение переменной известно уже на стадии компиляции. Данный факт продемонстрирован на примере инициализации

`i4` и `INT_5` с использованием случайных чисел. Эта часть программы также показывает разницу между статическими и нестатическими константами. Она проявляется только при инициализации во время исполнения, так как все величины времени компиляции обрабатываются компилятором одинаково (и обычно просто устраняются с целью оптимизации). Различие проявляется в результатах запуска программы. Заметьте, что значения поля `i4` для объектов `fd1` и `fd2` уникальны, но значение поля `INT_5` не изменяется при создании второго объекта `finalData`. Дело в том, что поле `INT_5` объявлено как `static`, поэтому оно инициализируется только один раз во время загрузки класса.

Переменные от `v1` до `VAL_3` поясняют смысл объявления ссылок с ключевым словом `final`. Как видно из метода `main()`, объявление ссылки `v2` как `final` еще не означает, что ее объект неизменен. Однако присоединить ссылку `v2` к новому объекту не получится, как раз из-за того, что она была объявлена как `final`. Именно такой смысл имеет ключевое слово `final` по отношению к ссылкам. Вы также можете убедиться, что это верно и для массивов, которые являются просто другой разновидностью ссылки. Пожалуй, для ссылок ключевое слово `final` обладает меньшей практической ценностью, чем для примитивов.

- 18.** (2) Создайте класс, содержащий два поля: `static final` и `final`. Продемонстрируйте различия между ними.

Пустые константы

В Java разрешается создавать *пустые константы* — поля, объявленные как `final`, но которым не было присвоено начальное значение. Во всех случаях пустую константу *обязательно* нужно инициализировать перед использованием, и компилятор следит за этим. Впрочем, пустые константы расширяют свободу действий при использовании ключевого слова `final`, так как, например, поле `final` в классе может быть разным для каждого объекта и при этом оно сохраняет свою неизменность. Пример:

```
//: c06:BlankFinal.java
// "Пустые" неизменные поля.

class Poppet {
    private int i;
    Poppet(int ii) { i = ii; }
}

public class BlankFinal {
    private final int i = 0; // Инициализированная константа
    private final int j;     // Пустая константа
    private final Poppet p;  // Пустая константа-ссылка
    // Пустые константы НЕОБХОДИМО инициализировать
    // в конструкторе:
    public BlankFinal() {
        j = 1; // Инициализация пустой константы
        p = new Poppet(1); // Инициализация пустой неизменной ссылки
    }
    public BlankFinal(int x) {
        j = x; // Инициализация пустой константы
        p = new Poppet(x); // Инициализация пустой неизменной ссылки
    }
    public static void main(String[] args) {
```

```
    new BlankFinal();  
    new BlankFinal(47);  
}  
} ///:~
```

Значения неизменных (**final**) переменных обязательно должны присваиваться или в выражении, записываемом в точке определения переменной, или в каждом из конструкторов класса. Тем самым гарантируется инициализация полей, объявленных как **final**, перед их использованием.

- 19.** (2) Создайте класс с пустой **final**-ссылкой на объект. Проведите инициализацию пустой константы во всех конструкторах. Продемонстрируйте гарантированную инициализацию **final** перед использованием и невозможность ее изменения после инициализации.

Неизменные аргументы

Java позволяет вам объявлять неизменными аргументы метода, с помощью ключевого слова **final** в списке аргументов. Это значит, что метод не может изменить значение, на которое указывает передаваемая ссылка:

```
//: reusing/FinalArguments.java  
// Использование final с аргументами метода.  
  
class Gizmo {  
    public void spin() {}  
}  
  
public class FinalArguments {  
    void with(final Gizmo g) {  
        ///! g = new Gizmo(); // Запрещено -- g объявлено final  
    }  
    void without(Gizmo g) {  
        g = new Gizmo(); // Разрешено -- g не является final  
        g.spin();  
    }  
    // void f(final int i) { i++; } // Нельзя изменять,  
    // неизменные примитивы доступны только для чтения:  
    int g(final int i) { return i + 1; }  
    public static void main(String[] args) {  
        FinalArguments bf = new FinalArguments();  
        bf.without(null);  
        bf.with(null);  
    }  
} ///:~
```

Методы **f()** и **g()** показывают, что происходит при передаче методу примитивов с пометкой **final**: их значение можно прочесть, но изменить его не удастся. Эта возможность чаще всего используется при передаче данных анонимным внутренним классам, о которых вы узнаете в главе 10.

Неизменные методы

Неизменные методы используются по двум причинам. Первая причина — «блокировка» метода, чтобы производные классы не могли изменить его содержание. Это делается

по соображениям проектирования, когда вам точно надо знать, что поведение метода не изменится при наследовании.

Второй причиной в прошлом считалась эффективность. В более ранних реализациях Java объявление метода с ключевым словом `final` позволяло компилятору превратить все вызовы такого метода во *встроенные* (`inline`). Когда компилятор видит метод, объявленный как `final`, он может (на свое усмотрение) пропустить стандартный механизм вставки кода для проведения вызова метода (занести аргументы в стек, перейти к телу метода, исполнить находящийся там код, вернуть управление, удалить аргументы из стека и распорядиться возвращенным значением) и вместо этого подставить на место вызова копию реального кода, находящегося в теле метода. Таким образом устраняются издержки обычного вызова метода. Конечно, для больших методов подстановка приведет к «разбуханию» программы и, скорее всего, никаких преимуществ от использования прямого встраивания не будет.

В последних версиях Java виртуальная машина выявляет подобные ситуации и устраняет лишние передачи управления при оптимизации, поэтому использовать `final` для методов уже не обязательно — и более того, нежелательно. В Java SE5/6 стоит поручить проблемы эффективности компилятору и JVM и объявлять методы `final` только в том случае, если вы хотите явно запретить переопределение¹.

Спецификаторы `final` и `private`

Любой закрытый (`private`) метод в классе косвенно является неизменным (`final`) методом. Так как вы не в силах получить доступ к закрытому методу, то не сможете и переопределить его. Ключевое слово `final` можно добавить к закрытому методу, но его присутствие ни на что не повлияет.

Это может вызвать недоразумения, так как при попытке переопределения закрытого (`private`) метода, также неявно являющегося `final`, все вроде бы работает, и компилятор не выдает сообщений об ошибках:

```
//: reusing/FinalOverridingIllusion.java
// Все выглядит так, будто закрытый (и неизменный) метод
// можно переопределить, но это заблуждение.
import static net.mindview.util.Print.*;

class WithFinals {
    // То же, что и просто private:
    private final void f() { print("WithFinals.f()"); }
    // Также автоматически является final:
    private void g() { print("WithFinals.g()"); }
}

class OverridingPrivate extends WithFinals {
    private final void f() {
        print("OverridingPrivate.f()");
    }
}
```

¹ Не поддавайтесь искушению поспешной оптимизации. Если ваша система работает недостаточно быстро, вряд ли проблему удастся решить ключевым словом `final`. На странице <http://MindView.net/Books/BetterJava> рассказано о профилировании, которое может ускорить работу вашей программы.


```

    private void g() {
        print("OverridingPrivate.g()");
    }
}

class OverridingPrivate2 extends OverridingPrivate {
    public final void f() {
        print("OverridingPrivate2.f()");
    }
    public void g() {
        print("OverridingPrivate2.g()");
    }
}

public class FinalOverridingIllusion {
    public static void main(String[] args) {
        OverridingPrivate2 op2 = new OverridingPrivate2();
        op2.f();
        op2.g();
        // Можно провести восходящее преобразование:
        OverridingPrivate op = op2;
        // Но методы при этом вызвать невозможно:
        /// op.f();
        /// op.g();
        // И то же самое здесь:
        WithFinals wf = op2;
        /// wf.f();
        /// wf.g();
    }
} /* Output:
OverridingPrivate2.f()
OverridingPrivate2.g()
*///:~

```

«Переопределение» применимо только к компонентам интерфейса базового класса. Иначе говоря, вы должны иметь возможность выполнить восходящее преобразование объекта к его базовому типу и вызвать тот же самый метод (это утверждение подробнее обсуждается в следующей главе). Если метод объявлен как `private`, он не является частью интерфейса базового класса; это просто некоторый код, скрытый внутри класса, у которого оказалось то же имя. Если вы создаете в производном классе одноименный метод со спецификатором `public`, `protected` или с доступом в пределах пакета, то он никак не связан с закрытым методом базового класса. Так как `private` метод недоступен и фактически невидим для окружающего мира, он не влияет ни на что, кроме внутренней организации кода в классе, где он был описан.

20. (1) Продемонстрируйте, что аннотация `@Override` решает проблему, упомянутую в этом разделе.
21. Создайте класс с неизменным (`final`) методом. Создайте производный класс и попытайтесь переопределить этот метод.

Неизменные классы

Объявляя класс неизменным (записывая в его определении ключевое слово `final`), вы показываете, что не собираетесь использовать этот класс в качестве базового при

наследовании и запрещаете это делать другим. Другими словами, по какой-то причине структура вашего класса должна оставаться постоянной — или же появление субклассов нежелательно по соображениям безопасности.

```
//: reusing/Jurassic.java
// Объявление неизменным всего класса.

class SmallBrain {}

final class Dinosaur {
    int i = 7;
    int j = 1;
    SmallBrain x = new SmallBrain();
    void f() {}
}

//! class Further extends Dinosaur {}
// Ошибка: Нельзя расширить неизменный класс Dinosaur

public class Jurassic {
    public static void main(String[] args) {
        Dinosaur n = new Dinosaur();
        n.f();
        n.i = 40;
        n.j++;
    }
} ///:~
```

Заметьте, что поля класса могут быть, а могут и не быть неизменными, по вашему выбору. Те же правила верны и для неизменных методов, вне зависимости от того, объявлен ли класс целиком как `final`. Объявление класса со спецификатором `final` запрещает наследование от него — и ничего больше. Впрочем, из-за того, что это предотвращает наследование, все методы в неизменном классе также являются неизменными, поскольку нет способа переопределить их. Поэтому компилятор имеет тот же выбор для обеспечения эффективности выполнения, как и в случае с явным объявлением методов как `final`. И если вы добавите спецификатор `final` к методу в классе, объявленном всецело как `final`, то это ничего не будет значить.

22. Создайте неизменный (`final`) класс и попытайтесь создать класс, производный от него.

Предостережение

На первый взгляд идея объявления неизменных методов (`final`) во время разработки класса выглядит довольно заманчиво — никто не сможет переопределить ваши методы. Иногда это действительно так.

Но будьте осторожнее в своих допущениях. Трудно предусмотреть все возможности повторного использования класса, особенно для классов общего назначения. Определяя метод как `final`, вы блокируете возможность использования класса в проектах других программистов только потому, что сами не могли предвидеть такую возможность.

Хорошим примером служит стандартная библиотека Java. Класс `vector` Java 1.0/1.1 часто использовался на практике и был бы еще полезнее, если бы по соображениям эффективности (в данном случае эфемерной) все его методы не были объявлены как

`final`. Возможно, вам хотелось бы создать на основе `vector` производный класс и переопределить некоторые методы, но разработчики почему-то посчитали это излишним. Ситуация выглядит еще более парадоксальной по двум причинам. Во-первых, класс `Stack` унаследован от `Vector`, и это значит, что `Stack` *есть* `Vector`, а это неверно с точки зрения логики. Тем не менее мы видим пример ситуации, в которой сами проектировщики Java используют наследование от `Vector`. В тот момент, когда класс `Stack` был создан, они должны были осознать, что `final`-методы вводят излишние ограничения.

Во-вторых, многие полезные методы класса `Vector`, такие как `addElement()` и `elementAt()`, объявлены с ключевым словом `synchronized`. Как вы увидите в главе 12, синхронизация сопряжена со значительными издержками во время выполнения, которые, вероятно, сводят к нулю все преимущества от объявления метода как `final`. Все это лишь подтверждает теорию о том, что программисты не умеют правильно находить области для применения оптимизации. Очень плохо, что такой неуклюжий дизайн проник в стандартную библиотеку Java. (К счастью, современная библиотека контейнеров Java заменяет `Vector` классом `ArrayList`, который сделан гораздо более аккуратно и по общепринятым нормам. К сожалению, существует очень много готового кода, написанного с использованием старой библиотеки контейнеров.)

Также интересно, что класс `Hashtable`, другой важный класс стандартной библиотеки Java 1.0/1.1, не содержит ни одного `final`-метода. Очевидно, что классы стандартной библиотеки создавались совершенно разными людьми (в частности, имена методов `Hashtable` значительно короче имен методов `Vector` — это еще одно доказательство), а ведь именно этот факт не должен быть очевиден для пользователей библиотек. Непоследовательное проектирование лишь усложняет работу пользователя — лишний довод в пользу сквозного контроля архитектуры и кода (кстати, в современной библиотеке контейнеров Java класс `Hashtable` заменен классом `HashMap`).

Инициализация и загрузка классов

В традиционных языках программы загружаются целиком в процессе запуска. Далее следует инициализация, а затем программа начинает работу. Процесс инициализации в таких языках должен тщательно контролироваться, чтобы порядок инициализации статических объектов не создавал проблем. Например, в C++ могут возникнуть проблемы, когда один из статических объектов полагает, что другим статическим объектом уже можно пользоваться, хотя последний еще не был инициализирован.

В языке Java таких проблем не существует, поскольку в нем используется другой подход к загрузке. Вспомните, что скомпилированный код каждого класса хранится в отдельном файле. Этот файл не загружается, пока не возникнет такая необходимость. В сущности, код класса загружается только в точке его первого использования. Обычно это происходит при создании первого объекта класса, но загрузка также выполняется при обращениях к статическим полям или методам¹.

¹ Конструктор также является статическим методом, хотя ключевое слово `static` и не указывается явно. Таким образом, формально загрузка класса выполняется при первом обращении к любому из его статических методов.

Точкой первого использования также является точка выполнения инициализации статических членов. Все статические объекты и блоки кода инициализируются при загрузке класса в том порядке, в котором они записаны в определении класса. Конечно, статические объекты инициализируются только один раз.

Инициализация с наследованием

Полезно разобрать процесс инициализации полностью, включая наследование, чтобы получить общую картину происходящего. Рассмотрим следующий пример:

```
//: reusing/Beetle.java
// Полный процесс инициализации.
import static net.mindview.util.Print.*;

class Insect {
    private int i = 9;
    protected int j;
    Insect() {
        System.out.println("i = " + i + ", j = " + j);
        j = 39;
    }
    private static int x1 =
        printInit("Поле static Insect.x1 инициализировано");
    static int printInit(String s) {
        print(s);
        return 47;
    }
}

public class Beetle extends Insect {
    private int k = printInit("Поле Beetle.k инициализировано");
    public Beetle() {
        prt("k = " + k);
        prt("j = " + j);
    }
    private static int x2 =
        printInit("Поле static Beetle.x2 инициализировано");
    public static void main(String[] args) {
        print("Конструктор Beetle");
        Beetle b = new Beetle();
    }
} /*
Поле static Insect.x1 инициализировано
Поле static Beetle.x2 инициализировано
Конструктор Beetle
i = 9, j = 0
Поле Beetle.k инициализировано
k = 47
j = 39
*///:~
```

Запуск класса `Beetle` в Java начинается с выполнения метода `Beetle.main()` (статического), поэтому загрузчик пытается найти скомпилированный код класса `Beetle` (он должен находиться в файле `Beetle.class`). При этом загрузчик обнаруживает, что у класса имеется базовый класс (о чем говорит ключевое слово `extends`), который затем и загружается.

Это происходит независимо от того, собираетесь вы создавать объект базового класса или нет. (Чтобы убедиться в этом, попробуйте закомментировать создание объекта.)

Если у базового класса имеется свой базовый класс, этот второй базовый класс будет загружен в свою очередь и т. д. Затем проводится `static`-инициализация корневого базового класса (в данном случае это `Insect`), затем следующего за ним производного класса и т. д. Это важно, так как производный класс и инициализация его `static`-объектов могут зависеть от инициализации членов базового класса.

В этой точке все необходимые классы уже загружены и можно переходить к созданию объекта класса. Сначала всем примитивам данного объекта присваиваются значения по умолчанию, а ссылкам на объекты задается значение `null` — это делается за один проход посредством обнуления памяти. Затем вызывается конструктор базового класса. В нашем случае вызов происходит автоматически, но вы можете явно указать в программе вызов конструктора базового класса (записав его в первой строке описания конструктора `Beetle()`) с помощью ключевого слова `super`. Конструирование базового класса выполняется по тем же правилам и в том же порядке, что и для производного класса. После завершения работы конструктора базового класса инициализируются переменные, в порядке их определения. Наконец, выполняется оставшееся тело конструктора.

23. (2) Продемонстрируйте, что загрузка класса выполняется только один раз. Докажите, что загрузка может быть вызвана как созданием первого экземпляра класса, так и обращением к статическому члену.
24. (2) В файле `Beetle.java` создайте еще один тип, производный от `Beetle`, в таком же формате, как и у других классов. Проследите за работой программы и объясните результат.

Резюме

Как наследование, так и композиция позволяют создавать новые типы на основе уже существующих типов. Композиция обычно применяется для повторного использования реализации в новом типе, а наследование — для повторного использования интерфейса. Так как производный класс имеет интерфейс базового класса, к нему можно применить восходящее преобразование к базовому классу; это очень важно для работы полиморфизма (см. следующую главу).

Несмотря на особое внимание, уделяемое наследованию в ООП, при начальном проектировании обычно предпочтение отдается композиции, а к наследованию следует обращаться только там, где это абсолютно необходимо. Композиция обеспечивает несколько большую гибкость. Вдобавок, применяя хитрости наследования к встроенным типам, можно изменять точный тип и соответственно поведение этих встроенных объектов во время исполнения. Таким образом, появляется возможность изменения поведения составного объекта во время исполнения программы.

При проектировании системы вы стремитесь создать иерархию, в которой каждый класс имеет определенную цель, чтобы он не был ни излишне большим (не содержал слишком много функциональности, затрудняющей его повторное использование),

ни раздражающе мал (так, что его нельзя использовать сам по себе, не добавив перед этим дополнительные возможности). Если архитектура становится слишком сложной, часто стоит внести в нее новые объекты, разбивая существующие объекты на меньшие составные части.

Важно понимать, что проектирование программы является пошаговым, последовательным процессом, как и обучение человека. Оно основано на экспериментах; сколько бы вы ни анализировали и ни планировали, в начале работы над проектом у вас еще останутся неясности. Процесс пойдет более успешно — и вы быстрее добьетесь результатов, если начнете «выращивать» свой проект как живое, эволюционирующее существо, нежели «воздвигнете» его сразу, как небоскреб из стекла и металла. Наследование и композиция — два важнейших инструмента объектно-ориентированного программирования, которые помогут вам выполнять эксперименты такого рода.

8

Полиморфизм

Меня спрашивали: «Скажите, мистер Бэббидж, если заложить в машину неверные числа, на выходе она все равно выдаст правильный ответ?» Не представляю, какую же кашу надо иметь в голове, чтобы задавать подобные вопросы.

Чарльз Бэббидж (1791–1871)

Полиморфизм является третьей важнейшей особенностью объектно-ориентированных языков, вместе с абстракцией данных и наследованием.

Он предоставляет еще одну степень отделения интерфейса от реализации, разъединения *что* от *как*. Полиморфизм улучшает организацию кода и его читаемость, а также способствует созданию *расширяемых* программ, которые могут «расти» не только в процессе начальной разработки проекта, но и при добавлении новых возможностей.

Инкапсуляция создает новые типы данных за счет объединения характеристик и поведения. Соккрытие реализации отделяет интерфейс от реализации за счет изоляции технических подробностей в *private*-частях класса. Подобное механическое разделение понятно любому, кто имел опыт работы с процедурными языками. Но полиморфизм имеет дело с логическим разделением в контексте *типов*. В предыдущей главе вы увидели, что наследование позволяет работать с объектом, используя как его собственный тип, *так и* его базовый тип. Этот факт очень важен, потому что он позволяет работать со многими типами (производными от одного базового типа) как с единым типом, что позволяет единому коду работать с множеством разных типов единообразно. Вызов полиморфного метода позволяет одному типу выразить свое отличие от другого, сходного типа, хотя они и происходят от одного базового типа. Это отличие выражается различным действием методов, вызываемых через базовый класс.

В этой главе рассматривается полиморфизм (также называемый *динамическим связыванием*, или *поздним связыванием*, или *связыванием во время выполнения*). Мы начнем с азов, а изложение материала будет поясняться простыми примерами, полностью акцентированными на полиморфном поведении программы.

Снова о восходящем преобразовании

Как было показано в главе 7, с объектом можно работать с использованием как его собственного типа, так и его базового типа. Интерпретация ссылки на объект как ссылки на базовый тип называется *восходящим преобразованием*.

Также были представлены проблемы, возникающие при восходящем преобразовании и наглядно воплощенные в следующей программе с музыкальными инструментами. Поскольку мы будем проигрывать с их помощью объекты `Note` (нота), логично создать эти объекты в отдельном пакете:

```
//: polymorphism/music/Music.java
// Объекты Note для использования с Instrument.
package polymorphism.music;

public enum Note {
    MIDDLE_C, C_SHARP, B_FLAT; // и т.д.
} ///:~
```

Перечисления были представлены в главе 5.

В следующем примере `Wind` является частным случаем инструмента (`Instrument`), поэтому класс `Wind` наследует от `Instrument`:

```
//: polymorphism/music/Instrument.java
package polymorphism.music;
import static net.mindview.util.Print.*;

class Instrument {
    public void play(Note n) {
        print("Instrument.play()");
    }
}
///:~

//: polymorphism/music/Wind.java
package polymorphism.music;

// Объекты Wind также являются объектами Instrument,
// поскольку имеют тот же интерфейс:
public class Wind extends Instrument {
    // Переопределение метода интерфейса:
    public void play(Note n) {
        System.out.println("Wind.play() " + n);
    }
} ///:~

//: polymorphism/music/Music.java
// Наследование и восходящее преобразование
package polymorphism.music;

public class Music {
    public static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
}
```



```

public static void main(String[] args) {
    Wind flute = new Wind();
    tune(flute); // Восходящее преобразование
}
} /* Output:
Wind.play() MIDDLE_C
*///:~

```

Метод `Music.tune()` получает ссылку на `Instrument`, но последняя также может указывать на объект любого класса, производного от `Instrument`. В методе `main()` ссылка на объект `Wind` передается методу `tune()` без явных преобразований. Это нормально; интерфейс класса `Instrument` должен существовать и в классе `Wind`, поскольку последний был унаследован от `Instrument`. Восходящее преобразование от `Wind` к `Instrument` способно «сузить» этот интерфейс, но не сделает его «меньше», чем полный интерфейс класса `Instrument`.

Потеря типа объекта

Программа `Music.java` выглядит немного странно. Зачем умышленно *игнорировать* фактический тип объекта? Именно такое мы наблюдаем при восходящем преобразовании, и казалось бы, программа стала яснее, если бы методу `tune()` передавалась ссылка на объект `Wind`. Но при этом мы сталкиваемся с очень важным обстоятельством: если поступить подобным образом, то потом придется писать новый метод `tune()` для каждого типа `Instrument`, присутствующего в системе. Предположим, что в систему были добавлены новые классы `Stringed` и `Brass`:

```

//: polymorphism/music/Music2.java
// Перегрузка вместо восходящего преобразования.
package polymorphism.music;
import static net.mindview.util.Print.*;

class Stringed extends Instrument {
    public void play(Note n) {
        print("Stringed.play() " + n);
    }
}

class Brass extends Instrument {
    public void play(Note n) {
        print("Brass.play() " + n);
    }
}

public class Music2 {
    public static void tune(Wind i) {
        i.play(Note.MIDDLE_C);
    }
    public static void tune(Stringed i) {
        i.play(Note.MIDDLE_C);
    }
    public static void tune(Brass i) {
        i.play(Note.MIDDLE_C);
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        Stringed violin = new Stringed();
    }
}

```

```

    Brass frenchHorn = new Brass();
    tune(flute); // Без восходящего преобразования
    tune(violin);
    tune(frenchHorn);
}
} /* Output:
Wind.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
*///:~

```

Программа работает, но у нее есть огромный недостаток: для каждого нового `Instrument` приходится писать новый, зависящий от конкретного типа метод `tune()`. Объем программного кода увеличивается, а при добавлении нового метода (такого, как `tune()`) или нового типа инструмента придется выполнить немало дополнительной работы. А если учесть, что компилятор не выводит сообщений об ошибках, если вы забудете перегрузить один из ваших методов, то и весь процесс работы с типами станет совершенно неуправляемым.

Разве не лучше было бы написать единственный метод, в аргументе которого передается базовый класс, а не один из производных классов? Разве не удобнее было бы забыть о производных классах и написать обобщенный код для базового класса?

Именно это и позволяет делать полиморфизм. Тем не менее многие программисты с опытом работы на процедурных языках при работе с полиморфизмом испытывают некоторые затруднения.

1. (2) Создайте класс `Cycle` с производными классами `Unicycle`, `Bicycle` и `Tricycle`. Покажите, что экземпляр каждого из производных типов может быть преобразован в `Cycle`, на примере вызова метода `ride()`.

Особенности

Сложности с программой `Music.java` обнаруживаются после ее запуска. Она выводит строку `wind.play()`. Именно это и требуется, но непонятно, откуда берется такой результат. Взгляните на метод `tune()`:

```

public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
}

```

Метод получает ссылку на объект `Instrument`. Как компилятор узнает, что ссылка на `Instrument` в данном случае указывает на объект `wind`, а не на `Brass` или `Stringed`? Компилятор и не знает. Чтобы в полной мере разобраться в сути происходящего, необходимо рассмотреть понятие *связывания* (*binding*).

Связывание «метод-вызов»

Присоединение вызова метода к телу метода называется *связыванием*. Если связывание проводится перед запуском программы (компилятором и компоновщиком, если

он есть), оно называется *ранним связыванием* (early binding). Возможно, ранее вам не приходилось слышать этот термин, потому что в процедурных языках никакого выбора связывания не было. Компиляторы C поддерживают только один тип вызова — раннее связывание.

Неоднозначность предыдущей программы кроется именно в раннем связывании: компилятор не может знать, какой метод нужно вызывать, когда у него есть только ссылка на объект `Instrument`.

Проблема решается благодаря *позднему связыванию* (late binding), то есть связыванию, проводимому во время выполнения программы, в зависимости от типа объекта. Позднее связывание также называют *динамическим* (dynamic) или *связыванием на стадии выполнения* (runtime binding). В языках, реализующих позднее связывание, должен существовать механизм определения фактического типа объекта во время работы программы для вызова подходящего метода. Иначе говоря, компилятор не знает типа объекта, но механизм вызова методов определяет его и вызывает соответствующее тело метода. Механизм позднего связывания зависит от конкретного языка, но нетрудно предположить, что для его реализации в объекты должна включаться какая-то дополнительная информация.

Для всех методов Java используется механизм позднего связывания, если только метод не был объявлен как `final` (приватные методы являются `final` по умолчанию). Следовательно, вам не придется принимать решений относительно использования позднего связывания — оно осуществляется автоматически.

Зачем объявлять метод как `final`? Как уже было замечено в предыдущей главе, это запрещает переопределение соответствующего метода. Что еще важнее, это фактически «отключает» позднее связывание или, скорее, указывает компилятору на то, что позднее связывание не является необходимым. Поэтому для методов `final` компилятор генерирует чуть более эффективный код. Впрочем, в большинстве случаев влияние на производительность вашей программы незначительно, поэтому `final` лучше использовать в качестве продуманного элемента своего проекта, а не как средство улучшения производительности.

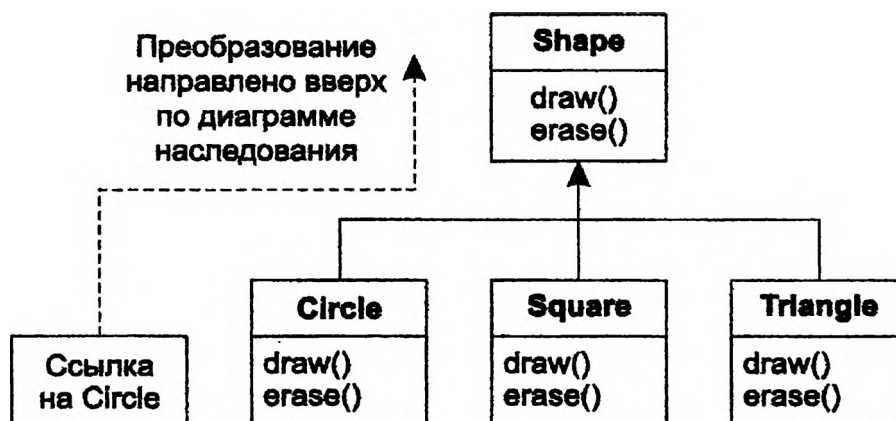
Получение нужного результата

Теперь, когда вы знаете, что связывание всех методов в Java осуществляется полиморфно, через позднее связывание, вы можете писать код для базового класса, не сомневаясь в том, что для всех производных классов он также будет работать верно. Другими словами, вы «посылаете сообщение объекту и позволяете ему решить, что следует делать дальше».

Классическим примером полиморфизма в ООП является пример с геометрическими фигурами. Он часто используется благодаря своей наглядности, но, к сожалению, некоторые новички начинают думать, что ООП подразумевает графическое программирование, а это, конечно же, неверно.

В примере с фигурами имеется базовый класс с именем `Shape` (фигура) и различные производные типы: `Circle` (окружность), `Square` (прямоугольник), `Triangle` (треугольник) и т. п. Выражения типа «окружность есть фигура» очевидны и не представляют

трудностей для понимания. Взаимосвязи показаны на следующей диаграмме наследования.



Восходящее преобразование имеет место даже в такой простой команде:

```
Shape s = new Circle();
```

Здесь создается объект `Circle`, и полученная ссылка немедленно присваивается типу `Shape`. На первый взгляд это может показаться ошибкой (присвоение одного типа другому), но в действительности все правильно, потому что тип `Circle` (окружность) является типом `Shape` (фигура) посредством наследования. Компилятор принимает команду и не выдает сообщения об ошибке.

Предположим, вызывается один из методов базового класса (из тех, что были переопределены в производных классах):

```
s.draw();
```

Опять можно подумать, что вызывается метод `draw()` из класса `Shape`, раз имеется ссылка на объект `Shape` — как компилятор может сделать что-то другое? И все же будет вызван правильный метод `Circle.draw()`, так как в программе используется позднее связывание (полиморфизм).

Следующий пример показывает несколько другой подход. Начнем с создания библиотеки типов `Shape`:

```
///  
package polymorphism.shape;
```

```
public class Shape {  
    public void draw() {}  
    public void erase() {}  
} ///:~
```

```
///  
package polymorphism.shape;  
import static net.mindview.util.Print.*;
```

```
public class Circle extends Shape {  
    public void draw() { print("Circle.draw()"); }  
    public void erase() { print("Circle.erase()"); }  
} ///:~
```

```
//: polymorphism/shape/Square.java
package polymorphism.shape;
import static net.mindview.util.Print.*;

public class Square extends Shape {
    public void draw() { print("Square.draw()"); }
    public void erase() { print("Square.erase()"); }
} ///:~

//: polymorphism/shape/Triangle.java
package polymorphism.shape;
import static net.mindview.util.Print.*;

public class Triangle extends Shape {
    public void draw() { print("Triangle.draw()"); }
    public void erase() { print("Triangle.erase()"); }
} ///:~

//: polymorphism/shape/RandomShapeGenerator.java
// "Фабрика", случайным образом создающая объекты:
package polymorphism.shape;
import java.util.*;

public class RandomShapeGenerator {
    private Random rand = new Random(47);
    public Shape next() {
        switch(rand.nextInt(3)) {
            default:
            case 0: return new Circle();
            case 1: return new Square();
            case 2: return new Triangle();
        }
    }
} ///:~

//: polymorphism/Shapes.java
// Polymorphism in Java.
import polymorphism.shape.*;

public class Shapes {
    private static RandomShapeGenerator gen =
        new RandomShapeGenerator();
    public static void main(String[] args) {
        Shape[] s = new Shape[9];
        // Заполняем массив фигурами:
        for(int i = 0; i < s.length; i++)
            s[i] = gen.next();
        // Полиморфные вызовы методов:
        for(Shape shp : s)
            shp.draw();
    }
} /* Output:
Triangle.draw()
Triangle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Triangle.draw()
*/
```

```

Square.draw()
Triangle.draw()
Circle.draw()
*///:~

```

Базовый класс `Shape` устанавливает общий интерфейс для всех классов, производных от `Shape`, — то есть любую фигуру можно нарисовать (`draw()`) и стереть (`erase()`). Производные классы переопределяют этот интерфейс, чтобы реализовать уникальное поведение для каждой конкретной фигуры.

Класс `RandomShapeGenerator` — своего рода «фабрика», при каждом вызове метода `next()` производящая ссылку на случайно выбираемый объект `Shape`. Заметьте, что восходящее преобразование выполняется в командах `return`, каждая из которых получает ссылку на объект `Circle`, `Square` или `Triangle`, а выдает ее за пределы `next()` в виде возвращаемого типа `Shape`. Таким образом, при вызове этого метода вы не сможете определить конкретный тип объекта, поскольку всегда получаете просто `Shape`.

Метод `main()` содержит массив ссылок на `Shape`, который заполняется последовательными вызовами `RandomShapeGenerator.next()`. К этому моменту вам известно, что имеются объекты `Shape`, но вы не знаете об этих объектах ничего конкретного (так же, как и компилятор). Но если перебрать содержимое массива и вызвать `draw()` для каждого его элемента, то как по волшебству произойдет верное, свойственное для определенного типа действие — в этом нетрудно убедиться, взглянув на результат работы программы.

Случайный выбор фигур в нашем примере всего лишь помогает понять, что компилятор во время компиляции кода не располагает информацией о том, какую реализацию следует вызывать. Все вызовы метода `draw()` проводятся с применением позднего связывания.

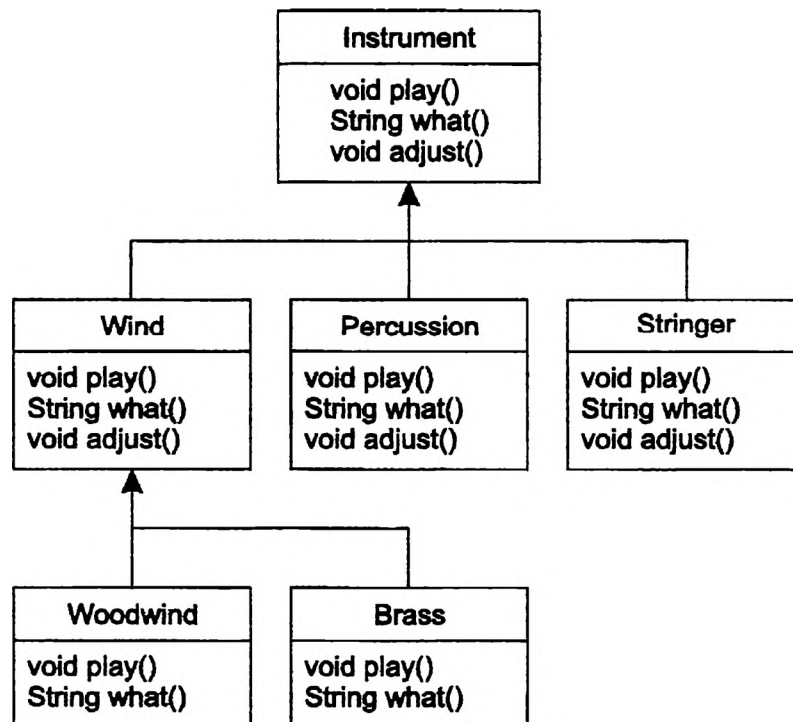
2. (1) Добавьте аннотацию `@Override` в пример с фигурами.
3. (1) Включите в базовый класс `Shapes.java` новый метод, выводящий сообщение, но не переопределяйте его в производных классах. Объясните результат. Переопределите его в одном из производных классов и посмотрите, что происходит. Наконец, переопределите метод во всех производных классах.
4. (2) Добавьте новый подтип `Shape` к программе `Shapes.java` и проверьте на методе `main()`, что полиморфизм работает правильно для вашего нового типа, так же как и для старых типов.
5. (1) В упражнении 1 добавьте в класс `Cycle` метод `wheels()`, возвращающий количество колес каждого транспортного средства. Измените метод `ride()` так, чтобы он вызывал `wheels()`, и убедитесь в том, что полиморфизм успешно работает.

Расширяемость

Теперь вернемся к программе `Music.java`. Благодаря полиморфизму вы можете добавить в нее сколько угодно новых типов, не изменяя метод `tune()`. В хорошо спроектированной ООП-программе большая часть ваших методов (или даже все методы) следуют модели метода `tune()`, оперируя только с интерфейсом базового класса. Такая программа является *расширяемой*, поскольку в нее можно добавить дополнительную функциональность, определяя новые типы данных от общего базового класса. Методы,

работающие на уровне интерфейса базового класса, совсем не нужно изменять, чтобы приспособить их к новым классам.

Давайте возьмем пример с объектами `Instrument` и включим дополнительные методы в базовый класс, а также определим несколько новых классов. Вот диаграмма.



Все новые классы правильно работают со старым, неизмененным методом `tune()`. Даже если метод `tune()` находится в другом файле, а к классу `Instrument` присоединяются новые методы, он все равно будет работать верно без повторной компиляции. Ниже приведена реализация рассмотренной диаграммы:

```

//: polymorphism/music3/Music3.java
// Расширяемая программа.
package polymorphism.music3;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

class Instrument {
    void play(Note n) { print("Instrument.play() " + n); }
    String what() { return "Instrument"; }
    void adjust() { print("Adjusting Instrument"); }
}

class Wind extends Instrument {
    void play(Note n) { print("Wind.play() " + n); }
    String what() { return "Wind"; }
    void adjust() { print("Adjusting Wind"); }
}

class Percussion extends Instrument {
    void play(Note n) { print("Percussion.play() " + n); }
    String what() { return "Percussion"; }
    void adjust() { print("Adjusting Percussion"); }
}
  
```

продолжение ➤

```

class Stringed extends Instrument {
    void play(Note n) { print("Stringed.play() " + n); }
    String what() { return "Stringed"; }
    void adjust() { print("Adjusting Stringed"); }
}

class Brass extends Wind {
    void play(Note n) { print("Brass.play() " + n); }
    void adjust() { print("Adjusting Brass"); }
}

class Woodwind extends Wind {
    void play(Note n) { print("Woodwind.play() " + n); }
    String what() { return "Woodwind"; }
}

public class Music3 {
    // Работа метода не зависит от фактического типа объекта,
    // поэтому типы, добавленные в систему, будут работать правильно:
    public static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
    public static void tuneAll(Instrument[] e) {
        for(Instrument i : e)
            tune(i);
    }
    public static void main(String[] args) {
        // Восходящее преобразование при добавлении в массив:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
        tuneAll(orchestra);
    }
} /* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*///:~

```

Новый метод `what()` возвращает строку (`String`) с информацией о классе, а метод `adjust()` предназначен для настройки инструментов.

В методе `main()` сохранение любого объекта в массиве `orchestra` автоматически приводит к выполнению восходящего преобразования к типу `Instrument`.

Вы можете видеть, что метод `tune()` изолирован от окружающих изменений кода, но при этом все равно работает правильно. Для достижения такой функциональности и используется полиморфизм. Изменения в коде не затрагивают те части программы, которые не зависят от них. Другими словами, полиморфизм помогает отделить «изменяемое от неизменного».

6. (1) Измените программу `Music3.java` так, чтобы метод `what()` стал методом корневого класса `Object` `toString()`. Попробуйте вывести информацию об объектах `Instrument` вызовом `System.out.println()` (без использования преобразований).
7. (2) Добавьте новый подтип `Instrument` в программу `Music3.java`. Убедитесь в том, что полиморфизм работает правильно и для этого нового типа.
8. (2) Измените программу `Music3.java`, чтобы в ней случайным образом генерировались объекты `Instrument`, как в программе `Shapes.java`.
9. (3) Создайте иерархию наследования, используя в качестве основы различные типы грызунов. Базовым классом станет `Rodent` (грызун), а производными классами будут `Mouse` (мышь), `Hamster` (хомяк) и т. п. В базовом классе определите несколько общих методов, которые затем переопределите в производных классах, для того чтобы они производили действие, свойственное конкретному типу объекта. Создайте массив из объектов `Rodent`, заполните его различными производными типами и вызовите методы базового класса, чтобы увидеть результат работы программы.
10. (3) Создайте базовый класс с двумя методами. В первом методе вызовите второй метод. Определите производный класс и переопределите второй метод. Создайте объект производного класса, выполните восходящее преобразование к базовому типу и вызовите первый метод. Объясните результат.

Проблема: «переопределение» закрытых методов

Перед вами одна из ошибок, совершаемых по наивности:

```
//: polymorphism/PrivateOverride.java
// Попытка переопределения приватного метода
package polymorphism;
import static net.mindview.util.Print.*;

public class PrivateOverride {
    private void f() { print("private f()"); }
    public static void main(String[] args) {
        PrivateOverride po = new Derived();
        po.f();
    }
}

class Derived extends PrivateOverride {
    public void f() { print("public f()"); }
} /* Output:
private f()
*///:~
```

Вполне естественно было бы ожидать, что программа выведет сообщение `public f()`, но закрытый (`private`) метод автоматически является неизменным (`final`), а заодно и скрытым от производного класса. Так что метод `f()` класса `Derived` в нашем случае является полностью новым — он даже не был перегружен, так как метод `f()` базового класса классу `Derived` недоступен.

Из этого можно сделать вывод, что переопределяются только методы, не являющиеся закрытыми. Будьте внимательны: компилятор в подобных ситуациях не выдает

сообщений об ошибке, но и не делает того, что вы от него ожидаете. Иными словами, методам производного класса следует присваивать имена, отличные от имен закрытых методов базового класса.

Проблема: поля и статические методы

После первого знакомства с полиморфизмом создается впечатление, что все обращения в программе осуществляются полиморфно. Тем не менее полиморфизм поддерживает только для обычных вызовов методов. Например, прямое обращение к полю будет обработано на стадии компиляции, как видно из следующего примера¹:

```
//: polymorphism/FieldAccess.java
// Прямое обращение к полю разрешается во время компиляции.

class Super {
    public int field = 0;
    public int getField() { return field; }
}

class Sub extends Super {
    public int field = 1;
    public int getField() { return field; }
    public int getSuperField() { return super.field; }
}

public class FieldAccess {
    public static void main(String[] args) {
        Super sup = new Sub(); // Upcast
        System.out.println("sup.field = " + sup.field +
            ", sup.getField() = " + sup.getField());
        Sub sub = new Sub();
        System.out.println("sub.field = " +
            sub.field + ", sub.getField() = " +
            sub.getField() +
            ", sub.getSuperField() = " +
            sub.getSuperField());
    }
} /* Output:
sup.field = 0, sup.getField() = 1
sub.field = 1, sub.getField() = 1, sub.getSuperField() = 0
*///:~
```

При восходящем преобразовании объекта `Sub` в ссылку на `Super` все обращения к полям разрешаются компилятором, и это поведение не является полиморфным. В этом примере для `Super.field` и `Sub.field` выделяются разные области памяти. Таким образом, `Sub` фактически содержит два поля с именем `field`: собственное и унаследованное от `Super`. При этом версия `Super` не используется по умолчанию при обращении к полю в `Sub`; чтобы получить доступ к полю из `Super`, необходимо использовать явную запись `super.field`.

Впрочем, эта проблема почти никогда не возникает на практике. Во-первых, обычно все поля объявляются закрытыми, а обращения к ним осуществляются не напрямую,

¹ Спасибо Рэнди Николсу за заданный вопрос.

а только в виде побочного эффекта от вызова методов. Кроме того, использовать одно имя для поля базового и производного класса вообще не рекомендуется, потому что это создает путаницу.

Статические методы не поддерживают полиморфного поведения:

```
//: polymorphism/StaticPolymorphism.java
// Статические методы не являются полиморфными.

class StaticSuper {
    public static String staticGet() {
        return "Базовая версия staticGet()";
    }
    public String dynamicGet() {
        return "Базовая версия dynamicGet()";
    }
}

class StaticSub extends StaticSuper {
    public static String staticGet() {
        return "Производная версия staticGet()";
    }
    public String dynamicGet() {
        return "Производная версия dynamicGet()";
    }
}

public class StaticPolymorphism {
    public static void main(String[] args) {
        StaticSuper sup = new StaticSub(); // Восходящее преобразование
        System.out.println(sup.staticGet());
        System.out.println(sup.dynamicGet());
    }
} /* Output:
Базовая версия staticGet()
Производная версия dynamicGet()
*///:~
```

Статические методы существуют на уровне класса, а не на уровне отдельных объектов.

Конструкторы и полиморфизм

Конструкторы отличаются от обычных методов, и эти отличия проявляются и при использовании полиморфизма. Хотя конструкторы сами по себе не полиморфны (фактически они представляют собой статические методы, только ключевое слово `static` опущено), вы должны хорошо понимать, как работают конструкторы в сложных полиморфных иерархиях. Такое понимание в дальнейшем поможет избежать некоторых затруднительных ситуаций.

Порядок вызова конструкторов

Порядок вызова конструкторов коротко обсуждался в главах 5 и 7, но в то время мы еще не рассматривали полиморфизм.

Конструктор базового класса всегда вызывается в процессе конструирования производного класса. Вызов автоматически проходит вверх по цепочке наследования, так что в конечном итоге вызываются конструкторы всех базовых классов по всей цепочке наследования. Это очень важно, поскольку конструктору отводится особая роль — обеспечивать правильное построение объектов. Производный класс обычно имеет доступ только к своим членам, но не к членам базового класса (которые чаще всего объявляются со спецификатором `private`). Только конструктор базового класса обладает необходимыми знаниями и правами доступа, чтобы правильно инициализировать свои внутренние элементы. Именно поэтому компилятор настаивает на вызове конструктора для любой части производного класса. Он незаметно подставит конструктор по умолчанию, если вы явно не вызовете конструктор базового класса в теле конструктора производного класса. Если конструктора по умолчанию не существует, компилятор сообщит об этом. (Если у класса вообще нет пользовательских конструкторов, компилятор автоматически генерирует конструктор по умолчанию.)

Следующий пример показывает, как композиция, наследование и полиморфизм влияют на порядок конструирования:

```
//: polymorphism/Sandwich.java
// Порядок вызова конструкторов.
package polymorphism;
import static net.mindview.util.Print.*;

class Meal {
    Meal() { print("Meal()"); }
}

class Bread {
    Bread() { print("Bread()"); }
}

class Cheese {
    Cheese() { print("Cheese()"); }
}

class Lettuce {
    Lettuce() { print("Lettuce()"); }
}

class Lunch extends Meal {
    Lunch() { print("Lunch()"); }
}

class PortableLunch extends Lunch {
    PortableLunch() { print("PortableLunch()"); }
}

public class Sandwich extends PortableLunch {
    private Bread b = new Bread();
    private Cheese c = new Cheese();
    private Lettuce l = new Lettuce();
    public Sandwich() { print("Sandwich()"); }
    public static void main(String[] args) {
        new Sandwich();
    }
}
```

```
} /* Output:  
Meal()  
Lunch()  
PortableLunch()  
Bread()  
Cheese()  
Lettuce()  
Sandwich()  
*///:~
```

В этом примере создается сложный класс, собранный из других классов, и в каждом классе имеется конструктор, который сообщает о своем выполнении. Самый важный класс — `Sandwich`, с тремя уровнями наследования (четырьмя, если считать неявное наследование от класса `Object`) и тремя встроенными объектами. Результат виден при создании объекта `Sandwich` в методе `main()`. Это значит, что конструкторы для сложного объекта вызываются в такой последовательности:

- ❑ Сначала вызывается конструктор базового класса. Этот шаг повторяется рекурсивно: сначала конструируется корень иерархии, затем следующий за ним класс, затем следующий за этим классом класс и т. д., пока не достигается «низший» производный класс.
- ❑ Проводится инициализация членов класса в порядке их объявления.
- ❑ Вызывается тело конструктора производного класса.

Порядок вызова конструкторов немаловажен. При наследовании вы располагаете полной информацией о базовом классе и можете получить доступ к любому из его открытых (`public`) или защищенных (`protected`) членов. Следовательно, при этом подразумевается, что все члены базового класса являются действительными в производном классе. При вызове нормального метода известно, что конструирование уже было проведено, поэтому все части объекта инициализированы. Однако в конструкторе вы также должны быть уверены в том, что все используемые члены уже проинициализированы. Это можно гарантировать только одним способом — сначала вызывать конструктор базового класса. В дальнейшем при выполнении конструктора производного класса можно быть уверенным в том, что все члены базового класса уже инициализированы. Гарантия действительности всех членов в конструкторе — важная причина, по которой все встроенные объекты (то есть объекты, помещенные в класс посредством композиции) инициализируются на месте их определения (как в рассмотренном примере сделано с объектами `b`, `c` и `l`). Если вы будете следовать этому правилу, это усилит уверенность в том, что все члены базового класса и объекты-члены были проинициализированы. К сожалению, это помогает не всегда, в чем вы убедитесь в следующем разделе.

11. (1) Включите класс `Pickle` в программу `Sandwich.java`.

Наследование и завершающие действия

Если при создании нового класса используется композиция и наследование, то вам не придется беспокоиться о проведении завершающих действий — подобъекты уничтожаются уборщиком мусора. Но если вам необходимо провести завершающие действия,

создайте в своем классе метод `dispose()` (в данном разделе я решил использовать такое имя; возможно, вы придумаете более удачное название). Переопределяя метод `dispose()` в производном классе, важно помнить о вызове версии этого метода из базового класса, поскольку иначе не будут выполнены завершающие действия базового класса. Следующий пример доказывает справедливость этого утверждения.

```

//: polymorphism/Frog.java
// Наследование и завершающие действия.
package polymorphism;
import static net.mindview.util.Print.*;

class Characteristic {
    private String s;
    Characteristic(String s) {
        this.s = s;
        print("Создаем Characteristic " + s);
    }
    protected void dispose() {
        print("Завершаем Characteristic " + s);
    }
}

class Description {
    private String s;
    Description(String s) {
        this.s = s;
        print("Создаем Description " + s);
    }
    protected void dispose() {
        print("Завершаем Description " + s);
    }
}

// живое существо
class LivingCreature {
    private Characteristic p =
        new Characteristic("живое существо");
    private Description t =
        new Description("обычное живое существо");
    LivingCreature() {
        print("LivingCreature()");
    }
    protected void dispose() {
        print("dispose() в LivingCreature ");
        t.dispose();
        p.dispose();
    }
}

// животное
class Animal extends LivingCreature {
    private Characteristic p =
        new Characteristic("имеет сердце");
    private Description t =
        new Description("животное, не растение");
    Animal() { print("Animal()"); }
    protected void dispose() {

```

```
        print("dispose() в Animal ");
        t.dispose();
        p.dispose();
        super.dispose();
    }
}

// земноводное
class Amphibian extends Animal {
    private Characteristic p =
        new Characteristic("может жить в воде");
    private Description t =
        new Description("и в воде, и на земле");
    Amphibian() {
        print("Amphibian()");
    }
    protected void dispose() {
        print("dispose() в Amphibian ");
        t.dispose();
        p.dispose();
        super.dispose();
    }
}

// лягушка
public class Frog extends Amphibian {
    private Characteristic p = new Characteristic("квакает");
    private Description t = new Description("ест жуков");
    public Frog() { print("Frog()"); }
    protected void dispose() {
        print("завершение Frog");
        t.dispose();
        p.dispose();
        super.dispose();
    }
    public static void main(String[] args) {
        Frog frog = new Frog();
        print("Пока!");
        frog.dispose();
    }
} /* Output:
Создаем Characteristic живое существо
Создаем Description обычное живое существо
LivingCreature()
Создаем Characteristic имеет сердце
Создаем Description животное, не растение
Animal()
Создаем Characteristic может жить в воде
Создаем Description и в воде, и на земле
Amphibian()
Создаем Characteristic квакает
Создаем Description ест жуков
Frog()
Пока!
завершение Frog
Завершаем Description ест жуков
Завершаем Characteristic квакает
dispose() в Amphibian
```

```

Завершаем Description и в воде, и на земле
Завершаем Characteristic может жить в воде
dispose() в Animal
Завершаем Description животное, не растение
Завершаем Characteristic имеет сердце
dispose() в LivingCreature
Завершаем Description обычное живое существо
Завершаем Characteristic живое существо
*///:~

```

Каждый класс в иерархии содержит объекты классов `Characteristic` и `Description`, которые также необходимо «завершать». Очередность завершения должна быть обратной порядку инициализации в том случае, если объекты зависят друг от друга. Для полей это означает порядок, обратный последовательности объявления полей в классе (инициализация соответствует порядку объявления). В базовых классах сначала следует выполнять финализацию для производного класса, а затем для базового класса. Это объясняется тем, что завершающий метод производного класса может вызывать некоторые методы базового класса, для которых необходимы действительные компоненты базового класса. Из результатов работы программы видно, что все части объекта `Frog` будут финализированы в порядке, противоположном очередности их создания.

12. (3) Измените упражнение 9 так, чтобы программа демонстрировала порядок инициализации базовых и производных классов. Включите встроенные объекты и в базовые, и в производные классы, и покажите, в каком порядке проходит их инициализация при конструировании объекта.
13. Также обратите внимание на то, что в описанном примере объект `Frog` является «владельцем» встроенных объектов. Он создает их, определяет продолжительность их существования (до тех пор, пока существует `Frog`) и знает, когда вызывать `dispose()` для встроенных объектов. Но если встроенный объект используется совместно с другими объектами, ситуация усложняется, и вы уже не можете просто вызвать `dispose()`. В таких случаях для отслеживания количества объектов, работающих со встроенным объектом, приходится использовать *подсчет ссылок*. Вот как это выглядит:

```

//: polymorphism/ReferenceCounting.java
// Уничтожение совместно используемых встроенных объектов
import static net.mindview.util.Print.*;

class Shared {
    private int refcount = 0;
    private static long counter = 0;
    private final long id = counter++;
    public Shared() {
        print("Создаем " + this);
    }
    public void addRef() { refcount++; }
    protected void dispose() {
        if(--refcount == 0)
            print("Disposing " + this);
    }
    public String toString() { return "Shared " + id; }
}

```



```
class Composing {
    private Shared shared;
    private static long counter = 0;
    private final long id = counter++;
    public Composing(Shared shared) {
        print("Создаем " + this);
        this.shared = shared;
        this.shared.addRef();
    }
    protected void dispose() {
        print("disposing " + this);
        shared.dispose();
    }
    public String toString() { return "Composing " + id; }
}

public class ReferenceCounting {
    public static void main(String[] args) {
        Shared shared = new Shared();
        Composing[] composing = { new Composing(shared),
            new Composing(shared), new Composing(shared),
            new Composing(shared), new Composing(shared) };
        for(Composing c : composing)
            c.dispose();
    }
} /* Output:
Создаем Shared 0
Создаем Composing 0
Создаем Composing 1
Создаем Composing 2
Создаем Composing 3
Создаем Composing 4
уничтожаем Composing 0
уничтожаем Composing 1
уничтожаем Composing 2
уничтожаем Composing 3
уничтожаем Composing 4
уничтожаем Shared 0
*///:~
```

В переменной `static long counter` хранится количество созданных экземпляров `Shared`. Для счетчика выбран тип `long` вместо `int` для того, чтобы предотвратить переполнение (это всего лишь хороший стиль программирования; в рассматриваемых примерах переполнение вряд ли возможно). Поле `id` объявлено со спецификатором `final`, поскольку его значение остается постоянным на протяжении жизненного цикла объекта.

Присоединяя к классу общий объект, необходимо вызвать `addRef()`, но метод `dispose()` будет следить за состоянием счетчика ссылок и сам решит, когда нужно выполнить завершающие действия. Подсчет ссылок требует дополнительных усилий со стороны программиста, но при совместном использовании объектов, требующих завершения, у вас нет особого выбора.

14. (3) Включите метод `finalize()` в `ReferenceCounting.java`, чтобы проверить условие завершения (см. главу 5).
15. (4) Измените упражнение 12 так, чтобы один из встроенных объектов был общим и для него использовался подсчет ссылок. Покажите, что он правильно работает.

Поведение полиморфных методов при вызове из конструкторов

В иерархиях конструкторов возникает интересный вопрос. Что произойдет, если вызвать в конструкторе динамически связываемый метод конструируемого объекта?

В обычных методах представить происходящее нетрудно — динамически связываемый вызов обрабатывается во время выполнения, так как объект не знает, принадлежит ли этот вызов классу, в котором определен метод, или классу, производному от этого класса. Казалось бы, то же самое должно происходить и в конструкторах.

Но ничего подобного. При вызове динамически связываемого метода в конструкторе используется переопределенное описание этого метода. Однако последствия такого вызова могут быть весьма неожиданными, и здесь могут крыться некоторые коварные ошибки.

По определению задача конструктора — дать объекту жизнь (и это отнюдь не простая задача). Внутри любого конструктора объект может быть сформирован лишь частично — известно лишь то, что объекты базового класса были проинициализированы. Если конструктор является лишь очередным шагом на пути построения объекта класса, производного от класса данного конструктора, «производные» части еще не были инициализированы на момент вызова текущего конструктора. Однако динамически связываемый вызов может перейти во «внешнюю» часть иерархии, то есть к производным классам. Если он вызовет метод производного класса в конструкторе, это может привести к манипуляциям с неинициализированными данными, а это наверняка приведет к катастрофе.

Следующий пример поясняет суть проблемы:

```
//: polymorphism/PolyConstructors.java
// Конструкторы и полиморфизм дают не тот
// результат, который можно было бы ожидать.
import static net.mindview.util.Print.*;

class Glyph {
    void draw() { print("Glyph.draw()"); }
    Glyph() {
        print("Glyph() перед вызовом draw()");
        draw();
        print("Glyph() после вызова draw()");
    }
}

class RoundGlyph extends Glyph {
    private int radius = 1;
    RoundGlyph(int r) {
        radius = r;
        print("RoundGlyph.RoundGlyph(), radius = " + radius);
    }
    void draw() {
        print("RoundGlyph.draw(), radius = " + radius);
    }
}
```

```
public class PolyConstructors {
    public static void main(String[] args) {
        new RoundGlyph(5);
    }
} /* Output:
Glyph() перед вызовом draw()
RoundGlyph.draw(), radius = 0
Glyph() после вызова draw()
RoundGlyph.RoundGlyph(), radius = 5
*///:~
```

Метод `Glyph.draw()` изначально предназначен для переопределения в производных классах, что и происходит в `RoundGlyph`. Но конструктор `Glyph` вызывает этот метод, и в результате это приводит к вызову метода `RoundGlyph.draw()`, что вроде бы и предполагалось. Но из результатов работы программы видно — когда конструктор класса `Glyph` вызывает метод `draw()`, переменной `radius` еще не присвоено даже значение по умолчанию 1. Переменная равна 0. В итоге класс может не выполнить свою задачу, а вам придется долго всматриваться в код программы, чтобы определить причину неверного результата.

Порядок инициализации, описанный в предыдущем разделе, немного неполон, и именно здесь кроется ключ к этой загадке. На самом деле процесс инициализации проходит следующим образом.

- ❑ Память, выделенная под новый объект, заполняется двоичными нулями.
- ❑ Конструкторы базовых классов вызываются в описанном ранее порядке. В этот момент вызывается переопределенный метод `draw()` (да, *перед* вызовом конструктора класса `RoundGlyph`), где обнаруживается, что переменная `radius` равна нулю из-за первого этапа.
- ❑ Вызываются инициализаторы членов класса в порядке их определения.
- ❑ Исполняется тело конструктора производного класса.

У происходящего есть и положительная сторона — по крайней мере, данные инициализируются нулями (или тем, что понимается под нулевым значением для определенного типа данных), а не случайным «мусором» в памяти. Это относится и к ссылкам на объекты, внедренные в класс с помощью композиции. Они принимают особое значение `null`. Если вы забудете инициализировать такую ссылку, то получите исключение во время выполнения программы. Остальные данные заполняются нулями, а это обычно легко заметить по выходным данным программы.

С другой стороны, результат программы выглядит довольно жутко. Вроде бы все логично, а программа ведет себя загадочно и некорректно без малейших объяснений со стороны компилятора. (В языке C++ такие ситуации обрабатываются более рациональным способом.) Поиск подобных ошибок занимает много времени.

При написании конструктора руководствуйтесь следующим правилом: «Не пытайтесь сделать больше для того, чтобы привести объект в нужное состояние, и по возможности избегайте вызова каких-либо методов». Единственные методы, которые можно вызывать в конструкторе без опаски, — неизменные (`final`) методы базового класса. (Сказанное относится и к закрытым (`private`) методам, поскольку они автоматически

являются неизменными.) Такие методы невозможно переопределить, и поэтому они застрахованы от «сюрпризов».

- 16.** (2) Включите класс `RectangularGlyph` в `PolyConstructors.java`. Продемонстрируйте проблему, описанную в этом разделе.

Ковариантность возвращаемых типов

В Java SE5 появилась концепция *ковариантности возвращаемых типов*; этот термин означает, что переопределенный метод производного класса может вернуть тип, производный от типа, возвращаемого методом базового класса:

```
//: polymorphism/CovariantReturn.java

class Grain {
    public String toString() { return "Grain"; }
}

class Wheat extends Grain {
    public String toString() { return "Wheat"; }
}

class Mill {
    Grain process() { return new Grain(); }
}

class WheatMill extends Mill {
    Wheat process() { return new Wheat(); }
}

public class CovariantReturn {
    public static void main(String[] args) {
        Mill m = new Mill();
        Grain g = m.process();
        System.out.println(g);
        m = new WheatMill();
        g = m.process();
        System.out.println(g);
    }
} /* Output:
Grain
Wheat
*///:~
```

Главное отличие Java SE5 от предыдущих версий Java заключается в том, что старые версии заставляли переопределение `process()` возвращать `Grain` вместо `Wheat`, хотя тип `Wheat`, производный от `Grain`, является допустимым возвращаемым типом. Ковариантность возвращаемых типов позволяет вернуть более специализированный тип `Wheat`.

Наследование при проектировании

После знакомства с полиморфизмом может показаться, что его следует применять везде и всегда. Однако злоупотребление полиморфизмом ухудшит архитектуру ваших приложений.

Лучше для начала использовать композицию, если пока вы точно не уверены в том, какой именно механизм следует выбрать. Композиция не стесняет разработку рамками иерархии наследования. К тому же механизм композиции более гибок, так как он позволяет динамически выбирать тип (а следовательно, и поведение), тогда как наследование требует, чтобы точный тип был известен уже во время компиляции. Следующий пример демонстрирует это:

```
//: polymorphism/Transmoglify.java
// Динамическое изменение поведения объекта
// с помощью композиции (паттерн проектирования "Состояние")
import static net.mindview.util.Print.*;

class Actor {
    public void act() {}
}

class HappyActor extends Actor {
    public void act() { print("HappyActor"); }
}

class SadActor extends Actor {
    public void act() { print("SadActor"); }
}

class Stage {
    private Actor actor = new HappyActor();
    public void change() { actor = new SadActor(); }
    public void performPlay() { actor.act(); }
}

public class Transmoglify {
    public static void main(String[] args) {
        Stage stage = new Stage();
        stage.performPlay();
        stage.change();
        stage.performPlay();
    }
} /* Output:
HappyActor
SadActor
*///:~
```

Объект `Stage` содержит ссылку на объект `Actor`, которая инициализируется объектом `HappyActor`. Это значит, что метод `performPlay()` имеет определенное поведение. Но так как ссылку на объект можно заново присоединить к другому объекту во время выполнения программы, ссылке `actor` назначается объект `SadActor`, и после этого поведение метода `performPlay()` изменяется. Таким образом значительно улучшается динамика поведения на стадии выполнения программы. С другой стороны, переключиться на

другой способ наследования во время работы программы невозможно; иерархия наследования раз и навсегда определяется в процессе компиляции программы.

17. (3) По образцу Transmogrify.java создайте класс Starship со ссылкой на объект AlertStatus, который может обозначать одно из трех состояний. Включите методы для изменения состояния.

Нисходящее преобразование и динамическое определение типов

Так как при проведении *восходящего преобразования* (передвижение вверх по иерархии наследования) теряется информация, характерная для определенного типа, возникает естественное желание восстановить ее с помощью *нисходящего преобразования*. Впрочем, мы знаем, что восходящее преобразование абсолютно безопасно; базовый класс не может иметь «большой» интерфейс, чем производный класс, и поэтому любое сообщение, посланное базовому классу, гарантированно дойдет до получателя. Но при использовании нисходящего преобразования вы не знаете достоверно, что фигура (например) в действительности является окружностью. С такой же вероятностью она может оказаться треугольником, прямоугольником или другим типом.

Должен существовать какой-то механизм, гарантирующий правильность нисходящего преобразования; в противном случае вы можете случайно использовать неверный тип, посылая ему сообщение, которое он не в состоянии принять. Это было бы небезопасно.

В некоторых языках (подобных C++) для проведения безопасного нисходящего преобразования типов необходимо провести специальную операцию, но в Java *каждое преобразование* контролируется! Поэтому хотя внешне все выглядит как обычное приведение типов в круглых скобках, во время выполнения программы это преобразование проходит проверку на фактическое соответствие типу. Если типы не совпадают, происходит исключение ClassCastException. Процесс проверки типов во время выполнения программы называется *динамическим определением типов* (run-time type identification, RTTI). Следующий пример демонстрирует действие RTTI:

```
//: polymorphism/RTTI.java
// Нисходящее преобразование и динамическое определение типов (RTTI).
// {ThrowException}

class Useful {
    public void f() {}
    public void g() {}
}

class MoreUseful extends Useful {
    public void f() {}
    public void g() {}
    public void u() {}
    public void v() {}
    public void w() {}
}

public class RTTI {
    public static void main(String[] args) {
```

```

Useful[] x = {
    new Useful(),
    new MoreUseful()
};
x[0].f();
x[1].g();
// Стадия компиляции: метод не найден в классе Useful:
//! x[1].u();
((MoreUseful)x[1]).u(); // Нисходящее преобразование /RTTI
((MoreUseful)x[0]).u(); // Происходит исключение
}
} ///:~

```

Класс `MoreUseful` расширяет интерфейс класса `Useful`. Но благодаря наследованию он также может быть преобразован к типу `Useful`. Вы видите, как это происходит, при инициализации массива `x` в методе `main()`. Так как оба объекта в массиве являются производными от `Useful`, вы можете послать сообщения (вызвать методы) `f()` и `g()` для обоих объектов, но при попытке вызова метода `u()` (который существует только в классе `MoreUseful`) вы получите сообщение об ошибке компиляции.

Чтобы получить доступ к расширенному интерфейсу объекта `MoreUseful`, используйте нисходящее преобразование. Если тип указан правильно, все пройдет успешно; иначе произойдет исключение `ClassCastException`. Вам не понадобится писать дополнительный код для этого исключения, поскольку оно указывает на общую ошибку, которая может произойти в любом месте программы. Тег комментария `{ThrowsException}` сообщает системе построения кода, использованной в книге, что при выполнении программы следует ожидать возникновения исключения.

Впрочем, RTTI не сводится к простой проверке преобразований. Например, можно узнать, с каким типом вы имеете дело, *прежде чем* проводить нисходящее преобразование. Глава 11 полностью посвящена изучению различных аспектов динамического определения типов Java.

- 18.** (2) Используя иерархию `Cycle` из упражнения 1, включите метод `balance()` в классы `Unicycle` и `Bicycle`, но не в `Tricycle`. Создайте экземпляры всех трех типов и выполните их восходящее преобразование в массив `Cycle`. Попробуйте вызвать `balance()` для каждого элемента массива. Теперь выполните нисходящее преобразование, вызовите `balance()` и проанализируйте результат.

Резюме

Полиморфизм означает «многообразие форм». В объектно-ориентированном программировании базовый класс предоставляет общий интерфейс, а различные версии динамически связываемых методов — разные формы использования интерфейса.

Как было показано в этой главе, невозможно понять или создать примеры с использованием полиморфизма, не прибегая к абстракции данных и наследованию. Полиморфизм — это возможность языка, которая не может рассматриваться изолированно; она работает только согласованно, как часть «общей картины» взаимоотношений классов.

Чтобы эффективно использовать полиморфизм — а значит, все объектно-ориентированные приемы — в своих программах, необходимо расширить свои представления о программировании, чтобы они охватывали не только члены и сообщения отдельного класса, но и общие аспекты классов, их взаимоотношения. Хотя это потребует значительных усилий, результат стоит того. Наградой станет ускорение разработки программ, улучшение структуры кода, расширяемые программы и сокращение усилий по сопровождению кода.

9

Интерфейсы

Интерфейсы и абстрактные классы улучшают структуру кода и способствуют отделению интерфейса от реализации.

В традиционных языках программирования такие механизмы не получили особого распространения. Например, в C++ существует лишь косвенная поддержка этих концепций. Сам факт их существования в Java показывает, что эти концепции были сочтены достаточно важными для прямой поддержки в языке.

Мы начнем с понятия *абстрактного класса*, который представляет собой своего рода промежуточную ступень между обычным классом и интерфейсом. Абстрактные классы — важный и необходимый инструмент для создания классов, содержащих нереализованные методы. Применение «чистых» интерфейсов возможно не всегда.

Абстрактные классы и методы

В примере с классами музыкальных инструментов из предыдущей главы методы базового класса `Instrument` всегда оставались «фиктивными». Любая попытка вызова такого метода означала, что в программе произошла какая-то ошибка. Это было связано с тем, что класс `Instrument` создавался для определения *общего интерфейса* всех классов, производных от него.

В этих примерах общий интерфейс создавался для единственной цели — его разной реализации в каждом производном типе. Интерфейс определяет базовую форму, общность всех производных классов. Такие классы, как `Instrument`, также называют *абстрактными базовыми классами*, или просто *абстрактными классами*.

Если в программе определяется абстрактный класс вроде `Instrument`, создание объектов такого класса практически всегда бессмысленно. Абстрактный класс создается для работы с набором классов через общий интерфейс. А если `Instrument` только выражает интерфейс, а создание объектов того класса не имеет смысла, вероятно, пользователю лучше запретить создавать такие объекты. Конечно, можно заставить все методы `Instrument` выдавать ошибки, но в этом случае получение информации откладывается до стадии выполнения. Ошибки такого рода лучше обнаруживать во время компиляции.

В языке Java для решения подобных задач применяются *абстрактные методы*¹. Абстрактный метод незавершен; он состоит только из объявления и не имеет тела. Синтаксис объявления абстрактных методов выглядит так:

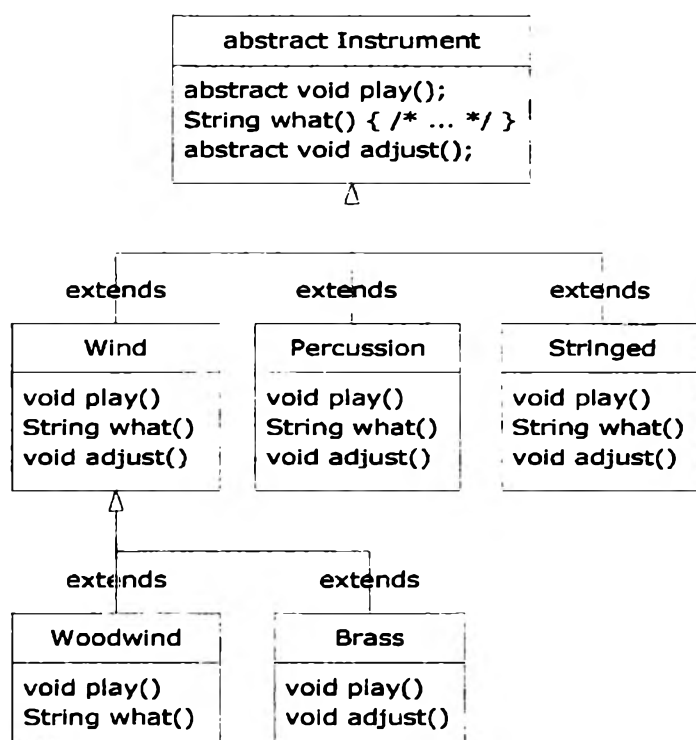
```
abstract void f();
```

Класс, содержащий абстрактные методы, называется *абстрактным классом*. Такие классы тоже должны помечаться ключевым словом `abstract` (в противном случае компилятор выдает сообщение об ошибке).

Если вы объявляете класс, производный от абстрактного класса, но хотите иметь возможность создания объектов нового типа, вам придется предоставить определения для всех абстрактных методов базового класса. Если вы этого не сделаете (возможно, вполне сознательно), производный класс тоже останется абстрактным, и компилятор заставит пометить *новый* класс ключевым словом `abstract`.

Можно создавать класс с ключевым словом `abstract` даже тогда, когда в нем не имеется ни одного абстрактного метода. Это бывает полезно в ситуациях, где в классе абстрактные методы просто не нужны, но необходимо запретить создание экземпляров этого класса.

Класс `Instrument` очень легко можно сделать абстрактным. Только некоторые из его методов станут абстрактными, поскольку объявление класса как `abstract` не подразумевает, что все его методы должны быть абстрактными. Вот что получится.



А вот как выглядит реализация примера оркестра с использованием абстрактных классов и методов:

```
//: interfaces/music4/Music4.java
// Абстрактные классы и методы.
package interfaces.music4;
import polymorphism.music.Note;
```

¹ Аналог *чисто виртуальных методов* языка C++.

```
import static net.mindview.util.Print.*;

abstract class Instrument {
    private int i; // Память выделяется для каждого объекта
    public abstract void play(Note n);
    public String what() { return "Instrument"; }
    public abstract void adjust();
}

class Wind extends Instrument {
    public void play(Note n) {
        print("Wind.play() " + n);
    }
    public String what() { return "Wind"; }
    public void adjust() {}
}

class Percussion extends Instrument {
    public void play(Note n) {
        print("Percussion.play() " + n);
    }
    public String what() { return "Percussion"; }
    public void adjust() {}
}

class Stringed extends Instrument {
    public void play(Note n) {
        print("Stringed.play() " + n);
    }
    public String what() { return "Stringed"; }
    public void adjust() {}
}

class Brass extends Wind {
    public void play(Note n) {
        print("Brass.play() " + n);
    }
    public void adjust() { print("Brass.adjust()"); }
}

class Woodwind extends Wind {
    public void play(Note n) {
        print("Woodwind.play() " + n);
    }
    public String what() { return "Woodwind"; }
}

public class Music4 {
    // Работа метода не зависит от фактического типа объекта,
    // поэтому типы, добавленные в систему, будут работать правильно:
    static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
    static void tuneAll(Instrument[] e) {
        for(Instrument i : e)
            tune(i);
    }
}
```

```

public static void main(String[] args) {
    // Восходящее преобразование при добавлении в массив:
    Instrument[] orchestra = {
        new Wind(),
        new Percussion(),
        new Stringed(),
        new Brass(),
        new Woodwind()
    };
    tuneAll(orchestra);
}
} /* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*///:~

```

Как видите, изменения ограничиваются базовым классом.

Создавать абстрактные классы и методы полезно, так как они подчеркивают абстрактность класса, а также сообщают и пользователю класса, и компилятору, как следует с ним обходиться. Кроме того, абстрактные классы играют полезную роль при переработке программ, потому что они позволяют легко перемещать общие методы вверх по иерархии наследования.

1. (1) Измените упражнение 9 из предыдущей главы так, чтобы класс `Rodent` стал абстрактным (`abstract`) классом. Сделайте некоторые методы класса `Rodent` абстрактными (там, где это оправданно).
2. (1) Создайте класс и объявите его как `abstract`, не включая в него ни одного абстрактного метода. Затем убедитесь, что вы не можете создавать экземпляры этого класса.
3. (2) Создайте базовый класс с определением метода `abstract print()`, переопределяемого производными классами. Переопределенная версия метода выводит значение переменной `int`, определенной в производном классе. В точке определения этой переменной присвойте ей ненулевое значение. Вызовите этот метод в конструкторе базового класса. В методе `main()` создайте объект производного типа, а затем вызовите его метод `print()`. Объясните результат работы программы.
4. (3) Создайте абстрактный (`abstract`) класс без методов. Произведите от него класс и добавьте метод. Создайте статический (`static`) метод, получающий ссылку на базовый класс, проведите нисходящее преобразование к производному типу и вызовите его метод. Продемонстрируйте, что такой способ работает, в методе `main()`. Теперь поместите в определение метода из базового класса ключевое слово `abstract`, и необходимость в нисходящем преобразовании исчезнет.

Интерфейсы

Ключевое слово `interface` становится следующим шагом на пути к абстракции. Ключевое слово `abstract` позволяет создать в классе один или несколько неопределенных

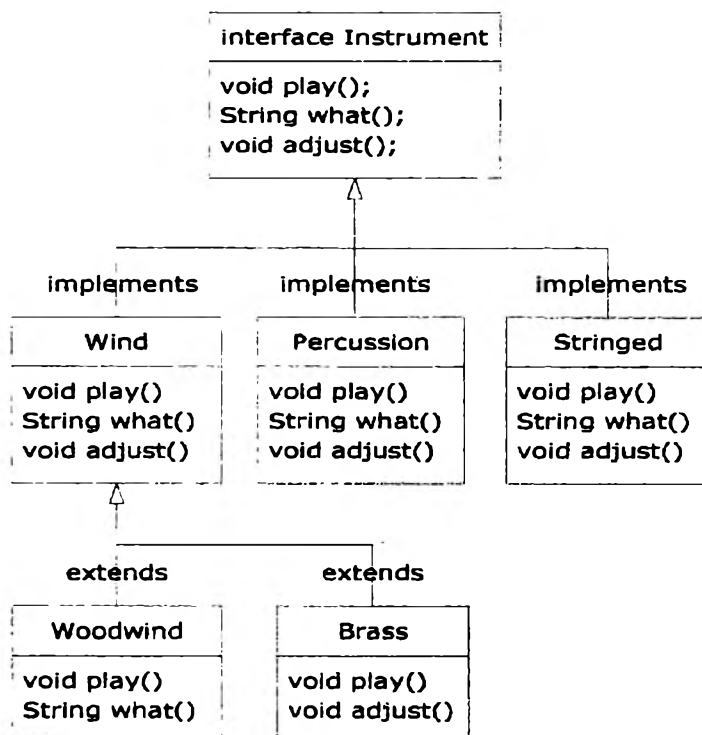
методов — разработчик предоставляет часть интерфейса без соответствующей реализации, которая должна предоставляться производными классами. Ключевое слово `interface` используется для создания полностью абстрактных классов, вообще не имеющих реализации. Создатель интерфейса определяет имена методов, списки аргументов и типы возвращаемых значений, но не тела методов. Интерфейс описывает форму, но не реализацию.

Ключевое слово `interface` фактически означает: «Именно так должны выглядеть все классы, которые *реализуют* данный интерфейс». Таким образом, любой код, использующий конкретный интерфейс, знает только то, какие методы вызываются для этого интерфейса, но не более того. Интерфейс определяет своего рода написание «протокол взаимодействия» между классами. (В некоторых языках программирования существует ключевое слово `protocol`, решающее ту же задачу.)

Однако интерфейс представляет собой нечто большее, чем абстрактный класс в своем крайнем проявлении, потому что он позволяет реализовать подобие «множественного наследования» C++: иначе говоря, создаваемый класс может быть преобразован к нескольким базовым типам.

Чтобы создать интерфейс, используйте ключевое слово `interface` вместо `class`. Как и в случае с классами, вы можете добавить перед словом `interface` спецификатор доступа `public` (но только если интерфейс определен в файле, имеющем то же имя) или оставить для него дружественный доступ, если он будет использоваться только в пределах своего пакета. Интерфейс также может содержать поля, но они автоматически являются статическими (`static`) и неизменными (`final`).

Для создания класса, реализующего определенный интерфейс (или группу интерфейсов), используется ключевое слово `implements`. Фактически оно означает: «Интерфейс лишь определяет форму, а здесь будет показано, как это *работает*». В остальном происходящее выглядит как обычное наследование. Рассмотрим реализацию на примере иерархии классов `Instrument`.



Классы `Woodwind` и `Brass` свидетельствуют, что реализация интерфейса представляет собой обычный класс, от которого можно создавать производные классы.

При описании методов в интерфейсе вы можете явно объявить их открытыми (`public`), хотя они являются таковыми даже без спецификатора. Однако при реализации интерфейса его методы *должны* быть объявлены как `public`. В противном случае будет использоваться доступ в пределах пакета, а это приведет к уменьшению уровня доступа во время наследования, что запрещается компилятором Java.

Все сказанное можно увидеть в следующем примере с объектами `Instrument`. Обратите, что каждый метод интерфейса ограничивается простым объявлением; ничего большего компилятор не разрешит. Вдобавок ни один из методов интерфейса `Instrument` не объявлен со спецификатором `public`, но все методы автоматически являются открытыми:

```
//: interfaces/music5/Music5.java
// Интерфейсы.
package interfaces.music5;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

interface Instrument {
    // Константа времени компиляции:
    int VALUE = 5; // является и static, и final
    // Определения методов недопустимы:
    void play(Note n); // Автоматически объявлен как public
    void adjust();
}

class Wind implements Instrument {
    public void play(Note n) {
        print(this + ".play() " + n);
    }
    public String toString() { return "Wind"; }
    public void adjust() { print(this + ".adjust()"); }
}

class Percussion implements Instrument {
    public void play(Note n) {
        print(this + ".play() " + n);
    }
    public String toString() { return "Percussion"; }
    public void adjust() { print(this + ".adjust()"); }
}

class Stringed implements Instrument {
    public void play(Note n) {
        print(this + ".play() " + n);
    }
    public String toString() { return "Stringed"; }
    public void adjust() { print(this + ".adjust()"); }
}

class Brass extends Wind {
    public String toString() { return "Brass"; }
}
```

```

class Woodwind extends Wind {
    public String toString() { return "Woodwind"; }
}

public class Music5 {
    // Работа метода не зависит от фактического типа объекта,
    // поэтому типы, добавленные в систему, будут работать правильно:
    static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
    static void tuneAll(Instrument[] e) {
        for(Instrument i : e)
            tune(i);
    }
    public static void main(String[] args) {
        // Восходящее преобразование при добавлении в массив:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
        tuneAll(orchestra);
    }
} /* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*///:~

```

В этой версии присутствует еще одно изменение: метод `what()` был заменен на `toString()`. Так как метод `toString()` входит в корневой класс `Object`, его присутствие в интерфейсе не обязательно.

Остальной код работает так же, как прежде. Неважно, проводите ли вы преобразование к «обычному» классу с именем `Instrument`, к абстрактному классу с именем `Instrument` или к интерфейсу с именем `Instrument` — действие будет одинаковым. В методе `tune()` ничто не указывает на то, является ли класс `Instrument` «обычным» или абстрактным, или это вообще не класс, а интерфейс.

5. (2) Создайте интерфейс, содержащий три метода, в отдельном пакете. Реализуйте этот интерфейс в другом пакете.
6. (2) Докажите, что все методы интерфейса автоматически являются открытыми (`public`).
7. (1) Измените упражнение 9 из главы 8 так, чтобы тип `Rodent` был оформлен как интерфейс.
8. (2) В программе `Sandwich.java` из главы 8 создайте интерфейс с именем `FastFood` (с подходящими методами); измените класс `Sandwich` так, чтобы он реализовал этот интерфейс.

9. (3) Переделайте программу `Music5.java`, переместив общие методы `Wind`, `Percussion` и `Stringed` в абстрактный класс.
10. (3) Измените программу `Music5.java`, добавив в нее интерфейс `Playable`. Переместите объявление `play()` из класса `Instrument` в интерфейс `Playable`. Добавьте `Playable` к производным классам, включив его в список `implements`. Измените метод `tune()` так, чтобы в аргументе ему передавался интерфейс `Playable`, а не класс `Instrument`.

Отделение интерфейса от реализации

В любой ситуации, когда метод работает с классом вместо интерфейса, вы ограничены использованием этого класса или его subclasses. Если метод должен быть применен к классу, не входящему в эту иерархию, — значит, вам не повезло. Интерфейсы в значительной мере ослабляют это ограничение. В результате код становится более универсальным, пригодным для повторного использования.

Представьте, что у нас имеется класс `Processor` с методами `name()` и `process()`. Последний получает входные данные, изменяет их и выдает результат. Базовый класс расширяется для создания разных специализированных типов `Processor`. В следующем примере производные типы изменяют объекты `String` (обратите внимание: ковариантными могут быть возвращаемые значения, но не типы аргументов):

```
//: interfaces/classprocessor/Apply.java
package interfaces.classprocessor;
import java.util.*;
import static net.mindview.util.Print.*;

class Processor {
    public String name() {
        return getClass().getSimpleName();
    }
    Object process(Object input) { return input; }
}

class Upcase extends Processor {
    String process(Object input) { // Ковариантный возвращаемый тип
        return ((String)input).toUpperCase();
    }
}

class Downcase extends Processor {
    String process(Object input) {
        return ((String)input).toLowerCase();
    }
}

class Splitter extends Processor {
    String process(Object input) {
        // Аргумент split() используется для разбиения строки
        return Arrays.toString(((String)input).split(" "));
    }
}

public class Apply {
```



```

public static void process(Processor p, Object s) {
    print("Используем Processor " + p.name());
    print(p.process(s));
}
public static String s =
    "Disagreement with beliefs is by definition incorrect";
public static void main(String[] args) {
    process(new Upcase(), s);
    process(new Downcase(), s);
    process(new Splitter(), s);
}
} /* Output:
Используем Processor Upcase
DISAGREEMENT WITH BELIEFS IS BY DEFINITION INCORRECT
Используем Processor Downcase
disagreement with beliefs is by definition incorrect
Используем Processor Splitter
[Disagreement, with, beliefs, is, by, definition, incorrect]
*///:~

```

Метод `Apply.process()` получает любую разновидность `Processor`, применяет ее к `Object`, а затем выводит результат. Решение, при котором поведение метода изменяется в зависимости от переданного объекта-аргумента, называется паттерном «*Стратегия*». Метод содержит фиксированную часть алгоритма, а объект стратегии — переменную часть. Под «объектом стратегии» понимается передаваемый объект, который содержит выполняемый код. В данном случае объект `Processor` является объектом стратегии, а в методе `main()` мы видим три разные стратегии, применяемые к `String s`.

Метод `split()` является частью класса `String`. Он получает объект `String`, разбивает его на несколько фрагментов по ограничителям, определяемым переданным аргументом, и возвращает `String[]`. Здесь он используется как более компактный способ создания массива `String`.

Теперь предположим, что вы обнаружили некое семейство электронных фильтров, которые тоже было бы уместно использовать с методом `Apply.process()`:

```

//: interfaces/filters/Waveform.java
package interfaces.filters;

public class Waveform {
    private static long counter;
    private final long id = counter++;
    public String toString() { return "Waveform " + id; }
} ///:~

//: interfaces/filters/Filter.java
package interfaces.filters;

public class Filter {
    public String name() {
        return getClass().getSimpleName();
    }
    public Waveform process(Waveform input) { return input; }
} ///:~

//: interfaces/filters/LowPass.java

```

```

package interfaces.filters;

public class LowPass extends Filter {
    double cutoff;
    public LowPass(double cutoff) { this.cutoff = cutoff; }
    public Waveform process(Waveform input) {
        return input; // Фиктивная обработка
    }
} ///:~

//: interfaces/filters/HighPass.java
package interfaces.filters;

public class HighPass extends Filter {
    double cutoff;
    public HighPass(double cutoff) { this.cutoff = cutoff; }
    public Waveform process(Waveform input) { return input; }
} ///:~

//: interfaces/filters/BandPass.java
package interfaces.filters;

public class BandPass extends Filter {
    double lowCutoff, highCutoff;
    public BandPass(double lowCut, double highCut) {
        lowCutoff = lowCut;
        highCutoff = highCut;
    }
    public Waveform process(Waveform input) { return input; }
} ///:~

```

Класс `Filter` содержит те же интерфейсные элементы, что и `Processor`, но поскольку он не является производным от `Processor` (создатель класса `Filter` и не подозревал, что вы захотите использовать его как `Processor`), он не может использоваться с методом `Apply.process()`, хотя это выглядело бы вполне естественно. Логическая привязка между `Apply.process()` и `Processor` оказывается более сильной, чем реально необходимо, и это обстоятельство препятствует повторному использованию кода `Apply.process()`. Также обратите внимание, что входные и выходные данные относятся к типу `Waveform`.

Но если преобразовать класс `Processor` в интерфейс, ограничения ослабляются и появляется возможность повторного использования `Apply.process()`. Обновленные версии `Processor` и `Apply` выглядят так:

```

//: interfaces/interfaceprocessor/Processor.java
package interfaces.interfaceprocessor;

public interface Processor {
    String name();
    Object process(Object input);
} ///:~

//: interfaces/interfaceprocessor/Apply.java
package interfaces.interfaceprocessor;
import static net.mindview.util.Print.*;

public class Apply {

```

```

public static void process(Processor p, Object s) {
    print("Using Processor " + p.name());
    print(p.process(s));
}
} ///:~

```

В первом варианте повторного использования кода клиентские программисты пишут свои классы с поддержкой интерфейса:

```

//: interfaces/interfaceprocessor/StringProcessor.java
package interfaces.interfaceprocessor;
import java.util.*;

public abstract class StringProcessor implements Processor{
    public String name() {
        return getClass().getSimpleName();
    }
    public abstract String process(Object input);
    public static String s =
        "If she weighs the same as a duck, she's made of wood";
    public static void main(String[] args) {
        Apply.process(new Upcase(), s);
        Apply.process(new Downcase(), s);
        Apply.process(new Splitter(), s);
    }
}

class Upcase extends StringProcessor {
    public String process(Object input) { // Ковариантный возвращаемый тип
        return ((String)input).toUpperCase();
    }
}

class Downcase extends StringProcessor {
    public String process(Object input) {
        return ((String)input).toLowerCase();
    }
}

class Splitter extends StringProcessor {
    public String process(Object input) {
        return Arrays.toString(((String)input).split(" "));
    }
} /* Output:
Используем Processor Upcase
IF SHE WEIGHS THE SAME AS A DUCK, SHE'S MADE OF WOOD
Используем Processor Downcase
if she weighs the same as a duck, she's made of wood
Используем Processor Splitter
[If, she, weighs, the, same, as, a, duck,, she's, made, of, wood]
*///:~

```

Впрочем, довольно часто модификация тех классов, которые вы собираетесь использовать, невозможна. Например, в примере с электронными фильтрами библиотека была получена из внешнего источника. В таких ситуациях применяется паттерн проектирования «Адаптер» — вы пишете код, который получает имеющийся интерфейс, и создаете тот интерфейс, который вам нужен:

```

//: interfaces/interfaceprocessor/FilterProcessor.java
package interfaces.interfaceprocessor;
import interfaces.filters.*;

class FilterAdapter implements Processor {
    Filter filter;
    public FilterAdapter(Filter filter) {
        this.filter = filter;
    }
    public String name() { return filter.name(); }
    public Waveform process(Object input) {
        return filter.process((Waveform)input);
    }
}

public class FilterProcessor {
    public static void main(String[] args) {
        Waveform w = new Waveform();
        Apply.process(new FilterAdapter(new LowPass(1.0)), w);
        Apply.process(new FilterAdapter(new HighPass(2.0)), w);
        Apply.process(
            new FilterAdapter(new BandPass(3.0, 4.0)), w);
    }
} /* Output:
Используем Processor LowPass
Waveform 0
Используем Processor HighPass
Waveform 0
Используем Processor BandPass
Waveform 0
*///:~

```

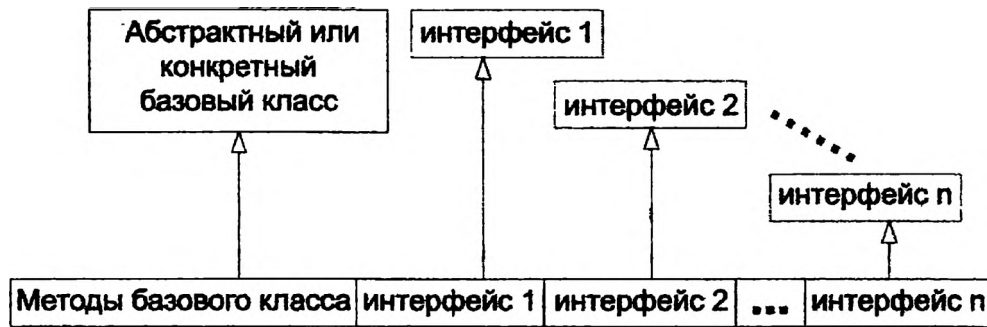
Конструктор `FilterAdapter` получает исходный интерфейс (`Filter`) и создает объект с требуемым интерфейсом `Processor`. Также обратите внимание на применение делегирования в классе `FilterAdapter`.

Отделение интерфейса от реализации позволяет применять интерфейс к разным реализациям, а следовательно, расширяет возможности повторного использования кода.

11. (4) Создайте класс с методом, который получает аргумент `String` и переставляет местами каждую пару символов в полученной строке. Адаптируйте класс так, чтобы он работал с `interfaceprocessor.Apply.process()`.

«Множественное наследование» в Java

Так как интерфейс по определению не имеет реализации (то есть не обладает памятью для хранения данных), нет ничего, что могло бы помешать совмещению нескольких интерфейсов. Это очень полезная возможность, так как в некоторых ситуациях требуется выразить утверждение: «Икс является и А, и Б, и В одновременно». В C++ подобное совмещение интерфейсов нескольких классов называется *множественным наследованием*, и оно имеет ряд очень неприятных аспектов, поскольку каждый класс может иметь свою реализацию. В Java можно добиться аналогичного эффекта, но поскольку реализацией обладает всего один класс, проблемы, возникающие при совмещении нескольких интерфейсов в C++, в Java принципиально невозможны.



При наследовании базовый класс вовсе не обязан быть абстрактным или «конкретным» (без абстрактных методов). Если наследование *действительно* осуществляется не от интерфейса, то среди прямых «предков» класс может быть только один — все остальные должны быть интерфейсами. Имена интерфейсов перечисляются вслед за ключевым словом `implements` и разделяются запятыми. Интерфейсов может быть сколько угодно, причем к ним можно проводить восходящее преобразование. Следующий пример показывает, как создать новый класс на основе конкретного класса и нескольких интерфейсов:

```

//: interfaces/Adventure.java
// Использование нескольких интерфейсов.

interface CanFight {
    void fight();
}

interface CanSwim {
    void swim();
}

interface CanFly {
    void fly();
}

class ActionCharacter {
    public void fight() {}
}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
    public void swim() {}
    public void fly() {}
}

public class Adventure {
    public static void t(CanFight x) { x.fight(); }
    public static void u(CanSwim x) { x.swim(); }
    public static void v(CanFly x) { x.fly(); }
    public static void w(ActionCharacter x) { x.fight(); }
    public static void main(String[] args) {
        Hero h = new Hero();
        t(h); // Используем объект в качестве типа CanFight
        u(h); // Используем объект в качестве типа CanSwim
        v(h); // Используем объект в качестве типа CanFly
        w(h); // Используем объект в качестве ActionCharacter
    }
}
//:~

```

Мы видим, что класс `Hero` сочетает конкретный класс `ActionCharacter` с интерфейсами `CanFight`, `CanSwim` и `CanFly`. При объединении конкретного класса с интерфейсами на первом месте должен стоять конкретный класс, а за ним следуют интерфейсы (иначе компилятор выдаст ошибку).

Заметьте, что объявление метода `fight()` в интерфейсе `CanFight` совпадает с тем, что имеется в классе `ActionCharacter`, и поэтому в классе `Hero` *нет* определения метода `fight()`. Интерфейсы можно расширять, но при этом получается другой интерфейс. Необходимым условием для создания объектов нового типа является наличие всех определений. Хотя класс `Hero` не имеет явного определения метода `fight()`, это определение существует в классе `ActionCharacter`, что и делает возможным создание объектов класса `Hero`.

Класс `Adventure` содержит четыре метода, которые принимают в качестве аргументов разнообразные интерфейсы и конкретный класс. Созданный объект `Hero` передается всем этим методам, а это значит, что выполняется восходящее преобразование объекта к каждому интерфейсу по очереди. Система интерфейсов Java спроектирована так, что она нормально работает без особых усилий со стороны программиста.

Помните, что главная причина введения в язык интерфейсов представлена в приведенном примере: это возможность выполнять восходящее преобразование к нескольким базовым типам. Вторая причина для использования интерфейсов совпадает с предназначением абстрактных классов: запретить программисту-клиенту создание объектов этого класса.

Возникает естественный вопрос: что лучше — интерфейс или абстрактный класс? Если возможно создать базовый класс без определений методов и переменных-членов, выбирайте именно интерфейс, а не абстрактный класс. Вообще говоря, если известно, что нечто будет использоваться как базовый класс, первым делом постарайтесь сделать это «нечто» интерфейсом.

12. (2) Добавьте в пример `Adventure.java` интерфейс `CanClimb`, созданный по образцу других интерфейсов.
13. (2) Создайте интерфейс и определите два новых интерфейса, производных от него. Выполните множественное наследование третьего интерфейса от первых двух¹.

Расширение интерфейса через наследование

Наследование позволяет легко добавить в интерфейс объявления новых методов, а также совместить несколько интерфейсов в одном. В обоих случаях получается новый интерфейс, как показано в следующем примере:

```
//: interfaces/HorrorShow.java
// Расширение интерфейса с помощью наследования.

interface Monster {
    void menace();
}
```

¹ Пример показывает, как в Java решается проблема «ромбовидного наследования», присущая множественному наследованию C++.

```
interface DangerousMonster extends Monster {
    void destroy();
}

interface Lethal {
    void kill();
}

class DragonZilla implements DangerousMonster {
    public void menace() {}
    public void destroy() {}
}

interface Vampire extends DangerousMonster, Lethal {
    void drinkBlood();
}

class VeryBadVampire implements Vampire {
    public void menace() {}
    public void destroy() {}
    public void kill() {}
    public void drinkBlood() {}
}

public class HorrorShow {
    static void u(Monster b) { b.menace(); }
    static void v(DangerousMonster d) {
        d.menace();
        d.destroy();
    }
    static void w(Lethal l) { l.kill(); }
    public static void main(String[] args) {
        DangerousMonster barney = new DragonZilla();
        u(barney);
        v(barney);
        Vampire vlad = new VeryBadVampire();
        u(vlad);
        v(vlad);
        w(vlad);
    }
} ///:~
```

`DangerousMonster` представляет собой простое расширение `Monster`, в результате которого образуется новый интерфейс. Он реализуется классом `DragonZilla`.

Синтаксис, использованный в интерфейсе `Vampire`, работает *только* при наследовании интерфейсов. Обычно ключевое слово `extends` может использоваться всего с одним классом, но так как интерфейс можно составить из нескольких других интерфейсов, `extends` подходит для написания нескольких имен интерфейсов при создании нового интерфейса. Как нетрудно заметить, имена нескольких интерфейсов разделяются при этом запятыми.

- 14.** (2) Создайте три интерфейса, каждый из которых содержит два метода. Объявите новый интерфейс, производный от первых трех, включите в него новый метод. Создайте класс, который реализует новый интерфейс, а также является производным от конкретного класса. Напишите четыре метода, каждый из которых получает

один из четырех интерфейсов в аргументе. Создайте в `main()` объект этого класса и передайте его каждому из методов.

15. (2) Измените предыдущее упражнение — создайте абстрактный класс и унаследуйте производный класс от него.

Конфликты имен при совмещении интерфейсов

При реализации нескольких интерфейсов может возникнуть небольшая проблема. В только что рассмотренном примере интерфейс `CanFight` и класс `ActionCharacter` имеют идентичные методы `void fight()`. Хорошо, если методы полностью тождественны, но что, если они различаются по сигнатуре или типу возвращаемого значения? Рассмотрим такой пример:

```
//: interfaces/InterfaceCollision.java
package interfaces;

interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }

class C2 implements I1, I2 {
    public void f() {}
    public int f(int i) { return 1; } // перегружен
}

class C3 extends C implements I2 {
    public int f(int i) { return 1; } // перегружен
}

class C4 extends C implements I3 {
    // Идентичны, все нормально:
    public int f() { return 1; }
}

// Методы различаются только по типу возвращаемого значения:
//! class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {} ///:~
```

Трудность возникает из-за того, что переопределение, реализация и перегрузка образуют опасную «смесь». Кроме того, перегруженные методы не могут различаться только возвращаемыми значениями. Если убрать комментарий в двух последних строках программы, сообщение об ошибке разъясняет суть происходящего:

```
InterfaceCollision.java:23: f() в C не может
реализовать f() в I1; попытка использовать
несовместимые возвращаемые типы
обнаружено: int
требуется: void
InterfaceCollision.java:24: интерфейсы I3 и I1
несовместимы; оба определяют f(),
но с различными возвращаемыми типами
```

Использование одинаковых имен методов в интерфейсах, предназначенных для совмещения, обычно приводит к запутанному и трудному для чтения коду. Постарайтесь по возможности избегать таких ситуаций.

Интерфейсы как средство адаптации

Одной из самых убедительных причин для использования интерфейсов является возможность определения нескольких реализаций для одного интерфейса. В простых ситуациях такая схема принимает вид метода, который при вызове передается интерфейсу; от вас потребуется реализовать интерфейс и передать объект методу.

Соответственно интерфейсы часто применяются в паттерне проектирования «*Стратегия*». Вы пишете метод, выполняющий несколько операций; при вызове метод получает интерфейс, который тоже указываете вы. Фактически вы говорите: «Мой метод может использоваться с любым объектом, удовлетворяющим моему интерфейсу». Метод становится более гибким и универсальным.

Например, конструктор класса Java SE5 `Scanner` (см. главу 13) получает интерфейс `Readable`. Анализ показывает, что `Readable` не является аргументом любого другого метода из стандартной библиотеки Java — этот интерфейс создавался исключительно для `Scanner`, чтобы его аргументы не ограничивались определенным классом. При таком подходе можно заставить `Scanner` работать с другими типами. Если вы хотите создать новый класс, который может использоваться со `Scanner`, реализуйте в нем интерфейс `Readable`:

```
//: interfaces/RandomWords.java
// Реализация интерфейса для выполнения требований метода
import java.nio.*;
import java.util.*;

public class RandomWords implements Readable {
    private static Random rand = new Random(47);
    private static final char[] capitals =
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();
    private static final char[] lowers =
        "abcdefghijklmnopqrstuvwxyz".toCharArray();
    private static final char[] vowels =
        "aeiou".toCharArray();
    private int count;
    public RandomWords(int count) { this.count = count; }
    public int read(CharBuffer cb) {
        if(count-- == 0)
            return -1; // Признак конца входных данных
        cb.append(capitals[rand.nextInt(capitals.length)]);
        for(int i = 0; i < 4; i++) {
            cb.append(vowels[rand.nextInt(vowels.length)]);
            cb.append(lowers[rand.nextInt(lowers.length)]);
        }
        cb.append(" ");
        return 10; // Количество присоединенных символов
    }
    public static void main(String[] args) {
        Scanner s = new Scanner(new RandomWords(10));
        while(s.hasNext())
            System.out.println(s.next());
    }
} /* Output:
Yazeruyac
Fowenucor
```

```
Goeazimom
Raeuuacio
Nuoadesiw
Hageaikux
Ruqicibui
Numasetih
Kuuuuozog
Waqizeyou
*///:~
```

Интерфейс `Readable` требует только присутствия метода `read()`. Метод `read()` либо добавляет данные в аргумент `CharBuffer` (это можно сделать несколькими способами; обращайтесь к документации `CharBuffer`), либо возвращает `-1` при отсутствии входных данных.

Допустим, у нас имеется класс, не реализующий интерфейс `Readable`, — как заставить его работать с `Scanner`? Перед вами пример класса, генерирующего вещественные числа:

```
//: interfaces/RandomDoubles.java
import java.util.*;

public class RandomDoubles {
    private static Random rand = new Random(47);
    public double next() { return rand.nextDouble(); }
    public static void main(String[] args) {
        RandomDoubles rd = new RandomDoubles();
        for(int i = 0; i < 7; i++)
            System.out.print(rd.next() + " ");
    }
} /* Output:
0.7271157860730044 0.5309454508634242 0.16020656493302599 0.18847866977771732
0.5166020801268457 0.2678662084200585 0.2613610344283964
*///:~
```

Мы снова можем воспользоваться паттерном «Адаптер», но на этот раз адаптируемый класс создается наследованием и реализацией интерфейса `Readable`. Псевдомножественное наследование, обеспечиваемое ключевым словом `interface`, позволяет создать новый класс, который одновременно является и `RandomDoubles`, и `Readable`:

```
//: interfaces/AdaptedRandomDoubles.java
// Создание адаптера посредством наследования
import java.nio.*;
import java.util.*;

public class AdaptedRandomDoubles extends RandomDoubles
implements Readable {
    private int count;
    public AdaptedRandomDoubles(int count) {
        this.count = count;
    }
    public int read(CharBuffer cb) {
        if(count-- == 0)
            return -1;
        String result = Double.toString(next()) + " ";
        cb.append(result);
        return result.length();
    }
}
```

```
public static void main(String[] args) {
    Scanner s = new Scanner(new AdaptedRandomDoubles(7));
    while(s.hasNextDouble())
        System.out.print(s.nextDouble() + " ");
}
} /* Output:
0.7271157860730044 0.5309454508634242 0.16020656493302599 0.18847866977771732
0.5166020801268457 0.2678662084200585 0.2613610344283964
*///:~
```

Так как интерфейсы можно добавлять подобным образом только к существующим классам, это означает, что любой класс может быть адаптирован для метода, получающего интерфейс. В этом проявляется одно из преимуществ интерфейсов перед классами.

- 16.** (3) Создайте класс, который генерирует серию `char`. Адаптируйте его так, чтобы он мог использоваться для передачи данных `Scanner`.

Поля в интерфейсах

Так как все поля, помещаемые в интерфейс, автоматически являются статическими (`static`) и неизменными (`final`), объявление `interface` хорошо подходит для создания групп постоянных значений. До выхода Java SE5 только так можно было имитировать перечисляемый тип `enum` из языков C и C++. Это выглядело примерно так:

```
//: interfaces/Months.java
// Использование интерфейсов для создания групп констант.
package interfaces;

public interface Months {
    int
        JANUARY = 1, FEBRUARY = 2, MARCH = 3,
        APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
        AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
        NOVEMBER = 11, DECEMBER = 12;
} ///:~
```

Отметьте стиль Java — использование только заглавных букв (с разделением слов подчеркиванием) для полей со спецификаторами `static` и `final`, которым присваиваются фиксированные значения на месте описания. Поля интерфейса автоматически являются открытыми (`public`), поэтому нет необходимости явно указывать это.

В Java SE5 появилось гораздо более мощное и удобное ключевое слово `enum`, поэтому надобность в применении интерфейсов для определения констант отпала. Впрочем, эта старая идиома еще может встречаться в некоторых старых программах (в приложении к книге на сайте www.MindView.net приведено полное описание способа моделирования перечислимых типов с использованием интерфейсов до Java SE5). Дополнительная информация об использовании `enum` приведена в главе 19.

- 17.** (2) Покажите, что поля интерфейса автоматически являются статическими (`static`) и неизменными (`final`).

Инициализация полей интерфейсов

Поля, определяемые в интерфейсах, не могут быть «пустыми константами», но могут инициализироваться не-константными выражениями. Например:

```
//: interfaces/RandVals.java
// Инициализация полей интерфейсов
// не-константными выражениями.
import java.util.*;

public interface RandVals {
    Random RAND = new Random(47);
    int RANDOM_INT = RAND.nextInt(10);
    long RANDOM_LONG = RAND.nextLong() * 10;
    float RANDOM_FLOAT = RAND.nextLong() * 10;
    double RANDOM_DOUBLE = RAND.nextDouble() * 10;
} ///:~
```

Так как поля являются статическими, они инициализируются при первой загрузке класса, которая происходит при первом обращении к любому из полей интерфейса. Простой тест:

```
//: interfaces/TestRandVals.java
import static net.mindview.util.Print.*;

public class TestRandVals {
    public static void main(String[] args) {
        print(RandVals.RANDOM_INT);
        print(RandVals.RANDOM_LONG);
        print(RandVals.RANDOM_FLOAT);
        print(RandVals.RANDOM_DOUBLE);
    }
} /* Output:
8
-32032247016559954
-8.5939291E18
5.779976127815049
*///:~
```

Конечно, поля не являются частью интерфейса. Данные хранятся в статической области памяти, отведенной для данного интерфейса.

Вложенные интерфейсы

Интерфейсы могут вкладываться в классы и в другие интерфейсы¹. При этом обнаруживается несколько весьма интересных особенностей:

```
//: interfaces/nesting/NestingInterfaces.java
package interfaces.nesting;

class A {
    interface B {
        void f();
    }
    public class BImp implements B {
        public void f() {}
    }
    private class BImp2 implements B {
        public void f() {}
    }
}
```

¹ Благодарю Мартина Даннера за то, что он задал этот вопрос на семинаре.

```
public interface C {
    void f();
}
class CImp implements C {
    public void f() {}
}
private class CImp2 implements C {
    public void f() {}
}
private interface D {
    void f();
}
private class DImp implements D {
    public void f() {}
}
public class DImp2 implements D {
    public void f() {}
}
public D getD() { return new DImp2(); }
private D dRef;
public void receiveD(D d) {
    dRef = d;
    dRef.f();
}
}

interface E {
    interface G {
        void f();
    }
    // Избыточное объявление public:
    public interface H {
        void f();
    }
    void g();
    // Не может быть private внутри интерфейса:
    ///! private interface I {}
}

public class NestingInterfaces {
    public class BImp implements A.B {
        public void f() {}
    }
    class CImp implements A.C {
        public void f() {}
    }
    // Нельзя реализовать private-интерфейс, кроме как
    // внутри класса, где он был определен:
    ///! class DImp implements A.D {
    ///! public void f() {}
    ///! }
    class EImp implements E {
        public void g() {}
    }
    class EGImp implements E.G {
        public void f() {}
    }
    class EImp2 implements E {
        public void g() {}
    }
}
```

```

class EG implements E.G {
    public void f() {}
}
}
public static void main(String[] args) {
    A a = new A();
    // Нет доступа к A.D:
    //! A.D ad = a.getD();
    // Не возвращает ничего, кроме A.D:
    //! A.DImp2 di2 = a.getD();
    // Нельзя получить доступ к члену интерфейса:
    //! a.getD().f();
    // Только другой класс A может использовать getD():
    A a2 = new A();
    a2.receiveD(a.getD());
}
} ///:~

```

Синтаксис вложения интерфейса в класс достаточно очевиден, и подобно любым другим обыкновенным интерфейсам, вложенным интерфейсам не запрещается иметь «дружественную» или открытую (`public`) видимость.

Любопытная подробность: интерфейсы могут быть объявлены закрытыми (`private`), как видно на примере `A.D` (используется тот же синтаксис описания, что и для вложенных классов). Что же хорошего в закрытом вложенном интерфейсе? Поначалу кажется, что такой интерфейс реализуется только в виде закрытого (`private`) вложенного класса, подобного `DImp`, но `A.DImp2` показывает, что он также может иметь форму открытого (`public`) класса. Тем не менее класс `A.DImp2` «замкнут» сам на себя. Вам не разрешается упоминать тот факт, что он реализует `private`-интерфейс, поэтому реализация такого интерфейса — просто способ принудительно определить методы этого интерфейса, не добавляя информации о дополнительном типе (то есть запрещая восходящее преобразование).

Метод `getD()` усугубляет сложности, связанные с `private`-интерфейсом, — это открытый (`public`) метод, возвращающий ссылку на закрытый (`private`) интерфейс. Что вы в силах сделать с возвращаемым значением этого метода? В методе `main()` вы можете видеть, как было предпринято несколько попыток использовать это возвращаемое значение, и все они оказались неудачными. Единственный способ заставить метод работать — передать возвращаемое значение некоторому «уполномоченному» объекту, в нашем случае это еще один объект `A`, у которого имеется необходимый метод `receiveD()`.

Интерфейс `E` показывает, что интерфейсы могут быть вложены друг в друга. Впрочем, правила для интерфейсов — в особенности то, что все элементы интерфейса должны быть открытыми (`public`), — здесь строго соблюдаются, поэтому интерфейс, вложенный внутри другого интерфейса, автоматически объявляется открытым и его нельзя сделать закрытым (`private`).

`NestingInterfaces` демонстрирует разнообразные способы реализации вложенных интерфейсов. В особенности отметьте тот факт, что вам необязательно реализовывать вложенные в один интерфейс другие интерфейсы. Также закрытые (`private`) интерфейсы нельзя реализовать за пределами классов, в которых они описываются.

Первой, как правило, бывает мысль, что вложенные интерфейсы добавлены в язык для синтаксической целостности, но я постоянно убеждаюсь в том, что как только вы

узнаете об определенном свойстве языка, причины для использования этого свойства начинают возникать сами собой.

Интерфейсы и фабрики

На концептуальном уровне интерфейс представляет собой «шлюз», ведущий к разным реализациям, а типичным механизмом создания объектов, реализующих интерфейс, является паттерн проектирования «Фабричный метод». Вместо прямого вызова конструктора вызывается метод *объекта-фабрики*, который создает реализацию интерфейса — таким образом код теоретически полностью изолируется от реализации интерфейса, что позволяет прозрачно заменять одну реализацию другой. Следующая программа демонстрирует структуру паттерна «Фабричный метод»:

```
//: interfaces/Factories.java
import static net.mindview.util.Print.*;

interface Service {
    void method1();
    void method2();
}

interface ServiceFactory {
    Service getService();
}

class Implementation1 implements Service {
    Implementation1() {} // Доступ в пределах пакета
    public void method1() {print("Implementation1 method1");}
    public void method2() {print("Implementation1 method2");}
}

class Implementation1Factory implements ServiceFactory {
    public Service getService() {
        return new Implementation1();
    }
}

class Implementation2 implements Service {
    Implementation2() {} // Доступ в пределах пакета
    public void method1() {print("Implementation2 method1");}
    public void method2() {print("Implementation2 method2");}
}

class Implementation2Factory implements ServiceFactory {
    public Service getService() {
        return new Implementation2();
    }
}

public class Factories {
    public static void serviceConsumer(ServiceFactory fact) {
        Service s = fact.getService();
        s.method1();
        s.method2();
    }
}
```

```

public static void main(String[] args) {
    serviceConsumer(new Implementation1Factory());
    // Реализации полностью взаимозаменяемы:
    serviceConsumer(new Implementation2Factory());
}
} /* Output:
Implementation1 method1
Implementation1 method2
Implementation2 method1
Implementation2 method2
*///:~

```

Без паттерна «Фабричный метод» вашему коду пришлось бы в каком-то месте задать тип создаваемого объекта *Service*, чтобы он мог вызвать подходящий конструктор.

Для чего добавлять в систему лишний логический уровень? Допустим, вы создаете систему для настольных игр — скажем, для шахмат и шашек, в которые можно играть на одной доске:

```

//: interfaces/Games.java
// Система для настольных игр с использованием Фабричного метода.
import static net.mindview.util.Print.*;

interface Game { boolean move(); }
interface GameFactory { Game getGame(); }

class Checkers implements Game {
    private int moves = 0;
    private static final int MOVES = 3;
    public boolean move() {
        print("Checkers move " + moves);
        return ++moves != MOVES;
    }
}

class CheckersFactory implements GameFactory {
    public Game getGame() { return new Checkers(); }
}

class Chess implements Game {
    private int moves = 0;
    private static final int MOVES = 4;
    public boolean move() {
        print("Chess move " + moves);
        return ++moves != MOVES;
    }
}

class ChessFactory implements GameFactory {
    public Game getGame() { return new Chess(); }
}

public class Games {
    public static void playGame(GameFactory factory) {
        Game s = factory.getGame();
        while(s.move())
            ;
    }
}

```



```
public static void main(String[] args) {  
    playGame(new CheckersFactory());  
    playGame(new ChessFactory());  
}  
} /* Output:  
Checkers move 0  
Checkers move 1  
Checkers move 2  
Chess move 0  
Chess move 1  
Chess move 2  
Chess move 3  
*///:~
```

Если класс `Games` представляет сложный блок кода, этот механизм позволяет повторно использовать этот код для разных типов игр. Нетрудно представить более сложные игры, которые могут извлечь пользу из этого паттерна. В следующей главе будет представлен более элегантный способ реализации фабрик на базе анонимных внутренних классов.

18. (2) Создайте интерфейс `Cycle` с реализациями `Unicycle`, `Bicycle` и `Tricycle`. Создайте фабрику для каждой разновидности `Cycle` и код, использующий эти фабрики.
19. (3) Создайте на базе паттерна «Фабричный метод» программную среду, моделирующую броски монет и броски кубиков.

Резюме

После знакомства с интерфейсами возникает соблазнительная мысль: интерфейсы — это хорошо, и их всегда стоит использовать вместо конкретных классов. Конечно, практически везде, где создается класс, вместо него можно создать интерфейс и фабрику.

Многие программисты поддались этому искушению и стали создавать интерфейсы и фабрики там, где это возможно. Не имея полной уверенности в том, когда может потребоваться другая реализация, они всегда добавляли эту абстракцию. Интерфейсы стали своего рода скороспелой оптимизацией.

Любая абстракция должна базироваться на реальных потребностях. Интерфейсы относятся к числу аспектов системы, которые вводятся по мере необходимости в результате рефакторинга, а не являются лишним логическим уровнем, добавляемым сплошь и рядом со всей сопутствующей сложностью. Лишняя сложность весьма существенна, и если вдруг становится ясно, что она появилась вследствие добавления интерфейсов «на всякий случай», а не по реальной необходимости — я начну сомневаться во всех решениях из области проектирования, принятых этим человеком.

В общем случае рекомендуется отдавать предпочтение классам перед интерфейсами. Начните с классов, и если вдруг станет ясно, что в данной ситуации необходимы интерфейсы, — проведите рефакторинг. Интерфейсы — замечательный инструмент, но и им можно злоупотреблять.

10

Внутренние классы

Определение класса можно разместить в определении другого класса. Класс, созданный подобным образом, называется внутренним классом (inner class).

Внутренние классы чрезвычайно полезны, поскольку они позволяют вам группировать классы, логически принадлежащие друг другу, и управлять доступом к ним. Однако важно понимать, что внутренние классы заметно отличаются от композиции.

На первый взгляд внутренние классы напоминают простой механизм сокрытия кода: вы размещаете классы внутри других классов. Однако на самом деле полезность внутренних классов этим не ограничивается — внутренний класс может взаимодействовать со своим внешним классом, а код, написанный с использованием внутренних классов, получается более элегантным и понятным (хотя, конечно, это преимущество не гарантировано).

Чтобы привыкнуть к внутренним классам и начать использовать их в программах, вам потребуется некоторое время. Необходимость внутренних классов не всегда очевидна, но после знакомства с базовым синтаксисом и семантикой внутренних классов раздел «Внутренние классы: зачем?» поможет вам понять их преимущества.

После этого раздела в оставшейся части этой главы синтаксис внутренних классов рассматривается более подробно. Это описание приводится для полноты материала, но, скорее всего, эти возможности вам не понадобятся (по крайней мере не сразу). Итак, начальной части главы пока будет достаточно, а более подробные объяснения можно использовать в будущем как справочный материал.

Создание внутренних классов

Создать внутренний класс несложно — достаточно разместить определение класса внутри окружающего класса:

```
//: innerclasses/Parcel1.java  
// Создание внутренних классов
```

```

public class Parcel1 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    // Использование внутренних классов очень похоже
    // на использование любых других классов,
    // в пределах Parcel1:
    public void ship(String dest) {
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
        Parcel1 p = new Parcel1();
        p.ship("Танзания");
    }
} /* Output:
Танзания
*///:~

```

Внутренний класс, используемый внутри метода `ship()`, выглядит так же, как и все остальные классы. Очевидное отличие одно — имена классов вложены в класс `Parcel1`. Вскоре вы увидите, что это не единственное отличие.

Как правило, внешний класс содержит метод, возвращающий ссылку на внутренний класс, как в методах `to()` и `contents()` в следующем примере:

```

//: innerclasses/Parcel2.java
// Возврат ссылки на внутренний класс.

public class Parcel2 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    public Destination to(String s) {
        return new Destination(s);
    }
    public Contents contents() {
        return new Contents();
    }
    public void ship(String dest) {

```

продолжение ➤

```

    Contents c = contents();
    Destination d = to(dest);
    System.out.println(d.readLabel());
}
public static void main(String[] args) {
    Parcel2 p = new Parcel2();
    p.ship("Танзания");
    Parcel2 q = new Parcel2();
    // Определение ссылок на внутренние классы:
    Parcel2.Contents c = q.contents();
    Parcel2.Destination d = q.to("Борнео");
}
} /* Output:
Танзания
*///:~

```

Если вам понадобится создать объект внутреннего класса где-либо еще, кроме как в нестатическом методе внешнего класса, нужно будет указать тип этого объекта следующим образом: *ИмяВнешнегоКласса.ИмяВнутреннегоКласса*, что и делается в методе `main()`.

1. (1) Напишите класс с именем `Outer`, содержащий внутренний класс с именем `Inner`. Добавьте в `Outer` метод, возвращающий объект типа `Inner`. В методе `main()` создайте и инициализируйте ссылку на `Inner`.

Ссылка на внешний класс

Пока что внутренние классы выглядят как механизм сокрытия имен и организации кода — полезный, но не особенно впечатляющий. Однако существует один нюанс: объект внутреннего класса получает *ссылку на внешний объект, который его создал*, и поэтому может обращаться к членам внешнего объекта без дополнительных уточнений. Кроме того, для внутренних классов доступны все элементы внешнего класса¹. Следующий пример поясняет сказанное:

```

//: innerclasses/Sequence.java
// Хранение последовательности Object.

interface Selector {
    boolean end();
    Object current();
    void next();
}

public class Sequence {
    private Object[] items;
    private int next = 0;
    public Sequence(int size) { items = new Object[size]; }
    public void add(Object x) {
        if(next < items.length)
            items[next++] = x;
    }
}

```

¹ Этот подход сильно отличается от присущего C++, где вложенные классы — просто механизм для сокрытия имен. Вложенные классы C++ не имеют связи с объектом-оболочкой и соответствующих прав на доступ к его элементам.

```

private class SequenceSelector implements Selector {
    private int i = 0;
    public boolean end() { return i == items.length; }
    public Object current() { return items[i]; }
    public void next() { if(i < items.length) i++; }
}
public Selector selector() {
    return new SequenceSelector();
}
public static void main(String[] args) {
    Sequence sequence = new Sequence(10);
    for(int i = 0; i < 10; i++)
        sequence.add(Integer.toString(i));
    Selector selector = sequence.selector();
    while(!selector.end()) {
        System.out.print(selector.current() + " ");
        selector.next();
    }
}
} /* Output:
0 1 2 3 4 5 6 7 8 9
*///:~

```

Класс `Sequence` — это просто «обертка» для массива с элементами `Object`, имеющего фиксированный размер. Для добавления нового объекта в конец последовательности используется метод `add()` (при наличии свободного места в массиве). Для выборки каждого объекта в последовательности `Sequence` предусмотрен интерфейс с именем `Selector`; это пример использования паттерна «Итератор», который будет описан позже. Интерфейс `Selector` позволяет узнать, достигнут ли конец последовательности (метод `end()`), обратиться к текущему объекту (метод `current()`) и передвинуться к следующему объекту последовательности (метод `next()`). Так как `Selector` является интерфейсом, многие другие классы вправе реализовать его по-своему, и многие методы могут брать его в качестве параметра, чтобы получить единообразный код.

Здесь `SequenceSelector` является закрытым (`private`) классом, предоставляющим функциональность интерфейса `Selector`. В методе `main()` вы можете наблюдать за процессом создания последовательности, с последующим заполнением ее объектами `String`. Затем объект `Selector` создается вызовом `selector()` для получения интерфейса `Selector`, который используется для перемещения по последовательности и выбора ее элементов.

На первый взгляд создание класса `SequenceSelector` похоже на создание обычного внутреннего класса. Но рассмотрите его повнимательнее. Заметьте, что каждый из методов: и `end()`, и `current()`, и `next()` — обращается к ссылке `items`, которая не является частью класса `SequenceSelector`, а относится к закрытому (`private`) полю внешнего класса. Впрочем, внутренний класс может обращаться ко всем полям и методам внешнего класса, как будто они описаны в нем самом. Такая возможность очень удобна, как наглядно доказывает приведенный пример.

Итак, внутренний класс автоматически имеет доступ к членам внешнего класса. Как же это происходит? Внутренний класс должен содержать ссылку на определенный объект окружающего класса, ответственного за его создание. При обращении к члену окружающего класса эта ссылка используется для вызова нужного члена. К счастью, все технические подробности берет на себя компилятор, но теперь вы знаете, что

объект внутреннего класса можно создать только в сочетании с объектом внешнего класса (если, как вы вскоре увидите, внутренний класс не является статическим). Для конструирования объекта внутреннего класса необходима ссылка на объект внешнего класса; если компилятор не сумеет обнаружить ее, он сообщит об ошибке. В основном весь процесс происходит без всякого участия со стороны программиста.

2. (2) Создайте класс, который содержит `String` и метод `toString()` для вывода хранимой строки. Добавьте несколько экземпляров нового класса в объект `Sequence` и выведите их.
3. (1) Измените упражнение 1 так, чтобы класс `Outer` содержал закрытое поле `String` (инициализируемое в конструкторе), а класс `Inner` содержал метод `toString()` для вывода этого поля. Создайте объект типа `Inner` и выведите его.

.this и .new

Если вам потребуется получить ссылку на объект внешнего класса, укажите имя внешнего класса с точкой и `this`. Полученная ссылка автоматически относится к правильному типу, который известен и проверяется во время компиляции, не требуя лишних затрат ресурсов во время выполнения. Следующий пример показывает, как использовать конструкцию `.this`:

```
//: innerclasses/DotThis.java
// Доступ к объекту внешнего класса.

public class DotThis {
    void f() { System.out.println("DotThis.f()"); }
    public class Inner {
        public DotThis outer() {
            return DotThis.this;
            // "this" без уточнения соответствует объекту Inner
        }
    }
    public Inner inner() { return new Inner(); }
    public static void main(String[] args) {
        DotThis dt = new DotThis();
        DotThis.Inner dti = dt.inner();
        dti.outer().f();
    }
} /* Output:
DotThis.f()
*///:~
```

Иногда в программе требуется приказать объекту создать объект одного из его внутренних классов. Для этого в выражение `new` включается ссылка на другой объект внешнего класса с синтаксисом `.new`, как в следующем примере:

```
//: innerclasses/DotNew.java
// Прямое создание объекта внутреннего класса с синтаксисом .new.

public class DotNew {
    public class Inner {}
```

```

public static void main(String[] args) {
    DotNew dn = new DotNew();
    DotNew.Inner dni = dn.new Inner();
}
} ///:~

```

Для непосредственного создания объекта внутреннего класса указывается не имя внешнего класса DotNew, как можно было бы ожидать, а объект внешнего класса, как в приведенном примере. Такой синтаксис решает проблему области действия имен внутреннего класса, поэтому в этом случае не используется (да и не может использоваться) запись `dn.new DotNew.Inner()`.

Чтобы создать объект внутреннего класса, обязательно должен существовать объект внешнего класса. Это объясняется тем, что объект внутреннего класса незаметно связывается с объектом внешнего класса, на базе которого он был создан. С другой стороны, при создании вложенного класса (статического внутреннего класса) ссылка на объект внешнего класса не нужна.

В следующем примере конструкция `.new` используется в примере Parcel:

```

//: innerclasses/Parcel3.java
// Использование конструкции .new для создания экземпляров внутренних классов.

```

```

public class Parcel3 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) { label = whereTo; }
        String readLabel() { return label; }
    }
    public static void main(String[] args) {
        Parcel3 p = new Parcel3();
        // Must use instance of outer class
        // to create an instance of the inner class:
        Parcel3.Contents c = p.new Contents();
        Parcel3.Destination d = p.new Destination("Танзания");
    }
} ///:~

```

4. (2) Добавьте в класс `Sequence.SequenceSelector` метод для получения ссылки на внешний класс `Sequence`.
5. (1) Создайте класс с внутренним классом. В отдельном классе создайте экземпляр внутреннего класса.

Внутренние классы и восходящее преобразование

Потенциал внутренних классов раскрывается при восходящем преобразовании к базовому классу, и прежде всего к интерфейсу. (Эффект получения ссылки на интерфейс по объекту, реализующему его, по сути не отличается от восходящего преобразования

к базовому классу.) Дело в том, что внутренний класс — реализация интерфейса — может оставаться невидимым и недоступным, что может быть удобно для сокрытия реализации. Пользователь получает лишь ссылку на базовый класс или интерфейс.

Мы можем создать интерфейсы для приведенных выше примеров:

```
//: innerclasses/Destination.java
public interface Destination {
    String readLabel();
} ///:~
```

```
//: innerclasses/Contents.java
public interface Contents {
    int value();
} ///:~
```

Теперь `Contents` и `Destination` являются интерфейсами, доступными программисту-клиенту. (Помните, что в объявлении `interface` все члены класса автоматически являются открытыми (`public`).)

При получении из метода ссылки на базовый класс или интерфейс возможны ситуации, в которых вам даже не удастся определить ее точный тип, как здесь:

```
//: innerclasses/TestParcel.java

class Parcel4 {
    private class PContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected class PDestination implements Destination {
        private String label;
        private PDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
    }
    public Destination destination(String s) {
        return new PDestination(s);
    }
    public Contents contents() {
        return new PContents();
    }
}

public class TestParcel {
    public static void main(String[] args) {
        Parcel4 p = new Parcel4();
        Contents c = p.contents();
        Destination d = p.destination("Танзания");
        // Обращение к закрытому классу невозможно:
        //! Parcel4.PContents pc = p.new PContents();
    }
} ///:~
```

В класс `Parcel4` было добавлено кое-что новое: внутренний класс `PContents` является закрытым (объявлен как `private`), поэтому нигде, кроме как во внешнем для него классе

`Parcel14`, нельзя получить к нему доступ. Класс `PDestination` объявлен как `protected`, следовательно, доступ к нему имеют только класс `Parcel14`, классы из одного пакета с `Parcel14` (так как спецификатор доступа `protected` также дает доступ в пределах своего пакета) и наследники класса `Parcel14`. Отсюда вывод, что программист-клиент обладает ограниченной информацией и доступом к этим членам класса. Вообще говоря, нельзя даже провести нисходящее преобразование к закрытому (`private`) внутреннему классу (или к защищенному (`protected`) внутреннему классу, кроме как из его наследника), так как у вас нет доступа к его имени, что показано в классе `TestParcel`. Таким образом, закрытый внутренний класс позволяет разработчику класса полностью запретить использование определенных типов и скрыть все детали реализации класса. Вдобавок расширение интерфейса с точки зрения программиста-клиента не будет иметь смысла, поскольку он не сможет получить доступ к дополнительным методам, не принадлежащим к открытой части класса. Также у компилятора Java появится возможность оптимизировать код.

6. (2) Создайте интерфейс, содержащий хотя бы один метод, в отдельном пакете. Создайте класс в другом пакете. Добавьте защищенный внутренний класс, реализующий интерфейс. В третьем пакете создайте производный класс; внутри метода верните объект защищенного внутреннего класса, преобразованный в интерфейс.
7. (2) Создайте класс, содержащий закрытое поле и закрытый метод. Создайте внутренний класс с методом, который изменяет поле внешнего класса и вызывает метод внешнего класса. Во втором методе внешнего класса создайте объект внутреннего класса и вызовите его метод; продемонстрируйте эффект на объекте внешнего класса.
8. (2) Проверьте, доступны ли для внешнего класса закрытые элементы внутреннего класса.

Внутренние классы в методах и областях действия

Примеры, приводившиеся до настоящего момента, дают представление о типичных применениях внутренних классов. Как правило, в коде, который вам предстоит читать и писать, задействованы «обычные» внутренние классы — простые и понятные. Однако синтаксис внутренних классов также подразумевает ряд других, не столь очевидных применений. Внутренние классы могут создаваться в методах и даже в произвольных областях действия. Обычно для такого применения есть две причины:

1. Вы хотите создавать и возвращать ссылки на некоторый интерфейс (как было показано ранее).
2. Вы решаете сложную задачу и хотите создать класс, который будет задействован в ее решении, но при этом сделать его недоступным для посторонних.

В следующих примерах приводившийся ранее код будет изменен так, чтобы в нем использовался:

- 1) класс, определенный в методе;
- 2) класс, определенный в области действия внутри метода;

- 3) анонимный класс, реализованный в интерфейсе;
- 4) анонимный класс, расширяющий класс с конструктором, который не является конструктором по умолчанию;
- 5) анонимный класс, выполняющий инициализацию поля;
- 6) анонимный класс, выполняющий конструирование с использованием инициализации экземпляра (анонимные внутренние классы не могут иметь конструкторов).

Первый пример демонстрирует создание всего класса в области действия метода (вместо области действия другого класса). Такая конструкция называется *локальным внутренним классом*:

```
//: innerclasses/Parcel5.java
// Вложение класса в метод.

public class Parcel5 {
    public Destination destination(String s) {
        class PDestination implements Destination {
            private String label;
            private PDestination(String whereTo) {
                label = whereTo;
            }
            public String readLabel() { return label; }
        }
        return new PDestination(s);
    }
    public static void main(String[] args) {
        Parcel5 p = new Parcel5();
        Destination d = p.destination("Танзания");
    }
} ///:~
```

Класс `PDestination` является частью `destination()`, а не частью `Parcel5`. Следовательно, к `PDestination` невозможно обратиться за пределами `destination()`. Обратите внимание на восходящее преобразование, выполняемое в команде `return`, — из `destination()` не выходит ничего, кроме ссылки на базовый класс `Destination`. Конечно, тот факт, что имя класса `PDestination` помещено в `destination()`, не означает, что `PDestination` не является действительным объектом после возвращения управления из `destination()`.

Идентификатор класса `PDestination` можно использовать для внутреннего класса внутри всех классов, находящихся в одном подкаталоге, и это не вызовет конфликта имен.

Следующий пример показывает, как происходит вложение внутреннего класса в произвольную область действия:

```
//: innerclasses/Parcel6.java
// Вложение класса в область действия.

public class Parcel6 {
    private void internalTracking(boolean b) {
        if(b) {
            class TrackingSlip {
                private String id;
            }
        }
    }
}
```

```

    TrackingSlip(String s) {
        id = s;
    }
    String getSlip() { return id; }
}
TrackingSlip ts = new TrackingSlip("slip");
String s = ts.getSlip();
}
// Здесь использовать нельзя! Вне области действия:
//!! TrackingSlip ts = new TrackingSlip("x");
}
public void track() { internalTracking(true); }
public static void main(String[] args) {
    Parcel6 p = new Parcel6();
    p.track();
}
} ///:~

```

Класс `TrackingSlip` вложен в область действия команды `if`. Это не означает, что класс создается условно, — он компилируется вместе с остальным кодом. Тем не менее этот класс недоступен за пределами области действия, в которой он определяется. В остальном он выглядит как самый обычный класс.

7. (1) Создайте интерфейс, содержащий минимум один метод. Реализуйте его, определяя внутренний класс в методе, который возвращает ссылку на ваш интерфейс.
8. (1) Повторите предыдущее упражнение, но определите внутренний класс в области действия внутри метода.
9. (2) Создайте закрытый внутренний класс, реализующий открытый интерфейс. Напишите метод, который возвращает ссылку на экземпляр закрытого внутреннего класса, преобразованную к интерфейсу восходящим преобразованием. Чтобы показать, что внутренний класс полностью скрыт, попробуйте выполнить нисходящее преобразование к нему.

Анонимные внутренние классы

Следующий пример выглядит довольно странно:

```

//: innerclasses/Parcel7.java
// Возвращение экземпляра анонимного внутреннего класса.

public class Parcel7 {
    public Contents contents() {
        return new Contents() { // Вставка определения класса
            private int i = 11;
            public int value() { return i; }
        }; // Точка с запятой здесь необходима
    }
    public static void main(String[] args) {
        Parcel7 p = new Parcel7();
        Contents c = p.contents();
    }
} ///:~

```

Метод `contents()` объединяет создание возвращаемого значения с определением класса, представляющего это возвращаемое значение! Кроме того, класс анонимен, то есть у него нет имени. Ситуация усугубляется тем, что на первый взгляд мы собираемся создать объект `Contents`. А потом, не добравшись до точки с запятой, мы говорим: «Погодите-ка, надо вставить определение класса». В действительности этот странный синтаксис означает: «Создать объект анонимного класса, производного от `Contents`». Ссылка, возвращаемая новым выражением, автоматически преобразуется в ссылку на `Contents`. В действительности синтаксис анонимного внутреннего класса представляет собой сокращенную запись:

```
//: innerclasses/Parcel7b.java
// Расширенная версия Parcel7.java

public class Parcel7b {
    class MyContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    public Contents contents() { return new MyContents(); }
    public static void main(String[] args) {
        Parcel7b p = new Parcel7b();
        Contents c = p.contents();
    }
} ///:~
```

В анонимном внутреннем классе `Contents` создается конструктором по умолчанию.

Следующий код показывает, как следует действовать, если базовому классу нужен конструктор с аргументом:

```
//: innerclasses/Parcel8.java
// Вызов конструктора базового класса.

public class Parcel8 {
    public Wrapping wrapping(int x) {
        // Вызов конструктора базового класса:
        return new Wrapping(x) { // Передача аргумента конструктору
            public int value() {
                return super.value() * 47;
            }
        }; // Точка с запятой необходима
    }
    public static void main(String[] args) {
        Parcel8 p = new Parcel8();
        Wrapping w = p.wrapping(10);
    }
} ///:~
```

А именно вы просто передаете соответствующий аргумент конструктору базового класса — в нашем примере это значение `x`, передаваемое `new Wrapping(x)`. И хотя это самый обычный класс с реализацией, `Wrapping` также используется как общий «интерфейс» для его производных классов:

```
//: innerclasses/Wrapping.java
public class Wrapping {
    private int i;
```

```
public Wrapping(int x) { i = x; }
public int value() { return i; }
} ///:~
```

Класс `Wrapping` содержит конструктор с аргументом, чтобы ситуация была чуть более интересной.

Точка с запятой в конце анонимного внутреннего класса отмечает не конец тела класса, а конец выражения, содержащего анонимный класс. Таким образом, данное использование точки с запятой ничем не отличается от ее использования в любом другом месте.

Инициализация также может выполняться при определении полей в анонимном классе:

```
//: innerclasses/Parcel9.java
// Анонимный внутренний класс, выполняющий инициализацию
// (сокращенная версия Parcel5.java).
```

```
public class Parcel9 {
    // Для использования в анонимном внутреннем классе
    // аргумент должен быть объявлен как final.
    public Destination destination(final String dest) {
        return new Destination() {
            private String label = dest;
            public String readLabel() { return label; }
        };
    }
    public static void main(String[] args) {
        Parcel9 p = new Parcel9();
        Destination d = p.destination("Танзания");
    }
} ///:~
```

Если вы определяете анонимный внутренний класс и хотите использовать объект, определенный внутри анонимного внутреннего класса, компилятор требует, чтобы ссылка на аргумент была объявлена как `final` (см. аргумент `destination()`). При нарушении этого требования компилятор выдает сообщение об ошибке.

Пока вы ограничиваетесь простым присваиванием, приведенное в примере решение работает нормально. А если вам потребуется выполнить операции, сходные с операциями конструктора? Анонимный класс не может содержать именованный конструктор (так как у такого конструктора не может быть имени!), но при инициализации экземпляра можно фактически моделировать конструктор для анонимного внутреннего класса:

```
//: innerclasses/AnonymousConstructor.java
// Создание конструктора для анонимного внутреннего класса.
import static net.mindview.util.Print.*;
```

```
abstract class Base {
    public Base(int i) {
        print("Базовый конструктор, i = " + i);
    }
    public abstract void f();
}
```

```
public class AnonymousConstructor {
    public static Base getBase(int i) {
        return new Base(i) {
```

продолжение ➤

```

    { print("В инициализаторе экземпляра"); }
    public void f() {
        print("В анонимном f()");
    }
};
}
public static void main(String[] args) {
    Base base = getBase(47);
    base.f();
}
} /* Output:
Базовый конструктор, i = 47
В инициализаторе экземпляра
В анонимном f()
*///:~

```

В данном случае переменная *i* *не обязана* быть `final`. Хотя значение *i* передается базовому конструктору анонимного класса, оно никогда не используется напрямую в анонимном классе.

Ниже приведена вариация на тему `Parcel` с инициализацией экземпляра. Помните, что аргументы `destination()` должны быть объявлены с ключевым словом `final`, так как они будут использоваться в анонимном классе:

```

//: innerclasses/Parcel10.java
// Использование "инициализации экземпляра" для выполнения
// конструирования в анонимном внутреннем классе.

public class Parcel10 {
    public Destination
    destination(final String dest, final float price) {
        return new Destination() {
            private int cost;
            // Инициализация экземпляра для каждого объекта:
            {
                cost = Math.round(price);
                if(cost > 100)
                    System.out.println("Превышение бюджета!");
            }
            private String label = dest;
            public String readLabel() { return label; }
        };
    }
    public static void main(String[] args) {
        Parcel10 p = new Parcel10();
        Destination d = p.destination("Танзания", 101.395F);
    }
} /* Output:
Превышение бюджета!
*///:~

```

В инициализаторе экземпляра присутствует код, который не может быть выполнен как часть инициализатора поля (а именно команда `if`). Таким образом, по сути инициализатор экземпляра является конструктором для анонимного внутреннего класса. Конечно, эта модель имеет свои ограничения; инициализаторы экземпляров не могут перегружаться, поэтому такой «конструктор» может быть только один.

Анонимные внутренние классы обладают ограниченными возможностями по сравнению с обычным наследованием: они могут *либо* расширять класс, *либо* реализовать интерфейс. И если вы реализуете интерфейс, то можете реализовать только один.

10. (1) Повторите упражнение 7 с анонимным внутренним классом.
11. (1) Повторите упражнение 9 с анонимным внутренним классом.
12. (1) Измените пример `interfaces/HorrorShow.java`, реализовав `DangerousMonster` и `Vampire` как анонимные классы.
13. (2) Создайте класс, не содержащий конструктор по умолчанию (конструктор без аргументов). При этом класс должен содержать конструктор, получающий аргументы. Создайте второй класс с методом, который возвращает ссылку на объект первого класса. Возвращаемый объект должен создаваться посредством анонимного внутреннего класса, производного от первого.

Снова о паттерне «Фабричный метод»

Сравните, насколько элегантнее выглядит пример `interfaces/Factories.java` при использовании анонимных внутренних классов:

```
//: innerclasses/Factories.java
import static net.mindview.util.Print.*;

interface Service {
    void method1();
    void method2();
}

interface ServiceFactory {
    Service getService();
}

class Implementation1 implements Service {
    private Implementation1() {}
    public void method1() {print("Implementation1 method1");}
    public void method2() {print("Implementation1 method2");}
    public static ServiceFactory factory =
        new ServiceFactory() {
            public Service getService() {
                return new Implementation1();
            }
        };
}

class Implementation2 implements Service {
    private Implementation2() {}
    public void method1() {print("Implementation2 method1");}
    public void method2() {print("Implementation2 method2");}
    public static ServiceFactory factory =
        new ServiceFactory() {
            public Service getService() {
                return new Implementation2();
            }
        };
}
```

продолжение ➤

```

public class Factories {
    public static void serviceConsumer(ServiceFactory fact) {
        Service s = fact.getService();
        s.method1();
        s.method2();
    }
    public static void main(String[] args) {
        serviceConsumer(Implementation1.factory);
        // Реализации полностью взаимозаменяемы:
        serviceConsumer(Implementation2.factory);
    }
} /* Output:
Implementation1 method1
Implementation1 method2
Implementation2 method1
Implementation2 method2
*///:~

```

Теперь конструкторы `Implementation1` и `Implementation2` могут быть закрытыми, и создавать в качестве фабрики именованный класс уже нет необходимости. Кроме того, часто необходим только один фабричный объект, поэтому в данном примере он создается как статическое поле в реализации `Service`. В результате синтаксис также становится более осмысленным.

Пример `interfaces/Games.java` также можно улучшить при помощи анонимных внутренних классов:

```

//: innerclasses/Games.java
// Использование анонимных внутренних классов в системе Game.
import static net.mindview.util.Print.*;

interface Game { boolean move(); }
interface GameFactory { Game getGame(); }

class Checkers implements Game {
    private Checkers() {}
    private int moves = 0;
    private static final int MOVES = 3;
    public boolean move() {
        print("Checkers move " + moves);
        return ++moves != MOVES;
    }
    public static GameFactory factory = new GameFactory() {
        public Game getGame() { return new Checkers(); }
    };
}

class Chess implements Game {
    private Chess() {}
    private int moves = 0;
    private static final int MOVES = 4;
    public boolean move() {
        print("Chess move " + moves);
        return ++moves != MOVES;
    }
    public static GameFactory factory = new GameFactory() {
        public Game getGame() { return new Chess(); }
    };
}

```



```
};  
}  
  
public class Games {  
    public static void playGame(GameFactory factory) {  
        Game s = factory.getGame();  
        while(s.move())  
            ;  
    }  
    public static void main(String[] args) {  
        playGame(Checkers.factory);  
        playGame(Chess.factory);  
    }  
}  
/* Output:  
Checkers move 0  
Checkers move 1  
Checkers move 2  
Chess move 0  
Chess move 1  
Chess move 2  
Chess move 3  
*///:~
```

Запомните совет, приведенный в конце предыдущей главы: в общем случае рекомендуется отдавать предпочтение классам перед интерфейсами. Если в вашей архитектуре нужен интерфейс, вы это поймете. В остальных случаях не используйте интерфейс без необходимости.

14. (1) Измените решение упражнения 11 из главы 9 так, чтобы в нем использовались анонимные внутренние классы.
15. (1) Измените решение упражнения 19 из главы 9 так, чтобы в нем использовались анонимные внутренние классы.

Вложенные классы

Если связь между объектом внутреннего класса и объектом внешнего класса вам не нужна, внутренний класс можно сделать статическим (объявить его как `static`). Часто такой класс называют *вложенным*¹ (nested). Для того чтобы понять значение ключевого слова `static` в отношении внутренних классов, вы должны вспомнить, что объект обычного внутреннего класса скрыто хранит ссылку на объект создавшего его объемлющего внешнего класса. Это условие не выполняется для внутренних классов, объявленных с ключевым словом `static`. Вложенный класс обладает следующими характеристиками.

1. Для создания его объекта не понадобится объект внешнего класса.
2. Вы не можете обращаться к членам не-статического объекта внешнего класса из объекта вложенного класса.

¹ Примерно то же самое, что и вложенные классы C++, за тем исключением, что в Java они способны обращаться к закрытым членам внешнего класса.

Есть и еще одно отличие между вложенными и обычными внутренними классами. Поля и методы обычного внутреннего класса определяются только на уровне внешнего класса, поэтому обычные внутренние классы не могут объявлять свои данные, поля и классы как `static`. Но вложенные классы не имеют таких ограничений:

```
//: innerclasses/Parcel11.java
// Вложенные (статические внутренние) классы.

public class Parcel11 {
    private static class ParcelContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected static class ParcelDestination
    implements Destination {
        private String label;
        private ParcelDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
        // Вложенные классы могут содержать другие статические элементы:
        public static void f() {}
        static int x = 10;
        static class AnotherLevel {
            public static void f() {}
            static int x = 10;
        }
    }
    public static Destination destination(String s) {
        return new ParcelDestination(s);
    }
    public static Contents contents() {
        return new ParcelContents();
    }
    public static void main(String[] args) {
        Contents c = contents();
        Destination d = destination("Танзания");
    }
} ///:~
```

В методе `main()` объект класса `Parcel11` не нужен; вместо этого используется обычная форма обращения к статическим членам класса для вызова методов, возвращающих ссылки на `Contents` и `Destination`.

Как вы вскоре узнаете, в обычном (нестатическом) внутреннем классе связь с объектом внешнего класса создается при помощи специальной ссылки `this`. Во вложенном классе этой специальной ссылки нет, и здесь наблюдается аналогия со статическим методом.

- 16.** (1) Создайте класс, содержащий вложенный класс. Создайте в методе `main()` экземпляр вложенного класса.
- 17.** (2) Создайте класс, содержащий внутренний класс, который, в свою очередь, содержит внутренний класс. Повторите управление с вложенными классами. Обратите внимание на имена файлов `.class`, создаваемых компилятором.

Классы внутри интерфейсов

Обычно в интерфейс нельзя помещать код, но вложенный класс *может* быть частью интерфейса. Любой класс, помещенный в интерфейс, автоматически объявляется как открытый (`public`) и статический (`static`). Так как класс объявляется как `static`, он не нарушает правил обращения с интерфейсом — этот вложенный класс всего лишь размещается в пространстве имен интерфейса. Вы даже можете реализовать окружающий интерфейс во внутреннем классе:

```
//: innerclasses/ClassInInterface.java
// {main: ClassInInterface$Test}

public interface ClassInInterface {
    void howdy();
    class Test implements ClassInInterface {
        public void howdy() {
            System.out.println("Привет!");
        }
        public static void main(String[] args) {
            new Test().howdy();
        }
    }
} /* Output:
Привет!
*///:~
```

Вложение классов в интерфейсы особенно удобно при создании общего кода, который должен использоваться со всеми реализациями этого интерфейса.

Ранее в книге я предлагал помещать в каждый класс метод `main()`, чтобы при необходимости проверять работоспособность класса. Недостатком такого подхода является дополнительный скомпилированный код, увеличивающий размеры программы. Если для вас это важно, вы можете использовать статический внутренний класс в качестве «контейнера» для тестового кода:

```
//: innerclasses/TestBed.java
// Размещение тестового кода во вложенном классе.
// {main: TestBed$Tester}

public class TestBed {
    public void f() { System.out.println("f()"); }
    public static class Tester {
        public static void main(String[] args) {
            TestBed t = new TestBed();
            t.f();
        }
    }
} /* Output:
f()
*///:~
```

При компиляции этого файла генерируется отдельный класс с именем `TestBed$Tester` (для запуска тестового кода наберите команду `java TestBed$Tester`). Вы можете использовать этот класс для тестирования, но включать его в окончательную версию программы необязательно — можно просто удалить файл `TestBed$Tester.class` перед окончательной сборкой программы.

- 18.** (1) Создайте интерфейс, содержащий вложенный класс. Реализуйте интерфейс и создайте экземпляр вложенного класса.
- 19.** (2) Создайте интерфейс с вложенным классом, содержащим статический метод, который вызывает методы вашего интерфейса и выводит результаты. Реализуйте интерфейс и передайте экземпляр своей реализации методу.

Доступ вовне из многократно вложенных классов

Неважно, насколько глубоко был вложен внутренний класс,¹ — у него есть непосредственный доступ ко всем членам всех классов, в которые он встроен. Проиллюстрируем это следующей программой:

```
///  
// Вложенные классы могут обращаться ко всем членам  
// всех уровней классов, в которые они вложены.
```

```
class MNA {  
    private void f() {}  
    class A {  
        private void g() {}  
        public class B {  
            void h() {  
                g();  
                f();  
            }  
        }  
    }  
}  
  
public class MultiNestingAccess {  
    public static void main(String[] args) {  
        MNA mna = new MNA();  
        MNA.A mnaa = mna.new A();  
        MNA.A.B mnaab = mnaa.new B();  
        mnaab.h();  
    }  
} ///:~
```

Вы можете видеть, что в классе `MNA.A` методы `f()` и `g()` вызываются без дополнительных уточнений (несмотря на то, что они объявлены как `private`). Этот пример также демонстрирует синтаксис, который следует использовать при создании объектов внутренних классов произвольного уровня вложенности из другого класса. Выражение `.new` дает верную область действия, и вам не приходится уточнять его полное имя при вызове конструктора.

Внутренние классы: зачем?

К настоящему моменту мы рассмотрели множество форм записи и способов работы внутренних классов, но это не дало ответа на вопрос, зачем они вообще нужны. Что же заставило фирму Sun добавить в язык настолько фундаментальное свойство?

¹ Снова благодарю Мартина Даннера.

Обычно внутренний класс наследует от класса или реализует интерфейс, а код внутреннего класса манипулирует объектом внешнего класса, в котором он был создан. Значит, можно сказать, что внутренний класс — это нечто вроде «окна» во внешний класс.

Возникает естественный вопрос: «Если мне нужна ссылка на интерфейс, почему бы внешнему классу не реализовать этот интерфейс?» Ответ здесь таков: «Если это все, что вам нужно, — значит, так и следует поступить». Но что же отличает внутренний класс, реализующий интерфейс, от внешнего класса, реализующего тот же интерфейс? Далеко не всегда удается использовать удобство интерфейсов — иногда приходится работать и с реализацией. Поэтому наиболее веская причина для использования внутренних классов формулируется так:

Каждый внутренний класс способен независимо наследовать определенную реализацию. Таким образом, внутренний класс не ограничен при наследовании в ситуациях, где внешний класс уже наследует реализацию.

Без способности внутренних классов наследовать (фактически) реализацию более чем одного конкретного или абстрактного класса некоторые задачи планирования и программирования имели бы крайне сложное решение. Поэтому внутренний класс выступает как «довесок» множественного наследования. Интерфейсы берут на себя часть этой задачи, в то время как внутренние классы фактически обеспечивают «множественное наследование реализации». То есть внутренние классы позволяют вам наследовать от нескольких «не-интерфейсов».

Чтобы понять сказанное, рассмотрим ситуацию, где два интерфейса тем или иным способом должны быть реализованы в классе. Вследствие гибкости интерфейсов у вас есть два варианта: одиночный класс или внутренний класс:

```
//: innerclasses/MultiInterfaces.java
// Два способа реализации нескольких интерфейсов
// в одном классе.
```

```
interface A {}
interface B {}
```

```
class X implements A, B {}
```

```
class Y implements A {
    B makeB() {
        // Анонимный внутренний класс:
        return new B() {};
    }
}
```

```
public class MultiInterfaces {
    static void takesA(A a) {}
    static void takesB(B b) {}
    public static void main(String[] args) {
        X x = new X();
        Y y = new Y();
        takesA(x);
        takesA(y);
        takesB(x);
        takesB(y.makeB());
    }
} ///:~
```

Конечно, выбор того или иного способа организации кода зависит от конкретной ситуации. Впрочем, сама решаемая вами задача должна подсказать, что для нее предпочтительно: один отдельный класс или внутренний класс. Но при отсутствии иных ограничений оба подхода, использованные в рассмотренном примере, ничем не отличаются с точки зрения реализации. Оба они работают.

Однако, если вместо интерфейсов у вас имеются конкретные или абстрактные классы, придется «звать на помощь» внутренние классы, если новый класс должен как-то задействовать функциональность двух других классов:

```

//: innerclasses/MultiImplementation.java
// При использовании конкретных или абстрактных классов
// внутренние классы предоставляют единственный способ
// провести "множественное наследование реализации".
package innerclasses;

class D {}
abstract class E {}

class Z extends D {
    E makeE() { return new E() {};}
}

public class MultiImplementation {
    static void takesD(D d) {}
    static void takesE(E e) {}
    public static void main(String[] args) {
        Z z = new Z();
        takesD(z);
        takesE(z.makeE());
    }
} ///:~

```

Если вам не приходится решать задачу «множественного наследования реализации», скорее всего, вы сможете написать любую программу без использования особенностей внутренних классов. С другой стороны, внутренние классы открывают следующие дополнительные возможности.

1. У внутреннего класса может существовать произвольное количество экземпляров, каждый из которых содержит собственную информацию, не зависящую от состояния объекта внешнего класса.
2. Один внешний класс может содержать несколько внутренних классов, которые по-разному реализуют один интерфейс или наследуют от единственного базового класса. Пример такой конструкции вскоре будет рассмотрен.
3. Место создания объекта внутреннего класса не привязано к месту и времени создания объекта внешнего класса.
4. Внутренний класс не создает взаимосвязи классов типа «является тем-то», способной вызвать путаницу; он представляет собой отдельную сущность.

Например, если в программе `Sequence.java` отсутствовали бы внутренние классы, то нам пришлось бы заявить, что «класс `Sequence` есть класс `Selector`», и при этом ограничиться только одним объектом `Selector` для конкретного объекта `Sequence`. А вы

можете с легкостью определить второй метод, `reverseSelector()`, создающий объект `Selector` для перебора элементов `Sequence` в обратном порядке. Такую гибкость дают только внутренние классы.

20. (2) Реализуйте `reverseSelector()` в `Sequence.java`.

21. (4) Создайте интерфейс `U` с тремя методами. Создайте класс `A` с методом, который создает ссылку на `U` посредством построения анонимного внутреннего класса. Создайте второй класс `B`, который содержит массив `U`. Класс `B` содержит один метод, который получает и сохраняет ссылку на `U` в массиве; второй метод, который сбрасывает ссылку в массиве (определяемую аргументом метода) в состояние `null`; и третий метод, который перебирает элементы массива и вызывает методы `U`. В методе `main()` создайте группу объектов `A` и один объект `B`. Заполните объект `B` ссылками `U`, произведенными объектами `A`. Используйте `B` для выполнения обратных вызовов по всем объектам `A`. Удалите некоторые ссылки на `U` из `B`.

Замыкания и обратные вызовы

Замыкание (closure) — это вызываемый объект, который сохраняет информацию о контексте, где он был создан. Из этого определения видно, что внутренний класс является объектно-ориентированным замыканием, поскольку он содержит не только всю информацию об объекте внешнего класса («место создания»), но к тому же у него есть ссылка на весь объект внешнего класса, с помощью которой он может манипулировать всеми членами этого объекта, включая его закрытые (`private`) члены.

При обсуждении того, стоит ли включать в Java некоторый род указателей, самым веским аргументом «за» была возможность *обратных вызовов* (callback). В механизме обратного вызова некоторому стороннему объекту дается информация, позволяющая ему затем вызвать объект, который произвел изначальный вызов. Это очень мощная концепция программирования, и вы убедитесь в этом позднее. С другой стороны, если обратный вызов реализуется по указателю, вам придется рассчитывать на то, что программист будет соблюдать правила и не станет злоупотреблять указателем. Вы уже видели и снова убедитесь, что создатели языка Java действовали более осторожно, поэтому указатели в него включены не были.

Замыкание, предоставляемое внутренним классом, — идеальное решение; гораздо более гибкое и безопасное, чем указатель. Рассмотрим пример:

```
//: innerclasses/Callbacks.java
// Использование внутренних классов
// для реализации обратных вызовов
package innerclasses;
import static net.mindview.util.Print.*;

interface Incrementable {
    void increment();
}

// ПРОСТО реализуем интерфейс:
class Callee1 implements Incrementable {
    private int i = 0;
    public void increment() {
        i++;
    }
}
```

```

        System.out.println(i);
    }
}

class MyIncrement {
    public void increment() { System.out.println("Другая операция"); }
    public static void f(MyIncrement mi) { mi.increment(); }
}

// Если ваш класс должен вызывать метод increment()
// по-другому, необходимо использовать внутренний класс:
class Callee2 extends MyIncrement {
    private int i = 0;
    private void increment() {
        super.increment();
        i++;
        print(i);
    }
    private class Closure implements Incrementable {
        public void increment() {
            // Указывается метод внешнего класса, иначе
            // возникнет бесконечная рекурсия:
            Callee2.this.increment();
        }
    }
    Incrementable getCallbackReference() {
        return new Closure();
    }
}

class Caller {
    private Incrementable callbackReference;
    Caller(Incrementable cbh) { callbackReference = cbh; }
    void go() { callbackReference.increment(); }
}

public class Callbacks {
    public static void main(String[] args) {
        Callee1 c1 = new Callee1();
        Callee2 c2 = new Callee2();
        MyIncrement.f(c2);
        Caller caller1 = new Caller(c1);
        Caller caller2 = new Caller(c2.getCallbackReference());
        caller1.go();
        caller1.go();
        caller2.go();
        caller2.go();
    }
} /* Output:
Другая операция
1
1
2
Другая операция
2
Другая операция
3
*///:~

```


Этот пример также показывает, какая существует разница при реализации интерфейса внешним или внутренним классом. Класс `Callee1` — наиболее очевидное решение задачи с точки зрения написания программы. Класс `Callee2` наследует от класса `MyIncrement`, в котором уже есть метод `increment()`, выполняющий действие, никак не связанное с тем, что ожидает от него интерфейс `Incrementable`. Когда класс `MyIncrement` расширяется для получения `Callee2`, метод `increment()` нельзя переопределить для использования в качестве метода интерфейса `Incrementable`, поэтому приходится обратиться к отдельной реализации во внутреннем классе. Также следует отметить, что создание внутреннего класса не затрагивает и не изменяет существующий интерфейс внешнего класса.

Обратите внимание: все элементы, за исключением метода `getCallbackReference()`, в классе `Callee2` являются закрытыми. Для того чтобы установить *любое* соединение с окружающим миром, необходим интерфейс `Incrementable`. Здесь можно видеть, как интерфейсы способствуют полному отделению интерфейса от реализации.

Внутренний класс `Closure` просто реализует интерфейс `Incrementable`, предоставляя при этом связь с объектом `Callee2`, но связь эта безопасна. Кто бы ни получил ссылку на `Incrementable`, он в состоянии вызвать только метод `increment()`, и других возможностей у него нет (в отличие от указателя, с которым программист может вытворять все, что вздумается).

Класс `Caller` берет ссылку на `Incrementable` в своем конструкторе (хотя передача ссылки для обратного вызова может происходить в любое время), а после этого, чуть позже, использует ссылку для «обратного вызова» объекта `Callee`.

Ценность обратного вызова кроется в его гибкости — вы можете динамически выбирать функции, выполняемые во время работы программы. Практическая выгода такого подхода станет более очевидной в главе 14, равномерно насыщенной обратными вызовами для реализации графического интерфейса пользователя (GUI).

Внутренние классы и система управления

Реальный пример использования внутренних классов можно обнаружить, рассмотрев то, что я буду называть здесь *системой управления* (control framework).

Каркас приложения (application framework) — это класс или набор классов, разработанных для решения определенного круга задач. Для применения каркаса приложения вы наследуете один или несколько классов и переопределяете некоторые методы. Код, который вы пишете в переопределенных методах, адаптирует предоставляемое каркасом приложения решение общего характера к вашим конкретным нуждам (это пример *паттерна проектирования* «Шаблонный метод» — см. книгу *Thinking In Patterns (with Java)*, доступную на www.MindView.net). Шаблонный метод содержит базовую структуру алгоритма и вызывает один или несколько переопределяемых методов для завершения работы алгоритма. Паттерн отделяет постоянные аспекты от переменных; в данном случае «Шаблонный метод» определяет постоянную часть, а переопределяемые методы — переменные аспекты.

Система управления представляет собой определенный тип каркаса приложения, построенный на необходимости реагировать на события; система, в обязанности которой

входит в основном реакция на события, называется *системой, управляемой по событиям* (event-driven system). Одна из самых важных задач в программировании приложений — создание графического интерфейса пользователя (GUI), всецело и полностью обусловленного событиями. Как вы увидите в главе 14, библиотека Java Swing из стандартного набора Java представляет собой систему управления, элегантно решающую задачу создания GUI, при этом в ней широко используются внутренние классы.

Как наглядный пример легкости создания и использования системы управления с внутренними классами, рассмотрим систему, ориентированную на обработку событий по их «готовности». Хотя «готовность» может значить все что угодно, в нашем случае по умолчанию будем отталкиваться от счетчика времени. Пусть эта система управления чем-то управляет, с точностью до неизвестного «чем-то». Нужная информация предоставляется посредством наследования, при реализации части `action()`.

Начнем с определения интерфейса, описывающего всякое событие системы. Здесь использован абстрактный класс, а не интерфейс, поскольку мы решили координировать поведение, основываясь на времени. Тогда реализация будет включать в себя:

```
//: innerclasses/controller/Event.java
// Общие для всякого управляющего события методы.
package innerclasses.controller;

public abstract class Event {
    private long eventTime;
    protected final long delayTime;
    public Event(long delayTime) {
        this.delayTime = delayTime;
        start();
    }
    public void start() { // С возможностью перезапуска
        eventTime = System.nanoTime() + delayTime;
    }
    public boolean ready() {
        return System.nanoTime() >= eventTime;
    }
    public abstract void action();
} ///:~
```

Конструктор просто запоминает время (отсчитанное от момента создания объекта), когда необходимо выполнить событие `Event`, и после этого вызывает метод `start()`, который прибавляет к текущему времени интервал задержки, чтобы вычислить время происхождения события. Метод `start()` отделен от конструктора, благодаря чему становится возможным «перезапустить» событие после того, как его время уже истекло, таким образом объект `Event` можно использовать многократно. К примеру, если вам понадобится повторяющееся событие, нужно будет просто добавить вызов `start()` в метод `action()`.

Метод `ready()` сообщает, что пора действовать — вызывать метод `action()`. Конечно, метод `ready()` может быть переопределен любым производным классом, если ему понадобится не временное, а иное условие для события `Event`.

Следующий файл описывает саму систему управления, которая распоряжается событиями и запускает их обработку. Объекты `Event` содержатся в контейнере `List<Event>`, узнать подробнее о котором вы сможете в главе 11. На данный момент достаточно

знать, что метод `add()` присоединяет объект `Event` к концу контейнера `List`, метод `size()` возвращает количество хранимых в контейнере элементов, конструкция `foreach` последовательно перебирает объекты `Event` из `List`, а метод `remove()` удаляет элемент из заданной позиции списка:

```
//: innerclasses/controller/Controller.java
// Вместе с классом Event составляет систему
// управления общего характера:
import java.util.*;

public class Controller {
    // Класс из пакета java.util для хранения объектов Event:
    private List<Event> eventList = new ArrayList<Event>();
    public void addEvent(Event c) { eventList.add(c); }
    public void run() {
        while(eventList.size() > 0)
            // Создать копию, чтобы избежать модификации списка
            // во время выборки элементов:
            for(Event e : new ArrayList<Event>(eventList))
                if(e.ready()) {
                    System.out.println(e);
                    e.action();
                    eventList.remove(e);
                }
    }
} ///:~
```

Метод `run()` циклически просматривает копию `eventList` в поиске событий `Event`, которые готовы для выполнения. Для каждого элемента списка, метод `ready()` которого возвращает `true`, он распечатывает объект с помощью метода `toString()`, вызывает метод `action()`, а после этого удаляет событие из списка.

Заметьте, пока что вам ничего не известно о том, *что* конкретно выполняет некое событие `Event`. В этом и состоит «изюминка» разработанной системы; она отделяет постоянную составляющую от изменяющейся. Или, если использовать мой термин «вектор изменения» — разницу в ответных действиях на разнообразные события `Event`, то описание новых событий состоит в создании различных подклассов `Event`.

На данном этапе в игру вступают внутренние классы. Они позволяют достичь двух целей:

1. Создать полную реализацию приложения на основе системы управления в одном классе, таким образом инкапсулируя все, что относится только к этой реализации. Внутренние классы используются для описания различных событий в отдельных методах `action()`, чтобы решить поставленную задачу.
2. Внутренние классы помогают поддерживать реализацию в «нужной форме», так как у них есть доступ к внешнему классу. Без этой возможности любое изменение кода могло бы стать проблемой и пришлось бы искать другие пути решения.

Рассмотрим конкретную реализацию системы управления, разработанную для управления функциями оранжереи¹. Каждое событие абсолютно самостоятельно: включение

¹ По некоторым причинам я всегда решал эту задачу с особым удовольствием; это началось еще с одной из первых моих книг *C++ Inside & Out*, но Java-реализация — гораздо более элегантная.

света, воды и нагревателей, звонок и перезапуск системы. Но система управления спроектирована так, что легко разделяет этот разный код. Внутренние классы помогают унаследовать несколько разных классов от одного базового класса `Event` в пределах одного класса. Для каждого типа события от `Event` наследуется новый внутренний класс, и в его методе `action()` записывается управляющий код.

Как общепринято при использовании каркасов приложений, класс `GreenhouseControls` унаследован от класса `Controller`:

```
//: innerclasses/GreenhouseControls.java
// Пример конкретного приложения на основе системы
// управления, все находится в одном классе. Внутренние
// классы дают возможность инкапсулировать различную
// функциональность для каждого отдельного события.
import innerclasses.controller.*;

public class GreenhouseControls extends Controller {
    private boolean light = false;
    public class LightOn extends Event {
        public LightOn(long delayTime) { super(delayTime); }
        public void action() {
            // Поместите сюда код управления оборудованием,
            // выполняющий непосредственное включение света.
            light = true;
        }
        public String toString() { return "Свет включен"; }
    }
    public class LightOff extends Event {
        public LightOff(long delayTime) { super(delayTime); }
        public void action() {
            // Поместите сюда код управления оборудованием,
            // выполняющий выключение света.
            light = false;
        }
        public String toString() { return "Свет выключен"; }
    }

    private boolean water = false;
    public class WaterOn extends Event {
        public WaterOn(long delayTime) { super(delayTime); }
        public void action() {
            // Здесь размещается код управления оборудованием.
            water = true;
        }
        public String toString() {
            return "Полив включен";
        }
    }
    public class WaterOff extends Event {
        public WaterOff(long delayTime) { super(delayTime); }
        public void action() {
            // Здесь размещается код управления оборудованием.
            water = false;
        }
        public String toString() {
            return "Полив выключен";
        }
    }
}
```

```
private String thermostat = "День";
public class ThermostatNight extends Event {
    public ThermostatNight(long delayTime) {
        super(delayTime);
    }
    public void action() {
        // Здесь размещается код управления оборудованием.
        thermostat = "Ночь";
    }
    public String toString() {
        return "Термостат использует ночной режим";
    }
}
public class ThermostatDay extends Event {
    public ThermostatDay(long delayTime) {
        super(delayTime);
    }
    public void action() {
        // Здесь размещается код управления оборудованием.
        thermostat = "День";
    }
    public String toString() {
        return "Термостат использует дневной режим";
    }
}
// Пример метода action(), вставляющего новый экземпляр
// самого себя в список событий:
public class Bell extends Event {
    public Bell(long delayTime) { super(delayTime); }
    public void action() {
        addEvent(new Bell(delayTime));
    }
    public String toString() { return "Бам!"; }
}
public class Restart extends Event {
    private Event[] eventList;
    public Restart(long delayTime, Event[] eventList) {
        super(delayTime);
        this.eventList = eventList;
        for(Event e : eventList)
            addEvent(e);
    }
    public void action() {
        for(Event e : eventList) {
            e.start(); // Перезапуск каждого события
            addEvent(e);
        }
        start(); // Перезапуск текущего события
        addEvent(this);
    }
    public String toString() {
        return "Перезапуск системы";
    }
}
public static class Terminate extends Event {
    public Terminate(long delayTime) { super(delayTime); }
    public void action() { System.exit(0); }
    public String toString() { return "Отключение"; }
}
} ///:~
```

Заметьте, что поля `light`, `water` и `thermostat` принадлежат внешнему классу `GreenhouseControls`, и все же внутренние классы имеют возможность обращаться к ним, не используя особой записи и не запрашивая особых разрешений. Также в методы `action()` обычно включается код управления оборудованием.

Большая часть событий `Event` выглядит схоже, однако классы `Bell` и `Restart` представляют собой особые случаи. `Bell` выдает звуковой сигнал, а затем добавляет новый объект `Bell` в список ожидания событий, чтобы звонок сработал снова. Заметьте, что внутренние классы действуют *почти* как множественное наследование: классы `Bell` и `Restart` содержат все методы класса `Event`, а также выглядят так, словно они содержат все методы внешнего класса `GreenhouseControls`.

Классу `Restart` передается массив событий `Event`, которые он добавляет в контроллер. Так как `Restart` также является объектом `Event`, вы можете добавить этот объект в список событий в методе `Restart.action()`, чтобы система регулярно перезапускалась.

Следующий класс настраивает систему, создавая объект `GreenhouseControls` и добавляя в него разнообразные типы объектов `Event`. Это пример паттерна проектирования «Команда» — каждый объект в `eventList` представляет запрос, инкапсулирующий объект:

```

//: innerclasses/GreenhouseController.java
// Настройка и запуск системы управления.
// {Args: 5000}
import innerclasses.controller.*;

public class GreenhouseController {
    public static void main(String[] args) {
        GreenhouseControls gc = new GreenhouseControls();
        // Вместо жесткой фиксации параметров в коде
        // можно было бы считать информацию
        // из текстового файла:
        gc.addEvent(gc.new Bell(900));
        Event[] eventList = {
            gc.new ThermostatNight(0),
            gc.new LightOn(200),
            gc.new LightOff(400),
            gc.new WaterOn(600),
            gc.new WaterOff(800),
            gc.new ThermostatDay(1400)
        };
        gc.addEvent(gc.new Restart(2000, eventList));
        if(args.length == 1)
            gc.addEvent(
                new GreenhouseControls.Terminate(
                    new Integer(args[0])));
        gc.run();
    }
} /* Output:
Бам!
Термостат использует ночной режим
Свет включен
Свет выключен
Полив включен
Полив выключен
Термостат использует дневной режим
Перезапуск системы
Отключение
*///:~

```

Этот класс инициализирует систему и добавляет в нее нужные события. Событие `Restart` выполняется многократно, при этом оно каждый раз загружает список `eventList` в объект `GreenhouseControls`. Если передать в командной строке аргумент с интервалом в миллисекундах, программа завершится после истечения заданного интервала (эта возможность используется для тестирования).

Конечно, более гибким способом стала бы не запись последовательности событий прямо в код, а чтение их из файла. (В упражнении главы 12 вам будет предложено внести именно такое изменение в данный пример.)

Разобравшись с этим примером, вы сможете гораздо лучше понять всю ценность механизма внутренних классов, особенно в случае с системами управления. В главе 14 вы увидите, как непринужденно используются внутренние классы для описания событий графического интерфейса пользователя. После того как вы ее прочитаете, вы, без сомнения, станете убежденным поклонником внутренних классов.

- 22.** (2) В программу `GreenhouseControls.java` добавьте внутренние классы для события `Event`, отключающие и включающие проветривание оранжереи. Настройте программу `GreenhouseController.java` на использование нового типа события.
- 23.** (3) Унаследуйте от класса `GreenhouseControls`, чтобы добавить в него новый внутренний класс событий `Event`, включающий и отключающий поддержку увлажнения оранжереи. Напишите новую версию программы `GreenhouseController.java`, обеспечивающую поддержку нового типа события.

Наследование от внутренних классов

Так как конструктор внутреннего класса обязан присоединить к объекту ссылку на окружающий внешний объект, наследование от внутреннего класса получается чуть сложнее, чем обычное наследование. Проблема состоит в том, что «потаянная» ссылка на объект объемлющего внешнего класса *должна быть* инициализирована, однако в производном классе больше не существует внешнего объекта по умолчанию, с которым можно было бы связаться. Ответ на этот вопрос — использование формы записи, позволяющей явно указать объемлющий внешний объект:

```
//: innerclasses/InheritInner.java
// Наследование от внутреннего класса.

class WithInner {
    class Inner {}
}

public class InheritInner extends WithInner.Inner {
    //! InheritInner() {} // Не компилируется
    InheritInner(WithInner wi) {
        wi.super();
    }
    public static void main(String[] args) {
        WithInner wi = new WithInner();
        InheritInner ii = new InheritInner(wi);
    }
} ///:~
```

Здесь класс `InheritInner` расширяет только внутренний класс, а не внешний. Но когда приходит время создавать конструктор, конструктор по умолчанию не подходит, и вы не можете просто передать ссылку на внешний объект. Необходимо использовать синтаксис следующего вида:

```
ссылкаНаВнешнийКласс.super();
```

в теле конструктора. Он обеспечит недостающую ссылку, и программа скомпилируется.

- 24.** (2) Создайте класс с внутренним классом, имеющим конструктор с аргументами (не являющийся конструктором по умолчанию). Создайте второй класс с внутренним классом, наследующим от первого внутреннего класса.

Можно ли переопределить внутренний класс?

Что происходит, если вы создаете внутренний класс, затем наследуете от его внешнего класса, а после этого переопределяете внутренний класс в производном классе? То есть можно ли переопределить весь внутренний класс? Это было бы довольно интересно, но «переопределение» внутреннего класса, как если бы он был еще одним методом внешнего класса, фактически не имеет никакого эффекта:

```
//: innerclasses/BigEgg.java
// Внутренний класс нельзя переопределить
// подобно обычному методу.
import static net.mindview.util.Print.*;

class Egg {
    private Yolk y;
    protected class Yolk {
        public Yolk() { print("Egg.Yolk()"); }
    }
    public Egg() {
        print("New Egg()");
        y = new Yolk();
    }
}

public class BigEgg extends Egg {
    public class Yolk {
        public Yolk() { print("BigEgg.Yolk()"); }
    }
    public static void main(String[] args) {
        new BigEgg();
    }
} /* Output:
New Egg()
Egg.Yolk()
*///:~
```

Конструктор по умолчанию автоматически синтезируется компилятором, а в нем вызывается конструктор по умолчанию из базового класса. Вы могли подумать, что если создается объект `BigEgg`, должен использоваться «переопределенный» класс `Yolk`, но это отнюдь не так, в чем вы можете убедиться, посмотрев на результат работы программы.

Этот пример просто показывает, что при наследовании от внешнего класса никаких дополнительных действий для внутренних классов не производится. Два внутренних класса — совершенно отдельные составляющие, с независимыми пространствами имен. Впрочем, сохранилась возможность явно унаследовать от внутреннего класса:

```
//: innerclasses/BigEgg2.java
// Правильное наследование внутреннего класса.
import static net.mindview.util.Print.*;

class Egg2 {
    protected class Yolk {
        public Yolk() { print("Egg2.Yolk()"); }
        public void f() { print("Egg2.Yolk.f()"); }
    }
    private Yolk y = new Yolk();
    public Egg2() { print("New Egg2()"); }
    public void insertYolk(Yolk yy) { y = yy; }
    public void g() { y.f(); }
}

public class BigEgg2 extends Egg2 {
    public class Yolk extends Egg2.Yolk {
        public Yolk() { print("BigEgg2.Yolk()"); }
        public void f() { print("BigEgg2.Yolk.f()"); }
    }
    public BigEgg2() { insertYolk(new Yolk()); }
    public static void main(String[] args) {
        Egg2 e2 = new BigEgg2();
        e2.g();
    }
} /* Output:
Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()
*///:~
```

Теперь класс `BigEgg2.Yolk` явно расширяет класс `Egg2.Yolk` и переопределяет его методы. Метод `insertYolk()` позволяет классу `BigEgg2` передать один из своих собственных объектов `Yolk` в класс `Egg2`, где тот присоединяется к ссылке `y`, поэтому, когда в методе `g()` вызывается метод `y.f()`, используется переопределенная версия `f()`. Вторым вызовом `Egg2.Yolk()` — это запуск конструктора базового класса из конструктора класса `BigEgg2.Yolk`. Вы также можете обнаружить, что при вызове метода `g()` использовалась переопределенная версия метода `g()`.

Локальные внутренние классы

Как было замечено чуть раньше, внутренние классы также могут создаваться в блоках кода — обычно это тело метода. Локальный внутренний класс не может располагать спецификатором доступа, так как он не является частью внешнего класса, но доступ ко всем неизменным (`final`) переменным текущего блока кода и ко всем членам внешнего класса он имеет. Следующий пример сравнивает процессы создания локального внутреннего класса и анонимного внутреннего класса:

```

//: innerclasses/LocalInnerClass.java
// Хранит последовательность объектов.
import static net.mindview.util.Print.*;

interface Counter {
    int next();
}

public class LocalInnerClass {
    private int count = 0;
    Counter getCounter(final String name) {
        // Локальный внутренний класс:
        class LocalCounter implements Counter {
            public LocalCounter() {
                // У локального внутреннего класса
                // может быть собственный конструктор:
                print("LocalCounter()");
            }
            public int next() {
                printb(name); // Неизменный аргумент
                return count++;
            }
        }
        return new LocalCounter();
    }
    // То же с анонимным внутренним классом:
    Counter getCounter2(final String name) {
        return new Counter() {
            // Анонимный внутренний класс не может содержать
            // именованного конструктора, только инициализатор экземпляра:
            {
                print("Counter()");
            }
            public int next() {
                printb(name); // Неизменный аргумент
                return count++;
            }
        };
    }
    public static void main(String[] args) {
        LocalInnerClass lic = new LocalInnerClass();
        Counter
            c1 = lic.getCounter("Локальный "),
            c2 = lic.getCounter2("Анонимный ");
        for(int i = 0; i < 5; i++)
            print(c1.next());
        for(int i = 0; i < 5; i++)
            print(c2.next());
    }
} /* Output:
LocalCounter()
Counter()
Локальный 0
Локальный 1
Локальный 2
Локальный 3
Локальный 4
*/

```

```
Анонимный 5  
Анонимный 6  
Анонимный 7  
Анонимный 8  
Анонимный 9  
*///:~
```

Объект `Counter` возвращает следующее по порядку значение. Он реализован и как локальный класс, и как анонимный внутренний класс, с одинаковым поведением и характеристиками. Поскольку имя локального внутреннего класса недоступно за пределами метода, единственным доводом в пользу локального внутреннего класса против анонимного внутреннего класса может быть только необходимость в именованном конструкторе и/или перегруженных конструкторах; безымянные внутренние классы вправе задействовать только инициализацию экземпляра.

Единственная причина использования локального внутреннего класса вместо анонимного внутреннего класса — возможность создания более чем одного объекта такого класса.

Идентификаторы внутренних классов

Так как каждый класс компилируется в файл с расширением `.class`, в котором хранятся инструкции по созданию его экземпляров (эта информация помещается в «мета»-класс с именем `Class`), вы могли бы предположить, что внутренние классы также хранят информацию для *своих* объектов `Class` в отдельных файлах. Имена этих файлов/классов строятся по четко определенной «формуле»: имя внешнего класса, затем символ `$` и только после имя внутреннего класса. Например, для программы `LocalInnerClass.java` создаются следующие файлы с расширением `.class`:

```
Counter.class  
LocalInnerClass$2.class  
LocalInnerClass$1LocalCounter.class  
LocalInnerClass.class
```

Если внутренние классы являются анонимными, компилятор использует номера в качестве идентификаторов для таких классов. Если внутренние классы вложены в другие внутренние классы, их имена просто присоединяются после символа `$` и перечисления имен всех внешних классов.

Хотя такая схема генерации встроенных имен проста и прямолинейна, она в то же время надежна и работает практически во всех ситуациях¹. Так как она является стандартной для языка Java, все получаемые файлы автоматически становятся платформенно-независимыми. (Заметьте, чтобы внутренние классы были работоспособны, компилятор изменяет их различными способами.)

¹ С другой стороны, символ `$` является управляющим в системе UNIX, и при просмотре файлов `.class` в этой ОС могут возникнуть проблемы. Это довольно странно, ведь компания Sun (разработчик Java) в основном использует систему UNIX. Я полагаю, что они вообще не обратили внимания на этот аспект, предполагая, что в основном внимание программиста будет привлечено к файлам с исходными текстами программы.

Резюме

Интерфейсы и внутренние классы — весьма утонченные понятия, и во многих других объектно-ориентированных языках вы их не найдете. Например, в C++ нет ничего похожего. Вместе они решают те задачи, которые C++ пытается осилить, используя множественное наследование. Однако множественное наследование C++ создает массу проблем, по сравнению с ним интерфейсы и внутренние классы Java гораздо более доступны.

Хотя рассмотренные конструкции довольно просты, необходимость вовлечения их в проектирование программ сначала не совсем очевидна, так же, как и в случае с полиморфизмом. По прошествии определенного времени вы научитесь сразу оценивать, где большую выгоду даст интерфейс, где внутренний класс, а где нужны обе возможности сразу. Но на данном этапе достаточно ознакомиться хотя бы с их синтаксисом и семантикой. Со временем, когда вы увидите примеры практического применения, вы привыкнете к этим возможностям языка и только будете удивляться, как раньше обходились без них.

11

Коллекции объектов

Ограниченное количество объектов с фиксированным временем жизни встречается разве что в самых простых программах.

В основном ваши программы будут создавать новые объекты, основываясь на таких критериях, которые станут известны лишь во время их работы. До того как программа начнет выполняться, вы обычно не знаете ни количества, ни даже типов нужных вам объектов. А для решения задач программирования в общем вам понадобится создавать неограниченное их число, когда угодно и где угодно. Нельзя рассчитывать, что для каждого из возможных объектов можно будет завести отдельную ссылку:

```
MyObject myReference;
```

Вы ведь не знаете заранее, сколько таких ссылок потребуется.

Большинство языков обеспечивают некоторые пути решения такой чрезвычайно насущной задачи. В Java имеется несколько способов хранения объектов (или, точнее, ссылок на объекты). На уровне компилятора поддерживаются массивы, которые уже рассматривались ранее. Массив обеспечивает самый эффективный способ хранения групп объектов и является первым кандидатом при хранении группы примитивов. Однако массив имеет фиксированный размер, а в общем случае во время написания программы разработчик может не знать точное количество объектов или же ему могут понадобиться более эффективные способы хранения объектов — в таких ситуациях ограничение фиксированного размера создает слишком много проблем.

Библиотека утилит Java (`java.util.*`) также содержит достаточно полный набор *классов контейнеров*, важнейшими из которых являются `List`, `Set`, `Queue` и `Map`. Эти типы объектов также называются *классами коллекций*, но поскольку имя `Collection` используется для обозначения определенного подмножества библиотеки, я буду употреблять общий термин «контейнер». Контейнеры предоставляют весьма изощренные и разнообразные средства для хранения объектов и работы с ними, с помощью которых можно решить множество задач.

Классы контейнеров Java помимо прочих характеристик (например, `Set` не содержит дубликатов, а `Map` представляет собой ассоциативный массив, который позволяет связывать объекты с другими объектами) способны автоматически изменяться в размерах.

Таким образом, в отличие от массивов, в класс можно поместить любое количество объектов, не беспокоясь о размере контейнера во время написания программы.

И хотя классы контейнеров не имеют прямой поддержки на уровне ключевых слов Java¹, это важнейший инструмент, который значительно повысит вашу квалификацию программиста. В этой главе вы получите практические навыки использования библиотеки контейнеров Java, причем особое внимание будет уделяться типичным примерам использования. Мы сосредоточимся на контейнерах, которые вы будете использовать в своей повседневной работе. Позднее, в главе 17, будут рассмотрены другие контейнеры, их функциональность и особенности использования.

Обобщенные типы и классы, безопасные по отношению к типам

До появления контейнеров Java SE5 одна из главных проблем заключалась в том, что компилятор позволял вставить в контейнер объект неправильного типа. Допустим, вы храните набор объектов `Apple` с использованием простейшего контейнера `ArrayList`. Пока считайте, что `ArrayList` — массив, который автоматически расширяется по мере надобности. Использовать `ArrayList` несложно: создайте объект контейнера, вставьте в него объекты методом `add()` и обращайтесь к ним методом `get()` с указанием индекса точно так же, как с массивами, но без квадратных скобок². Класс `ArrayList` также содержит метод `size()`, который сообщает, сколько элементов было добавлено в контейнер, чтобы случайное обращение к элементу за концом массива не привело к ошибке (с выдачей исключения времени выполнения — см. главу 12).

В следующем примере в контейнер помещаются объекты `Apple` и `Orange`, которые затем извлекаются из него. В общем случае компилятор Java выдает предупреждение, потому что в этом примере обобщенные типы не используются. Для подавления предупреждения используется специальная *аннотация* Java SE5. Аннотации начинаются со знака `@` и могут получать аргумент; в данном случае используется аннотация `@SuppressWarnings`, а аргумент указывает, что подавляться должны только «непроверяемые» предупреждения:

```
//: holding/ApplesAndOrangesWithoutGenerics.java
// Простой пример использования контейнера (с предупреждением компилятора).
// {ThrowsException}
import java.util.*;

class Apple {
    private static long counter;
    private final long id = counter++;
    public long id() { return id; }
}
```

¹ В некоторых языках — таких, как Perl, Python и Ruby, — реализована встроенная поддержка контейнеров.

² Здесь безусловно пригодилась бы перегрузка операторов. Классы контейнеров C++ и C# предоставляют более элегантный синтаксис за счет использования перегрузки операторов.

```
class Orange {}

public class ApplesAndOrangesWithoutGenerics {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        ArrayList apples = new ArrayList();
        for(int i = 0; i < 3; i++)
            apples.add(new Apple());
        // Не мешает добавить Orange в apples:
        apples.add(new Orange());
        for(int i = 0; i < apples.size(); i++)
            ((Apple)apples.get(i)).id();
        // Объект Orange обнаруживается только во время выполнения
    }
} /* (Выполните, чтобы увидеть результат) *///:~
```

Аннотации Java SE5 более подробно рассматриваются в главе 20.

Apple и Orange — совершенно разные классы; они не имеют ничего общего, кроме того, что оба являются разновидностью Object (помните: если класс, от которого вы наследуете, не указан явно, то автоматически используется наследование от Object). Так как контейнер ArrayList содержит элементы Object, в него можно добавлять методом add() не только объекты Apple, но и объекты Orange; ни компилятор, ни исполнительная среда вам в этом не помешают. Но когда придет время прочитать то, что вы считаете объектом Apple, с использованием метода get() класса ArrayList, вы получите ссылку на Object, которую необходимо преобразовать в Apple. Все выражение заключается в круглые скобки, чтобы преобразование было выполнено перед вызовом метода id() класса Apple; в противном случае вы получите синтаксическую ошибку.

Во время выполнения попытка преобразовать объект Orange в Apple приводит к ошибке в форме исключения (см. выше).

В главе 15 вы узнаете, что создание классов с использованием механизма обобщенных типов Java может быть достаточно нетривиальным делом. Впрочем, использование готовых обобщенных классов обычно проходит весьма прямолинейно. Например, чтобы определить контейнер ArrayList для хранения объектов Apple, следует использовать запись ArrayList<Apple> вместо простого имени ArrayList. В угловые скобки заключаются параметры-типы (их может быть несколько); они определяют типы, которые могут храниться в данном экземпляре контейнера.

Использование обобщенных типов предотвращает помещение неверного типа объекта в контейнер *на стадии компиляции*¹. Рассмотрим тот же пример с использованием обобщенных типов:

```
//: holding/ApplesAndOrangesWithGenerics.java
import java.util.*;

public class ApplesAndOrangesWithGenerics {
    public static void main(String[] args) {
        ArrayList<Apple> apples = new ArrayList<Apple>();
```

продолжение ➤

¹ В конце главы 15 мы разберемся, насколько серьезна эта проблема. Впрочем, там же будет показано, что полезность обобщенных типов Java не ограничивается контейнерами, безопасными по отношению к типам.

```

    for(int i = 0; i < 3; i++)
        apples.add(new Apple());
    // Ошибка во время компиляции:
    // apples.add(new Orange());
    for(int i = 0; i < apples.size(); i++)
        System.out.println(apples.get(i).id());
    // Использование синтаксиса foreach:
    for(Apple c : apples)
        System.out.println(c.id());
}
} /* Output:
0
1
2
0
1
2
*///:~

```

Теперь компилятор не позволяет поместить в `apples` объект `Orange`; таким образом, разработчик узнает об ошибке не во время выполнения, а во время компиляции.

Также обратите внимание на то, что при выборке данных из `List` преобразование типа становится лишним. Контейнер знает, какой тип в нем хранится, и автоматически выполняет преобразование при вызове `get()`. Таким образом, с обобщенными типами вы не только знаете, что компилятор проверит тип объекта, помещаемого в контейнер, но и можете использовать более компактный синтаксис при выборке объектов из контейнера.

Из приведенного примера также видно, что если вам не требуется использовать индекс каждого элемента, то для последовательной выборки элементов можно использовать синтаксис `foreach`.

При помещении объекта в контейнер вы не ограничены точным типом, указанным в параметре обобщенного типа. Восходящее преобразование работает с обобщенными типами точно так же, как и с любыми другими типами:

```

//: holding/GenericsAndUpcasting.java
import java.util.*;

class GrannySmith extends Apple {}
class Gala extends Apple {}
class Fuji extends Apple {}
class Braeburn extends Apple {}

public class GenericsAndUpcasting {
    public static void main(String[] args) {
        ArrayList<Apple> apples = new ArrayList<Apple>();
        apples.add(new GrannySmith());
        apples.add(new Gala());
        apples.add(new Fuji());
        apples.add(new Braeburn());
        for(Apple c : apples)
            System.out.println(c);
    }
} /* Output: (Sample)

```



```
GrannySmith@7d772e
Gala@11b86e7
Fuji@35ce36
Braeburn@757aef
*///:~
```

Таким образом, в контейнер для объектов Apple можно поместить объект одного из субтипов Apple.

Выходные данные были получены от реализации метода `toString()` класса `Object`, которая выводит имя класса с последующим шестнадцатеричным числом без знака — представлением хеш-кода объекта (сгенерированным методом `hashCode()`). Хеш-коды более подробно рассматриваются в главе 17.

1. (2) Создайте новый класс с именем `Gerbil` с полем `gerbilNumber` типа `int`, инициализируемым в конструкторе. Определите в нем метод `hop()`, который выводит значение `gerbilNumber` и короткое сообщение. Создайте контейнер `ArrayList` и добавьте в него объекты `Gerbil`. Используйте метод `get()` для перебора элементов и вызова `hop()` для каждого объекта `Gerbil`.

Основные концепции

Библиотека контейнеров Java решает вопрос «хранения ваших объектов», рассматривая его как совокупность двух различных концепций, выраженных основными интерфейсами библиотеки:

- ❑ **Коллекция (Collection):** последовательность отдельных элементов, формируемая по некоторым правилам. Интерфейс `List` (список) хранит элементы в определенной последовательности, а в интерфейсе `Set` (множество) нельзя хранить повторяющиеся элементы. Интерфейс `Queue` (очередь) выдает элементы в порядке, определяемом *дисциплиной очереди* (обычно совпадающем с порядком вставки).
- ❑ **Карта (Map):** набор пар объектов «ключ-значение», с возможностью выборки значения по ключу. Контейнер `ArrayList` позволяет получить объект по числу, так что он в каком-то смысле связывает числа с объектами. *Карта* позволяет получить объект *по другому объекту*. Также часто встречается термин *ассоциативный массив* (потому что объекты ассоциируются с другими объектами) и *словарь* (потому что объект-значение ищется по объекту-ключу, по аналогии с поиском определения по слову). Интерфейсы, производные от `Map`, — мощный и полезный инструмент.

В идеале большая часть кода должна взаимодействовать с этими интерфейсами (хотя это не всегда возможно), а точный тип указывается только в точке создания контейнера. Таким образом, команда создания контейнера `List` может выглядеть так:

```
List<Apple> apples = new ArrayList<Apple>();
```

Обратите внимание: тип `ArrayList` преобразуется в `List` посредством восходящего преобразования (в отличие от предыдущих примеров). Если вы вдруг решите изменить свою реализацию, для этого достаточно внести изменение в точку создания:

```
List<Apple> apples = new LinkedList<Apple>();
```

Итак, обычно вы создаете объект конкретного класса, преобразуете его в соответствующий интерфейс восходящим преобразованием и затем используете интерфейс в оставшейся части кода.

Такое решение работает не всегда, потому что некоторые классы обладают дополнительной функциональностью. Например, `LinkedList` содержит методы, отсутствующие в интерфейсе `List`, а `TreeMap` содержит методы, не входящие в интерфейс `Map`. Если вы намерены использовать эти методы, то восходящее преобразование к более общему интерфейсу становится невозможным.

Интерфейс `Collection` обобщает концепцию последовательности — способа хранения группы объектов. Следующий простой пример заполняет объект `Collection` (представленный классом `ArrayList`) объектами `Integer` и выводит каждый элемент в полученном контейнере:

```
//: holding/SimpleCollection.java
import java.util.*;

public class SimpleCollection {
    public static void main(String[] args) {
        Collection<Integer> c = new ArrayList<Integer>();
        for(int i = 0; i < 10; i++)
            c.add(i); // Автоматическая упаковка
        for(Integer i : c)
            System.out.print(i + ", ");
    }
} /* Output:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
*///:~
```

Так как в этом примере используются только методы `Collection`, подойдет объект любого класса, производного от `Collection`, но самым распространенным типом последовательности является `ArrayList`.

Имя `add()` создает впечатление, что метод добавляет новый элемент в коллекцию. Однако в документации используется тщательно выбранная формулировка: `add()` «...убеждается в том, что в `Collection` присутствует заданный элемент». Эта формулировка подразумевает функциональность множества (`Set`), для которого элемент добавляется только в том случае, если он еще не присутствует в контейнере. Для `ArrayList` или любой разновидности `List` вызов `add()` всегда эквивалентен добавлению, потому что объекты `List` не проверяют наличие дубликатов.

Для перебора всех контейнеров `Collection` может применяться синтаксис `foreach`, как в приведенном примере. Позднее в этой главе будет представлена более гибкая концепция *итераторов*.

2. (1) Измените пример `SimpleCollection.java` так, чтобы в качестве контейнера с использовалось множество (`Set`).
3. (2) Измените пример `innerclasses/Sequence.java` так, чтобы в контейнер можно было добавить произвольное количество элементов.

Добавление групп элементов

В классах `Arrays` и `Collections` из библиотеки `java.util` определены вспомогательные методы для добавления групп элементов в `Collection`. Метод `Arrays.asList()` получает массив или список элементов, разделенных запятыми (с использованием списка аргументов переменной длины), который преобразуется в объект `List`. Метод `Collections.addAll()` получает объект `Collection` и либо массив, либо список, разделенный запятыми, и добавляет элементы в `Collection`. Следующий пример демонстрирует оба метода, а также более традиционный метод `addAll()`, присутствующий во всех типах `Collection`:

```
//: holding/AddingGroups.java
// Добавление групп элементов в объекты Collection.
import java.util.*;

public class AddingGroups {
    public static void main(String[] args) {
        Collection<Integer> collection =
            new ArrayList<Integer>(Arrays.asList(1, 2, 3, 4, 5));
        Integer[] moreInts = { 6, 7, 8, 9, 10 };
        collection.addAll(Arrays.asList(moreInts));
        // Работает значительно быстрее, но Collection
        // так сконструировать невозможно:
        Collections.addAll(collection, 11, 12, 13, 14, 15);
        Collections.addAll(collection, moreInts);
        // Создает список "на базе" массива:
        List<Integer> list = Arrays.asList(16, 17, 18, 19, 20);
        list.set(1, 99); // OK -- изменить элемент
        // list.add(21); // Ошибка времени выполнения, потому что
        // размер базового массива изменяться не может.
    }
} ///:~
```

Конструктор `Collection` может получить другой объект `Collection`, который используется для инициализации, так что вы можете использовать `Arrays.asList()` для получения входных данных конструктора. Однако метод `Collections.addAll()` работает намного быстрее, а решение с созданием `Collection` без элементов и последующим вызовом `Collections.addAll()` очень просто реализуется и поэтому считается предпочтительным.

Метод `Collection.addAll()` может получать в аргументе только другой объект `Collection`, поэтому он не обладает такой гибкостью, как методы `Arrays.asList()` или `Collections.addAll()`, использующие списки аргументов переменной длины.

Также вывод `Arrays.asList()` можно использовать непосредственно как `List`, но базовым представлением данных в этом случае является массив, не поддерживающий изменение размеров. Попытка добавления (`add()`) или удаления (`delete()`) элементов из такого списка приводит к попытке изменения размера массива, и вы получите ошибку «Операция не поддерживается» во время выполнения.

Ограничение метода `Arrays.asList()` заключается в том, что он пытается «угадать» итоговый тип `List`, не обращая внимания на присвоенные значения. Иногда это создает проблемы:

```

//: holding/AsListInference.java
// Arrays.asList() пытается угадать тип.
import java.util.*;

class Snow {}
class Powder extends Snow {}
class Light extends Powder {}
class Heavy extends Powder {}
class Crusty extends Snow {}
class Slush extends Snow {}

public class AsListInference {
    public static void main(String[] args) {
        List<Snow> snow1 = Arrays.asList(
            new Crusty(), new Slush(), new Powder());

        // Не откомпилируется:
        // List<Snow> snow2 = Arrays.asList(
        //     new Light(), new Heavy());
        // Компилятор сообщает:
        // получено : java.util.List<Powder>
        // требуется : java.util.List<Snow>
        // У Collections.addAll() проблем нет:
        List<Snow> snow3 = new ArrayList<Snow>();
        Collections.addAll(snow3, new Light(), new Heavy());

        // Подсказка с явно указанным аргументом типа:
        List<Snow> snow4 = Arrays.<Snow>asList(
            new Light(), new Heavy());
    }
} ///:~

```

При попытке создания snow2 метод `Arrays.asList()` обнаруживает только типы `Powder`, поэтому он создает `List<Powder>` вместо `List<Snow>`, тогда как метод `Collections.addAll()` работает нормально, потому что он по первому аргументу определяет правильный целевой тип.

Как видно из создания snow4, в середину последовательности `Arrays.asList()` можно вставить «подсказку», которая сообщит компилятору фактический целевой тип `List`, создаваемый `Arrays.asList()`. Этот прием называется *явным указанием аргумента типа*.

Как вы вскоре убедитесь, с `Map` дело обстоит сложнее. Стандартная библиотека Java не предоставляет никаких средств их автоматической инициализации, кроме как по содержимому другого объекта `Map`.

Вывод контейнеров

Для создания печатного представления массива необходимо использовать `Arrays.toString()`, но контейнеры запросто выводятся без посторонней помощи. Следующий пример также демонстрирует использование основных типов контейнеров:

```

//: holding/PrintingContainers.java
// Контейнеры распечатываются автоматически.
import java.util.*;
import static net.mindview.util.Print.*;

```

```

public class PrintingContainers {
    static Collection fill(Collection<String> collection) {
        collection.add("крыса");
        collection.add("кошка");
        collection.add("собака");
        collection.add("собака");
        return collection;
    }

    static Map fill(Map<String,String> map) {
        map.put("крыса", "Анфиса");
        map.put("кошка", "Мурка");
        map.put("собака", "Шарик");
        map.put("собака", "Бобик");
        return map;
    }

    public static void main(String[] args) {
        print(fill(new ArrayList<String>()));
        print(fill(new LinkedList<String>()));
        print(fill(new HashSet<String>()));
        print(fill(new TreeSet<String>()));
        print(fill(new LinkedHashSet<String>()));
        print(fill(new HashMap<String,String>()));
        print(fill(new TreeMap<String,String>()));
        print(fill(new LinkedHashMap<String,String>()));
    }
} /* Output:
[крыса, кошка, собака, собака]
[крыса, кошка, собака, собака]
[собака, кошка, крыса]
[кошка, собака, крыса]
[крыса, кошка, собака]
{собака=Бобик, кошка=Мурка, крыса=Анфиса}
{кошка=Мурка, собака=Бобик, крыса=Анфиса}
{крыса=Анфиса, кошка=Мурка, собака=Бобик}
*///:~

```

Как уже было упомянуто, в библиотеке контейнеров Java существует две категории. Разница между ними основана на том, сколько в одной «ячейке» контейнера помещается элементов. Коллекции (*Collection*) содержат только один элемент в каждой ячейке (имя немного не соответствует их реальному предназначению, иногда «коллекциями» называют сами библиотеки контейнеров). К коллекциям относятся список (*List*), где в определенной последовательности хранится группа элементов; множество (*Set*), в которое можно добавлять только по одному элементу определенного типа; и очередь (*Queue*), позволяющая вставлять объекты с одного «конца» контейнера и извлекать их с другого «конца» (в контексте данного примера очередь представляет собой лишь другую разновидность последовательности, поэтому она не представлена). У карты (*Map*) в каждой ячейке хранятся два объекта: *ключ* и связанное с ним *значение*.

Если просмотреть результат работы программы, вы увидите, что вывод на печать содержимого контейнеров по умолчанию (который осуществляется в их методах *toString()*) дает достаточно приемлемые результаты и поэтому дополнительной поддержки для печати, как было нужно для массивов, не требуется. Коллекция распечатывается в квадратных скобках, ее элементы разделяются запятыми. Карта выводится

в фигурных скобках, ключ и ассоциированное с ним значение связываются знаком равенства (ключи слева, значения справа).

Первый метод `fill()` работает со всеми разновидностями `Collection`, каждая из которых реализует метод `add()` для включения новых элементов. `ArrayList` и `LinkedList` являются реализациями `List`; как видно из выходных данных, элементы хранятся в них в порядке вставки. Различия между ними не сводятся к скорости выполнения некоторых видов операций; `LinkedList` также поддерживает больше операций, чем `ArrayList`. Эти отличия будут более подробно рассмотрены далее в этой главе.

`HashSet`, `TreeSet` и `LinkedHashSet` относятся к разновидностям `Set`. Из выходных данных видно, что в `Set` не встречаются одинаковые элементы, а разные реализации `Set` по-разному хранят свое содержимое. `HashSet` хранит элементы с применением относительно сложного алгоритма, который будет рассмотрен в главе 17, а пока достаточно знать, что этот класс обеспечивает самую быструю выборку элементов, но порядок следования элементов выглядит бессмысленно (но часто вас интересует лишь то, присутствует ли некоторый объект в `Set`, а не порядок следования элементов). Если порядок хранения для вас важен, используйте контейнер `TreeSet`, который хранит объекты упорядоченными по возрастанию, или класс `LinkedHashSet`, хранящий элементы в порядке их добавления.

Карта (`Map`) позволяет найти объект по ключу (наподобие простой базы данных). Объект, ассоциированный с ключом, называется *значением*. Если у вас есть карта, в которой страны ассоциируются со своими столицами, и понадобится узнать столицу Канады — вы производите нужный поиск, почти так же, как получают элемент массива по его порядковому номеру. Из-за такого поведения карта не может содержать повторные вхождения ключа.

Метод `Map.put(ключ, значение)` добавляет значение (нужная информация) и связывает его с ключом (данные, по которым будет осуществляться поиск). Метод `Map.get(ключ)` возвращает значение, связанное с заданным ключом. В рассмотренном примере в класс только добавляются пары «ключ-значение», а выборка не выполняется. Эта операция будет продемонстрирована позднее.

Вам не нужно указывать размер контейнера `Map` (и даже задумываться о нем), потому что размеры контейнера изменяются автоматически. Кроме того, контейнер `Map` умеет выводить себя с представлением связей между ключами и значениями. Порядок хранения ключей и значений в `Map` отличен от порядка вставки, потому что реализация `HashMap` использует очень быстрый алгоритм выборки, определяющий порядок элементов.

В приведенном примере используются три основные разновидности `Map`: `HashMap`, `TreeMap` и `LinkedHashMap`. `HashMap`, как и `HashSet`, обеспечивает самую быструю выборку, но с непредсказуемым порядком хранения элементов. `TreeMap` хранит ключи отсортированными по возрастанию, а `LinkedHashMap` хранит ключи в порядке вставки без потери высокой скорости выборки `HashMap`.

4. (3) Создайте класс-генератор, который при каждом вызове `next()` выдает имена персонажей вашего любимого фильма в виде объектов `String`. Когда список заканчивается, программа снова возвращается к началу. Используйте генератор для заполнения массива и контейнеров `ArrayList`, `LinkedList`, `HashSet`, `LinkedHashSet` и `TreeSet`, после чего выведите содержимое каждого контейнера.

List

Контейнер `List` гарантирует хранение списка элементов в определенной последовательности. Интерфейс `List` добавляет в `Collection` методы вставки и удаления элементов в середине списка.

Существуют две основные разновидности `List`:

- ❑ Базовый контейнер `ArrayList` с превосходной скоростью произвольного доступа к элементам, но относительно медленными операциями вставки и удаления элементов в середине.
- ❑ Связанный список `LinkedList` с оптимальным последовательным доступом и низкотратными операциями вставки и удаления в середине списка. Операции произвольного доступа `LinkedList` выполняет относительно медленно, но обладает более широкой функциональностью, чем `ArrayList`.

В следующем примере мы немного забегаем вперед, импортируя библиотеку `typeinfo.pets` из главы 14. Эта библиотека содержит иерархию классов `Pet` и средства случайного генерирования объектов `Pet`. Знать все подробности пока не обязательно; достаточно того, что (1) существует класс `Pet` и различные классы, производные от `Pet`, и (2) статический метод `Pets.arrayList()` возвращает объект `ArrayList`, заполненный случайно выбранными объектами `Pet`.

```
//: holding/ListFeatures.java
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ListFeatures {
    public static void main(String[] args) {
        Random rand = new Random(47);
        List<Pet> pets = Pets.arrayList(7);
        print("1: " + pets);
        Hamster h = new Hamster();
        pets.add(h); // С автоматическим изменением размера
        print("2: " + pets);
        print("3: " + pets.contains(h));
        pets.remove(h); // Удаление заданного объекта
        Pet p = pets.get(2);
        print("4: " + p + " " + pets.indexOf(p));
        Pet cymric = new Cymric();
        print("5: " + pets.indexOf(cymric));
        print("6: " + pets.remove(cymric));
        // Удаление заданного объекта:
        print("7: " + pets.remove(p));
        print("8: " + pets);
        pets.add(3, new Mouse()); // Вставка по индексу
        print("9: " + pets);
        List<Pet> sub = pets.subList(1, 4);
        print("Частичный список: " + sub);
        print("10: " + pets.containsAll(sub));
        Collections.sort(sub); // Сортировка "на месте"
        print("После сортировки: " + sub);
        // Для containsAll() порядок не важен:
```

продолжение ➤

```

print("11: " + pets.containsAll(sub));
Collections.shuffle(sub, rand); // Перемешивание
print("После перемешивания: " + sub);
print("12: " + pets.containsAll(sub));
List<Pet> copy = new ArrayList<Pet>(pets);
sub = Arrays.asList(pets.get(1), pets.get(4));
print("sub: " + sub);
copy.retainAll(sub);
print("13: " + copy);
copy = new ArrayList<Pet>(pets); // Копирование
copy.remove(2); // Удаление по индексу
print("14: " + copy);
copy.removeAll(sub); // Удаление заданных объектов
print("15: " + copy);
copy.set(1, new Mouse()); // Замена элемента
print("16: " + copy);
copy.addAll(2, sub); // Вставка списка в середину
print("17: " + copy);
print("18: " + pets.isEmpty());
pets.clear(); // Удаление всех элементов
print("19: " + pets);
print("20: " + pets.isEmpty());
pets.addAll(Pets.arrayList(4));
print("21: " + pets);
Object[] o = pets.toArray();
print("22: " + o[3]);
Pet[] pa = pets.toArray(new Pet[0]);
print("23: " + pa[3].id());
}
} /* Output:
1: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug]
2: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug, Hamster]
3: true
4: Cymric 2
5: -1
6: false
7: true
8: [Rat, Manx, Mutt, Pug, Cymric, Pug]
9: [Rat, Manx, Mutt, Mouse, Pug, Cymric, Pug]
Частичный список: [Manx, Mutt, Mouse]
10: true
После сортировки: [Manx, Mouse, Mutt]
11: true
После перемешивания: [Mouse, Manx, Mutt]
12: true
sub: [Mouse, Pug]
13: [Mouse, Pug]
14: [Rat, Mouse, Mutt, Pug, Cymric, Pug]
15: [Rat, Mutt, Cymric, Pug]
16: [Rat, Mouse, Cymric, Pug]
17: [Rat, Mouse, Mouse, Pug, Cymric, Pug]
18: false
19: []
20: true
21: [Manx, Cymric, Rat, EgyptianMau]
22: EgyptianMau
23: 14
*///:~

```


Строки выходных данных пронумерованы, чтобы вывод можно было связать с исходным кодом. В первой строке приведен исходный контейнер объектов `Pet`. В отличие от массива контейнер `List` позволяет добавлять и удалять элементы после создания, а также изменяет свои размеры. Это очень принципиальный момент: последовательность может изменяться. Результат добавления `Hamster` продемонстрирован в главе 2: объект добавился в конец списка.

Метод `contains()` проверяет, присутствует ли объект в списке. Чтобы удалить объект, передайте ссылку на него методу `remove()`. Кроме того, если у вас имеется ссылка на объект, вы можете определить индекс объекта в `List` при помощи метода `indexOf()` (см. строку 4).

При принятии решения о том, присутствует ли элемент в `List`, определении индекса элемента и удалении элемента из `List` по ссылке используется метод `equals()` (часть корневого класса `Object`). Объекты `Pet` определяются как уникальные, и хотя в списке уже присутствуют два объекта `Cymric`, если я создам новый объект `Cymric` и передам его `indexOf()`, результат будет равен `-1` (объект не найден), а попытка вызвать `remove()` для этого объекта вернет `false`. Для других классов метод `equals()` может быть определен иначе — объекты `String`, например, считаются равными, если два объекта `String` имеют одинаковое содержимое. Таким образом, чтобы избежать неприятных сюрпризов, важно помнить, что поведение `List` изменяется в зависимости от поведения `equals()`.

В строках вывода 7 и 8 удаление объекта, точно совпадающего с объектом в контейнере `List`, выполняется успешно.

Как видно из строки вывода 9 и предшествующего ей кода, элемент можно вставить в середину `List`, но при этом возникает проблема: для `LinkedList` операции вставки и удаления в середине списка выполняются быстро (не считая непосредственного позиционирования на элементе в середине списка), но для `ArrayList` эта операция обходится дорого. Означает ли это, что вам не следует вставлять элементы в середину `ArrayList`, а при необходимости выполнения таких операций переключаться на `LinkedList`? Нет, просто вы должны знать об этой проблеме, и при большом количестве вставок в середину `ArrayList`, замедляющем работу программы, причины можно поискать в реализации `List` (как показано в приложении <http://MindView.net/Books/BetterJava>, для выявления подобных «узких мест» лучше всего воспользоваться профилировщиком). Оптимизация — дело тонкое, и лучше всего отложить ее до того момента, когда вы будете уверены в ее необходимости (хотя иметь представление о потенциальных проблемах всегда полезно).

Метод `subList()` позволяет легко выделить часть большего списка; естественно, при передаче созданного сегмента методу `containsAll()` для большего списка будет получен истинный результат. Также интересно заметить, что порядок следования не важен — из строк 11 и 12 видно, что вызов методов `Collections.sort()` и `Collections.shuffle()` для `sub` не влияет на результат `containsAll()`. Метод `subList()` создает список на базе исходного списка. Таким образом, изменения в возвращенном списке отражаются на исходном списке, и наоборот.

Метод `retainAll()` фактически выполняет операцию *пересечения множеств*; сохраняя все элементы `copy`, также присутствующие в `sub`. И снова итоговое поведение зависит от поведения метода `equals()`.

В строке 14 выводится результат удаления элемента по индексу. Эта операция проще удаления по ссылке на объект, потому что при использовании индекса не нужно задумываться о поведении `equals()`. Метод `removeAll()` также использует метод `equals()`. Как подсказывает имя, он удаляет из `List` все объекты, присутствующие в аргументе (также типа `List`).

Имя метода `set()` выбрано неудачно из-за возможной путаницы с классом `Set` — пожалуй, метод стоило назвать `replace`, потому что он заменяет элемент с заданным индексом (первый аргумент) вторым аргументом.

Строка 17 показывает, что у `List` существует перегруженная версия `addAll()`, которая позволяет вставить новый список в середину исходного списка (вместо простого присоединения в конец методом `addAll()`, унаследованным от `Collection`).

Строки 18–20 демонстрируют эффект вызова методов `isEmpty()` и `clear()`.

Строки 22 и 23 показывают, как произвольный контейнер `Collection` преобразуется в массив вызовом `toArray()`. Это перегруженный метод; версия без аргументов возвращает массив `Object`, но при передаче перегруженной версии массива целевого типа будет создан массив заданного типа (при условии, что проверка типов пройдет успешно). Если массив-аргумент слишком мал для хранения всех объектов `List` (как в нашем случае), метод `toArray()` создает новый массив соответствующего размера. Объекты `Pet` содержат метод `id()`, который, как мы видим, вызывается для одного из объектов полученного массива.

5. (3) Измените пример `ListFeatures.java`, чтобы вместо объектов `Pet` в нем использовались значения `Integer` (не забудьте про автоматическую упаковку!). Объясните различия в результатах.
6. (2) Измените пример `ListFeatures.java`, чтобы вместо объектов `Pet` в нем использовались объекты `String`. Объясните различия в результатах.
7. (3) Создайте класс, затем создайте инициализированный массив объектов этого класса. Заполните контейнер `List` данными массива. Создайте подмножество `List` методом `subList()`, после чего удалите это подмножество из вашего контейнера `List`.

Итераторы

При использовании любого класса контейнера должен быть способ помещать в него что-либо и получать это что-либо обратно. Помимо прочего, именно хранение объектов — задача номер один для контейнера. В контейнере `List` для добавления объектов может использоваться метод `add()`, а для их получения — метод `get()`.

Но если взглянуть на происходящее на более высоком уровне, обнаруживается одна проблема: чтобы начать что-то делать с контейнером, необходимо знать его точный тип. Сначала может показаться, что это не так уж плохо, но что если вы начали использовать в программе контейнер `List`, а затем обнаружили, что в вашем случае гораздо более эффективным будет множество (`Set`)? Или предположим, что вы хотели бы написать код общего вида, который не зависит от типа контейнера, с которым он работает, так, чтобы он был применим к любому контейнеру?

Для реализации этой абстракции можно воспользоваться концепцией *итератора* (iterator) — кстати, это еще один паттерн проектирования. Итератором называется объект, обеспечивающий перемещение по последовательности объектов с выбором каждого объекта этой последовательности; при этом программисту-клиенту не надо знать или заботиться о лежащей в ее основе структуре. Вдобавок, итератор обычно является так называемым «легковесным» (light-weight) объектом: его создание не должно занимать много ресурсов. Из-за этого для итераторов часто устанавливаются ограничения, которые на первый взгляд кажутся странными; например, в Java `Iterator` поддерживает перемещение только в одном направлении. С итератором можно выполнять следующие операции.

1. Запросить у `Collection` итератор посредством метода с именем `iterator()`. Этот итератор готов вернуть начальный элемент последовательности.
2. Получить следующий элемент последовательности вызовом метода `next()`.
3. Проверить, есть ли еще объекты в последовательности (метод `hasNext()`).
4. Удалить из последовательности последний элемент, возвращенный итератором, методом `remove()`.

Чтобы понять, как работают итераторы, мы снова воспользуемся инструментарием `Pet` из главы 14:

```
//: holding/SimpleIteration.java
import typeinfo.pets.*;
import java.util.*;

public class SimpleIteration {
    public static void main(String[] args) {
        List<Pet> pets = Pets.arrayList(12);
        Iterator<Pet> it = pets.iterator();
        while(it.hasNext()) {
            Pet p = it.next();
            System.out.print(p.id() + ":" + p + " ");
        }
        System.out.println();
        // Более простой синтаксис (там, где возможно):
        for(Pet p : pets)
            System.out.print(p.id() + ":" + p + " ");
        System.out.println();
        // Итератор также позволяет удалять элементы:
        it = pets.iterator();
        for(int i = 0; i < 6; i++) {
            it.next();
            it.remove();
        }
        System.out.println(pets);
    }
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
8:Cymric 9:Rat 10:EgyptianMau 11:Hamster
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
8:Cymric 9:Rat 10:EgyptianMau 11:Hamster
[Pug, Manx, Cymric, Rat, EgyptianMau, Hamster]
*///:~
```

С итератором можно не беспокоиться о том, сколько элементов содержится в контейнере. Об этом позаботятся методы `hasNext()` и `next()`.

Если вы просто перемещаетесь вперед по контейнеру `List`, не пытаясь изменять сам объект `List`, синтаксис `foreach` получается более компактным.

Итератор также может удалить последний элемент, полученный при вызове `next()`; это означает, что вызову `remove()` должен предшествовать вызов `next()`¹.

Передача контейнера объектов для выполнения некоторой операции с каждым объектом — полезная идея, с которой мы еще встретимся на страницах книги.

В качестве другого примера рассмотрим создание метода вывода содержимого, не зависящего от контейнера:

```
//: holding/CrossContainerIteration.java
import typeinfo.pets.*;
import java.util.*;

public class CrossContainerIteration {
    public static void display(Iterator<Pet> it) {
        while(it.hasNext()) {
            Pet p = it.next();
            System.out.print(p.id() + ":" + p + " ");
        }
        System.out.println();
    }
    public static void main(String[] args) {
        ArrayList<Pet> pets = Pets.arrayList(8);
        LinkedList<Pet> petsLL = new LinkedList<Pet>(pets);
        HashSet<Pet> petsHS = new HashSet<Pet>(pets);
        TreeSet<Pet> petsTS = new TreeSet<Pet>(pets);
        display(pets.iterator());
        display(petsLL.iterator());
        display(petsHS.iterator());
        display(petsTS.iterator());
    }
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat
5:Cymric 2:Cymric 7:Manx 1:Manx 3:Mutt 6:Pug 4:Pug 0:Rat
*///:~
```

Обратите внимание: метод `display()` не содержит информации о типе последовательности. В этом проявляется мощь итераторов: операция перебора элементов последовательности отделяется от базовой структуры этой последовательности. По этой причине иногда говорят, что итераторы *унифицируют доступ к контейнерам*.

8. (1) Измените упражнение 1 так, чтобы для перемещения по контейнеру `List` при вызовах `hop()` использовался итератор `Iterator`.

¹ Метод `remove()` принадлежит к числу так называемых «необязательных» методов (есть и другие); это означает, что он не обязан быть реализованным всеми реализациями `Iterator`. Эта тема рассматривается в главе 17. Впрочем, контейнеры стандартной библиотеки Java реализуют `remove()`, поэтому пока вам не нужно беспокоиться о нем.

9. (4) Измените пример `innerclasses/Sequence.java` так, чтобы контейнер `Sequence` работал с `Iterator` вместо `Selector`.
10. (2) Измените упражнение 9 из главы 8, чтобы объекты `Rodent` хранились в контейнере `ArrayList`, а для перебора последовательности `Rodent` использовался итератор `Iterator`.
11. (2) Напишите метод, который использует `Iterator` для перебора `Collection` и выводит результат вызова `toString()` для каждого объекта в контейнере. Заполните объектами разные типы `Collection` и примените свой метод к каждому контейнеру.

ListIterator

Интерфейс `ListIterator` представляет собой более мощную разновидность `Iterator`, которая работает только с классами `List`. Хотя `Iterator` может перемещаться только вперед, итератор `ListIterator` является двусторонним. Он может выдать индексы следующего и предыдущего элементов относительно текущей позиции итератора в списке, а также заменить последний посещенный элемент методом `set()`. Метод `listIterator()` возвращает объект `ListIterator`, указывающий в начало `List`, а вызов `listIterator(n)` создает объект `ListIterator`, изначально установленный в позицию списка с индексом `n`. Следующий пример демонстрирует эти возможности:

```
//: holding/ListIteration.java
import typeinfo.pets.*;
import java.util.*;

public class ListIteration {
    public static void main(String[] args) {
        List<Pet> pets = Pets.arrayList(8);
        ListIterator<Pet> it = pets.listIterator();
        while(it.hasNext())
            System.out.print(it.next() + ", " + it.nextIndex() +
                ", " + it.previousIndex() + "; ");
        System.out.println();
        // В обратном направлении:
        while(it.hasPrevious())
            System.out.print(it.previous().id() + " ");
        System.out.println();
        System.out.println(pets);
        it = pets.listIterator(3);
        while(it.hasNext()) {
            it.next();
            it.set(Pets.randomPet());
        }
        System.out.println(pets);
    }
} /* Output:
Rat, 1, 0; Manx, 2, 1; Cymric, 3, 2; Mutt, 4, 3; Pug, 5, 4;
Cymric, 6, 5; Pug, 7, 6; Manx, 8, 7;
7 6 5 4 3 2 1 0
[Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug, Manx]
[Rat, Manx, Cymric, Cymric, Rat, EgyptianMau, Hamster,
EgyptianMau]
*///:~
```

Метод `Pets.randomPet()` используется для замены всех объектов `Pet` в контейнере `List` начиная с позиции 3.

12. (3) Создайте и заполните контейнер `List<Integer>`. Создайте второй контейнер `List<Integer>` того же размера. Используйте итераторы `ListIterator` для чтения элементов из первого контейнера `List` и вставки их во второй контейнер в обратном порядке. (Проанализируйте несколько разных способов решения этой задачи.)

LinkedList

Класс `LinkedList`, как и `ArrayList`, реализует базовый интерфейс `List`, но при этом выполняет некоторые операции (вставка и удаление в середине списка) более эффективно, чем `ArrayList`. И наоборот, с операциями произвольного доступа он работает менее эффективно.

В `LinkedList` также добавляются методы, которые позволяют использовать его как стек, очередь или двустороннюю очередь (дек).

Некоторые из этих методов представляют собой синонимы или небольшие видоизменения для создания имен, более знакомых в контексте конкретного применения (прежде всего `Queue`). Например, методы `getFirst()` и `element()` идентичны — они возвращают начало (первый элемент) списка без его удаления и выдают исключение `NoSuchElementException`, если список пуст. Метод `peek()` — вариация на тему этих двух методов, которая возвращает `null` для пустого списка.

Метод `addFirst()` вставляет элемент в начало списка.

Метод `offer()` делает то же, что `add()` и `addLast()`. Все эти методы добавляют элемент в конец списка.

Метод `removeLast()` удаляет и возвращает последний элемент списка.

Следующий пример демонстрирует основные сходства и различия между этими операциями. Он не повторяет поведение, представленное в примере `ListFeatures.java`:

```
//: holding/LinkedListFeatures.java
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class LinkedListFeatures {
    public static void main(String[] args) {
        LinkedList<Pet> pets =
            new LinkedList<Pet>(Pets.arrayList(5));
        print(pets);
        // Идентично:
        print("pets.getFirst(): " + pets.getFirst());
        print("pets.element(): " + pets.element());
        // Отличается только поведение для пустого списка:
        print("pets.peek(): " + pets.peek());
        // Идентично; удаление и возвращение первого элемента:
        print("pets.remove(): " + pets.remove());
        print("pets.removeFirst(): " + pets.removeFirst());
    }
}
```

```
// Отличается только поведение для пустого списка:
print("pets.poll(): " + pets.poll());
print(pets);
pets.addFirst(new Rat());
print("После addFirst(): " + pets);
pets.offer(Pets.randomPet());
print("После offer(): " + pets);
pets.add(Pets.randomPet());
print("После add(): " + pets);
pets.addLast(new Hamster());
print("После addLast(): " + pets);
print("pets.removeLast(): " + pets.removeLast());
}
} /* Output:
[Rat, Manx, Cymric, Mutt, Pug]
pets.getFirst(): Rat
pets.element(): Rat
pets.peek(): Rat
pets.remove(): Rat
pets.removeFirst(): Manx
pets.poll(): Cymric
[Mutt, Pug]
После addFirst(): [Rat, Mutt, Pug]
После offer(): [Rat, Mutt, Pug, Cymric]
После add(): [Rat, Mutt, Pug, Cymric, Pug]
После addLast(): [Rat, Mutt, Pug, Cymric, Pug, Hamster]
pets.removeLast(): Hamster
*///:~
```

Результат вызова `Pets.arrayList()` передается конструктору `LinkedList` для заполнения объекта. Взглянув на интерфейс `Queue`, вы найдете в нем методы `element()`, `offer()`, `peek()`, `poll()` и `remove()`, добавленные в класс `LinkedList` для того, чтобы он мог быть реализацией `Queue`. Полные примеры работы с очередью будут приведены далее в этой главе.

13. (3) В примере `innerclasses/GreenhouseController.java` класс `Controller` использует `ArrayList`. Измените код так, чтобы в нем использовался класс `LinkedList`, и организуйте перебор множества событий с использованием `Iterator`.
14. (3) Создайте пустой контейнер `LinkedList<Integer>`. Используя итератор `ListIterator`, добавьте в `List` значения `Integer`; все операции вставки должны осуществляться в середине списка.

Стек

Стек часто называют контейнером, построенным на принципе «первый вошел, последний вышел» (LIFO). То есть что бы вы ни поместили (`push`) в стек в последнюю очередь, это будет первым, что вы получите при «выталкивании» (`pop`) элемента из стека. Стек нередко сравнивают со стопкой тарелок — тарелка, положенная в стопку последней, будет первой снята с нее.

В классе `LinkedList` имеются методы, напрямую реализующие функциональность стека, поэтому вы просто используете `LinkedList`, не создавая для стека новый класс. Впрочем, иногда отдельный класс для контейнера-стека решит задачу лучше:

```

//: net/mindview/util/Stack.java
// Создание стека на основе LinkedList.
package net.mindview.util;
import java.util.LinkedList;

public class Stack<T> {
    private LinkedList<T> storage = new LinkedList<T>();
    public void push(T v) { storage.addFirst(v); }
    public T peek() { return storage.getFirst(); }
    public T pop() { return storage.removeFirst(); }
    public boolean empty() { return storage.isEmpty(); }
    public String toString() { return storage.toString(); }
} ///:~

```

В этом примере представлен простейший случай определения класса с применением обобщения. `<T>` после имени класса сообщает компилятору, что тип является *параметризованным*, а параметр-тип (тот, который будет заменен реальным типом при использовании класса) называется `T`. По сути объявление `Stack<T>` означает: «Мы определяем стек для хранения объектов типа `T`». Стек реализуется с использованием `LinkedList`, а контейнер `LinkedList` также узнает, что в нем будут храниться объекты типа `T`. Обратите внимание: метод `push()` получает объект типа `T`, а `peek()` и `pop()` возвращают объект типа `T`. Метод `peek()` предоставляет верхний элемент без извлечения его из стека, а метод `pop()` удаляет и возвращает верхний элемент.

Если вам нужен только стек, не стоит привлекать наследование, так как при его применении получится класс, в который перейдут все методы списка `LinkedList` (в главе 17 вы увидите, что именно такую ошибку допустили разработчики `java.util.Stack` из стандартной библиотеки Java версии 1.0).

Простой пример использования класса `Stack`:

```

//: holding/StackTest.java
import net.mindview.util.*;

public class StackTest {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<String>();
        for(String s : "My dog has fleas".split(" "))
            stack.push(s);
        while(!stack.empty())
            System.out.print(stack.pop() + " ");
    }
} /* Output:
fleas has dog My
*///:~

```

Если вы захотите использовать класс `Stack` в своем коде, то при создании объекта вам придется полностью указать пакет или изменить имя класса; в противном случае, скорее всего, возникнет конфликт имен с классом `Stack` из пакета `java.util`. Например, если импортировать `java.util.*` в предыдущем примере, придется использовать имена пакетов для предотвращения конфликта:

```

//: holding/StackCollision.java
import net.mindview.util.*;

public class StackCollision {
    public static void main(String[] args) {

```



```

net.mindview.util.Stack<String> stack =
    new net.mindview.util.Stack<String>();
for(String s : "My dog has fleas".split(" "))
    stack.push(s);
while(!stack.empty())
    System.out.print(stack.pop() + " ");
System.out.println();
java.util.Stack<String> stack2 =
    new java.util.Stack<String>();
for(String s : "My dog has fleas".split(" "))
    stack2.push(s);
while(!stack2.empty())
    System.out.print(stack2.pop() + " ");
}
} /* Output:
fleas has dog My
fleas has dog My
*///:~

```

Два класса `Stack` обладают одинаковым интерфейсом, но в `java.util` нет общего интерфейса `Stack` — вероятно потому, что исходный, плохо спроектированный класс `java.util.Stack` из Java 1.0 «захватил» это имя. И хотя `java.util.Stack` существует, `LinkedList` предоставляет более качественную реализацию стека, поэтому решение `net.mindview.util.Stack` является предпочтительным.

Для выбора «предпочтительной» реализации `Stack` также можно воспользоваться явным импортированием:

```
import net.mindview.util.Stack;
```

Теперь при любой ссылке на `Stack` будет выбираться версия `net.mindview.util`, а для выбора `java.util.Stack` необходимо использовать полностью уточненное имя.

- 15.** (4) Стеки часто используются для вычисления выражений в языках программирования. Используя реализацию `net.mindview.util.Stack`, вычислите результат следующего выражения, в котором «+» означает «занести следующую букву в стек», а «-» — «извлечь верхний элемент стека и вывести его».

```
+U+n+c---+e+r+t---+a--+i--+n+t+y---+ -+r+u---+l+e+s---
```

Множество

Контейнер `Set` сохраняет не более одного экземпляра каждого объекта-значения. При попытке добавить более одного экземпляра эквивалентного объекта `Set` предотвращает появление дубликата. Чаще всего `Set` используется для тестирования принадлежности, чтобы пользователь мог легко узнать, присутствует ли объект в множестве. По этой причине поиск по ключу обычно является самой важной операцией для `Set`, и разработчик чаще всего выбирает реализацию `HashSet`, оптимизированную по скорости поиска по ключу.

`Set` обладает таким же интерфейсом, как `Collection`, поэтому в `Set` нет дополнительной функциональности, присутствующей в двух разновидностях `List`. Вместо этого `Set` представляет собой разновидность `Collection` — просто обладает другим поведением. (Идеальная ситуация для применения наследования и полиморфизма: выражение

другого поведения.) Set проверяет присутствие искомого объекта по «значению» объекта — более сложная тема, которая будет подробно рассмотрена в главе 17.

В следующем примере реализация HashSet используется с объектами Integer:

```
//: holding/SetOfInteger.java
import java.util.*;

public class SetOfInteger {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Set<Integer> intset = new HashSet<Integer>();
        for(int i = 0; i < 10000; i++)
            intset.add(rand.nextInt(30));
        System.out.println(intset);
    }
} /* Output:
[15, 8, 23, 16, 7, 22, 9, 21, 6, 1, 29, 14, 24, 4, 19, 26,
11, 18, 3, 12, 27, 17, 2, 13, 28, 20, 25, 10, 5, 0]
*///:~
```

В Set добавляются десять тысяч случайных чисел от 0 до 29; понятно, что в каждое значение добавляется огромное количество дубликатов. Однако мы видим, что в итоговом множестве каждое значение присутствует только в одном экземпляре.

Также обратите внимание на то, что в выходных данных не прослеживается никакого видимого порядка. Это объясняется тем, что HashSet для скорости использует хеширование (см. главу 17). Порядок, поддерживаемый HashSet, отличается от порядка TreeSet или LinkedHashSet, так как каждая реализация использует свой механизм хранения элементов. TreeSet хранит элементы отсортированными в специальной структуре данных — *красно-черном дереве*, тогда как HashSet применяет хеширование. LinkedHashSet также использует хеширование для повышения скорости поиска по ключу, но *выглядит* все так, словно элементы сохраняются в связанном списке в порядке вставки.

Если вы хотите, чтобы результаты были отсортированы, используйте TreeSet вместо HashSet:

```
//: holding/SortedSetOfInteger.java
import java.util.*;

public class SortedSetOfInteger {
    public static void main(String[] args) {
        Random rand = new Random(47);
        SortedSet<Integer> intset = new TreeSet<Integer>();
        for(int i = 0; i < 10000; i++)
            intset.add(rand.nextInt(30));
        System.out.println(intset);
    }
} /* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
*///:~
```

Одной из самых частых операций, выполняемых с множествами, является проверка присутствия значений методом contains(). Впрочем, есть и другие операции, при виде которых вспоминаются диаграммы Венна, знакомые по школьному курсу математики:

```

//: holding/SetOperations.java
import java.util.*;
import static net.mindview.util.Print.*;

public class SetOperations {
    public static void main(String[] args) {
        Set<String> set1 = new HashSet<String>();
        Collections.addAll(set1,
            "A B C D E F G H I J K L".split(" "));
        set1.add("M");
        print("H: " + set1.contains("H"));
        print("N: " + set1.contains("N"));
        Set<String> set2 = new HashSet<String>();
        Collections.addAll(set2, "H I J K L".split(" "));
        print("set2 in set1: " + set1.containsAll(set2));
        set1.remove("H");
        print("set1: " + set1);
        print("set2 in set1: " + set1.containsAll(set2));
        set1.removeAll(set2);
        print("set2 removed from set1: " + set1);
        Collections.addAll(set1, "X Y Z".split(" "));
        print("'X Y Z' added to set1: " + set1);
    }
} /* Output:
H: true
N: false
set2 in set1: true
set1: [D, K, C, B, L, G, I, M, A, F, J, E]
set2 in set1: false
set2 removed from set1: [D, C, B, G, M, A, F, E]
'X Y Z' added to set1: [Z, D, C, B, G, M, A, F, Y, X, E]
*///:~

```

Кроме представленных методов, есть и другие; за дополнительной информацией обращайтесь к документации JDK.

Возможность получения списка уникальных элементов может быть достаточно полезной. Допустим, вы хотите получить перечень всех слов в файле `SetOperations.java` (см. выше). При помощи инструментария `net.mindview.TextFile`, который будет представлен позднее, можно открыть и прочитать файл в контейнер `Set`:

```

//: holding/UniqueWords.java
import java.util.*;
import net.mindview.util.*;

public class UniqueWords {
    public static void main(String[] args) {
        Set<String> words = new TreeSet<String>(
            new TextFile("SetOperations.java", "\\W+"));
        System.out.println(words);
    }
} /* Output:
[A, B, C, Collections, D, E, F, G, H, HashSet, I, J, K, L,
M, N, Output, Print, Set, SetOperations, String, X, Y, Z,
add, addAll, added, args, class, contains, containsAll,
false, from, holding, import, in, java, main, mindview,
net, new, print, public, remove, removeAll, removed, set1,
set2, split, static, to, true, util, void]
*///:~

```

Класс `TextFile` является производным от `List<String>`. Конструктор `TextFile` открывает файл и разбивает его на слова при помощи регулярного выражения `\\W+`, которое означает «одна и более букв» (регулярные выражения будут представлены в главе 13). Результат передается конструктору `TreeSet`, который добавляет содержимое `List` в свои данные. Так как мы используем класс `TreeSet`, результат отсортирован. В данном случае применяется лексикографическая сортировка, так что буквы верхнего и нижнего регистра попадают в разные группы. Если вы предпочитаете алфавитную сортировку, передайте конструктору `TreeSet` компаратор `String.CASE_INSENSITIVE_ORDER` (компаратор — объект, определяющий порядок):

```
//: holding/UniqueWordsAlphabetic.java
// Получение алфавитного списка слов.
import java.util.*;
import net.mindview.util.*;

public class UniqueWordsAlphabetic {
    public static void main(String[] args) {
        Set<String> words =
            new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
        words.addAll(
            new TextFile("SetOperations.java", "\\W+"));
        System.out.println(words);
    }
} /* Output:
[A, add, addAll, added, args, B, C, class, Collections,
contains, containsAll, D, E, F, false, from, G, H, HashSet,
holding, I, import, in, J, java, K, L, M, main, mindview,
N, net, new, Output, Print, public, remove, removeAll,
removed, Set, set1, set2, SetOperations, split, static,
String, to, true, util, void, X, Y, Z]
*///:~
```

Интерфейс `Comparator` более подробно рассматривается в главе 16.

- 16. (5).** Создайте контейнер `Set` со всеми гласными буквами. Взяв за основу пример `UniqueWords.java`, подсчитайте и выведите количество гласных в каждом входном слове, а также выведите общее количество гласных во входном файле.

Map

Возможность отображения объектов на другие объекты может оказаться чрезвычайно полезной при решении задач программирования. Для примера рассмотрим программу, которая проверяет «случайность» чисел, вырабатываемых классом для производства случайных чисел `Random`. В идеале этот метод должен равномерно распределять сгенерированные числа, но для проверки необходимо создать набор случайных чисел и подсчитать количество чисел в разных диапазонах. Контейнер `Map` легко решает эту задачу; в качестве ключа используется сгенерированное число, а в качестве значения — количество сгенерированных экземпляров этого числа):

```
//: holding/Statistics.java
// Простой пример использования HashMap.
import java.util.*;
```

```

public class Statistics {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Map<Integer,Integer> m =
            new HashMap<Integer,Integer>();
        for(int i = 0; i < 10000; i++) {
            // Produce a number between 0 and 20:
            int r = rand.nextInt(20);
            Integer freq = m.get(r);
            m.put(r, freq == null ? 1 : freq + 1);
        }
        System.out.println(m);
    }
} /* Output:
{15=497, 4=481, 19=464, 8=468, 11=531, 16=533, 18=478,
3=508, 7=471, 12=521, 17=509, 2=489, 13=506, 9=549, 6=519,
1=502, 14=477, 10=513, 5=503, 0=481}
*///:~

```

В методе `main()` автоматическая упаковка преобразует каждое сгенерированное число в класс-«обертку» `Integer`, чтобы его можно было использовать в контейнере `HashMap`. (Ведь в контейнере нельзя хранить примитивы, только ссылки на объекты.) Метод `get()` возвращает `null`, если ключ отсутствует в контейнере (то есть число было сгенерировано впервые). В противном случае метод `get()` возвращает для этого ключа ассоциированное значение `Integer`, которое увеличивается на 1 (и снова автоматическая упаковка упрощает выражение, но в действительности к `Integer` и обратно).

Следующий пример позволяет использовать описание в формате `String` для поиска объектов `Pet`. Он также показывает, как проверить присутствие ключа или значения в контейнере `Map` методами `containsKey()` и `containsValue()`:

```

//: holding/PetMap.java
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class PetMap {
    public static void main(String[] args) {
        Map<String,Pet> petMap = new HashMap<String,Pet>();
        petMap.put("My Cat", new Cat("Molly"));
        petMap.put("My Dog", new Dog("Ginger"));
        petMap.put("My Hamster", new Hamster("Bosco"));
        print(petMap);
        Pet dog = petMap.get("My Dog");
        print(dog);
        print(petMap.containsKey("My Dog"));
        print(petMap.containsValue(dog));
    }
} /* Output:
{My Cat=Cat Molly, My Hamster=Hamster Bosco, My Dog=Dog
Ginger}
Dog Ginger
true
true
*///:~

```

Контейнеры Map, как и массивы и Collection, легко расширяются до нескольких измерений; достаточно создать контейнер Map, значениями которого являются контейнеры Map (значениями которых могут быть другие контейнеры, в том числе и другие контейнеры Map). Как видите, контейнеры легко объединяются для быстрого получения нетривиальных структур данных. Предположим, вы хотите хранить информацию о людях, у каждого из которых может быть несколько домашних животных, — для этого достаточно создать контейнер Map<Person, List<Pet>>:

```
//: holding/MapOfList.java
package holding;

import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class MapOfList {
    public static Map<Person, List<? extends Pet>>
        petPeople = new HashMap<Person, List<? extends Pet>>();
    static {
        petPeople.put(new Person("Dawn"),
            Arrays.asList(new Cymric("Molly"), new Mutt("Spot")));
        petPeople.put(new Person("Kate"),
            Arrays.asList(new Cat("Shackleton"),
                new Cat("Elsie May"), new Dog("Margrett")));
        petPeople.put(new Person("Marilyn"),
            Arrays.asList(
                new Pug("Louie aka Louis Snorkelstein Dupree"),
                new Cat("Stanford aka Stinky el Negro"),
                new Cat("Pinkola")));
        petPeople.put(new Person("Luke"),
            Arrays.asList(new Rat("Fuzzy"), new Rat("Fizzy")));
        petPeople.put(new Person("Isaac"),
            Arrays.asList(new Rat("Freckly")));
    }
    public static void main(String[] args) {
        print("People: " + petPeople.keySet());
        print("Pets: " + petPeople.values());
        for(Person person : petPeople.keySet()) {
            print(person + " has:");
            for(Pet pet : petPeople.get(person))
                print("    " + pet);
        }
    }
}

/* Output:
People: [Person Luke, Person Marilyn, Person Isaac, Person
Dawn, Person Kate]
Pets: [[Rat Fuzzy, Rat Fizzy], [Pug Louie aka Louis
Snorkelstein Dupree, Cat Stanford aka Stinky el Negro, Cat
Pinkola], [Rat Freckly], [Cymric Molly, Mutt Spot], [Cat
Shackleton, Cat Elsie May, Dog Margrett]]
Person Luke has:
    Rat Fuzzy
    Rat Fizzy
Person Marilyn has:
    Pug Louie aka Louis Snorkelstein Dupree
    Cat Stanford aka Stinky el Negro
    Cat Pinkola
```

```

Person Isaac has:
    Rat Freckly
Person Dawn has:
    Cymric Molly
    Mutt Spot
Person Kate has:
    Cat Shackleton
    Cat Elsie May
    Dog Margrett
*///:~

```

Контейнер Map может вернуть множество (Set) своих ключей, коллекцию (Collection) своих значений или множество (Set) их пар. Метод `keySet()` возвращает контейнер Set, содержащий все ключи из `petPeople`, который используется в команде `foreach` для перебора элементов Map.

17. (2) Возьмите класс `Gerbil` из упражнения 1 и поместите его в контейнер Map. Используйте объект `String`, содержащий имя каждого объекта `Gerbil`, в качестве ключа для связывания с объектом `Gerbil` (значение), помещаемым в таблицу. Получите `Iterator` для `keySet()` и используйте его для перемещения по Map, с выборкой объекта `Gerbil` для каждого ключа, выводом ключа и вызовом метода `hop()`.
18. (3) Заполните контейнер `HashMap` парами «ключ-значение». Выведите результаты, чтобы продемонстрировать упорядочение по хеш-коду. Извлеките пары, отсортируйте по ключу и поместите результат в `LinkedHashMap`. Покажите, что элементы хранятся в порядке вставки.
19. (2) Повторите предыдущее упражнение с `HashSet` и `LinkedHashSet`.
20. (3) Измените упражнение 16 так, чтобы в контейнере хранилось количество вхождений каждой гласной.
21. (3) Используя контейнер `Map<String,Integer>`, создайте по образцу `UniqueWords.java` программу для подсчета вхождений слов в файле. Отсортируйте результаты методом `Collections.sort()` со вторым аргументом `String.CASE_INSENSITIVE_ORDER` (для получения алфавитной сортировки) и выведите результат.
22. (5) Измените предыдущее упражнение так, чтобы для хранения слов в нем использовался класс с объектом `String` и полем счетчика. Для хранения списка слов должен использоваться контейнер Set, содержащий такие объекты.
23. (4) Взяв за отправную точку программу `Statistics.java`, создайте программу, которая циклически повторяет этот тест, проверяя, не появляется ли какое-либо из полученных случайных чисел чаще других.
24. (2) Заполните карту `LinkedHashMap` строковыми ключами и такими же значениями, взятыми по вашему усмотрению. После этого извлеките пары, отсортируйте их по ключам и заново вставьте в карту.
25. (3) Создайте контейнер `Map<String,ArrayList<Integer>>`. Используя `net.mindview.TextFile`, откройте текстовый файл и прочитайте его по словам (передайте `"\\w+"` во втором аргументе конструктора `TextFile`). Подсчитывайте слова в процессе чтения; для каждого слова в файле сохраните в `ArrayList<Integer>` счетчик слов, связанный с этим словом (то есть фактически позицию файла, в которой было обнаружено данное слово).

26. (4) Возьмите контейнер `Map` из предыдущего упражнения и воссоздайте порядок слов в исходном файле.

Очередь

Очередь (`queue`) — это контейнер, работающий по принципу «первый вошел, первый вышел» (FIFO). Иначе говоря, объекты помещаются в один «конец» очереди, а извлекаются с другого «конца». Таким образом, порядок занесения объектов в контейнер будет совпадать с порядком их извлечения оттуда. Соответственно, очереди часто используются для надежного перемещения объектов из одной области программы в другую. Очереди также играют важную роль в параллельном программировании, как будет показано в главе 21, потому что они обеспечивают безопасную передачу объектов между задачами.

Класс `LinkedList` содержит несколько методов для поддержки поведения очередей и реализует интерфейс `Queue`; соответственно, `LinkedList` может использоваться как реализация `Queue`. Восходящее преобразование `LinkedList` в `Queue` позволяет использовать в этом примере методы, специфические для `Queue`:

```
//: holding/QueueDemo.java
// Восходящее преобразование Queue в LinkedList.
import java.util.*;

public class QueueDemo {
    public static void printQ(Queue queue) {
        while(queue.peek() != null)
            System.out.print(queue.remove() + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<Integer>();
        Random rand = new Random(47);
        for(int i = 0; i < 10; i++)
            queue.offer(rand.nextInt(i + 10));
        printQ(queue);
        Queue<Character> qc = new LinkedList<Character>();
        for(char c : "Brontosaurus".toCharArray())
            qc.offer(c);
        printQ(qc);
    }
} /* Output:
8 1 1 1 5 14 3 1 0 1
B r o n t o s a u r u s
*///:~
```

Метод `offer()` относится к числу методов, специфических для `Queue`; он вставляет элемент в конец очереди или возвращает `false`. Методы `peek()` и `element()` возвращают элемент в начале очереди *без его извлечения*, но `peek()` возвращает `null` для пустой очереди, а `element()` выдает исключение `NoSuchElementException`. Методы `poll()` и `remove()` извлекают и возвращают элемент в начале очереди, но `poll()` возвращает `null` для пустой очереди, а `remove()` выдает `NoSuchElementException`.

Автоматическая упаковка преобразует значение `int`, полученное в результате вызова `nextInt()`, в объект `Integer` для `queue`, а `char c` — в объект `Character` для `qc`. Интерфейс

Queue сужает доступ к методам `LinkedList`, чтобы у разработчика не было соблазна использовать методы `LinkedList` (в данном примере `queue` можно преобразовать к `LinkedList`, но по крайней мере разработчик будет понимать, что это нежелательно).

Обратите внимание: методы, специфические для `Queue`, предоставляют полную и законченную функциональность. Иначе говоря, вы получаете работоспособную очередь без использования методов интерфейса `Collection`, от которого наследует `Queue`.

- 27.** (2) Напишите класс с именем `Command`, который содержит поле `String` и метод `operation()`, выводящему `String`. Напишите второй класс с методом, который заполняет контейнер `Queue` объектами `Command` и выводит его. Передайте заполненный контейнер `Queue` методу третьего класса, перебирающему объекты `Queue` и вызывающему их методы `operation()`.

PriorityQueue

Термин FIFO описывает самую типичную дисциплину очереди — правило, которое для группы элементов очереди определяет, какой элемент будет извлечен следующим. Согласно принципу FIFO, следующим должен извлекаться элемент, который дольше всех ожидает в очереди.

В приоритетной очереди `PriorityQueue` следующим извлекается элемент, обладающий наивысшим *приоритетом*. Например, в аэропорту клиент может быть обслужен вне очереди, если его самолет готовится к вылету. В системе передачи сообщений некоторые сообщения могут содержать более важную информацию; они должны быть срочно обработаны независимо от времени поступления. Контейнеры `PriorityQueue` были добавлены в Java SE5 для автоматической реализации такого поведения.

При вызове `offer()` для объекта, помещаемого в `PriorityQueue`, позиция этого объекта в очереди определяется сортировкой¹. Сортировка по умолчанию использует естественный порядок следования объектов в очереди, но вы можете изменить его, предоставив собственную реализацию `Comparator`. `PriorityQueue` гарантирует, что при вызове `peek()`, `poll()` или `remove()` будет получен элемент с наивысшим приоритетом.

Создать приоритетную очередь для работы со встроенными типами (`Integer`, `String`, `Character` и т. д.) совсем несложно. В следующем примере первая группа значений идентична случайным значениям из предыдущего примера; как видно из приведенного результата, они извлекаются из `PriorityQueue` в другом порядке:

```
//: holding/PriorityQueueDemo.java
import java.util.*;

public class PriorityQueueDemo {
    public static void main(String[] args) {
```

продолжение ➤

¹ Вообще говоря, это зависит от реализации. Алгоритмы приоритетных очередей обычно выполняют сортировку при вставке, но также возможна выборка наиболее приоритетного элемента при извлечении. Выбор алгоритма может быть важен, если приоритеты объектов могут изменяться во время нахождения в очереди.

```

PriorityQueue<Integer> priorityQueue =
    new PriorityQueue<Integer>();
Random rand = new Random(47);
for(int i = 0; i < 10; i++)
    priorityQueue.offer(rand.nextInt(i + 10));
QueueDemo.printQ(priorityQueue);

List<Integer> ints = Arrays.asList(25, 22, 20,
    18, 14, 9, 3, 1, 1, 2, 3, 9, 14, 18, 21, 23, 25);
priorityQueue = new PriorityQueue<Integer>(ints);
QueueDemo.printQ(priorityQueue);
priorityQueue = new PriorityQueue<Integer>(
    ints.size(), Collections.reverseOrder());
priorityQueue.addAll(ints);
QueueDemo.printQ(priorityQueue);

String fact = "EDUCATION SHOULD ESCHEW OBFUSCATION";
List<String> strings = Arrays.asList(fact.split(""));
PriorityQueue<String> stringPQ =
    new PriorityQueue<String>(strings);
QueueDemo.printQ(stringPQ);
stringPQ = new PriorityQueue<String>(
    strings.size(), Collections.reverseOrder());
stringPQ.addAll(strings);
QueueDemo.printQ(stringPQ);

Set<Character> charSet = new HashSet<Character>();
for(char c : fact.toCharArray())
    charSet.add(c); // Autoboxing
PriorityQueue<Character> characterPQ =
    new PriorityQueue<Character>(charSet);
QueueDemo.printQ(characterPQ);
}
} /* Output:
0 1 1 1 1 1 3 5 8 14
1 1 2 3 3 9 9 14 14 18 18 20 21 22 23 25 25
25 25 23 22 21 20 18 18 14 14 9 9 3 3 2 1 1
    A A B C C C D D E E E F H H I I L N N O O O S S S
T T U U U W
W U U U T T S S S O O O O N N L I I H H F E E E D D C C C B
A A
    A B C D E F H I L N O S T U W
*///:~

```

Как видите, дубликаты допустимы, а наименьшие значения обладают наивысшим приоритетом (в случае `String` пробелы также считаются значениями, которые обладают более высоким приоритетом, чем буквы). Для демонстрации изменения порядка приоритетной очереди посредством передачи собственного объекта `Comparator` третий вызов конструктора `PriorityQueue<Integer>` и второй вызов `PriorityQueue<String>` использует обратный компаратор, полученный при вызове `Collections.reverseOrder()` (метод был добавлен в Java SE5).

В последнем разделе добавляется контейнер `HashSet` для устранения дубликатов символов — просто для того, чтобы сделать задачу чуть более интересной.

Типы `Integer`, `String` и `Character` работают с `PriorityQueue`, потому что для этих классов существует естественный встроенный порядок. Если вы хотите использовать

собственный класс с `PriorityQueue`, включите дополнительную функциональность для введения естественного порядка или передайте собственный объект `Comparator`. В главе 17 приведен более сложный пример, демонстрирующий эти возможности.

- 28.** (2) Заполните контейнер `PriorityQueue` (с использованием метода `offer()`) значениями `Double`, созданными генератором `java.util.Random`. Извлеките элементы из очереди методом `poll()` и выведите их.
- 29.** (2) Создайте простой класс, производный от `Object` и не содержащий членов. Покажите, что множественные элементы этого класса не могут быть добавлены в `PriorityQueue`. Проблема будет более подробно рассмотрена в главе 17.

Collection и Iterator

`Collection` — корневой интерфейс, описывающий общие аспекты всех последовательных контейнеров. Считайте, что это своего рода «случайный интерфейс», появившийся из-за общности между другими интерфейсами. Кроме того, класс `java.util.AbstractCollection` предоставляет реализацию `Collection` по умолчанию, так что вы можете создать новый подтип `AbstractCollection` без избыточного дублирования кода.

Один из доводов в пользу существования такого интерфейса заключается в том, что он позволяет создавать более универсальный код. Код, написанный для интерфейса, а не для реализации, может применяться к большему количеству объектов¹. Итак, метод, получающий `Collection`, может быть применен к любому типу, реализующему `Collection`, а это позволяет разработчику нового класса реализовать `Collection`, чтобы класс мог использоваться с моим методом. Интересно, что в стандартной библиотеке C++ нет общего базового класса для всех контейнеров; все сходство между контейнерами обеспечивается итераторами. По аналогичному пути можно было пойти и в Java — так, чтобы сходство между контейнерами выражалось итератором, а не `Collection`. Однако эти два подхода тесно связаны, а реализация `Collection` также означает поддержку метода `iterator()`:

```
//: holding/InterfaceVsIterator.java
import typeinfo.pets.*;
import java.util.*;

public class InterfaceVsIterator {
    public static void display(Iterator<Pet> it) {
        while(it.hasNext()) {
            Pet p = it.next();
            System.out.print(p.id() + ":" + p + " ");
        }
        System.out.println();
    }
}
```

продолжение ➤

¹ Некоторые разработчики выступают за автоматическое создание интерфейса для каждой возможной комбинации методов класса. Я считаю, что смысл интерфейса не должен сводиться к механическому дублированию комбинаций методов, поэтому не тороплюсь создавать новые интерфейсы, пока польза от них не станет очевидной.

```

public static void display(Collection<Pet> pets) {
    for(Pet p : pets)
        System.out.print(p.id() + ":" + p + " ");
    System.out.println();
}
public static void main(String[] args) {
    List<Pet> petList = Pets.arrayList(8);
    Set<Pet> petSet = new HashSet<Pet>(petList);
    Map<String, Pet> petMap =
        new LinkedHashMap<String, Pet>();
    String[] names = ("Ralph, Eric, Robin, Lacey, " +
        "Britney, Sam, Spot, Fluffy").split(", ");
    for(int i = 0; i < names.length; i++)
        petMap.put(names[i], petList.get(i));
    display(petList);
    display(petSet);
    display(petList.iterator());
    display(petSet.iterator());
    System.out.println(petMap);
    System.out.println(petMap.keySet());
    display(petMap.values());
    display(petMap.values().iterator());
}
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat
{Ralph=Rat, Eric=Manx, Robin=Cymric, Lacey=Mutt,
Britney=Pug, Sam=Cymric, Spot=Pug, Fluffy=Manx}
[Ralph, Eric, Robin, Lacey, Britney, Sam, Spot, Fluffy]
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
*///:~

```

Обе версии `display()` работают с объектами `Map`, а также подтипами `Collection`, причем как интерфейс `Collection`, так и `Iterator` отделяют методы `display()` от информации о конкретной реализации используемого контейнера.

В данном случае эти два решения равноценны. Разве что решение с `Collection` немного предпочтительнее, потому что оно реализует `Iterable`, а следовательно, в реализации `display(Collection)` можно использовать конструкцию `foreach`, с которой код выглядит немного аккуратнее.

Решение с `Iterator` выглядит привлекательно при написании класса, в котором реализация интерфейса `Collection` затруднена или непрактична. Например, при создании реализации `Collection` наследованием от класса, содержащего объекты `Pet`, придется реализовать все методы `Collection`, даже если они не будут использоваться в методе `display()`. И хотя задача легко решается наследованием от `AbstractCollection`, вы все равно будете вынуждены реализовать `iterator()` вместе с `size()` для предоставления методов, которые не реализуются `AbstractCollection`, но используются другими методами `AbstractCollection`:

```

//: holding/CollectionSequence.java
import typeinfo.pets.*;
import java.util.*;

```

```

public class CollectionSequence
extends AbstractCollection<Pet> {
    private Pet[] pets = Pets.createArray(8);
    public int size() { return pets.length; }
    public Iterator<Pet> iterator() {
        return new Iterator<Pet>() {
            private int index = 0;
            public boolean hasNext() {
                return index < pets.length;
            }
            public Pet next() { return pets[index++]; }
            public void remove() { // Не реализован
                throw new UnsupportedOperationException();
            }
        };
    }
    public static void main(String[] args) {
        CollectionSequence c = new CollectionSequence();
        InterfaceVsIterator.display(c);
        InterfaceVsIterator.display(c.iterator());
    }
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
*///:~

```

Метод `remove()` является «необязательной операцией» (см. главу 17). В данном случае ее реализация не обязательна, а при вызове будет выдано исключение.

В этом примере мы видим, что при реализации `Collection` также реализуется метод `iterator()`, а одна лишь реализация `iterator()` требует немного меньших усилий, чем наследование от `AbstractCollection`. Тем не менее, если ваш класс уже наследует от другого класса, вы не сможете наследовать и от `AbstractCollection`. В этом случае для реализации `Collection` придется реализовать все методы интерфейса. В такой ситуации будет намного проще использовать наследование и добавить способность создания итератора:

```

//: holding/NonCollectionSequence.java
import typeinfo.pets.*;
import java.util.*;

class PetSequence {
    protected Pet[] pets = Pets.createArray(8);
}

public class NonCollectionSequence extends PetSequence {
    public Iterator<Pet> iterator() {
        return new Iterator<Pet>() {
            private int index = 0;
            public boolean hasNext() {
                return index < pets.length;
            }
            public Pet next() { return pets[index++]; }
            public void remove() { // Not implemented
                throw new UnsupportedOperationException();
            }
        };
    }
}

```

продолжение ➤

```

public static void main(String[] args) {
    NonCollectionSequence nc = new NonCollectionSequence();
    InterfaceVsIterator.display(nc.iterator());
}
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
*///:~

```

Получение `Iterator` обеспечивает связывание последовательности с методом, работающим с этой последовательностью, при минимальном уровне логических привязок и накладывает гораздо меньше ограничений на класс последовательности, чем реализация `Collection`.

- 30.** (5) Измените пример `CollectionSequence.java` так, чтобы в нем вместо наследования от `AbstractCollection` использовалась реализация `Collection`.

Foreach и итераторы

До настоящего момента синтаксис `foreach` использовался в основном с массивами, но он также работает для любого объекта `Collection`. Мы уже рассмотрели несколько примеров его использования с `ArrayList`, но здесь приводится более общее доказательство:

```

//: holding/ForEachCollections.java
// Все коллекции работают с foreach.
import java.util.*;

public class ForEachCollections {
    public static void main(String[] args) {
        Collection<String> cs = new LinkedList<String>();
        Collections.addAll(cs,
            "Один два три четыре пять".split(" "));
        for(String s : cs)
            System.out.print("'" + s + "' ");
    }
} /* Output:
'Один' 'два' 'три' 'четыре' 'пять'
*///:~

```

Так как `cs` является `Collection`, этот пример показывает, что конструкция `foreach` может использоваться для всех объектов `Collection`.

Почему это решение работает? В Java SE появился новый интерфейс `Iterable`, который содержит метод `iterator()` для получения объекта `Iterator`; именно интерфейс `Iterable` используется `foreach` для перемещения по последовательности. Таким образом, если вы создадите любой класс, реализующий `Iterable`, его можно будет использовать в команде `foreach`:

```

//: holding/IterableClass.java
// Любая реализация Iterable работает с foreach.
import java.util.*;

public class IterableClass implements Iterable<String> {
    protected String[] words = ("And that is how " +
        "we know the Earth to be banana-shaped.").split(" ");
}

```

```

public Iterator<String> iterator() {
    return new Iterator<String>() {
        private int index = 0;
        public boolean hasNext() {
            return index < words.length;
        }
        public String next() { return words[index++]; }
        public void remove() { // Не реализован
            throw new UnsupportedOperationException();
        }
    };
}

public static void main(String[] args) {
    for(String s : new IterableClass())
        System.out.print(s + " ");
}
} /* Output:
Из этого можно сделать вывод, что Земля имеет форму банана.
*///:~

```

Метод `iterator()` возвращает экземпляр анонимной внутренней реализации `Iterator<String>`, которая выдает каждое слово в массиве. В методе `main()` мы видим, что `IterableClass` действительно работает в синтаксисе `foreach`.

В Java SE5 интерфейс `Iterable` реализуется многими классами, прежде всего всеми реализациями `Collection` (но не `Map`). Например, следующая программа выводит все переменные окружения операционной системы:

```

//: holding/EnvironmentVariables.java
import java.util.*;

public class EnvironmentVariables {
    public static void main(String[] args) {
        for(Map.Entry entry: System.getenv().entrySet()) {
            System.out.println(entry.getKey() + ": " +
                               entry.getValue());
        }
    }
} /* (Выполните, чтобы просмотреть результат) *///:~

```

Метод `System.getenv()`¹ возвращает `Map`, метод `entrySet()` создает контейнер `Set` с элементами `Map.Entry`; контейнер `Set` реализует `Iterable`, поэтому он может использоваться в циклах `foreach`.

Команда `foreach` работает с массивом или любой другой реализацией `Iterable`, но это не означает, что массив автоматически реализует `Iterable`, и не подразумевает автоматической упаковки:

```

//: holding/ArrayIsNotIterable.java
import java.util.*;

public class ArrayIsNotIterable {

```

продолжение ➤

¹ До выхода Java SE5 этот метод был недоступен; считалось, что он слишком тесно связан с операционной системой, а следовательно, нарушает принцип «написано один раз, работает везде». Факт его включения наводит на мысль, что проектировщики Java становятся более прагматичными.

```

static <T> void test(Iterable<T> ib) {
    for(T t : ib)
        System.out.print(t + " ");
}
public static void main(String[] args) {
    test(Arrays.asList(1, 2, 3));
    String[] strings = { "A", "B", "C" };
    // Массив работает в foreach, но не является Iterable:
    //! test(strings);
    // Его необходимо явно преобразовать в Iterable:
    test(Arrays.asList(strings));
}
} /* Output:
1 2 3 A B C
*///:~

```

Попытка передачи массива в аргументе `Iterable` завершается неудачей. Преобразование к `Iterable` не выполняется автоматически; его необходимо выполнить вручную.

- 31.** (3) Измените пример `polymorphism/shape/RandomShapeGenerator.java`, чтобы он реализовал `Iterable`. Для этого необходимо добавить конструктор, получающий количество элементов, которые должен создать итератор перед остановкой. Убедитесь в том, что пример работает.

Идиома «Метод-Адаптер»

А если имеется существующий класс, который реализует `Iterable`, и вы хотите добавить новые способы использования этого класса в команде `foreach`? Предположим, вы хотите иметь возможность выбора между перебором списка слов в прямом или обратном направлении. Если просто создать производный класс и переопределить метод `iterator()`, вы замените существующий метод, и выбора не будет.

Одно из решений основано на том, что я называю «идиомой Метод-Адаптер». «Адаптер» в названии происходит от паттернов проектирования, потому что для команды `foreach` необходимо предоставить конкретный интерфейс. Если у вас имеется один интерфейс, а нужен другой, проблема решается написанием адаптера. В данном случае я хочу добавить возможность получения обратного итератора к прямому итератору по умолчанию, поэтому обычное переопределение не подходит. Вместо этого мы добавим метод, создающий объект `Iterable`, который может использоваться в команде `foreach`. Как видно из следующего примера, это позволяет нам предоставить разные способы использования `foreach`:

```

//: holding/AdapterMethodIdiom.java
// Идиома "Метод-Адаптер" позволяет использовать foreach
// с другими разновидностями Iterable.
import java.util.*;

class ReversibleArrayList<T> extends ArrayList<T> {
    public ReversibleArrayList(Collection<T> c) { super(c); }
    public Iterable<T> reversed() {
        return new Iterable<T>() {
            public Iterator<T> iterator() {

```



```

        return new Iterator<T>() {
            int current = size() - 1;
            public boolean hasNext() { return current > -1; }
            public T next() { return get(current--); }
            public void remove() { // Не реализован
                throw new UnsupportedOperationException();
            }
        };
    }
}
}
}
}

```

```

public class AdapterMethodIdiom {
    public static void main(String[] args) {
        ReversibleArrayList<String> ral =
            new ReversibleArrayList<String>(
                Arrays.asList("To be or not to be".split(" ")));
        // Получение обычного итератора при помощи iterator():
        for(String s : ral)
            System.out.print(s + " ");
        System.out.println();
        // Передача реализации Iterable по вашему выбору
        for(String s : ral.reversed())
            System.out.print(s + " ");
    }
}
/* Output:
To be or not to be
be to not or be To
*///:~

```

Если просто поместить объект `ral` в команду `foreach`, вы получите прямой итератор (по умолчанию). Но если вызвать для объекта метод `reversed()`, он обеспечит другое поведение.

Используя эту идиому, мы можем добавить в пример `IterableClass.java` два метода-адаптера:

```

//: holding/MultiIterableClass.java
// Добавление нескольких методов-адаптеров.
import java.util.*;

public class MultiIterableClass extends IterableClass {
    public Iterable<String> reversed() {
        return new Iterable<String>() {
            public Iterator<String> iterator() {
                return new Iterator<String>() {
                    int current = words.length - 1;
                    public boolean hasNext() { return current > -1; }
                    public String next() { return words[current--]; }
                    public void remove() { // Не реализован
                        throw new UnsupportedOperationException();
                    }
                };
            }
        };
    }
}
}
}
}

```

продолжение ➤

```

public Iterable<String> randomized() {
    return new Iterable<String>() {
        public Iterator<String> iterator() {
            List<String> shuffled =
                new ArrayList<String>(Arrays.asList(words));
            Collections.shuffle(shuffled, new Random(47));
            return shuffled.iterator();
        }
    };
}

public static void main(String[] args) {
    MultiIterableClass mic = new MultiIterableClass();
    for(String s : mic.reversed())
        System.out.print(s + " ");
    System.out.println();
    for(String s : mic.randomized())
        System.out.print(s + " ");
    System.out.println();
    for(String s : mic)
        System.out.print(s + " ");
}
} /* Output:
banana-shaped. be to Earth the know we how is that And
is banana-shaped. Earth that how the be And we know to
And that is how we know the Earth to be banana-shaped.
*///:~

```

Обратите внимание: второй метод `random()` не создает свой объект `Iterator`, а просто возвращает объект из перемешанного контейнера `List`.

Из выходных данных видно, что метод `Collections.shuffle()` не влияет на исходный массив, а только перемешивает ссылки в `shuffled`. Это истинно только потому, что метод `randomized()` «заворачивает» результат `Arrays.asList()` в `ArrayList`. Если контейнер `List`, полученный вызовом `Arrays.asList()`, будет перемешиваться напрямую, это повлияет на базовый массив, как видно из следующего примера:

```

//: holding/ModifyingArraysAsList.java
import java.util.*;
public class ModifyingArraysAsList {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Integer[] ia = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        List<Integer> list1 =
            new ArrayList<Integer>(Arrays.asList(ia));
        System.out.println("До перемешивания: " + list1);
        Collections.shuffle(list1, rand);
        System.out.println("После перемешивания: " + list1);
        System.out.println("массив: " + Arrays.toString(ia));

        List<Integer> list2 = Arrays.asList(ia);
        System.out.println("До перемешивания: " + list2);
        Collections.shuffle(list2, rand);
        System.out.println("После перемешивания: " + list2);
        System.out.println("массив: " + Arrays.toString(ia));
    }
} /* Output:
До перемешивания: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```

```

После перемешивания: [4, 6, 3, 1, 8, 7, 2, 5, 10, 9]
массив: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
До перемешивания: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
После перемешивания: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]
массив: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]
*///:~

```

В первом случае вывод `Arrays.asList()` передается конструктору `ArrayList()`; при этом создается объект `ArrayList`, ссылающийся на элементы `ia`. Перемешивание этих ссылок не изменяет массив. Однако если использовать результат `Arrays.asList(ia)` напрямую, перемешивание изменит порядок `ia`. Важно понимать, что `Arrays.asList()` создает объект `List`, использующий базовый массив как физическую реализацию. Если нужно выполнить с `List` какую-либо операцию, изменяющую его, и вы хотите оставить исходный массив неизменным, скопируйте его в другой контейнер.

- 32.** (2) По образцу `MultiIterableClass` добавьте методы `reversed()` и `randomized()` в `NonCollectionSequence.java`, а также заставьте `NonCollectionSequence` реализовать `Iterable`. Продемонстрируйте, что все решения работают в командах `foreach`.

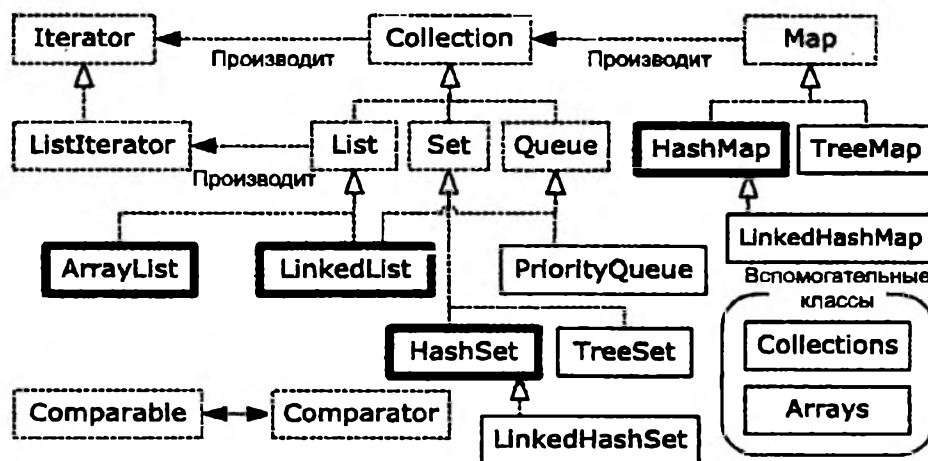
Резюме

В Java предусмотрены следующие способы хранения объектов.

1. Массив ассоциирует с объектом числовой индекс. Он содержит объекты известного типа, так что при поиске объекта нет необходимости проводить преобразование. Он может иметь несколько измерений и способен хранить примитивы, но после создания его размер изменить невозможно.
2. Коллекция (`Collection`) заполняется одиночными элементами, в то время как карта (`Map`) хранит ассоциированные пары «ключ-значение». При использовании обобщенных типов Java разработчик указывает тип объекта, хранимого в контейнере, так что поместить объект неправильного типа ему не удастся, а с извлекаемыми объектами не нужно выполнять преобразование типа. И `Collection`, и `Map` автоматически изменяют свои размеры с добавлением новых элементов. Контейнер не может хранить примитивы, но механизм автоматической упаковки обеспечит необходимые преобразования к типам-«оберткам», хранимым в контейнере.
3. Подобно массиву, список (`List`) также ассоциирует с объектами числовые индексы — массивы и списки можно представить себе как упорядоченные контейнеры.
4. Используйте список `ArrayList`, если имеется необходимость частого получения произвольных элементов, или `LinkedList`, если будет производиться множество вставок и удалений элементов в середине списка.
5. Возможности очередей и стеков предоставляет класс `LinkedList`.
6. Карта (`Map`) позволяет ассоциировать с объектами не числа, но объекты с *другими объектами*. Класс `HashMap` был специально разработан для быстрого доступа к элементам, в то время как `TreeMap` хранит набор ключей в отсортированном виде и поэтому уступает по скорости `HashMap`. `LinkedHashMap` упорядочивает свои элементы в порядке добавления, но обеспечивает быстрый доступ к элементам благодаря хешированию.

7. Множество (Set) позволяет хранить не более одного экземпляра определенного объекта. Множество `HashSet` предназначено для максимально ускоренного поиска, в то время как множество `TreeSet` хранит элементы отсортированными. Элементы множества `LinkedHashSet` хранятся в порядке добавления.
8. При написании новых программ нет никакой необходимости использовать устаревшие классы `Vector`, `Hashtable` и `Stack`.

Полезно взглянуть на упрощенную диаграмму контейнеров Java (без абстрактных классов или унаследованных компонентов). На диаграмме представлены только те интерфейсы и классы, которые часто используются в повседневной работе:



Простая таксономия контейнеров

Из диаграммы видно, что на самом деле существуют только четыре контейнерных компонента: карта (Map), список (List), множество (Set) и очередь (Queue) — и две или три реализации для каждого из них (реализации Queue из `java.util.concurrent` на диаграмме не представлены). Контейнеры, используемые чаще всего, выделены жирными линиями рамки.

Пунктирные прямоугольники представляют интерфейсы, прямоугольники со сплошным контуром — обычные (конкретные) классы. Пунктирные линии с пустыми стрелками обозначают, что определенный класс реализует интерфейс. Сплошные стрелки показывают, что класс может производить объекты того класса, на который нацелена стрелка. Например, любая коллекция (Collection) в состоянии произвести итератор (Iterator), в то время как список (List) может породить итератор `ListIterator` (вместе с обычным итератором `Iterator`, так как список (List) унаследован от Collection).

Следующий пример демонстрирует различия в методах между классами. Код приведен в главе 15; здесь я всего лишь вызываю его для получения выходных данных. В результатах также приведены интерфейсы, реализуемые каждым классом или интерфейсом:

```

//: holding/ContainerMethods.java
import net.mindview.util.*;

public class ContainerMethods {
    public static void main(String[] args) {
        ContainerMethodDifferences.main(args);
    }
}

```

```
} /* Output: (пример)
Collection: [add, addAll, clear, contains, containsAll,
equals, hashCode, isEmpty, iterator, remove, removeAll,
retainAll, size, toArray]
Interfaces in Collection: [Iterable]
Set extends Collection, adds: []
Interfaces in Set: [Collection]
HashSet extends Set, adds: []
Interfaces in HashSet: [Set, Cloneable, Serializable]
LinkedHashSet extends HashSet, adds: []
Interfaces in LinkedHashSet: [Set, Cloneable, Serializable]
TreeSet extends Set, adds: [pollLast, navigableHeadSet,
descendingIterator, lower, headSet, ceiling, pollFirst,
subSet, navigableTailSet, comparator, first, floor, last,
navigableSubSet, higher, tailSet]
Interfaces in TreeSet: [NavigableSet, Cloneable,
Serializable]
List extends Collection, adds: [listIterator, indexOf, get,
subList, set, lastIndexOf]
Interfaces in List: [Collection]
ArrayList extends List, adds: [ensureCapacity, trimToSize]
Interfaces in ArrayList: [List, RandomAccess, Cloneable,
Serializable]
LinkedList extends List, adds: [pollLast, offer,
descendingIterator, addFirst, peekLast, removeFirst,
peekFirst, removeLast, getLast, pollFirst, pop, poll,
addLast, removeFirstOccurrence, getFirst, element, peek,
offerLast, push, offerFirst, removeLastOccurrence]
Interfaces in LinkedList: [List, Deque, Cloneable,
Serializable]
Queue extends Collection, adds: [offer, element, peek,
poll]
Interfaces in Queue: [Collection]
PriorityQueue extends Queue, adds: [comparator]
Interfaces in PriorityQueue: [Serializable]
Map: [clear, containsKey, containsValue, entrySet, equals,
get, hashCode, isEmpty, keySet, put, putAll, remove, size,
values]
HashMap extends Map, adds: []
Interfaces in HashMap: [Map, Cloneable, Serializable]
LinkedHashMap extends HashMap, adds: []
Interfaces in LinkedHashMap: [Map]
SortedMap extends Map, adds: [subMap, comparator, firstKey,
lastKey, headMap, tailMap]
Interfaces in SortedMap: [Map]
TreeMap extends Map, adds: [descendingEntrySet, subMap,
pollLastEntry, lastKey, floorEntry, lastEntry, lowerKey,
navigableHeadMap, navigableTailMap, descendingKeySet,
tailMap, ceilingEntry, higherKey, pollFirstEntry,
comparator, firstKey, floorKey, higherEntry, firstEntry,
navigableSubMap, headMap, lowerEntry, ceilingKey]
Interfaces in TreeMap: [NavigableMap, Cloneable,
Serializable]
*///:~
```

Мы видим, что все разновидности Set, за исключением TreeSet, обладают точно таким же интерфейсом, как Collection. List и Collection существенно различаются, хотя для работы List необходимы методы, присутствующие в Collection. С другой стороны,

методы интерфейса `Queue` самостоятельны; для создания работоспособной реализации `Queue` методы `Collection` не нужны. Наконец, между `Map` и `Collection` существует только одна «точка соприкосновения»: `Map` может создавать объекты `Collection` методами `entrySet()` и `values()`.

Обратите внимание на идентификационный интерфейс `java.util.RandomAccess`, присоединенный к `ArrayList`, но не к `LinkedList`. Он предоставляет информацию для алгоритмов, которые могут динамически изменять свое поведение в зависимости от использования конкретной разновидности `List`.

Пожалуй, такая организация выглядит несколько странно с точки зрения объектно-ориентированных иерархий. Впрочем, по мере изучения контейнеров `java.util` (и особенно материалов главы 17) вы увидите, что сложности не ограничиваются непривычной структурой наследования. Библиотеки контейнеров всегда создавали непростые проблемы при проектировании — для решения этих проблем приходится учитывать факторы, которые часто противоречат друг другу, поэтому проектировщику приходится идти на компромиссы.

Несмотря на эти трудности, контейнеры Java — это инструменты, которые даны вам для повседневного применения, чтобы сделать ваши программы проще, мощнее и эффективнее. Возможно, вам понадобится какое-то время на изучение некоторых аспектов библиотеки, но я уверен, что вы быстро освоитесь с классами библиотеки и начнете пользоваться ими.

12

Обработка ошибок и исключения

Основной принцип языка Java состоит в том, что «плохо написанная программа не запустится».

Лучше всего обнаружить ошибку во время компиляции, еще до запуска программы. Однако не все ошибки поддаются обнаружению в это время. Остальные проблемы должны быть решены во время работы программы, с помощью определенного механизма, позволяющего источнику ошибки передать необходимую информацию о ней получателю, который сможет справиться с возникшими трудностями.

Улучшенный механизм восстановления после ошибок является одним из самых мощных способов увеличения отказоустойчивости вашей программы. Забота об исправлении ошибок важна для любой программы, но для Java особенно, поскольку наиболее приоритетной целью этого языка является создание программных компонентов. *Чтобы программа была надежной, каждая ее часть должна быть надежна.* Предоставляя согласованный механизм обработки исключений, Java дает компонентам надежный способ информировать клиентов об ошибках.

Механизм обработки исключений в Java создавался в расчете на написание больших и устойчивых программ, использующих меньше кода, чем было до этого возможно, с большей уверенностью в том, что ваша программа не имеет необрабатываемых ошибок. Исключения не сложно изучить, но они приносят немедленную и вполне ощутимую пользу в ваших проектах.

Обработка исключений является в Java единственным официальным способом уведомления об ошибках, а ее работа обеспечивается компилятором Java, поэтому в какой-то момент нам все равно пришлось бы задействовать обработку исключений в приводимых примерах. В этой главе вы узнаете, как создавать программы с правильной обработкой исключений, а также как генерировать свои собственные исключения, если какой-то из ваших методов сталкивается с непредусмотренными трудностями.

Основные концепции

В C и нескольких других ранних языках существовали подобные механизмы, хотя они обычно определялись как правила, которые должны были соблюдаться программистами,

и не были частью этих языков. Обычно программа возвращала какое-то специальное значение или устанавливала флаг, предполагая, что получатель просмотрит его, чтобы проверить, была ли ошибка. Но когда прошли годы, обнаружилось, что программисты, использующие готовые библиотеки, склонны верить в свою непогрешимость и думать: «Да, ошибки происходят у других, но никак *не у меня*». Неудивительно, что они не спешили проверять условия возникновения ошибок (впрочем, в некоторых ситуациях проверка этих условий была совершенно излишней¹). Даже *если* вы настолько тщательно писали программу, что при вызове каждого метода проверяли вероятность неверного исполнения, это оборачивалось тем, что текст программы превращался в нечитаемый кошмар. А так как разработчикам приходилось создавать все новые программные системы на тех же языках, им пришлось взглянуть правде в глаза: этот подход к обработке ошибок ограничивал их возможности по созданию больших, надежных и расширяемых программных систем.

Как следствие, было решено устранить элемент случайности из проверки ошибок и встроить механизм этой проверки в язык. Подобный подход имеет долгую историю, разработка механизма *обработки исключений* (exception handling) восходит еще к операционным системам 60-х годов и даже к конструкции BASIC'a «on error goto». Но используемая в C++ обработка исключений основана на языке Ada, а в Java она опирается в основном на конструкции C++ (хотя внешне более близка к Object Pascal).

Слово «исключение» следует воспринимать в смысле «возникла исключительная ситуация». Возможно, в точке возникновения проблемы вы еще не представляете, что же с ней делать, но точно знаете, что уже не можете просто игнорировать и продолжить выполнение; необходимо остановиться, чтобы кто-то где-то предпринял все полагающиеся действия. Но в текущем контексте вы не располагаете всей нужной информацией для преодоления проблемы. Тогда вы направляете свой вопрос на уровень выше, где «квалификация» достаточна, чтобы принять верное решение.

Другое важное преимущество использования механизма исключений состоит в том, что они значительно упрощают создание частей программы, отвечающих за обработку ошибок. Вам больше не понадобится выявлять определенные ошибки в различных местах вашей программы и проверять правильность исполнения метода именно там, где вы его вызываете (поскольку исключения гарантируют, что кто-то обработает их). Проблему можно решить в одном месте, так называемом *обработчике исключений* (exception handler). Такой подход улучшает структуру программы, так как части, отвечающие за выполнение рабочей задачи, отделяются от частей, ответственных за обработку ошибок. И в основном исключения приносят большую эффективность в чтение, написание и отладку программы, если сравнивать с традиционным способом обработки ошибок.

Основные исключения

Исключительная ситуация возникает, когда невозможно нормальное продолжение работы метода или части программы, выполняющихся в данный момент. Важно отличать исключительную ситуацию от «нормальной» ошибки, когда текущая обстановка

¹ С-программисты могут привести в пример возвращаемое значение `printf()`.

позволяет решить возникшее затруднение. В случае исключительной ситуации вы не можете продолжить работу программы, так как вам не хватает информации для разрешения исключения в *текущем контексте*. Все, что вам остается, — это покинуть текущий контекст и передать задачу на более высокий уровень. Именно это и происходит, когда вы возбуждаете исключение.

Самым простым примером является деление. Если вы вознамерились делить на ноль, неплохо было бы оглядеться по сторонам. Что же произойдет, если знаменатель равен нулю? Возможно, вам и известно в контексте проблемы, которую вы пытаетесь решить определенным методом, как поступить с нулевым знаменателем. Но если это значение оказалось для вас неожиданным, операцию выполнить не удастся, и тогда нужно возбудить исключение, а не продолжать исполнение программы по текущему пути.

Когда вы возбуждаете исключение, происходит сразу несколько вещей. Во-первых, создается объект, представляющий исключение, — точно так же, как и любой другой объект в Java: в куче, оператором `new`. Далее текущий поток исполнения (тот самый, где произошла ошибка) останавливается и ссылка на объект, представляющий исключение, извлекается из текущего контекста. С этого момента включается механизм обработки исключений, который начинает поиск подходящего для продолжения исполнения места в программе. Этим местом является *обработчик исключений*, который пытается решить возникшую проблему так, чтобы предпринять другую попытку выполнения задачи или просто продолжить.

Чтобы представить себе простой пример возбуждения исключения, вообразим ссылку на объект, названную `t`. Возможно, что вам передали ссылку, которая не была проинициализирована, и вы хотели бы уточнить это, перед тем как вызывать методы, используя эту ссылку. Вы можете послать информацию об ошибке на более высокий уровень, создав объект, представляющий то, что вы хотите сообщить, и «вытолкнув» его из вашего текущего контекста. Тем самым вы *возбудите исключение*. Вот как это выглядит на примере:

```
if(t == null)
    throw new NullPointerException( );
```

Вырабатывается исключение, которое позволяет вам — в текущем контексте — пережить с себя ответственность, не задумываясь о будущем. Как по волшебству, ошибка обрабатывается где-то в другом месте. Где именно — вы вскоре узнаете.

Исключения позволяют рассматривать все происходящее в программе как транзакции, для защиты которых используются исключения: «...Основополагающая посылка для использования транзакций связана с необходимостью обработки исключений в распределенных вычислениях. Транзакции — компьютерный аналог договорного права. Если что-то идет не так, мы просто отменяем все вычисление»¹. Исключения также можно рассматривать как встроенную систему отмены операций, потому что (при необходимой подготовке) в программе можно создать несколько точек восстановления. Если часть вычислений завершается неудачей, исключение производит «откат» к последней, заведомо стабильной точке программы.

Джим Грей, обладатель премии Тьюринга за вклад в теорию транзакций, в своем интервью на сайте www.actqueue.org.

У исключений есть одна важная особенность: если в программе происходит что-то плохое, исключения не позволяют программе продолжить выполнение по обычному пути. В таких языках, как С и С++, в этой области возникали серьезные проблемы — особенно в языке С, который не позволял принудительно прерывать выполнение ветви программы при возникновении проблемы, поэтому ошибка могла долго игнорироваться, а программа переходила в совершенно некорректное состояние. Исключения (среди прочего) позволяют вам заставить программу остановиться и сообщить о возникших проблемах или (в идеале) преодолеть последствия ошибки и вернуться в стабильное состояние.

Аргументы исключения

Исключения, как и любые другие объекты Java, создаются в куче оператором `new`, который выделяет память и вызывает конструктор. Существует два конструктора для всех стандартных исключений: конструктор по умолчанию и конструктор, который принимает в качестве аргумента строку, где можно поместить подходящую информацию об исключении:

```
throw new NullPointerException("t = null");
```

Указанная строка потом может быть извлечена различными методами, о чем будет рассказано позже.

Ключевое слово `throw` влечет за собой ряд довольно интересных действий. Как правило, сначала ключевое слово `new` используется для создания объекта, представляющего условие произошедшей ошибки. Ссылка на указанный объект передается команде `throw`. И этот объект фактически «возвращается» методом, несмотря на то, что для возвращаемого объекта обычно предусмотрен совсем другой тип. Таким образом, упрощенно можно говорить об обработке исключений как об альтернативном механизме возврата из исполняемого метода, хотя заходить слишком далеко с этой аналогией не стоит. Выдача исключений также позволяет выходить из простых областей действия. В любом случае, объект исключения возвращается, а исполнение текущего метода или области действия завершается.

Но на этом сходство с обычным возвратом из метода и заканчивается, поскольку при возврате с выдачей исключения управление передается *совсем* не в то место, куда оно было бы возвращено при нормальном вызове. (Управление передается подходящему обработчику исключения, который может находиться очень «далеко» — на расстоянии нескольких уровней в стеке вызова — от метода, где, собственно, и возникла исключительная ситуация.)

Вообще говоря, можно возбудить любой тип исключений, происходящих от объекта `Throwable` (корневой класс иерархии исключений). Обычно в программе возбуждаются разные типы исключений для разных типов ошибок. Информация о случившейся ошибке содержится внутри объекта исключения, а также указывается косвенно в самом типе этого объекта, чтобы кто-то на более высоком уровне сумел выяснить, как поступить с исключением. (Очень часто именно тип объекта исключения является единственной доступной информацией об ошибке, в то время как внутри объекта не содержится никакой содержательной информации.)

Перехват исключений

Чтобы увидеть, как перехватываются ошибки, сначала следует усвоить понятие *охраняемого участка*, который представляет собой часть программы, где вероятно появление исключений, и за которым следует специальный блок, отвечающий за обработку этих исключений.

Блок try

Если вы «находитесь» внутри метода и возбуждаете исключение (или это сделает другой вызванный вами метод), этот метод завершит работу при возникновении исключения. Но если вы не хотите, чтобы оператор `throw` завершил работу метода, создайте специальный блок внутри метода, в котором будет перехватываться исключение. В этом блоке, который называется *блоком try*, программа вызывает различные методы. Этот блок представляет собой простую область действия, введенную ключевым словом `try`:

```
try {  
    // Часть программы, способная возбуждать исключения  
}
```

При тщательной проверке ошибок в языке программирования, который не поддерживает обработку исключений, вам бы пришлось добавить к вызову каждого метода, даже одного и того же, дополнительный код для проверки ошибок. С обработкой исключений все помещается в блок `try`, который и перехватывает все возможные исключения в одном месте. А это означает, что вашу программу становится значительно легче писать и читать, поскольку выполняемая задача не смешивается с обработкой ошибок.

Обработчики исключений

Конечно, возбужденное исключение должно где-то найти свое «пристанище». Этим местом является *обработчик исключений*, организуемый для любого исключения, которое вы хотите перехватить. Обработчики исключений размещаются прямо за блоком `try` и вводятся ключевым словом `catch`:

```
try {  
    // Часть программы, способная возбуждать исключения  
} catch (Type1 id1) {  
    // Перехват исключения Type1  
} catch (Type2 id2) {  
    // Перехват исключения Type2  
} catch (Type3 id3) {  
    // Перехват исключения Type3  
}  
// и т. д.
```

Каждый блок `catch` (обработчик исключения) похож на маленький метод, принимающий один и только один аргумент определенного типа. Идентификатор (`id1`, `id2` и т. д.) может быть использован внутри обработчика, точно так же, как и метод распоряжается своими аргументами. Иногда этот идентификатор остается неиспользованным, так как тип исключения дает достаточно информации для обработки исключения, но тем не менее присутствует он всегда.

Обработчики всегда располагаются прямо за блоком `try`. Если возбуждается исключение, механизм обработки исключений ищет первый из обработчиков исключений, аргумент которого соответствует текущему типу исключения. После этого он входит в блок `catch`, и таким образом исключение считается обработанным. После выполнения блока `catch` поиск обработчиков исключения прекращается. Выполняется только один соответствующий типу исключения блок `catch`; в этом отношении конструкция `try-catch` отличается от оператора `switch`, где нужно дописывать `break` после каждой секции `case`, чтобы предотвратить исполнение других секций `case`.

Заметьте также, что внутри блока `try` могут вызываться различные методы, способные возбудить то же исключение, но обработчик понадобится всего один.

Прерывание и возобновление

В теории обработки исключений имеются две основные модели. В *прерывании* (которое используется в Java¹) предполагается, что ошибка настолько серьезна, что не существует способа продолжить исполнение там, где произошло исключение. Кто бы ни возбудил исключение, оно ясно дает понять, что исправить ситуацию невозможно, и даже *не собирается* это сделать.

Альтернативный путь называется *возобновлением*. Он подразумевает, что обработчик ошибок сделает что-то для исправления ситуации; после чего предпринимается попытка повторить неудавшуюся операцию в надежде на успешный исход. Если вы используете возобновление, значит, все еще надеетесь на благополучное продолжение работы программы после обработки исключения.

Если вы хотите реализовать модель возобновления в Java, не возбуждайте исключение при возникновении ошибки (вместо этого следует вызвать метод, способный решить проблему). Другое возможное решение — размещение блока `try` в цикле `while`, который станет снова и снова обращаться к этому блоку до тех пор, пока не будет достигнут нужный результат.

Исторически сложилось, что программисты, использующие операционные системы с поддержкой возобновления, со временем переходили к модели прерывания, забывая другую модель. Таким образом, несмотря на то что идея возобновления выглядит привлекательно, она не настолько полезна на практике. Основная причина кроется в *обратной связи*: обработчик ошибки часто должен знать, где произошло исключение, и содержать специальный код для каждого отдельного места ошибки. А это делает задачу написания и поддержки программы труднее, особенно для больших систем, где исключения могут быть сгенерированы во многих различных местах.

Создание собственных исключений

Разработчик не ограничивается использованием уже существующих в Java исключений. Иерархия исключений Java не может предусмотреть все те ошибки, которые вы хотели бы исправлять, таким образом, вы вправе создавать собственные типы исключений для обозначения специфических ошибок вашей программы.

¹ А также большинстве других языков, включая C++, C, Python, D и т. д.

Для создания собственного класса исключения вам придется унаследовать его от уже существующего типа, желательно того, который наиболее близок вашему по значению (хоть это и не всегда возможно). Простейший путь — создать класс с конструктором по умолчанию, что потребует от вас минимум работы:

```
//: exceptions/InheritingExceptions.java
// Создание собственного исключения.
import com.bruceeckel.simpletest.*;

class SimpleException extends Exception {}

public class InheritingExceptions {
    public void f() throws SimpleException {
        System.out.println("Возбуждаем SimpleException из f()");
        throw new SimpleException();
    }
    public static void main(String[] args) {
        InheritingExceptions sed = new InheritingExceptions();
        try {
            sed.f();
        } catch (SimpleException e) {
            System.out.println("Перехвачено!");
        }
    }
} /* Output:
Возбуждаем SimpleException из f()
Перехвачено!
*///:~
```

Компилятор создает конструктор по умолчанию, который автоматически (и незаметно) вызывает конструктор базового класса. Конечно, в этом случае у вас не будет конструктора вида `SimpleException(String)`, но на практике он не слишком часто используется. Как вы еще увидите, наиболее важно в исключении именно имя класса, так что в основном исключения, похожие на созданное выше, будут достаточны.

В примере результаты работы выводятся в стандартный поток для ошибок на консоль, что достигается использованием класса `System.err`. Обычно это лучше, чем выводить в поток `System.out`, который может быть перенаправлен. Если вы печатаете результаты с помощью `System.err`, существует большая вероятность того, что пользователь их заметит, чем в случае с `System.out`.

Создать класс исключения с конструктором, который получает аргумент-строку, также достаточно просто:

```
//: exceptions/FullConstructors.java

class MyException extends Exception {
    public MyException() {}
    public MyException(String msg) { super(msg); }
}

public class FullConstructors {
    public static void f() throws MyException {
        System.out.println("Возбуждаем MyException из f()");
        throw new MyException();
    }
}
```

```

public static void g() throws MyException {
    System.out.println("Возбуждаем MyException из g()");
    throw new MyException("Создано в g()");
}
public static void main(String[] args) {
    try {
        f();
    } catch(MyException e) {
        e.printStackTrace(System.out);
    }
    try {
        g();
    } catch(MyException e) {
        e.printStackTrace(System.out);
    }
}
} /* Output:
Возбуждаем MyException из f()
MyException
    at FullConstructors.f(FullConstructors.java:11)
    at FullConstructors.main(FullConstructors.java:19)
Возбуждаем MyException из g()
MyException: Создано в g()
    at FullConstructors.g(FullConstructors.java:15)
    at FullConstructors.main(FullConstructors.java:24)
*///:~

```

Добавленный код невелик — появилось два конструктора, определяющих способ создания объекта `MyException`. Во втором конструкторе используется конструктор родительского класса с аргументом `String`, вызываемый ключевым словом `super`.

В обработчике исключений вызывается метод `printStackTrace()` класса `Throwable` (от этого класса унаследован класс `Exception`). Этот метод позволяет восстановить последовательность вызванных методов, которая привела к точке возникновения исключения. По умолчанию эта информация отправляется в стандартный вывод программы, однако при вызове версии по умолчанию:

```
e.printStackTrace();
```

информация направляется в стандартный поток ошибок.

1. (2) Создайте класс с методом `main()`, возбуждающим исключение типа `Exception` из блока `try`. Задайте в конструкторе для `Exception` строковый аргумент. Перехватите исключение в блоке `catch` и распечатайте текст аргумента. Добавьте блок `finally` и выведите сообщение как доказательство его выполнения.
2. (1) Определите ссылку на объект и присвойте ей значение `null`. Попробуйте вызвать метод объекта, пользуясь этой ссылкой. Потом вставьте этот код в блок `try-catch` и перехватите исключение.
3. (1) Напишите код, который генерирует и перехватывает исключение `ArrayIndexOutOfBoundsException`.
4. (2) Создайте ваш собственный класс исключения, используя ключевое слово `extends`. Напишите конструктор, получающий строковый аргумент, и сохраните этот аргумент внутри объекта по ссылке на `String`. Напишите метод, который выводит эту строку. Подсоедините новый блок `try-catch` для проверки нового исключения.

5. (3) Создайте собственную реализацию модели возобновления, используя цикл `while`, который выполняется до тех пор, пока исключение не перестанет выдаваться.

Вывод информации об исключениях

Возможно, вы также захотите вывести информацию об исключениях средствами `java.util.logging`. Хотя подробная информация об использовании пакета приведена в приложении по адресу <http://MindView.net/Books/BetterJava>, базовые средства достаточно просты, чтобы мы могли их использовать здесь.

```
//: exceptions/LoggingExceptions.java
// Вывод информации об исключении через объект Logger.
import java.util.logging.*;
import java.io.*;

class LoggingException extends Exception {
    private static Logger logger =
        Logger.getLogger("LoggingException");
    public LoggingException() {
        StringWriter trace = new StringWriter();
        printStackTrace(new PrintWriter(trace));
        logger.severe(trace.toString());
    }
}

public class LoggingExceptions {
    public static void main(String[] args) {
        try {
            throw new LoggingException();
        } catch(LoggingException e) {
            System.err.println("Перехвачено " + e);
        }
        try {
            throw new LoggingException();
        } catch(LoggingException e) {
            System.err.println("Перехвачено " + e);
        }
    }
} /* Output: (85% match)
Aug 30, 2005 4:02:31 PM LoggingException <init>
SEVERE: LoggingException
    at
    LoggingExceptions.main(LoggingExceptions.java:19)

Перехвачено LoggingException
Aug 30, 2005 4:02:31 PM LoggingException <init>
SEVERE: LoggingException
    at
    LoggingExceptions.main(LoggingExceptions.java:24)

Перехвачено LoggingException
*///:~
```

Статический метод `Logger.getLogger()` создает объект `Logger`, связанный с аргументом `String` (обычно это имя пакета и класс, к которому относится ошибка), который

направляет результат в `System.err`. Простейший способ вывести данные в `Logger` заключается в вызове метода, соответствующего уровню регистрируемого сообщения; в нашем примере используется метод `severe()`. В объекте `String` для регистрируемого сообщения было бы полезно иметь данные трассировки стека в точке возбуждения исключения, но `printStackTrace()` по умолчанию не выдает данные в формате `String`. Чтобы получить объект `String`, необходимо использовать перегруженную версию `printStackTrace()`, которая получает в аргументе объект `java.io.PrintWriter` (эта тема более подробно рассматривается в главе 18). Если передать конструктору `PrintWriter` объект `java.io.StringWriter`, то вывод можно получить в формате `String` вызовом `toString()`.

Подход, использованный в `LoggingException`, чрезвычайно удобен тем, что вся инфраструктура регистрации встраивается в само исключение, а следовательно, работает автоматически без вмешательства со стороны программиста-клиента. Тем не менее на практике вам чаще придется перехватывать и регистрировать чужие исключения, поэтому в обработчике исключения необходимо сгенерировать соответствующее сообщение:

```
//: exceptions/LoggingExceptions2.java
// Регистрация перехваченных исключений.
import java.util.logging.*;
import java.io.*;

public class LoggingExceptions2 {
    private static Logger logger =
        Logger.getLogger("LoggingExceptions2");
    static void logException(Exception e) {
        StringWriter trace = new StringWriter();
        e.printStackTrace(new PrintWriter(trace));
        logger.severe(trace.toString());
    }
    public static void main(String[] args) {
        try {
            throw new NullPointerException();
        } catch (NullPointerException e) {
            logException(e);
        }
    }
} /* Output: (90% match)
Aug 30, 2005 4:07:54 PM LoggingExceptions2 logException
SEVERE: java.lang.NullPointerException
      at
LoggingExceptions2.main(LoggingExceptions2.java:16)
*///:~
```

Процесс создания собственных исключений можно продолжить — в класс исключения можно добавить дополнительные конструкторы и члены:

```
//: exceptions/ExtraFeatures.java
// Расширение функциональности классов исключений.
import static net.mindview.util.Print.*;

class MyException2 extends Exception {
    private int x;
    public MyException2() {}
```



```
public MyException2(String msg) { super(msg); }
public MyException2(String msg, int x) {
    super(msg);
    this.x = x;
}
public int val() { return x; }
public String getMessage() {
    return "Подробное сообщение: " + x + " " + super.getMessage();
}
}

public class ExtraFeatures {
    public static void f() throws MyException2 {
        print("Возбуждаем MyException2 из f()");
        throw new MyException2();
    }
    public static void g() throws MyException2 {
        print("Возбуждаем MyException2 из g()");
        throw new MyException2("Создано в g()");
    }
    public static void h() throws MyException2 {
        print("Возбуждаем MyException2 из h()");
        throw new MyException2("Создано в h()", 47);
    }
    public static void main(String[] args) {
        try {
            f();
        } catch(MyException2 e) {
            e.printStackTrace(System.out);
        }
        try {
            g();
        } catch(MyException2 e) {
            e.printStackTrace(System.out);
        }
        try {
            h();
        } catch(MyException2 e) {
            e.printStackTrace(System.out);
            System.out.println("e.val() = " + e.val());
        }
    }
} /* Output:
Возбуждаем MyException2 из f()
MyException2: Подробное сообщение: 0 null
    at ExtraFeatures.f(ExtraFeatures.java:22)
    at ExtraFeatures.main(ExtraFeatures.java:34)
Возбуждаем MyException2 из g()
MyException2: Подробное сообщение: 0 Создано в g()
    at ExtraFeatures.g(ExtraFeatures.java:26)
    at ExtraFeatures.main(ExtraFeatures.java:39)
Возбуждаем MyException2 из h()
MyException2: Подробное сообщение: 47 Создано в h()
    at ExtraFeatures.h(ExtraFeatures.java:30)
    at ExtraFeatures.main(ExtraFeatures.java:44)
e.val() = 47
*///:~
```

В класс исключения добавлено поле `x` вместе с методом, который читает это значение, и дополнительным конструктором, который его присваивает. Кроме того, метод `Throwable.getMessage()` был переопределен для получения более содержательного сообщения. Метод `getMessage()` представляет собой аналог `toString()` для классов исключений.

Поскольку исключение представляет собой всего лишь разновидность объекта, процесс расширения функциональности классов исключений можно продолжить. Однако помните, что все эти усовершенствования могут быть проигнорированы программистами-клиентами, использующими ваши пакеты, которые могут просто обнаруживать сам факт возбуждения исключения — и ничего более. (Именно так используется большинство исключений библиотеки Java.)

6. (1) Создайте два класса исключения, каждый из которых автоматически выводит информацию о себе. Продемонстрируйте, что эти классы работают.
7. (1) Измените упражнение 3 так, чтобы информация об исключении выводилась в блоке `catch`.

Спецификация исключений

В языке Java рекомендуется сообщать программисту, вызывающему ваш метод, об исключениях, которые данный метод способен возбуждать. Так как пользователь, вызывающий метод, может написать весь необходимый код для перехвата возможных исключений, это полезно. Конечно, когда доступен исходный код, программист-клиент может пролистать его в поиске предложений `throw`, но часто случается так, что библиотека поставляется без исходных текстов. Для решения этой проблемы в Java добавили синтаксис (*обязательный* для использования), позволяющий вам любезно сообщить потребителю метода об исключениях, возбуждаемых этим методом, чтобы он сумел правильно обработать их. Этот синтаксис называется *спецификацией исключений* (exception specification), является частью объявления метода и следует сразу за списком аргументов.

Спецификация исключений предусматривает дополнительное ключевое слово `throws`, за которым перечисляются все возможные типы исключений. Таким образом, определение метода выглядит примерно так:

```
void f() throws TooBig, TooSmall, DivZero { //...
```

Если вы пишете

```
void f() { //...
```

то это значит, что метод не вырабатывает исключений. (*Кроме* исключений, производных от `RuntimeException`, которые могут быть возбуждены практически в любом месте без привлечения спецификации исключений, — об этом еще будет сказано.)

Сообщить неверную информацию о возбуждаемых исключениях невозможно — если ваш метод возбуждает исключения и не обрабатывает их, то компилятор это увидит и предложит либо обработать исключение, либо отметить в спецификации исключений, что оно может быть возбуждено в вашем методе. Жесткий контроль за соблюдением

правил гарантирует правильность использования механизма исключений во время трансляции программы.

Правда, существует один способ обойти этот контроль: вы вправе объявить возбуждение исключения, которого на самом деле нет. Компилятор верит вам «на слово» и заставляет пользователей вашего метода поступать так, как будто им и в самом деле необходимо перехватывать исключение. Таким образом можно «запасться» исключением на будущее и потом уже возбуждать его на самом деле, не изменяя описания готовой программы. Это важно и для создания абстрактных базовых классов и интерфейсов, производным классам которых может быть необходимо возбуждение исключений.

Исключения, которые проверяются и навязываются еще на этапе компиляции программы, называют *контролируемыми* (checked).

8. (1) Напишите класс с методом, который возбуждает исключение, созданное вами в упражнении 4. Попробуйте откомпилировать его без спецификации исключений и посмотрите, что «скажет» компилятор. После этого добавьте необходимую спецификацию исключений. Протестируйте свой класс и его исключение внутри блока try-catch.

Перехват любого типа исключения

Вы можете написать обработчик для перехвата *любых* типов исключений. Для этого следует перехватывать базовый класс всех исключений `Exception` (существуют и другие базовые типы исключений, но класс `Exception` актуален практически для всех видов программирования):

```
catch(Exception e) {  
    System.err.println("Перехвачено исключение");  
}
```

Подобная конструкция не упустит ни одного исключения, так что имеет смысл помещать ее в *конец* списка обработчиков, во избежание блокировки следующих за ней обработчиков исключений.

Поскольку класс `Exception` является базовым для всех классов исключений, интересующих программиста, сам он не предоставит какой-либо нужной информации об исключительной ситуации, но можно вызвать методы из *его* собственного базового типа `Throwable`:

```
String getMessage()  
String getLocalizedMessage()
```

Получает детальное сообщение или локализованное сообщение.

```
String toString()
```

Возвращает короткое описание объекта `Throwable`, включая детальное сообщение, если оно присутствует.

```
void printStackTrace()  
void printStackTrace(PrintStream)  
void printStackTrace(java.io.PrintWriter)
```

Выводит информацию об объекте `Throwable` и трассировку стека вызовов этого объекта. Стек вызовов показывает последовательность вызова методов, которая привела к точке возникновения исключения. Первый вариант отправляет информацию в стандартный поток для ошибок, второй и третий — в поток по вашему выбору (в главе 18 вы поймете, почему типов потоков два).

`Throwable fillInStackTrace()`

Записывает информацию о текущем состоянии кадров стека относительно текущего объекта `Throwable`. Метод используется при повторном возбуждении ошибок или исключений.

Вдобавок, в вашем распоряжении имеются несколько методов из типа `Object`, базового для `Throwable` (и для всех остальных классов). При использовании исключений наиболее полезным кажется метод `getClass()`, который возвращает информацию о классе объекта. Из нее можно узнать имя класса, включающее информацию о пакете (метод `getName()`), или же простое имя класса без уточнений (`getSimpleName()`).

Рассмотрим пример с использованием основных методов класса `Exception`:

```
//: exceptions/ExceptionMethods.java
// Демонстрация методов класса Exception.
import static net.mindview.util.Print.*;

public class ExceptionMethods {
    public static void main(String[] args) {
        try {
            throw new Exception("Мое исключение");
        } catch (Exception e) {
            print("Перехвачено");
            print("getMessage():" + e.getMessage());
            print("getLocalizedMessage():" +
                e.getLocalizedMessage());
            print("toString():" + e);
            print("printStackTrace()");
            e.printStackTrace(System.out);
        }
    }
} /* Output:
Перехвачено
getMessage():Мое исключение
getLocalizedMessage():Мое исключение
toString():java.lang.Exception: Мое исключение
printStackTrace():
java.lang.Exception: Мое исключение
    at ExceptionMethods.main(ExceptionMethods.java:8)
*///:~
```

Можно заметить, что каждый из методов последовательно расширяет объем выдаваемой информации — информация каждого следующего метода образует надмножество информации предыдущего.

9. (2) Создайте три новых типа исключений. Напишите класс с методом, возбуждающим каждое из них. В методе `main()` вызовите этот метод, используя одно предположение `catch`, способное перехватить все три исключения.

Трассировка стека

К информации, предоставляемой методом `printStackTrace()`, также можно обратиться напрямую методом `getStackTrace()`. Этот метод возвращает массив элементов трассировки стека; каждый элемент представляет один *кадр стека*. Нулевой элемент находится на вершине стека и описывает последний вызов метода в последовательности. Следующая программа демонстрирует пример данных трассировки:

```
//: exceptions/WhoCalled.java
// Программный доступ к информации трассировки стека.

public class WhoCalled {
    static void f() {
        // Генерируем исключение для заполнения трассировки стека
        try {
            throw new Exception();
        } catch (Exception e) {
            for (StackTraceElement ste : e.getStackTrace())
                System.out.println(ste.getMethodName());
        }
    }
    static void g() { f(); }
    static void h() { g(); }
    public static void main(String[] args) {
        f();
        System.out.println("-----");
        g();
        System.out.println("-----");
        h();
    }
} /* Output:
f
main
-----
f
g
main
-----
f
g
h
main
*///:~
```

Здесь мы выводим имя метода, но также можно вывести все содержимое `StackTraceElement`, содержащее дополнительную информацию.

Повторное возбуждение исключения

Иногда вам может понадобиться заново возбудить уже перехваченное исключение, что особенно вероятно при использовании класса `Exception` для перехвата любого исключения. Так как вы уже получили ссылку на это исключение, вы его попросту возбуждаете еще раз:

```
catch (Exception e) {
    System.err.println("Было возбуждено исключение");
    throw e;
}
```

Повторное возбуждение исключения заставляет его перейти в распоряжение обработчика исключений более высокого уровня. Все последующие блоки `catch` текущего блока `try` игнорируются. Кроме того, вся информация из объекта, представляющего исключение, сохраняется, соответственно, обработчик более высокого уровня, перехватывающий подобные исключения, сможет ее извлечь.

Если вы просто заново возбуждаете исключение, информация о нем, печатаемая методом `printStackTrace()`, будет по-прежнему относиться к месту, где возникло исключение, но не к месту его повторного возбуждения. Если вам понадобится использовать новую информацию о трассировке стека, используйте метод `fillInStackTrace()`, который возвращает исключение (объект `Throwable`), созданное на основе старого включением туда текущей информации о стеке. Вот как это выглядит:

```
//: exceptions/Rethrowing.java
// Demonstrating fillInStackTrace()

public class Rethrowing {
    public static void f() throws Exception {
        System.out.println("Создание исключения в f()");
        throw new Exception("возбуждено из f()");
    }

    public static void g() throws Exception {
        try {
            f();
        } catch (Exception e) {
            System.out.println("В методе g(), e.printStackTrace()");
            e.printStackTrace(System.out);
            throw e;
        }
    }

    public static void h() throws Exception {
        try {
            f();
        } catch (Exception e) {
            System.out.println("В методе h(), e.printStackTrace()");
            e.printStackTrace(System.out);
            throw (Exception)e.fillInStackTrace();
        }
    }

    public static void main(String[] args) {
        try {
            g();
        } catch (Exception e) {
            System.out.println("main: printStackTrace()");
            e.printStackTrace(System.out);
        }
        try {
            h();
        } catch (Exception e) {
            System.out.println("main: printStackTrace()");
            e.printStackTrace(System.out);
        }
    }
} /* Output:
Создание исключения в f()
```

```

В методе g(),e.printStackTrace()
java.lang.Exception: возбуждено из f()
    at Rethrowing.f(Rethrowing.java:7)
    at Rethrowing.g(Rethrowing.java:11)
    at Rethrowing.main(Rethrowing.java:29)
main: printStackTrace()
java.lang.Exception: возбуждено из f()
    at Rethrowing.f(Rethrowing.java:7)
    at Rethrowing.g(Rethrowing.java:11)
    at Rethrowing.main(Rethrowing.java:29)
Создание исключения в f()
В методе h(),e.printStackTrace()
java.lang.Exception: возбуждено из f()
    at Rethrowing.f(Rethrowing.java:7)
    at Rethrowing.h(Rethrowing.java:20)
    at Rethrowing.main(Rethrowing.java:35)
main: printStackTrace()
java.lang.Exception: возбуждено из f()
    at Rethrowing.h(Rethrowing.java:24)
    at Rethrowing.main(Rethrowing.java:35)
*///:~

```

Строка, в которой вызывается метод `fillInStackTrace()`, становится новой точкой происхождения исключения.

Также возможно повторно возбудить исключение, отличающееся от изначально перехваченного. Если вы это делаете, получается такой же эффект, как и при использовании метода `fillInStackTrace()`, — информация о месте зарождения исключения теряется и остается только то, что относится к новой команде `throw`:

```

//: exceptions/RethrowNew.java
// Повторное возбуждение объекта исключения,
// отличного от перехваченного.

class OneException extends Exception {
    public OneException(String s) { super(s); }
}

class TwoException extends Exception {
    public TwoException(String s) { super(s); }
}

public class RethrowNew {
    public static void f() throws OneException {
        System.out.println("Создание исключения в f()");
        throw new OneException("из f()");
    }
    public static void main(String[] args) {
        try {
            try {
                f();
            } catch (OneException e) {
                System.out.println(
                    "Перехвачено во внутреннем блоке try, e.printStackTrace()");
                e.printStackTrace(System.out);
                throw new TwoException("из внутреннего блока try");
            }
        }
    }
}

```

```

    } catch(TwoException e) {
        System.out.println(
            "Перехвачено во внешнем блоке try, e.printStackTrace()");
        e.printStackTrace(System.out);
    }
}
} /* Output:
Создание исключения в f()
Перехвачено во внутреннем блоке try, e.printStackTrace()
OneException: из f()
    at RethrowNew.f(RethrowNew.java:15)
    at RethrowNew.main(RethrowNew.java:20)
Перехвачено во внешнем блоке try, e.printStackTrace()
TwoException: из внутреннего блока try
    at RethrowNew.main(RethrowNew.java:25)
*///:~

```

О последнем исключении известно только то, что оно возбуждено из внутреннего блока try, но не из метода f().

Вам никогда не придется беспокоиться об удалении предыдущих исключений, и исключений вообще. Все они являются объектами, созданными в общей куче оператором new, и уборщик мусора автоматически уничтожает их.

Цепочки исключений

Зачастую бывает нужно перехватить одно исключение и возбудить следующее, не теряя при этом информации о первом исключении, — это называется *цепочкой исключений* (exception chaining). До выпуска пакета JDK 1.4 программистам приходилось самостоятельно писать код, сохраняющий информацию о предыдущем исключении, однако теперь все подклассы Throwable могут принимать в качестве аргумента конструктора объект-причину. Под *причиной* подразумевается изначальное исключение, и передавая ее в новый объект, вы поддерживаете трассировку стека вплоть до самого его начала, даже если при этом создаете и возбуждаете новое исключение.

Интересно отметить, что единственными подклассами класса Throwable, принимающими объект-причину в качестве аргумента конструктора, являются три основополагающих класса исключений: Error (используется виртуальной машиной (JVM) для сообщений о системных ошибках), Exception и RuntimeException. Если вам понадобится организовать цепочку из других типов исключений, придется использовать метод `initCause()`, а не конструктор.

В следующем примере используется динамическое добавление полей в объект `DynamicFields` во время работы программы:

```

//: exceptions/DynamicFields.java
// Динамическое добавление полей в класс.
// Демонстрирует цепочку исключений.
import static net.mindview.util.Print.*;

class DynamicFieldsException extends Exception {}

public class DynamicFields {
    private Object[][] fields;

```



```

public DynamicFields(int initialSize) {
    fields = new Object[initialSize][2];
    for(int i = 0; i < initialSize; i++)
        fields[i] = new Object[] { null, null };
}
public String toString() {
    StringBuilder result = new StringBuilder();
    for(Object[] obj : fields) {
        result.append(obj[0]);
        result.append(": ");
        result.append(obj[1]);
        result.append("\n");
    }
    return result.toString();
}
private int hasField(String id) {
    for(int i = 0; i < fields.length; i++)
        if(id.equals(fields[i][0]))
            return i;
    return -1;
}
private int
getFieldNumber(String id) throws NoSuchFieldException {
    int fieldNum = hasField(id);
    if(fieldNum == -1)
        throw new NoSuchFieldException();
    return fieldNum;
}
private int makeField(String id) {
    for(int i = 0; i < fields.length; i++)
        if(fields[i][0] == null) {
            fields[i][0] = id;
            return i;
        }
    // Пустых полей нет, добавить новое:
    Object[][] tmp = new Object[fields.length + 1][2];
    for(int i = 0; i < fields.length; i++)
        tmp[i] = fields[i];
    for(int i = fields.length; i < tmp.length; i++)
        tmp[i] = new Object[] { null, null };
    fields = tmp;
    // Рекурсивный вызов с новыми полями:
    return makeField(id);
}
public Object
getField(String id) throws NoSuchFieldException {
    return fields[getFieldNumber(id)][1];
}
public Object setField(String id, Object value)
throws DynamicFieldsException {
    if(value == null) {
        // У большинства исключений нет конструктора,
        // принимающего объект-причину.
        // В таких случаях следует использовать
        // метод initCause(), доступный всем
        // подклассам Throwable.
        DynamicFieldsException dfe =
            new DynamicFieldsException();

```

```

        dfe.initCause(new NullPointerException());
        throw dfe;
    }
    int fieldNumber = hasField(id);
    if(fieldNumber == -1)
        fieldNumber = makeField(id);
    Object result = null;
    try {
        result = getField(id); // Получаем старое значение
    } catch(NoSuchFieldException e) {
        // Используем конструктор с "причиной":
        throw new RuntimeException(e);
    }
    fields[fieldNumber][1] = value;
    return result;
}
public static void main(String[] args) {
    DynamicFields df = new DynamicFields(3);
    print(df);
    try {
        df.setField("d", "Значение d");
        df.setField("number", 47);
        df.setField("number2", 48);
        print(df);
        df.setField("d", "Новое значение d");
        df.setField("number3", 11);
        print("df: " + df);
        print("df.getField(\"d\") : " + df.getField("d"));
        Object field = df.setField("d", null); // Исключение
    } catch(NoSuchFieldException e) {
        e.printStackTrace(System.out);
    } catch(DynamicFieldsException e) {
        e.printStackTrace(System.out);
    }
}
} /* Output:
null: null
null: null
null: null

d: Значение d
number: 47
number2: 48

df: d: Новое значение d
number: 47
number2: 48
number3: 11

df.getField("d") : Новое значение d
DynamicFieldsException
    at DynamicFields.setField(DynamicFields.java:64)
    at DynamicFields.main(DynamicFields.java:94)
Caused by: java.lang.NullPointerException
    at DynamicFields.setField(DynamicFields.java:66)
    ... 1 more
*///:~

```

Каждый объект `DynamicFields` содержит пары `Object-Object`. Первый объект — это идентификатор поля (`String`), а второй объект — значение поля, которое может быть любого типа, кроме не помещенных в оболочки примитивов. При создании объекта вам придется примерно оценить, сколько полей вам может понадобиться. При вызове метода `setField()` он либо находит уже существующее поле по его имени, либо создает новое поле и помещает в него новое значение. Когда пространство для полей заканчивается, метод наращивает его, создавая массив размером на единицу больше и копируя в него старые элементы. Если вы поместите в поле пустую ссылку `null`, то это вызовет исключение `DynamicFieldsException`, создавая объект нужного типа и передавая его методу `initCause()` в качестве причины исключения `NullPointerException`.

Для возвращаемого значения метод `setField()` использует старое значение поля, получая его методом `getField()`, который может вызвать исключение `NoSuchFieldException`. Если метод `getField()` вызывает программист-клиент, то он ответственен за обработку возможного исключения `NoSuchFieldException`, однако если последнее возникает в методе `setField()`, то это является ошибкой программы, соответственно, полученное исключение преобразуется в исключение `RuntimeException` с помощью конструктора, принимающего аргумент-причину.

Для построения результата `toString()` использует объект `StringBuilder`. Этот класс более подробно рассматривается при описании работы со строками, но в общем случае его следует применять в реализациях `toString()`, в основу которых заложен цикл (как в данном случае).

10. (2) Создайте класс с двумя методами, `f()` и `g()`. В методе `g()` возбуждите исключение того типа, который вы определили ранее. В методе `f()` вызовите `g()`, перехватите исключение и в предложении `catch` возбуждите новое исключение (второй тип, который вам необходимо определить). Проверьте этот код в методе `main()`.
11. (1) Повторите предыдущее упражнение, но на этот раз в предложении `catch` преобразуйте исключение метода `g()` в `RuntimeException`.

Стандартные исключения Java

Все, что может быть возбуждено как исключение, описывается классом `Throwable`. Существует две основные разновидности объектов `Throwable` (то есть классов, производных от `Throwable`). Класс `Error` представляет системные ошибки и ошибки времени компиляции, которые обычно не перехватываются (кроме нескольких особых случаев). Класс `Exception` представляет «исключение вообще» и может быть возбужден из любого метода стандартной библиотеки классов Java или вашего метода в случае неполадок в исполнении программы. Таким образом, для программиста интерес представляет в основном класс `Exception`.

Лучший способ получить представление об исключениях — просмотреть документацию JDK. Стоит сделать это хотя бы раз, чтобы почувствовать отличия между различными видами исключений, но вскоре вы легко убедитесь в том, что два разных класса исключений часто не отличаются ничем, кроме имени. К тому же количество исключений в Java постоянно возрастает и едва ли имеет смысл описывать их в книге. Любая программная библиотека от стороннего производителя, скорее всего, также будет иметь

собственный набор исключений. Здесь важнее понять принцип работы и поступать с исключениями сообразно.

Основная идея в том, что имя исключения представляет появившуюся проблему и подразумевается, что это имя говорит само за себя. Не все исключения определены в пакете `java.lang`, некоторые из них созданы для поддержки других библиотек, таких как `util`, `net` и `io`, что можно видеть из полных имен их классов или из базовых классов. Например, все исключения, связанные с вводом-выводом (I/O), унаследованы от `java.io.IOException`.

Особый случай: RuntimeException

Вспомним первый пример в этой главе:

```
if(t == null)
    throw new NullPointerException();
```

Было бы ужасно вот так проверять каждую ссылку, передаваемую вашим методам (так как вы не знаете, была ли передана верная ссылка). К счастью, вам не нужно этого делать — это входит в стандартную проверку во время исполнения Java-программы, и при попытке использования ссылки, содержащей `null`, автоматически возбуждается `NullPointerException`. Таким образом, использованная в примере конструкция избыточна (хотя, возможно, вам стоит выполнить другие проверки, предотвращающие появление `NullPointerException`).

Есть целая группа исключений, принадлежащих к этой категории. Они всегда возбуждаются в Java автоматически, и вам нет нужды включать их в спецификацию исключений. Они удобно сгруппированы, поскольку все унаследованы от одного базового класса `RuntimeException`. Это идеальный пример наследования: образование семейства классов, имеющих общие характеристики и поведение. Вам также не придется создавать спецификацию исключений, говорящую, что метод возбуждает `RuntimeException` (или любое унаследованное от него исключение), так как эти исключения относятся к *неконтролируемым* (*unchecked*). Такие исключения означают ошибки в программе, и фактически вам никогда не придется перехватывать их — это делается автоматически. Если бы вам пришлось проверять возможность возбуждения `RuntimeException`, программа стала бы слишком беспорядочной. И хотя обычно перехватывать `RuntimeException` не требуется, возможно, в своих собственных пакетах вы сочтете целесообразным возбуждать некоторые из них.

Что же происходит, когда подобные исключения не перехватываются? Так как компилятор не заставляет включать спецификацию таких исключений, можно предположить, что `RuntimeException` найдет способ проникнуть прямо в метод `main()` и не будет перехвачен. Чтобы увидеть все в действии, испытайте следующий пример:

```
//: exceptions/NeverCaught.java
// Игнорирование исключений RuntimeException.
// {ThrowsException}

public class NeverCaught {
    static void f() {
        throw new RuntimeException("From f()");
    }
}
```

```
static void g() {  
    f();  
}  
public static void main(String[] args) {  
    g();  
}  
} ///:~
```

Как видно из листинга, `RuntimeException` (и все производные классы) является особым случаем, так как компилятор не требует для него спецификацию исключений. Вывод направляется в `System.error`:

```
Exception in thread "main" java.lang.RuntimeException: From f()  
    at NeverCaught.f(NeverCaught.java:7)  
    at NeverCaught.g(NeverCaught.java:10)  
    at NeverCaught.main(NeverCaught.java:13)
```

Значит, ответ на наш вопрос выглядит так: если `RuntimeException` добирается до `main()` без перехвата, то при выходе из программы для исключения вызывается метод `printStackTrace()`.

Помните, что исключения `RuntimeException` (и их подклассы) могут быть проигнорированы в программном коде, в то время как обработка остальных исключений обеспечивается компилятором. Причина в том, что `RuntimeException` является следствием ошибки программиста, например:

- 1) непредвиденной ошибки (к примеру, передачи вашему методу ссылки `null`, которая находится вне вашего контроля);
- 2) ошибки, которую вы, как программист, должны были проверить в вашей программе (подобной исключению `ArrayIndexOutOfBoundsException`, для которого следовало проверить размер массива). Ошибки из пункта 1 часто становятся причиной ошибок пункта 2.

Разумеется, исключения в этом случае оказывают неоценимую помощь в процессе отладки.

Интересно отметить, что назвать механизм исключений Java направленным на достижение одной цели будет неверно. Да, он предназначен для обработки всех досадных ошибок, происходящих во время исполнения программы, ошибок, которые невозможно предусмотреть, но к тому же этот механизм необходим для обнаружения логических ошибок, которые не могут быть обнаружены компилятором.

- 12.** (3) Измените пример `innerclasses/Sequence.java` так, чтобы при попытке размещения слишком большого количества элементов программа выдавала соответствующее исключение.

Завершение с помощью finally

Часто возникает ситуация, когда часть программы должна выполняться независимо от того, было ли возбуждено исключение внутри блока `try`. Обычно это имеет отношение к операции, отличающейся от освобождения памяти (так как это входит в обязанности

уборщика мусора). Чтобы осуществить задуманное, необходимо использовать блок `finally`¹ в конце всех обработчиков исключений. Таким образом, полная конструкция обработки исключения выглядит так:

```
try {
    // Охраняемая секция: небезопасные операции,
    // которые могут возбуждать исключения А, В или С
} catch(A a1) {
    // Обработчик для ситуации А
} catch(B b1) {
    // Обработчик для ситуации В
} catch(C c1) {
    // Обработчик для ситуации С
} finally {
    // Действия, производимые в любом случае
}
```

Чтобы продемонстрировать, что предложение `finally` всегда выполняется, рассмотрим следующую программу:

```
//: exceptions/FinallyWorks.java
// Предложение finally выполняется всегда.

class ThreeException extends Exception {}

public class FinallyWorks {
    static int count = 0;
    public static void main(String[] args) {
        while(true) {
            try {
                // Постфиксный инкремент; при первом выполнении 0:
                if(count++ == 0)
                    throw new ThreeException();
                System.out.println("Нет исключения");
            } catch(ThreeException e) {
                System.out.println("ThreeException");
            } finally {
                System.out.println("В блоке finally");
                if(count == 2) break; // Выход из "while"
            }
        }
    }
} /* Output:
ThreeException
В блоке finally
Нет исключения
В блоке finally
*///:~
```

Результат работы программы показывает, что вне зависимости от того, было ли возбуждено исключение, блок `finally` выполняется всегда.

Этот пример также подсказывает возможное решение проблемы с невозможностью возврата в Java к месту, где было возбуждено исключение, о чем мы говорили чуть

¹ Механизм обработки исключений в языке C++ не имеет аналога `finally`, поскольку опирается на деструкторы в такого рода действиях.

раньше. Если расположить блок `try` в цикле, можно также определить условие, на основании которого будет решено, должна ли программа продолжаться. Вы можете также добавить статический счетчик или какое-то другое «устройство», чтобы позволить циклу попытаться решить задачу несколькими способами. Это один из способов повысить отказоустойчивость программ.

Для чего нужен блок finally?

В языке без уборки мусора и без автоматических вызовов деструкторов¹ блок `finally` позволяет программисту гарантировать освобождение ресурсов и памяти, что бы ни случилось в блоке `try`, и поэтому очень важно. Но в Java есть уборщик мусора, потому освобождение памяти практически никогда не создает проблем. Также нет необходимости вызывать деструкторы, их просто нет. Когда же нужно использовать `finally` в Java?

Блок `finally` используется, когда вам необходимо вернуть в первоначальное состояние не память, а что-то *другое*. Это может быть, к примеру, открытый файл или сетевое соединение, результат графического вывода на экран или даже какие-то внешние операции, наподобие представленных в следующем примере:

```
//: exceptions/Switch.java
import static net.mindview.util.Print.*;

public class Switch {
    private boolean state = false;
    public boolean read() { return state; }
    public void on() { state = true; print(this); }
    public void off() { state = false; print(this); }
    public String toString() { return state ? "вкл" : "выкл"; }
} ///:~
```

```
//: exceptions/OnOffException1.java
public class OnOffException1 extends Exception {} ///:~
```

```
//: exceptions/OnOffException2.java
public class OnOffException2 extends Exception {} ///:~
```

```
//: exceptions/OnOffSwitch.java
// Для чего используется finally?

public class OnOffSwitch {
    private static Switch sw = new Switch();

    public static void f()
        throws OnOffException1, OnOffException2 {}
    public static void main(String[] args) {
        try {
            sw.on();
            // Код, который может возбуждать исключения...
```

продолжение ➤

¹ Деструктор — это специальная функция, вызываемая всякий раз, когда объект перестает использоваться. Всегда точно известно, где и когда вызывается деструктор. В C++ деструкторы вызываются автоматически, а язык C# (который гораздо больше схож с Java) предлагает способ автоматического разрушения объектов.

```

        f();
        sw.off();
    } catch(OnOffException1 e) {
        System.out.println("OnOffException1");
        sw.off();
    } catch(OnOffException2 e) {
        System.out.println("OnOffException2");
        sw.off();
    }
}
} /* Output:
вкл
выкл
*///:~

```

Цель этой программы — обеспечить выключение переключателя по завершении метода `main()`, поэтому в конце блока `try` и в конце каждого обработчика исключения помещен вызов `sw.off()`. Но также может произойти исключение, которое здесь не перехватывается, и тогда вызов `sw.off()` будет пропущен. Однако с помощью `finally` можно поместить завершающий код в одном определенном месте:

```

//: exceptions/WithFinally.java
// Finally гарантирует выполнение завершающего кода.

```

```

public class WithFinally {
    static Switch sw = new Switch();
    public static void main(String[] args) {
        try {
            sw.on();
            // Код, способный возбуждать исключения...
            OnOffSwitch.f();
        } catch(OnOffException1 e) {
            System.out.println("OnOffException1");
        } catch(OnOffException2 e) {
            System.out.println("OnOffException2");
        } finally {
            sw.off();
        }
    }
} /* Output:
вкл
выкл
*///:~

```

Здесь вызов метода `sw.off()` был просто перемещен в то место, где он гарантированно будет выполнен, что бы ни случилось.

Даже в случае, когда исключение не будет перехвачено в текущем наборе блоков `catch`, блок `finally` отработает, перед тем как механизм обработки исключений продолжит поиск обработчика на более высоком уровне:

```

//: exceptions/AlwaysFinally.java
// Finally выполняется всегда.
import static net.mindview.util.Print.*;

class FourException extends Exception {}

public class AlwaysFinally {

```



```

public static void main(String[] args) {
    print("Входим в первый блок try");
    try {
        print("Входим во второй блок try");
        try {
            throw new FourException();
        } finally {
            print("finally во втором блоке try");
        }
    } catch(FourException e) {
        System.out.println(
            "Перехвачено FourException в первом блоке try");
    } finally {
        System.out.println("finally в первом блоке try");
    }
}
} /* Output:
Входим в первый блок try
Входим во второй блок try
finally во втором блоке try
Перехвачено FourException в первом блоке try
finally в первом блоке try
*///:~

```

Блок `finally` также выполняется в ситуациях, где используются операторы `break` и `continue`. Заметьте, что комбинация `finally` и операторов `break` и `continue` с метками устраняет для Java необходимость в операторе `goto`.

13. (2) Измените упражнение 9, добавив туда блок `finally`. Проверьте, что блок выполняется даже в случае возбуждения `NullPointerException`.
14. (2) Покажите, что программа `OnOffSwitch.java` может завершиться сбоем при возбуждении `RuntimeException` внутри блока `try`.
15. (2) Продемонстрируйте, что программа `WithFinally.java` работает корректно при возбуждении `RuntimeException` внутри блока `try`.

Использование `finally` при `return`

Поскольку блок `finally` выполняется всегда, метод может возвращать управление из нескольких точек — при этом все важные завершающие действия гарантированно будут выполнены:

```

//: exceptions/MultipleReturns.java
import static net.mindview.util.Print.*;

public class MultipleReturns {
    public static void f(int i) {
        print("Инициализация, требующая завершения");
        try {
            print("Точка 1");
            if(i == 1) return;
            print("Точка 2");
            if(i == 2) return;
            print("Точка 3");
            if(i == 3) return;
            print("Конец");
        }
    }
}

```

```

        return;
    } finally {
        print("Завершающие действия");
    }
}
public static void main(String[] args) {
    for(int i = 1; i <= 4; i++)
        f(i);
}
}

```

/* Output:

Инициализация, требующая завершения

Точка 1

Завершающие действия

Инициализация, требующая завершения

Точка 1

Точка 2

Завершающие действия

Инициализация, требующая завершения

Точка 1

Точка 2

Точка 3

Завершающие действия

Инициализация, требующая завершения

Точка 1

Точка 2

Точка 3

Конец

Завершающие действия

*///:~

Из выходных данных видно, что завершение выполняется независимо от того, какая из команд `return` вернула управление в классе с `finally`.

16. (2) Измените пример `reusing/CADSystem.java` и покажите, что при возврате управления из середины `try-finally` все равно выполняются необходимые завершающие действия.
17. (3) Измените пример `polymorphism/Frog.java`, чтобы он использовал `try-finally` для обеспечения необходимых завершающих действий. Покажите, что программа работает даже при возврате из середины `try-finally`.

Ловушка: потерянное исключение

К сожалению, без изъяна в реализации механизма исключений в Java не обошлось. Несмотря на то что исключение сигнализирует о кризисе в программе и никогда не должно игнорироваться, возможна его потеря. Это происходит при использовании `finally` в конструкции определенного вида:

```
//: exceptions/LostMessage.java
```

```
// Как можно потерять исключение.
```

```

class VeryImportantException extends Exception {
    public String toString() {
        return "Очень важное исключение!";
    }
}

```

```

class HoHumException extends Exception {
    public String toString() {
        return "Второстепенное исключение";
    }
}

public class LostMessage {
    void f() throws VeryImportantException {
        throw new VeryImportantException();
    }
    void dispose() throws HoHumException {
        throw new HoHumException();
    }
    public static void main(String[] args) {
        try {
            LostMessage lm = new LostMessage();
            try {
                lm.f();
            } finally {
                lm.dispose();
            }
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
/* Output:
Второстепенное исключение
*///:~

```

Вы видите, что от `VeryImportantException` не осталось и следа — это исключение замещено исключением `HoHumException` в предложении `finally`. Это очень серьезный недочет, так как потеря исключения может произойти в гораздо более скрытой и трудно диагностируемой ситуации, в отличие от той, что показана в примере. Для сравнения: в C++ подобная ситуация (возбуждение второго исключения без обработки первого) считается грубой ошибкой программиста. Возможно, в новых версиях Java эта проблема будет решена (хотя, с другой стороны, любой метод, способный возбуждать исключения, подобный `dispose()` в приведенном примере, можно заключить в блок `try-catch`).

Исключение может быть потеряно еще проще: достаточно включить в блок `finally` команду `return`:

```

//: exceptions/ExceptionSilencer.java

public class ExceptionSilencer {
    public static void main(String[] args) {
        try {
            throw new RuntimeException();
        } finally {
            // Использование 'return' в блоке finally
            // подавляет любое возбужденное исключение.
            return;
        }
    }
}
*///:~

```

При выполнении эта программа не выдает никаких результатов, хотя в ней возбуждается исключение.

18. (3) Добавьте в `LostMessage.java` второй уровень потери исключений, чтобы исключение `NoNumException` само замещалось третьим исключением.
19. (2) Исправьте недостаток `LostMessage.java`, защитив вызов в блоке `finally`.

Ограничения исключений

При переопределении метода вы вправе возбуждать только те исключения, которые были описаны в методе базового класса. Это полезное ограничение означает, что программа, работающая с базовым классом, автоматически сможет работать и с объектом, произошедшим от базового (конечно, это фундаментальный принцип ООП), включая и исключения.

Следующий пример демонстрирует виды ограничений (во время компиляции), наложенные на исключения:

```
//: exceptions/StormyInning.java
// Переопределенные методы могут возбуждать только
// исключения, описанные в версии базового класса,
// или исключения, производные от исключений
// базового класса.

class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
    public Inning() throws BaseballException {}
    public void event() throws BaseballException {
        // Фактически возбуждать исключение не нужно
    }
    public abstract void atBat() throws Strike, Foul;
    public void walk() {} // Не возбуждает контролируемых исключений
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
    public void event() throws RainedOut;
    public void rainHard() throws RainedOut;
}

public class StormyInning extends Inning implements Storm {
    // Можно добавлять новые исключения для
    // конструкторов, но вы должны обработать
    // и исключения базового конструктора:
    public StormyInning()
        throws RainedOut, BaseballException {}
    public StormyInning(String s)
        throws Foul, BaseballException {}
}
```

```

// Обычные методы должны соответствовать правилам базового класса:
//! void walk() throws PopFoul {} //Ошибка компиляции
// Интерфейс НЕ МОЖЕТ добавлять исключения
// к существующим методам базового класса:
//! public void event() throws RainedOut {}
// Если метод не был определен в базовом
// классе, исключение допускается:
public void rainHard() throws RainedOut {}
// Вы можете не возбуждать исключений вообще,
// даже если базовая версия это делает:
public void event() {}
// Переопределенные методы могут возбуждать
// унаследованные исключения:
public void atBat() throws PopFoul {}
public static void main(String[] args) {
    try {
        StormyInning si = new StormyInning();
        si.atBat();
    } catch(PopFoul e) {
        System.out.println("Pop foul");
    } catch(RainedOut e) {
        System.out.println("Rained out");
    } catch(BaseballException e) {
        System.out.println("Generic baseball exception");
    }
    // Strike не возбуждается в унаследованной версии.
    try {
        // Что произойдет при восходящем преобразовании?
        Inning i = new StormyInning();
        i.atBat();
        // Вы должны перехватывать исключения
        // из базовой версии метода:
    } catch(Strike e) {
        System.out.println("Strike");
    } catch(Foul e) {
        System.out.println("Foul");
    } catch(RainedOut e) {
        System.out.println("Rained out");
    } catch(BaseballException e) {
        System.out.println("Общее исключение");
    }
}
} ///:~

```

Взгляните на класс `Inning`: и конструктор, и метод `event()` объявляют, что будут возбуждать исключения, но никогда этого не делают. Это разрешается, поскольку подобный подход позволяет вам заставить пользователя перехватывать все виды исключений, которые потом могут быть добавлены в переопределенные версии метода `event()`. Этот же принцип остается в силе и для абстрактных методов, что и показано для метода `atBat()`.

Интерфейс `Storm` интересен тем, что содержит один метод (`event()`), уже определенный в классе `Inning`, и один уникальный. Оба метода возбуждают новый тип исключения `RainedOut`. Когда класс `StormyInning` расширяется от `Inning` и реализует интерфейс `Storm`, выясняется, что метод `event()` из `Storm` не способен изменить тип исключения для метода `event()` класса `Inning`. Опять-таки, это очень важно, так как иначе вы бы

никогда не знали, перехватываете ли нужное исключение в случае работы с базовым классом. Конечно, когда метод описан в интерфейсе и отсутствует в базовом классе, никаких проблем с возбуждением исключений нет.

Ограничения для исключений не распространяются на конструкторы. Вы можете заметить, что в классе `StormyInning` конструктор волен возбудить любое исключение по своему вкусу, не обращая внимания на то, какие исключения вырабатывает конструктор базового класса. Однако конструктор базового класса так или иначе вызывается (в нашем случае автоматически вызывается конструктор по умолчанию), и поэтому конструктор унаследованного класса должен объявить все исключения базового конструктора в своей спецификации исключений.

Также заметьте, что конструктор унаследованного класса не может перехватывать исключения, возбуждаемые конструктором базового класса.

Причина, по которой метод `StormyInning.walk()` не будет скомпилирован, заключается в том, что он возбуждает исключение, в то время как `Inning.walk()` такого не делает. Если бы это позволялось, вы могли бы написать код, вызывающий метод `Inning.walk()` и не перехватывающий никаких исключений, а потом столкнуться с классом, унаследованным от `Inning`, возникли бы исключения, и в программе произошел бы сбой. Таким образом, навязывая соответствие спецификации исключений в унаследованных и базовых версиях методов, Java добивается взаимозаменяемости объектов.

Переопределенный метод `event()` показывает, что в методах унаследованных классов можно вообще не возбуждать исключений, даже если это делается в базовой версии. Опять-таки, это верно, так как не влияет на уже написанный код — подразумевается, что метод базового класса возбуждает исключения. Примерно то же верно для метода `atBat()`, возбуждающего исключение `PopFoul`, унаследованное от `Foul`, которое, в свою очередь, возбуждается базовой версией `atBat()`. Итого, если кто-то работает с базовым классом `Inning`, то он должен перехватывать исключение `Foul`. Так как `PopFoul` унаследован от `Foul`, обработчик исключения для `Foul` перехватит и `PopFoul`.

И наконец, последний момент, представляющий интерес, — метод `main()`. Здесь вы видите, что при работе именно с объектом `StormyInning` компилятор заставляет перехватывать только те исключения, которые характерны для этого класса, но при восходящем преобразовании к базовому типу компилятор уже навязывает перехват исключений из базового класса. Все эти ограничения приводят к гораздо более надежному коду обработки исключений¹.

Полезно понимать, что, несмотря на то что компилятор заставляет описывать исключения при наследовании, спецификация исключений не является частью объявления метода, которое включает только имя метода и его аргументы. Соответственно, нельзя переопределять методы, опираясь на спецификацию исключений. Вдобавок, даже если оно присутствует в методе базового класса, это вовсе не гарантирует существования этой спецификации в методе унаследованного класса. Данная практика сильно отличается

¹ Язык C++ стандарта ISO вводит аналогичные ограничения при возбуждении исключений унаследованными версиями методов (исключения обязаны быть такими же или унаследованными от исключений базовых версий методов). Это единственный способ C++ для контроля верности описания исключений во время компиляции.

от правил наследования, по которым метод базового класса обязательно есть и в унаследованном классе. Другими словами, «интерфейс спецификации исключений» для определенного метода может сузиться в процессе наследования и переопределения, но никак не расшириться — и это прямая противоположность интерфейсу класса во время наследования.

- 20.** (3) Измените программу `StormyInning.java`, добавив туда исключение типа `UmpireException` и методы, возбуждающие это исключение. Протестируйте получившуюся иерархию.

Конструкторы

Во время написания кода с исключениями следует постоянно спрашивать себя: «Если произойдет исключение, будет ли все корректно завершено?» Большую часть времени вы в известной степени под защитой, но конструкторы привносят проблему. Конструктор приводит объект в определенное начальное состояние, но может начать выполнять какое-либо действие — такое, как открытие файла, — которое не будет правильно завершено, пока пользователь не освободит объект, вызвав специальный завершающий метод. Если вы возбуждаете исключения из конструктора, эти финальные действия могут быть исполнены ошибочно. И это означает, что при написании конструкторов вы должны быть особенно внимательны.

Казалось бы, проблема может быть решена при помощи блока `finally`. Но это не так-то просто, ведь `finally` выполняется *всегда*, и даже тогда, когда вы не хотите исполнять завершающий код до вызова какого-то метода. Если сбой в конструкторе происходит до того, как он будет выполнен полностью, то он может не успеть создать некоторую часть объекта, освобождаемого в `finally`.

В нижеследующем примере создается класс `InputFile`, который открывает файл и позволяет читать из него по одной строке (преобразованной в объект `String`). Он использует классы `FileReader` и `BufferedReader` из стандартной библиотеки ввода-вывода Java, которая будет изучена в главе 18, но эти классы достаточно просты, и вряд ли работа с ними вызовет вопросы:

```
//: exceptions/InputFile.java
// Исключения в конструкторах.
import java.io.*;

public class InputFile {
    private BufferedReader in;
    public InputFile(String fname) throws Exception {
        try {
            in = new BufferedReader(new FileReader(fname));
            // Остальной код, способный возбуждать исключения
        } catch (FileNotFoundException e) {
            System.out.println("Could not open " + fname);
            // Файл не был открыт, закрывать не нужно
            throw e;
        } catch (Exception e) {
            // При других исключениях необходимо закрыть файл
            try {
```

продолжение ➤

```

        in.close();
    } catch(IOException e2) {
        System.out.println("ошибка при выполнении in.close()");
    }
    throw e; // Rethrow
} finally {
    // Здесь файл не закрывается!!!
}
}
public String getLine() {
    String s;
    try {
        s = in.readLine();
    } catch(IOException e) {
        throw new RuntimeException("ошибка при выполнении readLine()");
    }
    return s;
}
public void dispose() {
    try {
        in.close();
        System.out.println("dispose() успешен");
    } catch(IOException e2) {
        throw new RuntimeException("ошибка при выполнении in.close()");
    }
}
} ///:~

```

Конструктор класса `InputFile` получает в качестве аргумента строку (`String`), содержащую имя открываемого файла. Внутри блока `try` он создает экземпляр класса `FileReader` для этого файла. Класс `FileReader` не особенно полезен сам по себе, поэтому мы встраиваем его в созданный `BufferedReader`, с которым и работаем, — заметьте, что одно из преимуществ `InputFile` состоит в том, что он объединяет эти два действия.

Если вызов конструктора `FileReader` проходит неудачно, он возбуждает исключение `FileNotFoundException`, которое должно быть перехвачено отдельно. В этом случае не нужно закрывать файл, так как он и не был открыт. Все *другие* блоки `catch` обязаны закрыть файл, так как он *уже был* открыт ко времени входа в эти предложения. (Конечно, все было бы сложнее в случае, если бы несколько методов могли возбуждать `FileNotFoundException`. Здесь бы вам потребовалось несколько блоков `try`.) Метод `close()` также может возбудить исключение, потому попытка его вызова предпринимается в новом блоке `try`, пусть даже и находящемся в предложении `catch`, — для компилятора Java это всего лишь еще одна пара фигурных скобок. После выполнения всех необходимых действий по месту исключение возбуждается заново, и это правильно — ведь вы не хотите, чтобы вызывающий метод полагал, что объект был благополучно создан.

В этом примере блок `finally` определенно не годится для закрытия файла, поскольку в таком варианте закрытие происходило бы каждый раз по завершении работы конструктора. Так как файл должен оставаться открытым на протяжении срока жизни объекта `InputFile`, подобное решение неверно.

Метод `getLine()` возвращает объект `String`, содержащий очередную строку из файла. Он вызывает метод `readLine()`, способный возбуждать исключения, но они

перехватываются, таким образом, сам `getLine()` исключений не возбуждает. При разработке программы можно полностью обработать исключение на определенном уровне, обработать его частично и его же передать дальше (или какое-то другое), наконец, просто передать дальше. Где это возможно, передача исключения дальше по цепочке значительно упрощает написание программы. В данной ситуации метод `getLine()` *преобразует* исключение в `RuntimeException`, чтобы указать на ошибку в программе.

Метод `dispose()` должен вызываться пользователем, когда объект `InputFile` становится ненужным. При этом освобождаются системные ресурсы (такие, как открытые файлы), закрепленные за объектами `BufferedReader` и/или `FileReader`. Не спешите с методом `dispose()` до тех пор, пока работа с объектом `InputFile` не будет действительно завершена. Вы можете решить, что неплохо было бы поместить подобные действия в метод `finalize()`, но как было упомянуто в главе 5, нельзя полностью полагаться на то, что этот метод будет вызван (и даже если вы знаете, что он *будет* вызван, то неизвестно, *когда*). Это один из недостатков Java: все завершающие действия, за исключением освобождения памяти, не производятся автоматически, так что вам придется уведомить пользователя о том, что он ответствен за их выполнение.

Самый безопасный способ использования класса, который может выдавать исключения во время выполнения конструктора и требует завершающих действий, заключается в использовании вложенных блоков `try`:

```
//: exceptions/Cleanup.java
// Гарантированное освобождение ресурса

public class Cleanup {
    public static void main(String[] args) {
        try {
            InputFile in = new InputFile("Cleanup.java");
            try {
                String s;
                int i = 1;
                while((s = in.getLine()) != null)
                    ; // Обработка данных по строкам...
            } catch(Exception e) {
                System.out.println("Перехвачено исключение Exception в main");
                e.printStackTrace(System.out);
            } finally {
                in.dispose();
            }
        } catch(Exception e) {
            System.out.println("Ошибка при конструировании InputFile");
        }
    }
} /* Output:
dispose() успешен
*///:~
```

Присмотритесь повнимательнее к логике происходящего: объект `InputFile` фактически конструируется в собственном блоке `try`. Если при конструировании произойдет ошибка, программа входит во внешний блок `catch`, а метод `dispose()` не вызывается. Но если конструирование проходит успешно, необходимо проследить за тем, чтобы с объектом были выполнены завершающие действия, поэтому сразу же после конструирования


```
        nc5.dispose();
    }
} catch(ConstructionException e) { // Конструктор nc5
    System.out.println(e);
} finally {
    nc4.dispose();
}
} catch(ConstructionException e) { // Конструктор nc4
    System.out.println(e);
}
}
} /* Output:
NeedsCleanup 1 освобожден
NeedsCleanup 3 освобожден
NeedsCleanup 2 освобожден
NeedsCleanup 5 освобожден
NeedsCleanup 4 освобожден
*///:~
```

В `main()` часть 1 выглядит тривиально: за вызовом конструктора объекта, для которого вызывается `dispose()`, следует `try-finally`. Если ошибки при конструировании объекта исключены, блок `catch` не нужен. В части 2 объекты с конструкторами, в которых ошибки невозможны, группируются при конструировании и завершении.

В части 3 показано, как следует поступать с объектами, в конструкторах которых могут произойти ошибки и которые нуждаются в завершении. Правильное решение выглядит довольно громоздко: каждому вызову конструктора сопоставляется своя команда `try-catch`, и за каждым конструированием объекта должна следовать команда `try-finally`, гарантирующая выполнение завершающих действий.

Громоздкость обработки исключений в таких случаях — веская причина для создания конструкторов, при выполнении которых не могут происходить ошибки (хотя это возможно не всегда). Обратите внимание: если `dispose()` может возбуждать исключения, могут понадобиться дополнительные блоки `try`. Фактически вы должны внимательно продумать все возможные сбои и защититься от каждого.

- 21.** (2) Продемонстрируйте, что конструктор производного класса не может перехватывать исключения, возбужденные конструктором базового класса.
- 22.** (2) Создайте класс `FailingConstructor` с конструктором, во время выполнения которого может произойти ошибка, приводящая к выдаче исключения. В методе `main()` напишите код, который защищает программу от таких сбоев.
- 23.** (4) Добавьте в предыдущее упражнение класс с методом `dispose()`. Измените класс `FailingConstructor` так, чтобы конструктор создавал один из таких объектов в поле класса; далее конструктор может выдать исключение, после чего создает второй объект с необходимостью вызова `dispose()`. Напишите код для защиты от ошибок; в методе `main()` убедитесь в том, что защита распространяется на все возможные ситуации с ошибками.
- 24.** (3) Добавьте в класс `FailingConstructor` метод `dispose()`. Напишите код для правильного использования этого класса.

Отождествление исключений

При возбуждении исключения механизм обработки исключений ищет в списке «ближайших» обработчиков подходящий, в том порядке, в каком они были записаны. Когда соответствие обнаруживается, исключение считается найденным, и дальнейшего поиска не происходит.

Отождествление исключений не требует точного соответствия между исключением и обработчиком. Объект порожденного класса подойдет и для обработчика, изначально написанного для базового класса:

```
//: exceptions/Human.java
// Перехват иерархий исключений.

class Annoyance extends Exception {}
class Sneeze extends Annoyance {}

public class Human {
    public static void main(String[] args) {
        // Перехват точного типа:
        try {
            throw new Sneeze();
        } catch(Sneeze s) {
            System.out.println("Перехвачено Sneeze");
        } catch(Annoyance a) {
            System.out.println("Перехвачено Annoyance");
        }
        // Перехват базового типа:
        try {
            throw new Sneeze();
        } catch(Annoyance a) {
            System.out.println("Перехвачено Annoyance");
        }
    }
} /* Output:
Перехвачено Sneeze
Перехвачено Annoyance
*///:~
```

Исключение `Sneeze` будет перехвачено в первом блоке `catch`, который ему соответствует — таковым, конечно, является первый блок. Но если удалить первый блок `catch`, оставив только `catch` для `Annoyance`, код все равно работает правильно, поскольку исключения `Sneeze` перехватывает базовый класс. Другими словами, блок `catch(Annoyance a)` перехватит `Annoyance` или *любой другой класс, унаследованный от него*. Это удобно, ведь если вы решите добавить больше производных исключений к вашему методу, программа пользователя этого метода не потребует изменений, так как клиент перехватывает исключения базового класса.

Если вы попытаетесь «замаскировать» исключения производного класса, поместив сначала предложение `catch` базового класса, как ниже в примере:

```
try {
    throw new Sneeze();
} catch(Annoyance a) {
```

```
// ...  
} catch(Sneeze s) {  
    // ...  
}
```

то получите сообщение об ошибке от компилятора, который заметит, что предложение `catch` для исключения `Sneeze` никогда не выполнится.

- 25.** (2) Создайте трехуровневую иерархию исключений. Далее сделайте базовый класс `A` с методом, который возбуждает исключение, являющееся основой иерархии. Унаследуйте класс `B` от `A` и переопределите метод так, чтобы он возбуждал исключение из второго уровня иерархии. Аналогично поступите при наследовании класса `C` от `B`. В методе `main()` создайте класс `C`, проведите восходящее преобразование к классу `A`, а затем вызовите метод.

Альтернативные решения

Система обработки исключений — это «черный ход», позволяющий вашей программе отказаться от нормального выполнения ее последовательности предложений. Черный ход «открывается» при возникновении «исключительных ситуаций», когда обычная работа далее невозможна или нежелательна. Исключения представляют собой условия, с которыми текущий метод справиться не в состоянии. Причина, по которой возникли системы обработки исключений, кроется в том, что программисты не желали иметь дело с обременительным подходом, навязывающим проверку всех возможных условий возникновения ошибок каждой функции. В результате ошибки ими просто игнорировались. Стоит отметить, что вопрос удобства программиста при обработке ошибок был для разработчиков Java первичной мотивацией.

Один из главных советов при использовании исключений таков: не обрабатывайте исключение до тех пор, пока вы не знаете, что с ним делать. По сути, отделение кода, ответственного за обработку ошибок, от места, где ошибка возникает, является одной из главных *целей* обработки исключений. Это позволяет вам сосредоточиться на том, что вы хотите сделать, в одном фрагменте кода и реализовать обработку ошибок в совершенно другом месте программы. В результате основной код не перемежается с логикой обработки ошибок и его легко поддерживать и понимать. Обработка исключений также сокращает объем кода, потому что один обработчик может обслуживать разные точки возникновения ошибок.

Контролируемые исключения немного усложняют происходящее, поскольку они заставляют вас добавлять блоки `catch` там, где вы не всегда еще готовы справиться с ошибкой. В итоге возникает проблема «пагубности поспешного поглощения»:

```
try {  
    // ... делает что-то полезное  
} catch(ОбязывающееИсключение e) {} // Поглощено!
```

Программисты (и я в том числе, в первом издании книги), не долго думая, делали самое бросающееся в глаза, и «поглощали» исключение, зачастую непреднамеренно, но как только дело было сделано, компилятор был удовлетворен, поэтому пока вы не вспоминали о необходимости пересмотреть и исправить код, то и не вспоминали об исключении.

Исключение происходит, но безвозвратно теряется при «съедании». Из-за того что компилятор заставляет вас писать код для обработки исключений прямо на месте, это кажется самым простым решением, хотя думается, что ничего хуже и не придумаешь.

Ужаснувшись тому, что я так поступил, во втором издании книги я «исправил» проблему, распечатывая в обработчике трассировку стека исключения (и сейчас это можно видеть — в подходящих местах — в некоторых примерах данной главы). Хотя эта информация пригодится для отслеживания поведения исключений, на самом деле она подразумевает, что вы так и не знаете, что же делать с исключением в данном фрагменте кода. В этом разделе мы рассмотрим некоторые тонкости и осложнения, порождаемые контролируемыми исключениями, и варианты работы с последними.

Несмотря на кажущуюся простоту, вопрос этот не только очень сложен, но к тому же является и предметом постоянных сомнений. Существуют приверженцы обеих точек зрения, которые считают, что верный ответ (их) попросту очевиден. Вероятно, одна из причин связана с несомненными преимуществами, получаемыми при переходе от слабо типизированного языка (такого, как C), не «связанного по рукам и ногам» стандартом ANSI, к языку со строгой статической проверкой типов (то есть с проверкой во время компиляции), подобного C++ или Java. После такого перехода преимущества выглядят настолько неоспоримыми, что строгая статическая проверка типов кажется панацеей от всех бед. Я надеюсь поставить под вопрос ту небольшую часть моей эволюции, отличающуюся *абсолютной* верой в строгую статическую проверку типов: без сомнения, большую часть времени она приносит пользу, но существует размытая граница, переходя которую такая проверка становится препятствием на вашем пути (одна из моих любимых цитат такова: «Все модели неверны, но некоторые полезны»).

Предыстория

Обработка исключений зародилась в таких системах, как PL/1 и Mesa, а затем мигрировала в CLU, Smalltalk, Modula-3, Ada, Eiffel, C++, Python, Java и в появившиеся после Java языки Ruby и C#. Конструкции Java сходны с конструкциями C++, за исключением тех мест, где создатели языка чувствовали, что решения C++ создают проблемы.

Обработка исключений была добавлена в C++ в процессе его стандартизации довольно поздно и предназначалась для предоставления программистам инфраструктуры, которую они с большей охотой стали бы использовать для обработки ошибок и восстановления после сбоя (за это ратовал Бьерн Страуструп, прародитель языка). Модель исключений в C++ в основном была заимствована из CLU. Впрочем, в то время существовали и другие языки с поддержкой обработки исключений: Ada, Smalltalk (в обоих были исключения, но отсутствовали их спецификации) и Modula-3 (в котором существовали и исключения, и их спецификации).

В своих первых заметках, посвященных данному вопросу¹, Лисков и Снайдер заметили, что основным недостатком языков, подобных C, рапортующих об ошибках лишь с переменным успехом, является следующее:

¹ Барбара Лисков и Алан Снайдер: *Exception Handling In CLU*, «Труды IEEE по программному обеспечению», том SE-5, номер 6, ноябрь 1979. Заметки эти недоступны в сети Интернет, так что для получения экземпляра вам придется обратиться в библиотеку.

«...после каждого вызова необходима проверка условия, определяющая, что было получено в результате. Такое требование приводит программам, трудным в чтении и, возможно, неэффективным, из-за чего программисты теряют всякое желание сигнализировать об исключениях и обрабатывать их».

Заметьте, что одной из основных причин создания систем обработки исключений была отмена приведенного требования, но с контролируемыми исключениями в Java мы обычно такой тип кода и видим. Они продолжают:

«...требование присоединения обработчика к вызову, ставшему причиной возникновения исключения, приведет к „неудобоваримым“ программам, в которых выражения будут перемежаться обработчиками».

Следуя подходу CLU при разработке исключений C++, Страуструп утверждал, что его целью было уменьшение количества кода, требуемого для восстановления после ошибки. Я полагаю, что он наблюдал за программистами, которые не писали обрабатывающий ошибки код на C, поскольку объем этого кода был устрашающим, а размещение создавало путаницу. В результате все происходило в стиле C, когда ошибки в коде игнорировались, а с проблемами справлялись при помощи отладчиков. Чтобы исключения использовались в программе, разработчиков необходимо было убедить писать «дополнительный» код, чего они обычно не делали. Таким образом, чтобы склонить их на сторону лучшего способа обработки ошибок, объем «лишнего» кода не должен был становиться им в тягость. Я думаю, важно учитывать эту цель при рассмотрении эффективности контролируемых исключений в Java.

C++ добавил к идее CLU дополнительную возможность: специфирование исключений, позволяющее программно описать, какие исключения могут возникать при вызове данного метода. В действительности у спецификации исключения два предназначения. Она может говорить: «Я возбуждаю это исключение в коде, а вы его обрабатываете». Но она также может утверждать: «Я игнорирую исключение, которое имеет шанс на возникновение в моем коде, но обрабатываете его снова вы». При освещении механизмов исключений мы концентрировались на утверждении «обрабатываете вы», но здесь мне хотелось бы поближе рассмотреть тот факт, что зачастую мы игнорируем исключение, и именно это может сформулировать спецификация исключения.

В C++ спецификация исключений не является частью информации о типе функции. Единственная проверка, осуществляемая во время компиляции, имеет целью удостовериться в последовательности использования исключений: к примеру, если функция или метод возбуждает исключения, то перегруженная или переопределенная версия должна возбуждать те же самые исключения. Однако, в отличие от Java, во время компиляции не проводится проверки на то, действительно ли функция или метод возбуждают данное исключение, или на полноту спецификации (то есть описывает ли она все исключения, возможные для этого метода). Проверка все же происходит, но уже во время работы программы. Если возбуждается исключение, не являющееся частью спецификации исключений, программа на C++ вызывает функцию `unexpected()` из стандартной библиотеки.

Интересно отметить, что из-за использования шаблонов (templates) вы вообще не найдете спецификации исключений в стандартной библиотеке C++. Таким образом, в Java существуют ограничения на возможности использования обобщенных типов Java со спецификациями исключений.

Перспективы

Во-первых, стоит заметить, что язык Java по сути стал первопроходцем в использовании контролируемых исключений (несомненно, из-за спецификаций исключений C++ и того факта, что программисты на C++ не уделяли им слишком много внимания). Это был эксперимент, повторить который с тех пор пока не решился еще ни один язык.

Во-вторых, контролируемые исключения кажутся, несомненно, хорошим средством при рассмотрении вводных примеров и в небольших программах. Оказывается, что трудноуловимые проблемы начинают проявляться при разрастании программы. Конечно, программы не становятся большими тут же и сразу, но они имеют тенденцию расти незаметно. И когда языки, не предназначенные для больших проектов, используются для небольших, но растущих проектов, мы в некоторый момент с удивлением обнаруживаем, что ситуация постепенно выходит из-под контроля. Именно это, как я полагаю, может произойти, когда проверок типов слишком много, и особенно в отношении контролируемых исключений.

Масштаб программы, похоже, является немаловажным вопросом. Это представляет собой проблему, поскольку, как правило, в дискуссиях демонстрируются небольшие программки. Один из проектировщиков C# подчеркнул¹, что:

«...Изучение небольших программ приводит к выводу, что требование спецификации исключений позволяет и увеличить продуктивность разработчика, и улучшить качество кода; однако опыт с большими проектами приводит к другому заключению — уменьшенная продуктивность и небольшое улучшение качества кода или отсутствие такового вовсе».

В отношении необрабатываемых исключений создатели CLU говорят²:

«Мы посчитали, что невозможно требовать от программиста написания обработчика в ситуациях, где нет возможности провести осмысленные действия».

Страуструп, объясняя, почему объявление функции без спецификации исключений означает, что функция может возбудить любое исключение, а не то, что исключений вообще не возникает, утверждает³:

«В таком случае понадобились бы спецификации исключения практически для любой функции, это стало бы серьезной причиной для перекомпиляции и препятствовало бы взаимодействию с программами, написанными на других языках. Все это заставило бы программистов низвергнуть механизм обработки исключений и писать фальшивый код, подавляющий исключения. Это дало бы ложное чувство безопасности людям, не сумевшим увидеть исключения».

Именно такое поведение — «низвержение» исключений — и происходит с управляемыми исключениями в Java.

Мартин Фаулер (автор книг *UML Distilled*, *Refactoring* и *Analysis Patterns*) написал мне следующее:

¹ <http://discuss.develop.com/archives/wa.exe?A2=ind0011A&L=DOTNET&P=R32820>.

² Exception Handling in CLU, Liskov & Snyder.

³ Bjarne Stroustrup, *The C++ Programming Language*, 3rd Edition (Addison-Wesley, 1997), с. 376.

«...в общем и целом исключения хороши, но контролируемые исключения в Java создают больше проблем, чем решают».

Сейчас я полагаю, что важным шагом Java стала унификация модели информирования об ошибках, так как обо всех ошибках *сообщается* посредством исключений. В C++ этого не было из-за обратной совместимости с C и возможности задействовать старую модель банального игнорирования исключений. Однако если вы были последовательны в отношении исключений, то по желанию их можно было использовать, а в противном случае они переходили на более высокий уровень (консоль или другой «контейнер» программы). Когда Java изменил модель C++ так, что сообщать об ошибках стало возможно только посредством исключений, дополнительные принуждения в виде управляемых исключений стали менее необходимы.

В прошлом я был убежденным сторонником того, что для разработки надежных программ необходимы и контролируемые исключения, и строгая статическая проверка типов. Однако опыт, полученный лично и со стороны¹, с языками, более динамичными, чем статичными, привел меня к мысли, что на самом деле наивысшие преимущества возникают, когда используется:

- 1) унифицированная модель сообщения об ошибках посредством исключений, вне зависимости от того, заставляет ли компилятор программиста их обрабатывать;
- 2) проверка типов, не привязанная к тому, *когда* она проводится. То есть если в итоге обеспечивается правильное использование типа, неважно, происходит ли это при компиляции или во время работы программы.

На вершине всего этого, при уменьшении ограничений времени компиляции, виднеются весьма значительные плюсы для увеличения продуктивности программиста. Действительно, *отражение* (и со временем *обобщенные типы*) приходит на помощь, компенсируя чрезмерную суровость строгой статической проверки типов, как вы убедитесь в следующей главе и в некоторых примерах книги.

Мне уже говорили, что это чуть ли не кощунство и что при публичном провозглашении этих слов моя репутация будет безнадежно испорчена, цивилизация погибнет, а большой процент программных проектов провален. Вера в то, что компилятор может спасти ваш проект посредством указания на ошибки при компиляции, весьма сильна, но гораздо более важно осознавать ограничения того, на что способен компьютер, — в приложении <http://MindView.net/Books/BetterJava> я подчеркиваю важность автоматизированной уборки и модульного тестирования, и это дает вам гораздо больше возможностей для достижения цели, чем попытки превратить все в синтаксическую ошибку. Стоит помнить, что:

«Хороший язык программирования помогает программистам писать хорошие программы. Ни один из языков программирования не может запретить своим пользователям писать плохие программы»².

¹ Косвенно через язык Smalltalk, после разговоров со многими опытными программистами на этом языке, и напрямую при работе с Python (www.Python.org).

² (Киз Костер, архитектор языка CDL, процитировано Бертраном Мейером, создателем языка Eiffel.) <http://www.elj.com/elj/v1/n1/bm/right>.

В любом случае, исчезновение когда-либо из Java контролируемых исключений весьма маловероятно. Это слишком радикальное изменение языка, и защитники их в Sun весьма сильны. История Sun неотделима от политики абсолютной обратной совместимости — фактически любое программное обеспечение Sun работает на любом аппаратном обеспечении Sun, как бы старо оно ни было. Однако, если вы чувствуете, что контролируемые исключения становятся для вас препятствием, особенно когда вас заставляют обрабатывать исключения, с которыми вы не знаете как поступать, вот несколько вариантов.

Передача исключений на консоль

В несложных программах, как во многих примерах данной книги, простейшим способом сберечь исключение, не громоздя лишний код, является передача его за пределы метода `main()`, на консоль. К примеру, если вы захотите открыть файл для чтения (подробности вы вскоре узнаете в главе 12), то должны открыть и закрыть поток `FileInputStream`, который возбуждает исключения. В небольшой программе можно поступить следующим образом (подобный подход характерен для многих примеров книги):

```
//: exceptions/MainException.java
import java.io.*;

public class MainException {
    // Передаем все исключения на консоль:
    public static void main(String[] args) throws Exception {
        // Открываем файл:
        FileInputStream file =
            new FileInputStream("MainException.java");
        // Используем файл ...
        // Закрываем файл:
        file.close();
    }
} ///:~
```

Заметьте, что `main()` — это такой же, как и прочие, метод, который может иметь спецификацию исключений, и здесь типом исключения является `Exception`, базовый класс всех управляемых исключений. Передавая его на консоль, вы освобождаетесь от необходимости написания конструкций `try-catch` в теле метода `main()`. (К сожалению, файловый вывод-вывод гораздо сложнее, чем может показаться из данного примера, поэтому до прочтения главы 18 особенно не обольщайтесь.)

- 26.** (1) Измените строку имени файла в примере `MainException.java` так, чтобы она содержала имя несуществующего файла. Запустите программу и обратите внимание на результат.

Преобразование контролируемых исключений в неконтролируемые

Рассмотренный выше подход хорош при написании метода `main()`, но в более общих ситуациях не слишком полезен. Подлинная проблема возникает при написании тела самого обычного метода, когда вы вызываете еще один метод и осознаете: «Я не имею ни малейшего представления о том, что делать с исключением дальше, но „поглощать“

его мне не хочется, так же как и печатать банальное сообщение». Благодаря цепочкам исключений у этой проблемы появляется простое решение. Управляемое исключение просто «заворачивается» в класс `RuntimeException`, примерно так:

```
try {
    // ... делаем что-нибудь полезное
} catch(НеЗнаюЧтоДелатьСЭтимКонтролируемымИсключением e) {
    throw new RuntimeException(e);
}
```

Кажется, что это идеальное решение, если вы хотите «отключить» контролируемое исключение, — вы не «поглощаете» его, вам не приходится описывать его в своей спецификации исключений, и благодаря цепочке исключений вы не теряете информацию об оригинальном исключении.

Такой прием дает возможность игнорировать исключение и пустить его «всплывать» вверх по стеку вызова без необходимости писать конструкции `try-catch` и/или спецификации исключений. Впрочем, вы все еще можете перехватить и обработать некоторое исключение, используя метод `getCause()`, как показано здесь:

```
//: exceptions/TurnOffChecking.java
// "Отключение" контролируемых исключений.
import java.io.*;
import static net.mindview.util.Print.*;

class WrapCheckedException {
    void throwRuntimeException(int type) {
        try {
            switch(type) {
                case 0: throw new FileNotFoundException();
                case 1: throw new IOException();
                case 2: throw new RuntimeException("Где я?");
                default: return;
            }
        } catch(Exception e) { // Превращаем в неконтролируемое:
            throw new RuntimeException(e);
        }
    }
}
```

```
class SomeOtherException extends Exception {}
```

```
public class TurnOffChecking {
    public static void main(String[] args) {
        WrapCheckedException wce = new WrapCheckedException();
        // Вы можете вызвать f() без блока try и позволить
        // исключению RuntimeException покинуть метод:
        wce.throwRuntimeException(3);
        // Или перехватить исключения:
        for(int i = 0; i < 4; i++)
            try {
                if(i < 3)
                    wce.throwRuntimeException(i);
                else
                    throw new SomeOtherException();
            } catch(SomeOtherException e) {
```

продолжение ➤

```

        print("SomeOtherException: " + e);
    } catch(RuntimeException re) {
        try {
            throw re.getCause();
        } catch(FileNotFoundException e) {
            print("FileNotFoundException: " + e);
        } catch(IOException e) {
            print("IOException: " + e);
        } catch(Throwable e) {
            print("Throwable: " + e);
        }
    }
}
} /* Output:
FileNotFoundException: java.io.FileNotFoundException
IOException: java.io.IOException
Throwable: java.lang.RuntimeException: Где я?
SomeOtherException: SomeOtherException
*///:~

```

Метод `wrapCheckedException.throwRuntimeException()` содержит код, генерирующий различные типы исключений. Они перехватываются и «заворачиваются» в объекты `RuntimeException`, становясь таким образом «причиной» этих исключений.

Глядя на класс `TurnOffChecking`, нетрудно заметить, что вызвать метод `throwRuntimeException()` можно и без блока `try`, поскольку он не возбуждает никаких контролируемых исключений. Однако когда вы будете готовы перехватить исключение, у вас будет возможность перехватить любое из них — достаточно поместить свой код в блок `try`. Начинаете вы с перехвата исключений, которые, как вы знаете, могут явно возникнуть в коде блока `try`, — в нашем случае первым делом перехватывается `SomeOtherException`. В конце вы перехватываете `RuntimeException` и заново возбуждаете исключение, являющееся его причиной (получая последнее методом `getCause()`, «вернутое» исключение). Так извлекаются изначальные исключения, обрабатываемые в своих предложениях `catch`.

Прием «упаковки» управляемых исключений в объектах `RuntimeException` будет по мере необходимости использоваться в оставшейся части книги. Другое возможное решение — создание собственного класса, производного от `RuntimeException`. В этом случае перехватывать его не обязательно, но другая сторона сможет перехватить его, если пожелает.

- 27.** (1) Измените упражнение 3 и преобразуйте исключение в `RuntimeException`.
- 28.** (1) Измените упражнение 4 так, чтобы класс исключения был производным от `RuntimeException`. Покажите, что компилятор позволяет опустить блок `try`.
- 29.** (1) Измените все типы исключений в `StormyInning.java` так, чтобы они расширяли `RuntimeException`. Покажите, что при этом не обязательны ни спецификации исключений, ни блоки `try`. Удалите комментарии `///` и продемонстрируйте, что эти методы могут компилироваться без спецификаций.
- 30.** (2) Измените пример `Human.java` так, чтобы исключения наследовали от `RuntimeException`. Измените метод `main()` так, чтобы прием из примера `TurnOffChecking.java` использовался для обработки разных типов исключений.

Рекомендации по использованию исключений

Используйте исключения для того, чтобы:

1. Обработать ошибку на текущем уровне. (Старайтесь не перехватывать исключения, если вы не знаете, как с ними поступить.)
2. Исправить проблему и снова вызвать метод, возбудивший исключение.
3. Предпринять все необходимые действия и продолжить выполнение без повторного вызова метода.
4. Попытаться найти альтернативный результат вместо того, который должен был бы произвести вызванный метод.
5. Сделать все, что можно в текущем контексте, и заново возбудить *это же* исключение, перенаправив его на более высокий уровень.
6. Сделать все, что можно в текущем контексте, и возбудить *новое* исключение, перенаправив его на более высокий уровень.
7. Завершить работу программы.
8. Упростить программу. (Если используемая вами схема обработки исключений делает все только сложнее, значит, она никуда не годится.)
9. Повысить уровень безопасности вашей библиотеки и программы. (Сначала это поможет в отладке программы, а в дальнейшем окупится ее надежностью.)

Резюме

Исключения являются неотъемлемым аспектом программирования на языке Java; без умения работать с ними вы далеко не уйдете. По этой причине исключения представлены на этой стадии изложения материала — хотя существует много библиотек (например, библиотека ввода-вывода, упоминавшаяся ранее), с которыми невозможно работать без обработки исключений.

Одно из преимуществ механизма обработки исключений заключается в том, что он позволяет сосредоточиться на решаемой задаче в одном месте, а затем разобраться с ошибками, которые могут возникнуть в этом коде, в другом месте. И хотя исключения обычно рассматриваются как инструменты передачи информации и восстановления работоспособности после ошибок во время выполнения, я не уверен в том, насколько часто реализуется аспект «восстановления» — да и насколько это возможно. По моим оценкам, это происходит менее чем в 10 % случаев, и даже в этом случае восстановление обычно сводится к раскрутке стека в последнее, заведомо стабильное состояние, а не к выполнению каких-то восстановительных операций. Не знаю, правда это или нет, но я пришел к убеждению, что подлинная ценность исключений проявляется именно в области передачи информации. Java фактически требует, чтобы программа сообщала обо всех ошибках в виде исключений, и в этом проявляется огромное преимущество Java перед такими языками, как C++, которые позволяют сообщать об ошибках многими разными способами — или не сообщать о них вовсе. Последовательная система передачи информации об ошибках означает, что вам уже не нужно задавать себе вопрос:

«А не закралась ли здесь какая-нибудь ошибка?» в каждом написанном фрагменте кода (конечно, при условии, что ваш код не «поглощает» исключения!).

Как будет показано в следующих главах, избавление от хлопот с ошибками (даже если оно сводится к обычному возбуждению `RuntimeException`) позволяет вам сосредоточить свои усилия по проектированию и реализации на более интересных и творческих вопросах.

13 Строки

Операции со строками являются одной из самых распространенных задач программирования.

Это утверждение особенно справедливо для веб-систем, в реализации которых широко применяется Java. В этой главе мы рассмотрим класс `String` — безусловно, один из важнейших классов Java, а также некоторые из сопутствующих классов и средств.

Постоянство строк

Объекты класса `String` *постоянны* (immutable). Просмотрев документацию класса `String` в JDK, вы увидите, что каждый метод класса, который на первый взгляд изменяет `String`, в действительности создает и возвращает новый объект `String` с включенными изменениями. Исходный объект `String` при этом не модифицируется.

Рассмотрим следующий пример:

```
//: strings/Immutable.java
import static net.mindview.util.Print.*;

public class Immutable {
    public static String upcase(String s) {
        return s.toUpperCase();
    }
    public static void main(String[] args) {
        String q = "howdy";
        print(q); // howdy
        String qq = upcase(q);
        print(qq); // HOWDY
        print(q); // howdy
    }
} /* Output:
howdy
HOWDY
howdy
*///:~
```

При передаче `q` методу `upcase()` в действительности передается копия ссылки на `q`. Физическое местонахождение объекта, с которым связана ссылка, при этом не изменяется. Ссылки копируются при передаче.

Из определения `upcase()` видно, что полученная ссылка с именем `s` существует только во время выполнения `upcase()`. После завершения `upcase()` локальная ссылка `s` пропадает. Метод `upcase()` возвращает результат — исходную строку, все символы которой преобразованы к верхнему регистру. Конечно, в действительности метод возвращает ссылку на результат. Но оказывается, что возвращаемая ссылка указывает на новый объект, а исходный объект `q` остается в прежнем виде.

Обычно требуется именно такое поведение. Допустим, программа содержит следующий фрагмент:

```
String s = "asdf";
String x = Immutable.upcase(s);
```

Должен ли метод `upcase()` *изменять* свой аргумент? С точки зрения читателя кода аргумент обычно представляет информацию, предоставляемую методу, а не ту, которая требует изменения. Это важное обстоятельство, упрощающее написание и понимание кода.

Перегрузка + и StringBuilder

Так как объекты `String` не модифицируются, для одного объекта `String` в программе можно создать сколько угодно «синонимов». Поскольку объект `String` доступен только для чтения, одна ссылка ни при каких условиях не изменит данные, используемые по другим ссылкам.

Оператор `+` был перегружен для объектов `String`. *Перегрузкой* называется изменение смысла оператора при его использовании с конкретным классом. (Перегрузка операторов в Java ограничивается операторами `+` и `+=` для класса `String`; Java не позволяет программисту перегружать другие операторы¹.)

Оператор `+` может использоваться для конкатенации объектов `String`:

```
//: strings/Concatenation.java

public class Concatenation {
    public static void main(String[] args) {
        String mango = "mango";
        String s = "abc" + mango + "def" + 47;
        System.out.println(s);
    }
} /* Output:
abcmangodef47
*///:~
```

¹ В C++ программист может перегружать любые операторы по своему усмотрению. Так как этот процесс часто приводит к сложностям, создатели Java решили, что это «плохая» возможность, которую не стоит включать в язык. Видимо, она все же была недостаточно плохой, если они сами не обошлись без перегрузки операторов, и как ни парадоксально, в Java использовать перегрузку операторов было бы намного проще, чем в C++. Например, в языках Python (www.Python.org) и C#, использующих уборку мусора, реализован упрощенный механизм перегрузки операторов.

Как можно было бы реализовать эту серию операций? Объект `String` "abc" может содержать метод `append()`, который создает новый объект `String` из подстроки "abc", объединенной с содержимым `mango`. Полученный объект `String` создает новый объект `String`, к которому добавляется подстрока "def" и т. д.

Безусловно, такое решение работает, но оно требует создания множества промежуточных объектов `String` для простого построения нового объекта `String`, которые потом должны уничтожаться уборщиком мусора. Подозреваю, что создатели Java сначала опробовали этот способ и решили, что его быстрое действие не приемлемо.

Чтобы понять, что происходит в действительности, можно декомпилировать приведенный код программой `javap`, входящей в поставку JDK. Командная строка выглядит так:

```
javap -c Concatenation
```

Флаг `-c` генерирует байт-код JVM. После удаления частей, которые не представляют интереса, и некоторого редактирования мы получаем следующий байт-код:

```
public static void main(java.lang.String[]);
Code:
    Stack=2, Locals=3, Args_size=1
    0:   ldc #2; //String mango
    2:   astore_1
    3:   new #3; //class StringBuilder
    6:   dup
    7:   invokespecial #4; //StringBuilder."<init>":()
    10:  ldc #5; //String abc
    12:  invokevirtual #6; //StringBuilder.append:(String)
    15:  aload_1
    16:  invokevirtual #6; //StringBuilder.append:(String)
    19:  ldc #7; //String def
    21:  invokevirtual #6; //StringBuilder.append:(String)
    24:  bipush 47
    26:  invokevirtual #8; //StringBuilder.append:(I)
    29:  invokevirtual #9; //StringBuilder.toString:()
    32:  astore_2
    33:  getstatic #10; //Field System.out:PrintStream;
    36:  aload_2
    37:  invokevirtual #11; // PrintStream.println:(String)
    40:  return
```

Читателям с опытом работы на ассемблере этот результат может показаться знакомым — такие команды, как `dup` и `invokevirtual`, являются аналогом ассемблера для виртуальной машины Java (JVM). Если вы никогда не видели язык ассемблера, не огорчайтесь — здесь важно то, что компилятор вводит в программу класс `java.lang.StringBuilder`. В исходном коде `StringBuilder` не упоминается, но компилятор все равно решил его использовать, потому что он работает намного эффективнее.

В нашем примере компилятор создает объект `StringBuilder` для построения `String s`, после чего вызывает `append()` четыре раза, по одному для каждого фрагмента. В завершение он вызывает метод `toString()` для получения результата и сохраняет его (командой `astore_2`) под именем `s`.

Итак, нужно повсюду использовать `String`, а компилятор позаботится об эффективности? Не торопитесь с выводами. Давайте повнимательнее присмотримся к тому, что

делает компилятор. В следующем примере результат `String` создается двумя способами: с использованием `String` и с «ручной» реализацией, использующей `StringBuilder`:

```
//: strings/WhitherStringBuilder.java
```

```
public class WhitherStringBuilder {
    public String implicit(String[] fields) {
        String result = "";
        for(int i = 0; i < fields.length; i++)
            result += fields[i];
        return result;
    }
    public String explicit(String[] fields) {
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < fields.length; i++)
            result.append(fields[i]);
        return result.toString();
    }
} ///:~
```

Выполнив команду `javap -c WitherStringBuilder`, вы сможете просмотреть (упрощенный) код двух разных методов. Начнем с `implicit()`:

```
public java.lang.String implicit(java.lang.String[]);
Code:
  0:   ldc #2; //String
  2:   astore_2
  3:   iconst_0
  4:   istore_3
  5:   iload_3
  6:   aload_1
  7:   arraylength
  8:   if_icmpge 38
 11:   new #3; //class StringBuilder
 14:   dup
 15:   invokespecial #4; // StringBuilder."<init>":()
 18:   aload_2
 19:   invokevirtual #5; // StringBuilder.append:()
 22:   aload_1
 23:   iload_3
 24:   aaload
 25:   invokevirtual #5; // StringBuilder.append:()
 28:   invokevirtual #6; // StringBuilder.toString:()
 31:   astore_2
 32:   iinc 3, 1
 35:   goto 5
 38:   aload_2
 39:   areturn
```

Обратите внимание на строки 8: и 35:, образующие цикл. В строке 8: выполняется целочисленная операция сравнения «больше или равно» с операндами в стеке, а при завершении цикла происходит переход к строке 38:. Строка 35: содержит команду перехода к началу цикла 5:. Здесь важно заметить, что конструирование `StringBuilder` выполняется внутри цикла; это означает, что новый объект `StringBuilder` создается при каждом прохождении цикла.

А вот как выглядит байт-код `explicit()`:

```
public java.lang.String explicit(java.lang.String[]);
Code:
  0:   new #3; //class StringBuilder
  3:   dup
  4:   invokespecial #4; // StringBuilder."<init>":()
  7:   astore_2
  8:   iconst_0
  9:   istore_3
 10:   iload_3
 11:   aload_1
 12:   arraylength
 13:   if_icmpge 30
 16:   aload_2
 17:   aload_1
 18:   iload_3
 19:   aaload
 20:   invokevirtual #5; // StringBuilder.append:()
 23:   pop
 24:   iinc 3, 1
 27:   goto 10
 30:   aload_2
 31:   invokevirtual #6; // StringBuilder.toString:()
 34:   areturn
```

Не только код цикла стал короче и проще, но и метод теперь создает только один объект `StringBuilder`. Явное создание `StringBuilder` также позволяет заранее выделить объект нужного размера (если вы располагаете соответствующей информацией), чтобы избежать многократных повторных выделений памяти для буфера.

Таким образом, если при создании метода `toString()` выполняются простые операции, в которых компилятор может разобраться самостоятельно, обычно можно доверить построение результата компилятору. Но если в вычислениях задействован цикл, лучше явно использовать `StringBuilder` в `toString()`:

```
//: strings/UsingStringBuilder.java
import java.util.*;

public class UsingStringBuilder {
    public static Random rand = new Random(47);
    public String toString() {
        StringBuilder result = new StringBuilder("");
        for(int i = 0; i < 25; i++) {
            result.append(rand.nextInt(100));
            result.append(", ");
        }
        result.delete(result.length()-2, result.length());
        result.append("]");
        return result.toString();
    }
    public static void main(String[] args) {
        UsingStringBuilder usb = new UsingStringBuilder();
        System.out.println(usb);
    }
}

/* Output:
58, 55, 93, 61, 61, 29, 68, 0, 22, 7, 88, 28, 51, 89, 9,
78, 98, 61, 20, 58, 16, 40, 11, 22, 4]
*///:~
```

Обратите внимание на последовательное добавление частей результата вызовом `append()`. Если вы попытаетесь схитрить и используете вызов вида `append(a + ": " + c)`, то компилятор снова начнет создавать дополнительные объекты `StringBuilder`. Если у вас нет уверенности в том, какой способ следует выбрать, вы всегда можете запустить `javap` и проверить.

Хотя класс `StringBuilder` содержит полный набор методов, включая `insert()`, `replace()`, `substring()` и даже `reverse()`, чаще всего вы будете использовать методы `append()` и `toString()`. Обратите внимание на использование `delete()` для удаления последней запятой и пробела перед добавлением закрывающей квадратной скобки.

Класс `StringBuilder` появился в Java SE5. До этого использовался класс `StringBuffer`, который обеспечивал потоковую безопасность (см. главу 21), поэтому операции с ним были связаны с большими затратами. Соответственно, строковые операции в Java SE5/6 должны выполняться быстрее.

1. (2) Проанализируйте метод `SprinklerSystem.toString()` из примера `reusing/SprinklerSystem.java` и определите, избавит ли написание метода `toString()` с явным созданием `StringBuilder` от лишних операций создания `StringBuilder`.

Непреднамеренная рекурсия

Так как стандартные контейнеры Java (как и все остальные классы) в конечном итоге наследуют от `Object`, они содержат метод `toString()`. Этот метод был переопределен, чтобы контейнеры могли выдать свое представление в формате `String`, включающее данные о хранящихся в них объектах. Например, метод `ArrayList.toString()` перебирает элементы `ArrayList` и вызывает для каждого элемента `toString()`:

```
//: strings/ArrayListDisplay.java
import generics.coffee.*;
import java.util.*;

public class ArrayListDisplay {
    public static void main(String[] args) {
        ArrayList<Coffee> coffees = new ArrayList<Coffee>();
        for(Coffee c : new CoffeeGenerator(10))
            coffees.add(c);
        System.out.println(coffees);
    }
} /* Output:
[Americano 0, Latte 1, Americano 2, Mocha 3, Mocha 4, Breve
5, Americano 6, Latte 7, Cappuccino 8, Cappuccino 9]
*///:~
```

Допустим, вы хотите, чтобы метод `toString()` выводил адрес объекта вашего класса. Казалось бы, для этого достаточно использовать ссылку `this`:

```
//: strings/InfiniteRecursion.java
// Accidental recursion.
// {RunByHand}
import java.util.*;
```

```

public class InfiniteRecursion {
    public String toString() {
        return " InfiniteRecursion address: " + this + "\n";
    }
    public static void main(String[] args) {
        List<InfiniteRecursion> v =
            new ArrayList<InfiniteRecursion>();
        for(int i = 0; i < 10; i++)
            v.add(new InfiniteRecursion());
        System.out.println(v);
    }
}
: ///:~

```

Но при попытке создать объект `InfiniteRecursion` и вывести его вы получите очень длинную последовательность исключений. То же произойдет, если вы поместите объекты `InfiniteRecursion` в `ArrayList` и выведете `ArrayList` так, как показано ниже. Проблемы возникают из-за автоматического преобразования типов для `String`. Обратите внимание на выражение:

```
"InfiniteRecursion address: " + this
```

Компилятор видит объект `String`, за которым следует `+` и нечто, не являющееся строкой; соответственно, он пытается преобразовать `this` в `String`. Для этого он вызывает метод `toString()`, порождающий рекурсивный вызов.

Если вам действительно потребуется вывести адрес объекта, задача решается вызовом метода `toString()` класса `Object`, который делает именно это. Таким образом, вместо `this` следует использовать выражение `super.toString()`.

2. (1) Исправьте ошибку в `InfiniteRecursion.java`.

Операции со строками

В таблице представлены некоторые основные методы объектов `String`. Перегруженные методы объединены в одну строку таблицы.

Метод	Аргументы, перегрузка	Использование
Конструктор	Перегруженные версии: по умолчанию, <code>String</code> , <code>StringBuilder</code> , <code>StringBuffer</code> , массивы <code>char</code> , массивы <code>byte</code>	Создание объектов <code>String</code>
<code>length()</code>		Количество символов в <code>String</code>
<code>charAt()</code>	Индекс типа <code>int</code>	Символ (<code>char</code>) в заданной позиции строки
<code>getChars()</code> , <code>getBytes()</code>	Начало и конец копируемого участка, приемный массив, индекс в приемном массиве	Копирование блоков <code>char</code> и <code>byte</code> во внешний массив
<code>toCharArray()</code>		Создает массив <code>char[]</code> , содержащий все символы <code>String</code>
<code>equals()</code> , <code>equalsIgnoreCase()</code>	Объект <code>String</code> для сравнения	Проверка равенства содержимого двух объектов <code>String</code>

продолжение ➤

Метод	Аргументы, перегрузка	Использование
<code>compareTo()</code>	Объект <code>String</code> для сравнения	Отрицательное число, нуль или положительное число в зависимости от лексикографического порядка <code>String</code> и аргумента. Символы верхнего и нижнего регистра не равны!
<code>contains()</code>	Искомая последовательность <code>CharSequence</code>	Результат равен <code>true</code> , если аргумент содержится в <code>String</code>
<code>contentEquals()</code>	<code>CharSequence</code> или <code>StringBuffer</code> для сравнения	Результат равен <code>true</code> при точном совпадении с аргументом
<code>equalsIgnoreCase()</code>	Объект <code>String</code> для сравнения	Результат равен <code>true</code> при равенстве содержимого без учета регистра символов
<code>regionMatches()</code>	Смещение в текущем объекте <code>String</code> , другой объект <code>String</code> , смещение и длина сравниваемого участка. Перегруженная версия добавляет признак игнорирования регистра	Результат типа <code>boolean</code> указывает, совпадают ли участки
<code>startsWith()</code>	Объект <code>String</code> , который может быть префиксом текущего объекта <code>String</code>	Результат типа <code>boolean</code> указывает, начинается ли объект <code>String</code> с аргумента
<code>endsWith()</code>	Объект <code>String</code> , который может быть суффиксом текущего объекта <code>String</code>	Результат типа <code>boolean</code> указывает, является ли аргумент суффиксом
<code>indexOf()</code> , <code>lastIndexOf()</code>	Перегруженные версии: <code>char</code> , <code>char</code> и начальный индекс, <code>String</code> , <code>String</code> и начальный индекс	Возвращает <code>-1</code> , если аргумент не найден в текущем объекте <code>String</code> ; в противном случае возвращает индекс, по которому начинается аргумент. Метод <code>lastIndexOf()</code> осуществляет поиск в обратном направлении от конца строки
<code>substring()</code> (также <code>subSequence()</code>)	Перегруженные версии: начальный индекс; начальный и конечный индексы	Возвращает новый объект <code>String</code> с заданным набором символов
<code>concat()</code>	Объект <code>String</code> для конкатенации	Возвращает новый объект <code>String</code> , состоящий из символов исходного объекта <code>String</code> , за которыми следуют символы аргумента
<code>replace()</code>	Искомый символ и новый символ, заменяющий его. Также возможна замена <code>CharSequence</code> на <code>CharSequence</code>	Возвращает новый объект <code>String</code> с внесенными изменениями. Если совпадения не найдены, используется старый объект <code>String</code>
<code>toLowerCase()</code> , <code>toUpperCase()</code>		Возвращает новый объект <code>String</code> , полученный изменением регистра символов. Если изменения отсутствуют, используется старый объект <code>String</code>
<code>trim()</code>		Возвращает новый объект <code>String</code> , в котором с обоих концов удалены пропуски. Если изменения отсутствуют, используется старый объект <code>String</code>

Метод	Аргументы, перегрузка	Использование
valueOf()	Перегруженные версии: Object, char[], char[] со смещением и количеством, boolean, char, int, long, float, double	Возвращает объект String с символьным представлением аргумента
intern()		Производит одну и только одну ссылку на String для каждой уникальной последовательности символов

Как видите, когда появляется необходимость в изменении содержимого, методы String возвращают новый объект String. Если же содержимое не нуждается в изменении, метод просто возвращает ссылку на исходный объект String для экономии памяти и вычислительных ресурсов.

Методы String с использованием *регулярных выражений* будут рассмотрены позднее в этой главе.

Форматирование вывода

Одним из долгожданных нововведений, появившихся в Java SE5, стал механизм форматирования вывода в стиле команды printf() языка C. Он не только упрощает код вывода, но и предоставляет разработчикам Java мощные средства управления форматированием и выравниванием¹.

printf()

Метод printf() языка C не «собирает» строки так, как это делается в Java; он получает одну *форматную строку* и вставляет в нее значения, форматируя их при подстановке. Вместо того чтобы использовать для конкатенации текста в кавычках и переменных перегруженный оператор + (который не перегружается в языке C), printf() использует специальные служебные комбинации для обозначения позиции данных. Аргументы, вставляемые в форматную строку, перечисляются в виде списка, разделенного запятыми.

Например:

```
printf("Строка 1: [%d %f]\n", x, y);
```

Во время выполнения значение x подставляется на место %d, а значение y подставляется на место %f. Эти заполнители называются *форматными спецификаторами*, и кроме обозначения позиции подстановки, они также указывают, какого рода переменная должна вставляться и как она должна форматироваться. Например, спецификатор %d в приведенном примере указывает, что x является целочисленным значением, а %f — что у является вещественным значением (float или double).

¹ Марк Уэлш помог мне написать этот раздел, а также раздел «Сканирование ввода».

System.out.format()

В Java SE5 появился новый метод `format()`, доступный в объектах `PrintStream` и `PrintWriter` (эти объекты более подробно рассматриваются в главе 18), к которым также относится `System.out`. Метод `format()` создан по образцу `printf()` языка C. Также существует вспомогательный метод `printf()`, который просто вызывает `format()`; используйте его, если это имя кажется вам более привычным. Простой пример:

```
//: strings/SimpleFormat.java
```

```
public class SimpleFormat {
    public static void main(String[] args) {
        int x = 5;
        double y = 5.332542;
        // Старый способ:
        System.out.println("Row 1: [" + x + " " + y + "]");
        // Новый способ:
        System.out.format("Row 1: [%d %f]\n", x, y);
        // или
        System.out.printf("Row 1: [%d %f]\n", x, y);
    }
} /* Output:
Строка 1: [5 5.332542]
Строка 1: [5 5.332542]
Строка 1: [5 5.332542]
*///:~
```

Как видите, вызовы `format()` и `printf()` эквивалентны. В обоих случаях передается одна форматная строка, за которой перечисляются аргументы — по одному для каждого форматного спецификатора.

Класс Formatter

Вся новая функциональность форматирования обеспечивается классом `Formatter` из пакета `java.util`. Класс `Formatter` можно рассматривать как преобразователь, приводящий форматную строку и данные к нужному результату. При создании объекта `Formatter` вы сообщаете ему, куда следует выдать результат, передавая эту информацию конструктору:

```
//: strings/Turtle.java
```

```
import java.io.*;
import java.util.*;

public class Turtle {
    private String name;
    private Formatter f;
    public Turtle(String name, Formatter f) {
        this.name = name;
        this.f = f;
    }
    public void move(int x, int y) {
        f.format("%s The Turtle is at (%d,%d)\n", name, x, y);
    }
    public static void main(String[] args) {
```



```

    PrintStream outAlias = System.out;
    Turtle tommy = new Turtle("Tommy",
        new Formatter(System.out));
    Turtle terry = new Turtle("Terry",
        new Formatter(outAlias));
    tommy.move(0,0);
    terry.move(4,8);
    tommy.move(3,4);
    terry.move(2,5);
    tommy.move(3,3);
    terry.move(3,3);
}
} /* Output:
Tommy The Turtle is at (0,0)
Terry The Turtle is at (4,8)
Tommy The Turtle is at (3,4)
Terry The Turtle is at (2,5)
Tommy The Turtle is at (3,3)
Terry The Turtle is at (3,3)
*///:~

```

Весь вывод, относящийся к объекту `tommy`, направляется в `System.out`, а весь вывод для `terry` отправляется в синоним `System.out`. Перегруженные версии конструктора получают разные варианты выходных потоков, но наиболее полезными являются `PrintStream` (как в приведенном примере), `OutputStream` и `File`. Они более подробно рассматриваются в главе 18.

3. (1) Измените пример `Turtle.java` так, чтобы весь вывод направлялся в поток `System.err`.

В предыдущем примере используется новый форматный спецификатор `%s`, обозначающий аргумент `String`. Это простейший форматный спецификатор, который определяет только тип преобразования.

Форматные спецификаторы

Для управления интервалами и выравниванием вставляемых данных потребуются более сложные форматные спецификаторы. Общий синтаксис выглядит так:

%[аргумент_индекс\$][флаги][ширина][.точность]преобразование

Спецификатор *ширина* управляет минимальным размером поля. Объект `Formatter` гарантирует, что выходное поле занимает не менее указанного количества символов; при необходимости данные дополняются пробелами. По умолчанию данные выравниваются по правому краю, но тип выравнивания можно переопределить, включив символ - в секцию *флаги*.

В отличие от ширины поле *точность* задает максимальное значение. И если *ширина* относится ко всем типам преобразований данных и работает одинаково для всех типов, *точность* имеет разный смысл для разных типов. Для объектов `String` *точность* задает максимальное количество выводимых символов. Для вещественных чисел *точность* задает количество выводимых знаков (шесть по умолчанию), с округлением или добавлением завершающих нулей в случае необходимости. Так как целые числа не имеют дробной части, *точность* на них не распространяется, и при попытке применения этого спецификатора с целочисленным преобразованием будет возбуждено исключение.

В следующем примере форматные спецификаторы используются для вывода кассового чека:

```
//: strings/Receipt.java
import java.util.*;

public class Receipt {
    private double total = 0;
    private Formatter f = new Formatter(System.out);
    public void printTitle() {
        f.format("%-15s %5s %10s\n", "Item", "Qty", "Price");
        f.format("%-15s %5s %10s\n", "----", "---", "-----");
    }

    public void print(String name, int qty, double price) {
        f.format("%-15.15s %5d %10.2f\n", name, qty, price);
        total += price;
    }

    public void printTotal() {
        f.format("%-15s %5s %10.2f\n", "Tax", "", total*0.06);
        f.format("%-15s %5s %10s\n", "", "", "-----");
        f.format("%-15s %5s %10.2f\n", "Total", "",
            total * 1.06);
    }

    public static void main(String[] args) {
        Receipt receipt = new Receipt();
        receipt.printTitle();
        receipt.print("Jack's Magic Beans", 4, 4.25);
        receipt.print("Princess Peas", 3, 5.1);
        receipt.print("Three Bears Porridge", 1, 14.29);
        receipt.printTotal();
    }
} /* Output:
```

```
Item           Qty      Price
----          ---      -
Jack's Magic Be    4       4.25
Princess Peas      3       5.10
Three Bears Por    1      14.29
Tax                1.42
-----
Total                25.06
*///:~
```

Как видите, класс `Formatter` предоставляет мощные средства управления интервалами и выравниванием при достаточно компактной записи. В данном случае форматные строки просто копируются для получения необходимых интервалов.

4. (3) Измените пример `Receipt.java` так, чтобы все ширины управлялись одним набором констант. Сделайте так, чтобы ширину вывода можно было изменить, модифицируя одно значение в одном месте.

Преобразования Formatter

Чаще всего на практике используются следующие преобразования

Символы преобразования	
d	Целое число (десятичное)
c	Символ Юникода
b	Логическое значение
s	Строка
f	Вещественное число (в десятичной записи)
e	Вещественное число (в экспоненциальной записи)
x	Целое число (шестнадцатеричное)
h	Хеш-код (в шестнадцатеричной записи)
%	Литерал «%»

Следующий пример демонстрирует использование этих преобразований:

```
//: strings/Conversion.java
import java.math.*;
import java.util.*;

public class Conversion {
    public static void main(String[] args) {
        Formatter f = new Formatter(System.out);

        char u = 'a';
        System.out.println("u = 'a'");
        f.format("s: %s\n", u);
        // f.format("d: %d\n", u);
        f.format("c: %c\n", u);
        f.format("b: %b\n", u);
        // f.format("f: %f\n", u);
        // f.format("e: %e\n", u);
        // f.format("x: %x\n", u);
        f.format("h: %h\n", u);

        int v = 121;
        System.out.println("v = 121");
        f.format("d: %d\n", v);
        f.format("c: %c\n", v);
        f.format("b: %b\n", v);
        f.format("s: %s\n", v);
        // f.format("f: %f\n", v);
        // f.format("e: %e\n", v);
        f.format("x: %x\n", v);
        f.format("h: %h\n", v);

        BigInteger w = new BigInteger("50000000000000");
        System.out.println(
            "w = new BigInteger(\"50000000000000\")");
        f.format("d: %d\n", w);
        // f.format("c: %c\n", w);
        f.format("b: %b\n", w);
        f.format("s: %s\n", w);
    }
}
```

продолжение ➤

```

// f.format("f: %f\n", w);
// f.format("e: %e\n", w);
f.format("x: %x\n", w);
f.format("h: %h\n", w);

double x = 179.543;
System.out.println("x = 179.543");
// f.format("d: %d\n", x);
// f.format("c: %c\n", x);
f.format("b: %b\n", x);
f.format("s: %s\n", x);
f.format("f: %f\n", x);
f.format("e: %e\n", x);
// f.format("x: %x\n", x);
f.format("h: %h\n", x);

Conversion y = new Conversion();
System.out.println("y = new Conversion()");
// f.format("d: %d\n", y);
// f.format("c: %c\n", y);
f.format("b: %b\n", y);
f.format("s: %s\n", y);
// f.format("f: %f\n", y);
// f.format("e: %e\n", y);
// f.format("x: %x\n", y);
f.format("h: %h\n", y);

boolean z = false;
System.out.println("z = false");
// f.format("d: %d\n", z);
// f.format("c: %c\n", z);
f.format("b: %b\n", z);
f.format("s: %s\n", z);
// f.format("f: %f\n", z);
// f.format("e: %e\n", z);
// f.format("x: %x\n", z);
f.format("h: %h\n", z);
}
} /* Output: (Sample)
u = 'a'
s: a
c: a
b: true
h: 61
v = 121
d: 121
c: y
b: true
s: 121
x: 79
h: 79
w = new BigInteger("5000000000000000")
d: 5000000000000000
b: true
s: 5000000000000000
x: 2d79883d2000
h: 8842a1a7
x = 179.543

```

```

b: true
s: 179.543
f: 179.543000
e: 1.795430e+02
h: 1ef462c
y = new Conversion()
b: true
s: Conversion@9cab16
h: 9cab16
z = false
b: false
s: false
h: 4d5
*///:~

```

В закомментированных строках приведены преобразования, недействительные для конкретного типа переменной; попытка их выполнения приводит к возбуждению исключения.

Обратите внимание: преобразование `b` работает для всех переменных. Хотя оно действительно для всех типов аргументов, оно может работать не так, как вы ожидаете. Для примитивов `boolean` или объектов `Boolean` результат будет равен `true` или `false` в зависимости от значения, но для любого другого аргумента, отличного от `null`, результат всегда равен `true`. Даже для числового значения `0`, которое является синонимом `false` во многих языках (включая C), будет получено значение `true`; будьте осторожны при использовании этого преобразования с типами, отличными от `boolean`.

Также существуют другие типы преобразований и другие параметры форматного спецификатора. О них можно прочитать в описании класса `Formatter` из документации JDK.

5. (5) Для каждого базового типа преобразования в приведенной таблице напишите самое сложное из возможных выражений форматирования. Другими словами, используйте все возможные форматные спецификаторы, доступные для этого типа преобразования.

String.format()

Проектировщики Java SE5 также создали аналог функции `sprintf()` языка C, предназначенной для создания строк. Статический метод `String.format()` получает те же аргументы, что и метод `format()` класса `Formatter`, но возвращает `String`. Он может пригодиться в ситуации, в которой `format()` нужно вызвать всего один раз:

```

//: strings/DatabaseException.java

public class DatabaseException extends Exception {
    public DatabaseException(int transactionID, int queryID,
        String message) {
        super(String.format("(t%d, q%d) %s", transactionID,
            queryID, message));
    }
    public static void main(String[] args) {
        try {
            throw new DatabaseException(3, 7, "Ошибка записи");
        } catch (Exception e) {

```

продолжение ➤

```

        System.out.println(e);
    }
}
} /* Output:
DatabaseException: (t3, q7) Ошибка записи
*///:~

```

Во внутренней реализации `String.format()` всего лишь создает экземпляр `Formatter` и передает ему аргументы, но вызвать этот вспомогательный метод часто оказывается проще и удобнее, чем делать все «вручную».

Вывод файла в шестнадцатеричном виде

Второй пример: часто возникает необходимость просмотра содержимого двоичного файла в шестнадцатеричном формате. Следующая программа выводит двоичный массив байтов в удобочитаемом шестнадцатеричном формате с использованием метода `String.format()`:

```

//: net/mindview/util/Hex.java
package net.mindview.util;
import java.io.*;

public class Hex {
    public static String format(byte[] data) {
        StringBuilder result = new StringBuilder();
        int n = 0;
        for(byte b : data) {
            if(n % 16 == 0)
                result.append(String.format("%05X: ", n));
            result.append(String.format("%02X ", b));
            n++;
            if(n % 16 == 0) result.append("\n");
        }
        result.append("\n");
        return result.toString();
    }

    public static void main(String[] args) throws Exception {
        if(args.length == 0)
            // Тестирование на примере файла класса:
            System.out.println(
                format(BinaryFile.read("Hex.class")));
        else
            System.out.println(
                format(BinaryFile.read(new File(args[0]))));
    }
} /* Output: (Sample)
00000: CA FE BA BE 00 00 00 31 00 52 0A 00 05 00 22 07
00010: 00 23 0A 00 02 00 22 08 00 24 07 00 25 0A 00 26
00020: 00 27 0A 00 28 00 29 0A 00 02 00 2A 08 00 2B 0A
00030: 00 2C 00 2D 08 00 2E 0A 00 02 00 2F 09 00 30 00
00040: 31 08 00 32 0A 00 33 00 34 0A 00 15 00 35 0A 00
00050: 36 00 37 07 00 38 0A 00 12 00 39 0A 00 33 00 3A
...
*///:~

```

Для открытия и чтения двоичного файла используется другая программа, которая будет представлена в главе 18: `net.mindview.util.BinaryFile`. Метод `read()` возвращает все содержимое файла в виде массива байтов.

6. (2) Создайте класс с полями `int`, `long`, `float` и `double`. Создайте для этого класса метод `toString()`, использующий `String.format()`, и продемонстрируйте, что ваш класс работает правильно.

Регулярные выражения

Регулярные выражения давно поддерживаются стандартными утилитами Unix (такими, как `sed` и `awk`), а также языками Python и Perl (некоторые разработчики считают, что именно регулярные выражения стали основной причиной успеха Perl). Ранее основные средства работы со строками были реализованы в Java в классах `String`, `StringBuffer` и `StringTokenizer`, которые обладают относительно простыми возможностями по сравнению с регулярными выражениями.

Регулярные выражения — мощный и гибкий инструмент обработки текстов. Они позволяют определять на программном уровне сложные шаблоны для поиска текста во входной строке. Обнаружив совпадение для шаблона, вы можете обработать его так, как считаете нужным. Синтаксис регулярных выражений поначалу выглядит устрашающе, но это компактный и динамичный язык, который может использоваться для решения самых разнообразных задач обработки строк, поиска и выделения совпадений, редактирования и проверки.

ОСНОВЫ

Регулярные выражения предназначены для обобщенного описания строк по принципу: «Если строка содержит такие-то элементы, то в ней находится совпадение для искомого критерия». Например, чтобы указать, что перед числом может стоять знак «-» (но его может и не быть), вы включаете в условие поиска знак «-», за которым следует вопросительный знак:

-?

Целое число описывается как последовательность из одной или нескольких цифр. В регулярных выражениях цифра обозначается комбинацией `\d`. Если у вас имеется опыт работы с регулярными выражениями в других языках, вы сразу заметите различия в работе с символами обратной косой черты (`\`). В других языках последовательность `\\` означает: «Я хочу вставить в регулярное выражение обычный (литеральный) символ обратной косой черты. У этого символа нет никакого специального значения». В Java `\\` означает: «Здесь вставляется обратная косая черта регулярного выражения, так что следующий символ имеет специальное значение». Например, строка регулярного выражения для обозначения цифры имеет вид `\\d`. Для вставки литерального символа обратной косой черты используется последовательность `\\\\`. Однако в обозначениях новой строки и табуляции используется только одна косая черта: `\n` и `\t`.

Для описания «одного или нескольких повторений предшествующего выражения» используется символ `+`. Таким образом, выражение «необязательный знак „минус“, за которым следует одна или несколько цифр» записывается следующим образом:

```
-?\d+
```

Простейший вариант использования регулярных выражений основан на функциональности класса `String`. Например, можно проверить, имеется ли в объекте `String` совпадение для приведенного выше регулярного выражения:

```
//: strings/IntegerMatch.java
```

```
public class IntegerMatch {
    public static void main(String[] args) {
        System.out.println("-1234".matches("-?\d+"));
        System.out.println("5678".matches("-?\d+"));
        System.out.println("+911".matches("-?\d+"));
        System.out.println("+911".matches("( -|\\+)?\d+"));
    }
} /* Output:
true
true
false
true
*///:~
```

В первых двух строках обнаруживаются совпадения, но третье число, начинающееся с `+`, является допустимым числом, но не совпадает с регулярным выражением. Таким образом, нужно включить условие «может начинаться с `+` или `-`». В регулярных выражениях круглые скобки используются для группировки, а вертикальная черта `|` означает ИЛИ. Итак, выражение

```
(-|\\+)?
```

означает, что эта часть строки может содержать `-`, `+` или ничего (из-за `?`). Так как символ `+` имеет особый смысл в регулярных выражениях, необходимо *экранировать* его символами `\\`, чтобы он воспринимался как обычный символ в выражении.

Еще один полезный инструмент, встроенный в `String` — метод `split()`, — разбивает строку по совпадениям заданного регулярного выражения.

```
//: strings/Splitting.java
import java.util.*;
```

```
public class Splitting {
    public static String knights =
        "Then, when you have found the shrubbery, you must " +
        "cut down the mightiest tree in the forest... " +
        "with... a herring!";
    public static void split(String regex) {
        System.out.println(
            Arrays.toString(knights.split(regex)));
    }
    public static void main(String[] args) {
        split(" "); // Выражение может не содержать специальных символов
        split("\\W+"); // Разбиение по символам, не являющимся символами слов
    }
}
```



```

split("n\\W+"); // Буква 'n', за которой следуют символы,
                // не являющиеся символами слов
}
} /* Output:
[Then,, when, you, have, found, the, shrubbery,, you, must,
cut, down, the, mightiest, tree, in, the, forest...,
with..., a, herring!]
[Then, when, you, have, found, the, shrubbery, you, must,
cut, down, the, mightiest, tree, in, the, forest, with, a,
herring]
[The, whe, you have found the shrubbery, you must cut dow,
the mightiest tree i, the forest... with... a herring!]
*///:~

```

Прежде всего обратите внимание на то, что в регулярных выражениях могут использоваться обычные символы — присутствие специальных символов не обязательно, как видно из первого вызова `split()`, в котором строка разбивается по пробелам.

Во втором и третьем вызовах `split()` используется `\w`, обозначающая символ, не являющийся символом слова (версия в нижнем регистре `\w` обозначает символ слова), — вы видите, что во втором случае знаки препинания были удалены. Третий вызов `split()` означает: «Буква `n`, за которой следует один или несколько символов, не являющихся символами слов». Как видно из листинга, шаблоны разбивки в результате не отображаются.

Перегруженная версия `String.split()` позволяет ограничить количество разбиений.

Последний инструмент `String`, связанный с регулярными выражениями, — замена. Операция может заменять как первое совпадение, так и все совпадения:

```

//: strings/Replacing.java
import static net.mindview.util.Print.*;

public class Replacing {
    static String s = Splitting.knights;
    public static void main(String[] args) {
        print(s.replaceFirst("f\\w+", "located"));
        print(s.replaceAll("shrubbery|tree|herring", "banana"));
    }
} /* Output:
Then, when you have located the shrubbery, you must cut
down the mightiest tree in the forest... with... a herring!
Then, when you have found the banana, you must cut down the
mightiest banana in the forest... with... a banana!
*///:~

```

Первое выражение совпадает с буквой `f`, за которой следует один или несколько символов слов (обратите внимание: на этот раз используется строчная буква `w`). Операция заменяет *только первое* найденное совпадение, поэтому слово «found» заменяется словом «located».

Второе выражение совпадает с любым из трех слов, разделенных символами `|`, при этом заменяются *все* найденные совпадения.

Как видите, регулярные выражения, не привязанные к `String`, обладают более мощными средствами замены — например, вы можете вызывать методы для выполнения замены. Регулярные выражения, не привязанные к `String`, также работают намного эффективнее, если регулярное выражение должно использоваться многократно.

7. (5) Взяв за основу документацию `java.util.regex.Pattern`, напишите и протестируйте регулярное выражение, которое проверяет, что предложение начинается с прописной буквы и завершается точкой.
8. (2) Разбейте строку `Splitting.knights` по словам «the» или «you».
9. (3) Взяв за основу документацию `java.util.regex.Pattern`, замените все гласные в `Splitting.knights` подчеркиваниями.

Создание регулярных выражений

Изучение регулярных выражений можно начать с подмножества возможных конструкций. Полный список конструкций, используемых при построении регулярных выражений, можно найти в описании класса `Pattern` из пакета `java.util.regex` в документации JDK.

Символы	
<code>\b</code>	Символ <code>B</code>
<code>\xhh</code>	Символ с шестнадцатеричным кодом <code>0xhh</code>
<code>\uhhhh</code>	Символ Юникода с шестнадцатеричным представлением <code>0xhhhh</code>
<code>\t</code>	Табуляция
<code>\n</code>	Новая строка
<code>\r</code>	Возврат курсора
<code>\f</code>	Подача страницы
<code>\e</code>	Escape

Мощь регулярных выражений начинает проявляться при определении символьных классов. Несколько типичных способов создания символьных классов, а также некоторые заранее определенные классы:

Символьные классы	
<code>.</code>	Любой символ
<code>[abc]</code>	Любой из символов <code>a</code> , <code>b</code> и <code>c</code> (то же, что <code>a b c</code>)
<code>[^abc]</code>	Любой символ, кроме <code>a</code> , <code>b</code> и <code>c</code> (отрицание)
<code>[a-zA-Z]</code>	Любой символ от <code>a</code> до <code>z</code> и от <code>A</code> до <code>Z</code> (диапазон)
<code>[abc[hij]]</code>	Любой из символов <code>a</code> , <code>b</code> , <code>c</code> , <code>h</code> , <code>i</code> , <code>j</code> (то же, что <code>a b c h i j</code>) (объединение)
<code>[a-z&&[hij]]</code>	Символ <code>h</code> , <code>i</code> или <code>j</code> (пересечение)
<code>\s</code>	Пропуск (пробел, табуляция, новая строка, подача страницы, возврат курсора)
<code>\S</code>	Символ, не являющийся пропуском (<code>[^\s]</code>)
<code>\d</code>	Цифра <code>[0-9]</code>
<code>\D</code>	Не цифра <code>[^0-9]</code>
<code>\w</code>	Символ слова <code>[a-zA-Z_0-9]</code>
<code>\W</code>	Символ, не являющийся символом слова <code>[^\w]</code>

Здесь приведена лишь небольшая подборка; за полным списком всех шаблонов регулярных выражений обращайтесь к странице `java.util.regex.Pattern` в документации JDK.

Логические операторы	
<code>XY</code>	X, за которым следует Y
<code>X Y</code>	X или Y
<code>(X)</code>	Сохраняющая группировка. Позднее в выражении к i-й сохраненной группе можно обратиться при помощи записи <code>\i</code>
Привязка к границам	
<code>^</code>	Начало строки
<code>\$</code>	Конец строки
<code>\b</code>	Граница слова
<code>\B</code>	Не граница слова
<code>\G</code>	Конец предыдущего совпадения

Например, каждое из регулярных выражений в следующем примере успешно совпадает с последовательностью символов «Rudolph»:

```
//: strings/Rudolph.java
```

```
public class Rudolph {
    public static void main(String[] args) {
        for(String pattern : new String[]{ "Rudolph",
            "[rR]udolph", "[rR][aeiou][a-z]ol.*", "R.*" })
            System.out.println("Rudolph".matches(pattern));
    }
} /* Output:
true
true
true
true
*///:~
```

Разумеется, вы должны стремиться написать не самое запутанное, а самое простое регулярное выражение, решающее задачу. При написании новых регулярных выражений вы часто будете брать за образец готовые выражения из своего кода.

Квантификаторы

Квантификатор описывает режим «поглощения» входного текста шаблоном:

- ❑ **Максимальные** квантификаторы используются по умолчанию. В максимальном режиме для выражения подбирается максимально возможное количество возможных совпадений. Одна из типичных ошибок — полагать, что шаблон совпадет только с первой возможной группой символов, тогда как в действительности механизм регулярных выражений продолжает двигаться вперед, пока не подберет возможное совпадение максимальной длины.
- ❑ **Минимальный** квантификатор (задается вопросительным знаком) старается ограничиться минимальным количеством символов, необходимых для соответствия шаблону.

- ❑ *Захватывающие* квантификаторы поддерживаются только в Java. Они сложнее других квантификаторов, поэтому, скорее всего, на первых порах вы не будете их использовать. При применении регулярного выражения к строке генерируются множественные состояния для возврата в случае неудачи при поиске. Захватывающие квантификаторы не поддерживают эти промежуточные состояния, что предотвращает возврат и может способствовать повышению эффективности.

Максимальный	Минимальный	Захватывающий	Совпадает
X?	X??	X?+	X, один или ни одного
X*	X*?	X*+	X, нуль и более
X+	X+?	X++	X, один и более
X{n}	X{n}?	X{n}+	X, ровно n раз
X{n,}	X{n,}?	X{n,}+	X, не менее n раз
X{n,m}	X{n,m}?	X{n,m}+	X, не менее n, но не более m раз

Помните, что выражение X часто приходится заключать в круглые скобки, чтобы оно работало так, как нужно. Для примера возьмем следующее выражение:

`abc+`

Может показаться, что оно совпадает с последовательностью символов «abc» один и более раз, а при применении его к входной строке «abcabcabc» вы получите три совпадения. Однако это выражение в действительности означает: «Найти совпадение для последовательности «ab», за которой следует один или несколько вхождений «с». Чтобы найти одно и более совпадений для всей строки «abc», следует использовать запись:

`(abc)+`

При использовании регулярных выражений легко ошибиться. Помните, что язык регулярных выражений является «надстройкой», работающей поверх Java.

CharSequence

Интерфейс `CharSequence` устанавливает обобщенное определение последовательности символов, выделенной в классе `CharBuffer`, `String`, `StringBuffer` или `StringBuilder`:

```
interface CharSequence {
    charAt(int i);
    length();
    subSequence(int start, int end);
    toString();
}
```

Перечисленные классы реализуют этот интерфейс. Многие операции регулярных выражений получают аргументы `CharSequence`.

Pattern и Matcher

В общем случае следует компилировать объекты регулярных выражений вместо того, чтобы использовать весьма ограниченные возможности `String`. Для этого импортируйте

`java.util.regex`, а затем откомпилируйте регулярное выражение статическим методом `Pattern.compile()`. В результате на базе аргумента `String` создается объект `Pattern`; чтобы использовать этот объект, вызовите его метод `matcher()` и передайте строку, в которой ведется поиск. Метод `matcher()` создает объект `Matcher` с набором операций (полный список которых представлен в описании `java.util.regex.Matcher` из документации JDK. Например, метод `replaceAll()` заменяет все совпадения своим аргументом.

В первом примере для тестирования регулярных выражений с входной строкой будет использоваться приведенный ниже класс. Первый аргумент командной строки содержит входную строку, в которой будет осуществляться поиск; за ней следует одно или несколько регулярных выражений, применяемых к входным данным. В Unix/Linux регулярные выражения в командной строке должны заключаться в кавычки. Программа может использоваться для тестирования регулярных выражений в процессе построения (чтобы разработчик мог проверить, соответствует ли поведение выражения его ожиданиям).

```
//: strings/TestRegularExpression.java
// Класс для простого тестирования регулярных выражений.
// {Args: abcabcabdefabc "abc+" "(abc)+" "(abc){2,}" }
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class TestRegularExpression {
    public static void main(String[] args) {
        if(args.length < 2) {
            print("Usage:\njava TestRegularExpression " +
                "characterSequence regularExpression+");
            System.exit(0);
        }
        print("Input: \"" + args[0] + "\"");
        for(String arg : args) {
            print("Regular expression: \"" + arg + "\"");
            Pattern p = Pattern.compile(arg);
            Matcher m = p.matcher(args[0]);
            while(m.find()) {
                print("Match \"" + m.group() + "\" at positions " +
                    m.start() + "-" + (m.end() - 1));
            }
        }
    }
}

/* Output:
Input: "abcabcabdefabc"
Regular expression: "abcabcabdefabc"
Match "abcabcabdefabc" at positions 0-14
Regular expression: "abc+"
Match "abc" at positions 0-2
Match "abc" at positions 3-5
Match "abc" at positions 6-8
Match "abc" at positions 12-14
Regular expression: "(abc)+"
Match "abcabcab" at positions 0-8
Match "abc" at positions 12-14
Regular expression: "(abc){2,}"
Match "abcabcab" at positions 0-8
*///:~
```

Объект `Pattern` представляет откомпилированную версию регулярного выражения. Как видно из приведенного примера, метод `matcher()` и входная строка используются для создания объекта `Matcher` на базе откомпилированного объекта `Pattern`. Класс `Pattern` также содержит статический метод:

```
static boolean matches(String regex, CharSequence input)
```

для проверки совпадения регулярного выражения `regex` со всей входной последовательностью `CharSequence`, и метод `split()`, который создает массив объектов `String`, полученных разбиением строки по совпадениям регулярного выражения.

Объект `Matcher` генерируется методом `Pattern.matcher()`, получающим в аргументе входную строку. Объект `Matcher` затем используется для обращения к результатам; для проверки успеха или неудачи при различных типах совпадений используются следующие методы:

```
boolean matches()
boolean lookingAt()
boolean find()
boolean find(int start)
```

Метод `matches()` выполняет успешную проверку, если шаблон совпадает со всей входной строкой, тогда как метод `lookingAt()` успешен, если совпадение находится в начале области.

- 10.** (2) Определите, будет ли найдено в строке «Java now has regular expressions» совпадение для следующих выражений:

```
^Java
\breg.*
n.w\s+h(a|i)s
s?
s*
s+
s{4}
s{1}.
s{0,3}
```

- 11.** (2) Примените регулярное выражение

```
(?i)((^[aeiou])|(\s+[aeiou]))\w+[aeiou]\b
```

к строке

```
"Arline ate eight apples and one orange while Anita hadn't any"
```

find()

Метод `Matcher.find()` может использоваться для поиска множественных совпадений шаблона в объекте `CharSequence`, к которому он применяется. Пример:

```
//: strings/Finding.java
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class Finding {
    public static void main(String[] args) {
        Matcher m = Pattern.compile("\\w+")
            .matcher("Evening is full of the linnet's wings");
```

```

while(m.find())
    printnb(m.group() + " ");
print();
int i = 0;
while(m.find(i)) {
    printnb(m.group() + " ");
    i++;
}
}
} /* Output:
Evening is full of the linnet s wings
Evening vening ening ning ing ng g is is s full full ull ll
l of of f the the he e linnet linnet innet nnet net et t s
s wings wings ings ngs gs s
*///:~

```

Шаблон `\w+` разбивает входные данные на слова. Метод `find()` действует как итератор, перемещаясь по входной строке. Второй версии `find()` может передаваться целочисленный аргумент с позицией символа, с которой должен начинаться поиск, — эта версия сбрасывает позицию поиска до значения аргумента, как видно из выходных данных.

Группы

Группы представляют собой части регулярного выражения, заключенные в круглые скобки, к которым позднее можно обращаться по номеру группы. Группа 0 соответствует совпадению всего выражения, группа 1 — совпадению первого подвыражения в круглых скобках и т. д. Таким образом, в выражении

`A(B(C))D`

задействованы три группы: группа 0 — `ABCD`, группа 1 — `BC` и группа 2 — `C`.

Объект `Matcher` содержит методы для получения информации о группах:

- ❑ `public int groupCount()` возвращает количество групп в шаблоне объекта `Matcher`. Группа 0 в это количество не включается.
- ❑ `public String group()` возвращает группу 0 (все совпадение) от предыдущей операции поиска совпадения (`find()`, например).
- ❑ `public String group(int i)` возвращает группу с заданным номером от предыдущей операции поиска совпадения. Если совпадение было найдено, но указанная группа не совпала ни с какой частью входной строки, возвращается `null`.
- ❑ `public int start(int group)` возвращает начальный индекс группы, найденной в предыдущей операции поиска совпадения.
- ❑ `public int end(int group)` возвращает индекс последнего символа группы, найденной в предыдущей операции поиска совпадения, увеличенный на 1.

Пример:

```

//: strings/Groups.java
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class Groups {
    static public final String POEM =
        "Twas brillig, and the slithy toves\n" +

```

продолжение ➤

```

"Did gyre and gimble in the wabe.\n" +
"All mimsy were the borogoves,\n" +
"And the mome raths outgrabe.\n\n" +
"Beware the Jabberwock, my son,\n" +
"The jaws that bite, the claws that catch.\n" +
"Beware the Jubjub bird, and shun\n" +
"The frumious Bandersnatch.";
public static void main(String[] args) {
    Matcher m =
        Pattern.compile("(?m)(\\S+)\\s+(\\S+)\\s+(\\S+))$")
            .matcher(POEM);
    while(m.find()) {
        for(int j = 0; j <= m.groupCount(); j++)
            printnb "[" + m.group(j) + " ");
        print();
    }
}
} /* Output:
[the slithy toves][the][slithy toves][slithy][toves]
[in the wabe.][in][the wabe.][the][wabe.]
[were the borogoves,][were][the
borogoves,][the][borogoves,]
[mome raths outgrabe.][mome][raths
outgrabe.][raths][outgrabe.]
[Jabberwock, my son,][Jabberwock,][my son,][my][son,]
[claws that catch.][claws][that catch.][that][catch.]
[bird, and shun][bird,][and shun][and][shun]
[The frumious Bandersnatch.][The][frumious
Bandersnatch.][frumious][Bandersnatch.]
*///:~

```

В качестве исходного текста используется первая часть стихотворения «Бармаглот» из книги Льюиса Кэрролла «Алиса в Зазеркалье». Шаблон регулярного выражения содержит несколько подвыражений, заключенных в круглые скобки; эти подвыражения состоят из произвольного количества символов, не являющихся пропусками (`\S+`), за которыми следует произвольное количество символов-пропусков (`\s+`). Группы предназначены для сохранения трех последних слов в каждой строке текста, конец которой обозначается знаком `$`. Однако обычно `$` совпадает только в конце всей входной последовательности, поэтому вы должны явно приказать регулярному выражению учитывать разрывы строк во входных данных. Задача решается при помощи флага шаблона `(?m)` в начале последовательности (флаги шаблонов вскоре будут рассмотрены более подробно).

- 12. (5)** Измените пример `Groups.java` так, чтобы в нем подсчитывались все уникальные слова, не начинающиеся с прописной буквы.

start() и end()

После успешного поиска совпадения метод `start()` возвращает начальный индекс предыдущего совпадения, а метод `end()` возвращает индекс последнего символа совпадения, увеличенный на 1. Вызов `start()` или `end()` после неуспешной операции поиска совпадения (или до ее начала) порождает исключение `IllegalStateException`. Следующая программа также демонстрирует применение методов `matches()` и `lookingAt()`¹:

¹ В листинге использована цитата из речи командора Таггарта из фильма «Galaxy Quest».


```

//: strings/StartEnd.java
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class StartEnd {
    public static String input =
        "As long as there is injustice, whenever a\n" +
        "Targathian baby cries out, wherever a distress\n" +
        "signal sounds among the stars ... We'll be there.\n" +
        "This fine ship, and this fine crew ...\n" +
        "Never give up! Never surrender!";
    private static class Display {
        private boolean regexPrinted = false;
        private String regex;
        Display(String regex) { this.regex = regex; }
        void display(String message) {
            if(!regexPrinted) {
                print(regex);
                regexPrinted = true;
            }
            print(message);
        }
    }
    static void examine(String s, String regex) {
        Display d = new Display(regex);
        Pattern p = Pattern.compile(regex);
        Matcher m = p.matcher(s);
        while(m.find())
            d.display("find() '" + m.group() +
                "' start = " + m.start() + " end = " + m.end());
        if(m.lookingAt()) // Вызов reset() не нужен
            d.display("lookingAt() start = "
                + m.start() + " end = " + m.end());
        if(m.matches()) // Вызов reset() не нужен
            d.display("matches() start = "
                + m.start() + " end = " + m.end());
    }
    public static void main(String[] args) {
        for(String in : input.split("\n")) {
            print("input : " + in);
            for(String regex : new String[]{"\\w*ere\\w*",
                "\\w*ever", "T\\w+", "Never.*?!"})
                examine(in, regex);
        }
    }
}
/* Output:
input : As long as there is injustice, whenever a
\\w*ere\\w*
find() 'there' start = 11 end = 16
\\w*ever
find() 'whenever' start = 31 end = 39
input : Targathian baby cries out, wherever a distress
\\w*ere\\w*
find() 'wherever' start = 27 end = 35
\\w*ever
find() 'wherever' start = 27 end = 35
T\\w+
find() 'Targathian' start = 0 end = 10

```

```

lookingAt() start = 0 end = 10
input : signal sounds among the stars ... We'll be there.
\w*ere\w*
find() 'there' start = 43 end = 48
input : This fine ship, and this fine crew ...
T\w+
find() 'This' start = 0 end = 4
lookingAt() start = 0 end = 4
input : Never give up! Never surrender!
\w*ever
find() 'Never' start = 0 end = 5
find() 'Never' start = 15 end = 20
lookingAt() start = 0 end = 5
Never.*?!
find() 'Never give up!' start = 0 end = 14
find() 'Never surrender!' start = 15 end = 31
lookingAt() start = 0 end = 14
matches() start = 0 end = 31
*///:~

```

Обратите внимание: `find()` находит совпадение регулярного выражения в любой позиции входных данных, а в случае `lookingAt()` и `matches()` совпадение обнаруживается успешно только в том случае, если совпадение начинается от самого начала входных данных. И если для `matches()` совпадение успешно только в том случае, если с регулярным выражением совпадают *все* входные данные, в случае `lookingAt()`¹ достаточно совпадения только начальной части входных данных.

- 13.** (2) Измените пример `StartEnd.java`, чтобы он использовал входные данные `Groups.POEM`, но при этом выдавал положительные результаты для `find()`, `lookingAt()` и `matches()`.

Флаги шаблонов

Альтернативный метод `compile()` получает флаги, управляющие процессом поиска совпадений:

```
Pattern Pattern.compile(String regex, int flag)
```

Параметр `flag` принимает значения из следующих констант класса `Pattern`.

Флаг <code>compile()</code>	Эффект
<code>Pattern.CANON_EQ</code>	Два символа считаются совпадающими в том (и только в том!) случае, если совпадают их полные канонические декомпозиции
<code>Pattern.CASE_INSENSITIVE</code> (<code>?</code> <i>i</i>)	По умолчанию режим поиска совпадения без учета регистра символов распространяется только на символы набора US-ASCII. Поиск без учета регистра символов с поддержкой Юникода включается указанием флага <code>UNICODE_CASE</code> вместе с этим флагом

¹ Понятия не имею, как создатели Java придумали это название и что оно должно означать. По крайней мере приятно знать, что в этом мире хоть что-то остается неизменным: люди, изобретающие невразумительные имена методов, продолжают работать в Sun, а политика отказа от рецензирования программных архитектур все еще действует. Простите за сарказм, но через несколько лет такие вещи начинают утомлять.

Флаг <code>compile()</code>	Эффект
<code>Pattern.COMMENTS</code> (?x)	В этом режиме пропуски игнорируются, а встроенные комментарии, начинающиеся с #, игнорируются до конца строки
<code>Pattern.DOTALL</code> (?s)	В этом режиме метасимвол «точка» (.) совпадает с любым символом, включая завершитель строки. По умолчанию точка не совпадает с завершителями строк
<code>Pattern.MULTILINE</code> (?m)	В этом режиме выражения ^ и \$ совпадают с началом и концом логических строк соответственно. ^ также совпадает с началом входной строки, а \$ — с концом входной строки. По умолчанию эти выражения совпадают только в начале и в конце всей входной строки
<code>Pattern.UNICODE_CASE</code> (?u)	Поиск совпадения без учета регистра символов, включаемый флагом <code>CASE_INSENSITIVE</code> , осуществляется способом, совместимым со стандартом Юникод. По умолчанию поиск без учета регистра символов распространяется только на символы из набора US-ASCII
<code>Pattern.UNIX_LINES</code> (?d)	В этом режиме в поведении метасимволов ., ^ и \$ распознается только завершитель строк \n

Из перечисленных флагов особенно полезны `Pattern.CASE_INSENSITIVE`, `Pattern.MULTILINE` и `Pattern.COMMENTS` (делает код более понятным и используется для документирования). Поведение большинства флагов также может быть реализовано вставкой символов в круглых скобках, приведенных в таблице под именами флагов, в регулярное выражение перед той позицией, в которой должен включиться заданный режим.

Эти и другие флаги также могут объединяться операцией | (ИЛИ):

```
//: strings/ReFlags.java
import java.util.regex.*;

public class ReFlags {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("^java",
            Pattern.CASE_INSENSITIVE | Pattern.MULTILINE);
        Matcher m = p.matcher(
            "java has regex\nJava has regex\n" +
            "JAVA has pretty good regular expressions\n" +
            "Regular expressions are in Java");
        while(m.find())
            System.out.println(m.group());
    }
} /* Output:
java
Java
JAVA
*///:~
```

Созданный шаблон совпадает в строках, начинающихся с префикса «java», «Java», «JAVA» и т. д., и пытается найти совпадение в каждой логической строке многострочного набора (совпадения начинаются в начале символьной последовательности и после каждого завершителя строки в последовательности). Обратите внимание: метод `group()` возвращает только совпавшую часть.

split()

Метод `split()` разбивает входную строку по совпадениям регулярного выражения и создает массив объектов `String`:

```
String[] split(CharSequence input)
String[] split(CharSequence input, int limit)
```

Это удобный способ разбиения входного текста по общему разделителю:

```
//: strings/SplitDemo.java
import java.util.regex.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class SplitDemo {
    public static void main(String[] args) {
        String input =
            "This!!unusual use!!of exclamation!!points";
        print(Arrays.toString(
            Pattern.compile("!!").split(input)));
        // Only do the first three:
        print(Arrays.toString(
            Pattern.compile("!!").split(input, 3)));
    }
} /* Output:
[This, unusual use, of exclamation, points]
[This, unusual use, of exclamation!!points]
*///:~
```

Вторая форма `split()` ограничивает количество выполняемых разбиений.

14. (1) Перепишите класс `SplitDemo` с использованием `String.split()`.

Операции замены

Регулярные выражения особенно полезны при замене текста. Для ее выполнения доступны следующие методы:

`replaceFirst(String replacement)` заменяет первое совпадение во входной строке строкой `replacement`.

`replaceAll(String replacement)` заменяет все совпадения во входной строке строкой `replacement`.

`appendReplacement(StringBuffer sbuf, String replacement)` выполняет пошаговые замены в `sbuf` (вместо замены первого совпадения или всех совпадений, как `replaceFirst()` и `replaceAll()` соответственно). Этот метод очень важен, поскольку он позволяет вызывать методы и выполнять другие операции для получения `replacement` (`replaceFirst()` и `replaceAll()` могут добавлять только фиксированные строки). С этим методом вы можете разбивать группы на программном уровне и строить мощные заменяющие конструкции.

`appendTail(StringBuffer sbuf, String replacement)` вызывается после одного или нескольких вызовов метода `appendReplacement()` для копирования остатка входной строки.

Следующий пример демонстрирует применение всех операций замены. Блок закомментированного текста в начале программы извлекается и обрабатывается с применением регулярных выражений, после чего результат образует входные данные для оставшейся части примера:

```
//: strings/TheReplacements.java
import java.util.regex.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

/*! Here's a block of text to use as input to
the regular expression matcher. Note that we'll
first extract the block of text by looking for
the special delimiters, then process the
extracted block. !*/

public class TheReplacements {
    public static void main(String[] args) throws Exception {
        String s = TextFile.read("TheReplacements.java");
        // Поиск блока текста, заключенного в специальные комментарии:
        Matcher mInput =
            Pattern.compile("/\\*!(.*)!\\*/", Pattern.DOTALL)
                .matcher(s);
        if(mInput.find())
            s = mInput.group(1); // Совпадение подвыражения в круглых скобках
        // Два и более пробела заменяются одним пробелом:
        s = s.replaceAll(" {2,}", " ");
        // Один и более пробелов в начале строки заменяются пустой строкой.
        // Для выполнения должен быть активен режим MULTILINE:
        s = s.replaceAll("(?m)^ +", "");
        print(s);
        s = s.replaceFirst("[aeiou]", "(VOWEL1)");
        StringBuffer sbuf = new StringBuffer();
        Pattern p = Pattern.compile("[aeiou]");
        Matcher m = p.matcher(s);
        // Обработка информации find при выполнении замены:
        while(m.find())
            m.appendReplacement(sbuf, m.group().toUpperCase());
        // Присоединение оставшегося текста:
        m.appendTail(sbuf);
        print(sbuf);
    }
} /* Output:
Here's a block of text to use as input to
the regular expression matcher. Note that we'll
first extract the block of text by looking for
the special delimiters, then process the
extracted block.
H(VOWEL1)R'E'S A bLcK Of tEXt tO UsE As InPUt tO
thE rEGULAr ExprESSIOn mAtChEr. NOTe thAt wE'll
fIrSt ExtrAct thE bLcK Of tEXt by lOOkIng fOr
thE spEcIAl dElImItErS, thEn prOcEsS thE
ExtrActEd bLcK.
*///:~
```

Файл открывается и читается с использованием класса `TextFile` в библиотеке `net.mindview.util` (соответствующий код будет приведен в главе 18). Статический метод

`read()` читает весь файл и возвращает его содержимое в виде объекта `String`. Переменная `mInput` предназначена для хранения текста совпадения (обратите внимание на группирующие круглые скобки) между `/*!` и `!*/`. Затем серия из двух и более пробелов сокращается до одного пробела, а пробелы в начале строк удаляются (чтобы удаление произошло во всех строках, а не только в начале ввода, должен быть активен многострочный режим). Эти две замены выполняются эквивалентным (но более удобным в данном случае) методом `replaceAll()`, который является частью `String`. Поскольку каждая замена используется в программе только один раз, такая реализация не снижает эффективность по сравнению с предварительной компиляцией в объект `Pattern`.

Метод `replaceFirst()` выполняет только первую найденную замену. Кроме того, заменяющие строки в `replaceFirst()` и `replaceAll()` представляют собой обычные литералы, и если вы хотите выполнить какую-либо дополнительную обработку при каждой замене, они вам не помогут. В таких случаях необходимо использовать метод `appendReplacement()`, который позволяет задать код, выполняемый при замене. В предыдущем примере при построении итогового буфера `sbuf` выбираются и обрабатываются данные `group()` (в данном случае гласные, найденные при помощи регулярного выражения, преобразуются к верхнему регистру). Обычно все замены выполняются одна за другой, после чего вызывается метод `appendTail()`, но если вы захотите смоделировать метод `replaceFirst()` (или выполнить замену для первых `n` вхождений), выполните замену и вызовите `appendTail()`, чтобы поместить остаток в `sbuf`.

Метод `appendReplacement()` также позволяет включить ссылку на сохраненную группу прямо в строку замены; для этого используется обозначение `$g`, где `g` — номер группы. Впрочем, эта возможность предназначена для более простых задач и не принесет желаемого результата в приведенной программе.

reset()

Существующий объект `Matcher` может быть применен к новой символьной последовательности методами `reset()`:

```
//: strings/Resetting.java
import java.util.regex.*;

public class Resetting {
    public static void main(String[] args) throws Exception {
        Matcher m = Pattern.compile("[frb][aiu][gx]")
            .matcher("fix the rug with bags");
        while(m.find())
            System.out.print(m.group() + " ");
        System.out.println();
        m.reset("fix the rig with rags");
        while(m.find())
            System.out.print(m.group() + " ");
    }
} /* Output:
fix rug bag
fix rig rag
*///:~
```

Вызов `reset()` без аргументов переводит `Matcher` в начало текущей последовательности.

Регулярные выражения и ввод-вывод в Java

В большинстве приведенных ранее примеров регулярные выражения применялись к статическим строкам. В следующем примере продемонстрирован один из способов применения регулярных выражений для поиска совпадений в файле. Пример `JGrep.java` (прототипом которого послужила Unix-программа `grep`) получает два аргумента: имя файла и регулярное выражение. В выходных данных представлены номера строк, в которых найдены совпадения, и позиция(-и) совпадения в строке.

```
//: strings/JGrep.java
// Очень простая версия программы "grep".
// {Args: JGrep.java "\\b[Ssct]\\w+"}
import java.util.regex.*;
import net.mindview.util.*;

public class JGrep {
    public static void main(String[] args) throws Exception {
        if(args.length < 2) {
            System.out.println("Usage: java JGrep file regex");
            System.exit(0);
        }
        Pattern p = Pattern.compile(args[1]);
        // Перебор строк входного файла:
        int index = 0;
        Matcher m = p.matcher("");
        for(String line : new TextFile(args[0])) {
            m.reset(line);
            while(m.find())
                System.out.println(index++ + ": " +
                                    m.group() + ": " + m.start());
        }
    }
} /* Output: (Sample)
0: strings: 4
1: simple: 10
2: the: 28
3: Ssct: 26
4: class: 7
5: static: 9
6: String: 26
7: throws: 41
8: System: 6
9: System: 6
10: compile: 24
11: through: 15
12: the: 23
13: the: 36
14: String: 8
15: System: 8
16: start: 31
*///:~
```

Файл открывается как объект `net.mindview.util.TextFile` (см. главу 18), который читает строки файла в `ArrayList`. Это позволяет использовать синтаксис `foreach` для перебора строк объекта `TextFile`.

И хотя новый объект `Matcher` можно создавать в цикле `for`, чуть эффективнее будет создать пустой объект `Matcher` вне цикла, а затем использовать метод `reset()` для

связывания `Matcher` с каждой строкой входных данных. Для поиска в полученной строке используется метод `find()`.

Тестовый пример открывает файл `JGrep.java` для чтения входных данных и ищет в нем слова, начинающиеся с `[Ssct]`.

Регулярные выражения подробно рассматриваются в книге Джеффри Фридля «*Mastering Regular Expressions*», 2nd Edition (O'Reilly, 2002). В Интернете можно найти много руководств начального уровня; кроме того, немало полезной информации содержится в документации таких языков, как Perl и Python.

15. (5) Измените пример `JGrep.java` так, чтобы в его аргументах могли передаваться флаги (например, `Pattern.CASE_INSENSITIVE`, `Pattern.MULTILINE`).
16. (5) Измените пример `JGrep.java`, чтобы в аргументе ему можно было передать имя каталога или файла (при передаче каталога в поиск должны включаться все файлы, находящиеся в указанном каталоге). Подсказка: список имен файлов можно построить командой


```
File[] files = new File(".").listFiles();
```
17. (8) Напишите программу, которая читает файл с исходным кодом Java (имя файла передается в командной строке) и выводит все комментарии, содержащиеся в файле.
18. (8) Напишите программу, которая читает файл с исходным кодом Java (имя файла передается в командной строке) и выводит все строковые литералы, содержащиеся в файле.
19. (8) На основе двух последних упражнений напишите программу, которая анализирует исходный код Java и выдает список всех имен классов, использованных в программе.

Сканирование ввода

Когда-то чтение данных из текстового файла или стандартного ввода считалось относительно хлопотным делом. Обычно программа читала строку текста, разбивала ее на лексемы, после чего использовала различные методы классов `Integer`, `Double` и т. д. для разбора данных:

```
//: strings/SimpleRead.java
import java.io.*;
public class SimpleRead {
    public static BufferedReader input = new BufferedReader(
        new StringReader("Sir Robin of Camelot\n22 1.61803"));
    public static void main(String[] args) {
        try {
            System.out.println("What is your name?");
            String name = input.readLine();
            System.out.println(name);
            System.out.println(
                "How old are you? What is your favorite double?");
            System.out.println("(input: <age> <double>)");
            String numbers = input.readLine();
            System.out.println(numbers);
        }
    }
}
```



```

String[] numArray = numbers.split(" ");
int age = Integer.parseInt(numArray[0]);
double favorite = Double.parseDouble(numArray[1]);
System.out.format("Hi %s.\n", name);
System.out.format("In 5 years you will be %d.\n",
    age + 5);
System.out.format("My favorite double is %f.",
    favorite / 2);
} catch(IOException e) {
    System.err.println("I/O exception");
}
}
} /* Output:
What is your name?
Sir Robin of Camelot
How old are you? What is your favorite double?
(input: <age> <double>)
22 1.61803
Hi Sir Robin of Camelot.
In 5 years you will be 27.
My favorite double is 0.809015.
*///:~

```

Поле `input` использует классы из пакета `java.io`, который будет рассматриваться в главе 18. Объект `StringReader` преобразует `String` в поток, доступный для чтения; этот объект используется для создания объекта `BufferedReader`, содержащего метод `readLine()`. В результате объект `input` читается по строкам так, как если бы программа работала со стандартным вводом с консоли.

При помощи метода `readLine()` программа получает объект `String` для каждой строки ввода. Если каждая строка данных содержит одно входное значение, все просто, но если одна строка может содержать два и более значения, ситуация усложняется — строку приходится разбивать для раздельной обработки каждого входного значения. В нашем примере разбиение выполняется при создании `numArray`, но учтите, что метод `split()` появился в J2SE1.4 и в предыдущих версиях приходится использовать другие средства.

Класс `Scanner`, появившийся в Java SE5, избавляет разработчика от многих сложностей по сканированию ввода:

```

//: strings/BetterRead.java
import java.util.*;

public class BetterRead {
    public static void main(String[] args) {
        Scanner stdin = new Scanner(SimpleRead.input);
        System.out.println("What is your name?");
        String name = stdin.nextLine();
        System.out.println(name);
        System.out.println(
            "How old are you? What is your favorite double?");
        System.out.println("(input: <age> <double>)");
        int age = stdin.nextInt();
        double favorite = stdin.nextDouble();
        System.out.println(age);
        System.out.println(favorite);
        System.out.format("Hi %s.\n", name);
    }
}

```

продолжение ➤

```

        System.out.format("In 5 years you will be %d.\n",
            age + 5);
        System.out.format("My favorite double is %f.",
            favorite / 2);
    }
} /* Output:
What is your name?
Sir Robin of Camelot
How old are you? What is your favorite double?
(input: <age> <double>)
22
1.61803
Hi Sir Robin of Camelot.
In 5 years you will be 27.
My favorite double is 0.809015.
*///:~

```

Конструктор `Scanner` может получать практически любые входные объекты, включая объекты `File` (которые будут рассматриваться в главе 18), `InputStream`, `String` или в нашем случае `Readable` — интерфейс, появившийся в Java SE5 для описания «объекта, содержащего метод `read()`». Объект `BufferedReader` из предыдущего примера попадает в эту категорию.

С классом `Scanner` ввод, разбиение на лексемы и разбор удобно распределяются по разным видам методов `next()`. Простой метод `next()` возвращает следующую лексему `String`; также существуют методы `next()` для всех примитивных типов (кроме `char`), `BigDecimal` и `BigInteger`. Все методы `next` выполняются в блокирующем режиме; это означает, что они возвращают управление только после появления следующей завершённой входной лексемы. Также существуют соответствующие методы `hasNext`, которые возвращают `true`, если следующая входная лексема относится к правильному типу.

Интересное различие между двумя предшествующими примерами проявляется в отсутствии блока `try` для исключений `IOException` в `BetterRead.java`. Класс `Scanner` предполагает, что `IOException` сигнализирует о конце входных данных и поэтому поглощает эти исключения. Впрочем, самое последнее исключение доступно в методе `IOException()`, так что вы можете проверить его при необходимости.

- 20.** (2) Создайте класс, содержащий поля типов `int`, `long`, `float`, `double` и `String`. Создайте для этого класса конструктор, который получает один аргумент `String`, сканирует полученную строку и разбирает ее по разным полям. Добавьте метод `toString()` и продемонстрируйте правильность работы своего класса.

Ограничители Scanner

По умолчанию `Scanner` разбивает входные данные по пропускам, но вы также можете задать собственный ограничитель в форме регулярного выражения:

```

//: strings/ScannerDelimiter.java
import java.util.*;

public class ScannerDelimiter {
    public static void main(String[] args) {
        Scanner scanner = new Scanner("12, 42, 78, 99, 42");
        scanner.useDelimiter("\\s*,\\s*");
    }
}

```

```

        while(scanner.hasNextInt())
            System.out.println(scanner.nextInt());
    }
} /* Output:
12
42
78
99
42
*///:~

```

В этом примере в качестве разделителей при чтении из `String` используются запятые (окруженные произвольным количеством пропусков). Тот же способ может использоваться для чтения из файлов, разделенных запятыми. Кроме метода `useDelimiter()`, назначающего шаблон разделителя, также существует метод `delimiter()`, который возвращает текущий объект `Pattern`, используемый в качестве разделителя.

Сканирование с использованием регулярных выражений

Кроме сканирования с чтением заранее определенных примитивных типов, вы также можете проводить сканирование ввода по собственным шаблонам; эта возможность полезна при поиске более сложных данных. В следующем примере сканируются данные о потенциальных угрозах из журнального файла, созданного программой-брандмауэром (firewall):

```

//: strings/ThreatAnalyzer.java
import java.util.regex.*;
import java.util.*;

public class ThreatAnalyzer {
    static String threatData =
        "58.27.82.161@02/10/2005\n" +
        "204.45.234.40@02/11/2005\n" +
        "58.27.82.161@02/11/2005\n" +
        "58.27.82.161@02/12/2005\n" +
        "58.27.82.161@02/12/2005\n" +
        "[Next log section with different data format]";
    public static void main(String[] args) {
        Scanner scanner = new Scanner(threatData);
        String pattern = "(\\d+\\.\\d+\\.\\d+\\.\\d+)@" +
            "(\\d{2}/\\d{2}/\\d{4})";
        while(scanner.hasNext(pattern)) {
            scanner.next(pattern);
            MatchResult match = scanner.match();
            String ip = match.group(1);
            String date = match.group(2);
            System.out.format("Threat on %s from %s\n", date, ip);
        }
    }
} /* Output:
Threat on 02/10/2005 from 58.27.82.161
Threat on 02/11/2005 from 204.45.234.40
Threat on 02/11/2005 from 58.27.82.161
Threat on 02/12/2005 from 58.27.82.161
Threat on 02/12/2005 from 58.27.82.161
*///:~

```

При использовании `next()` с конкретным шаблоном этот шаблон применяется к следующей входной лексеме. Для получения доступа к результату используется метод `match()`; как видно из приведенного примера, он работает так же, как и поиск совпадения по регулярному выражению, о котором говорилось ранее.

Будьте внимательны при сканировании по регулярным выражениям: совпадение шаблона ищется только к следующей входной лексеме, так что если ваш шаблон содержит разделитель, совпадение никогда не будет найдено.

StringTokenizer

До введения поддержки регулярных выражений (в J2SE1.4) или класса `Scanner` (в Java SE5) для разбиения строк использовался класс `StringTokenizer`. Теперь те же задачи решаются намного проще и компактнее, но ниже приведено простое сравнение `StringTokenizer` с двумя другими методами:

```
//: strings/ReplacingStringTokenizer.java
import java.util.*;

public class ReplacingStringTokenizer {
    public static void main(String[] args) {
        String input = "But I'm not dead yet! I feel happy!";
        StringTokenizer stoke = new StringTokenizer(input);
        while(stoke.hasMoreElements())
            System.out.print(stoke.nextToken() + " ");
        System.out.println();
        System.out.println(Arrays.toString(input.split(" ")));
        Scanner scanner = new Scanner(input);
        while(scanner.hasNext())
            System.out.print(scanner.next() + " ");
    }
} /* Output:
But I'm not dead yet! I feel happy!
[But, I'm, not, dead, yet!, I, feel, happy!]
But I'm not dead yet! I feel happy!
*///:~
```

С регулярными выражениями и объектами `Scanner` строку также можно разбить на части по более сложным шаблонам — с `StringTokenizer` это сделать сложнее. Можно уверенно сказать, что класс `StringTokenizer` устарел и пользоваться им не следует.

Резюме

В прошлом поддержка строковых операций в Java была весьма примитивной, но в последних версиях языка появились куда более совершенные средства, позаимствованные из других языков. На данный момент поддержка строк в Java достаточно полна, хотя иногда приходится обращать внимание на подробности, связанные с эффективностью. например на правильность использования `StringBuilder`.

14

Информация о типах

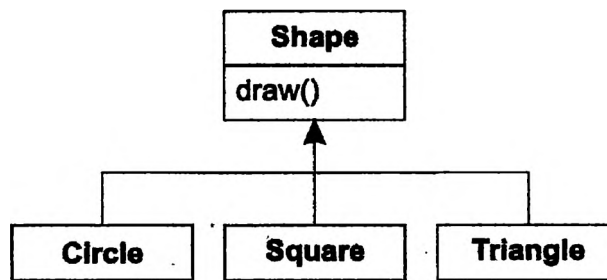
Механизм динамического определения типов (RTTI) позволяет получать и использовать информацию о типах во время выполнения программы.

RTTI избавляет от необходимости выполнять операции, использующие особенности конкретного типа, только во время компиляции и открывает ряд чрезвычайно мощных возможностей. Необходимость в динамическом определении типов выводит на первый ряд интересных (и зачастую сложных) аспектов объектно-ориентированного проектирования и поднимает фундаментальные вопросы относительно структурирования программ.

Эта глава показывает, как язык Java позволяет вам извлечь информацию об объектах и классах во время выполнения программы. Существует два способа получить эту информацию: «традиционное» динамическое определение типов (run-time type identification, RTTI), подразумевающее, что во время компиляции и последующего выполнения программы у вас есть все необходимые типы, а также механизм «отражения» (reflection), применяемый исключительно во время выполнения программы. Для начала мы обсудим традиционное определение типов, а потом уже перейдем к отражению.

Необходимость в динамическом определении типов (RTTI)

Рассмотрим хорошо знакомый нам пример с геометрическими фигурами. Иерархия классов этого примера использует полиморфизм. Общим базовым классом является фигура Shape, а конкретными производными классами — окружность Circle, прямоугольник Square и треугольник Triangle.



Это обычная диаграмма наследования — базовый класс располагается на вершине диаграммы, производные классы присоединяются к нему снизу. Обычно в объектно-ориентированном программировании разработчик стремится к тому, чтобы код манипулировал ссылками на типы базового класса (в нашем случае это фигура — `Shape`). Таким образом, если вы решите добавить в программу новый класс (например, производный от фигуры `Shape` ромб — `Rhomboid`), то код менять не придется. В нашем случае метод `draw()` класса является динамически связываемым, поэтому программист-клиент может вызывать этот метод по ссылке на базовый тип `Shape`. Метод `draw()` переопределяется во всех производных классах, и по природе динамического связывания вызов его с помощью ссылки базового класса все равно даст необходимый результат. Это полиморфизм.

Таким образом, обычно вы создаете объект конкретного класса (`Circle`, `Square` или `Triangle`), проводите восходящее преобразование к фигуре `Shape` («забывая» точный тип объекта) и используете ссылку на безымянную фигуру.

Реализация иерархии `Shape` может выглядеть так:

```

//: typeinfo/Shapes.java
import java.util.*;

abstract class Shape {
    void draw() { System.out.println(this + ".draw()"); }
    abstract public String toString();
}

class Circle extends Shape {
    public String toString() { return "Circle"; }
}

class Square extends Shape {
    public String toString() { return "Square"; }
}

class Triangle extends Shape {
    public String toString() { return "Triangle"; }
}

public class Shapes {
    public static void main(String[] args) {
        List<Shape> shapeList = Arrays.asList(
            new Circle(), new Square(), new Triangle()
        );
        for(Shape shape : shapeList)
            shape.draw();
    }
} /* Output:

```

```
Circle.draw()  
Square.draw()  
Triangle.draw()  
*///:~
```

Метод `draw()` базового класса `Shape` неявно использует метод `toString()` для вывода идентификатора класса, передавая методу `System.out.println()` ссылку `this` (обратите внимание: метод `toString()` объявлен абстрактным, чтобы производные классы были обязаны переопределить его и чтобы предотвратить создание экземпляров `Shape`). Когда при выводе строки в выражении конкатенации (с оператором `+` и объектами `String`) встречается объект `String`, автоматически вызывается его метод `toString()`, возвращающий строковое представление объекта. Каждый производный класс переопределяет метод `toString()` (из базового класса `Object`), чтобы метод `draw()` (полиморфно) выводил в каждом случае различную информацию.

В данном примере восходящее преобразование происходит во время помещения объекта-фигуры в контейнер `List<Shape>`. При восходящем преобразовании к `Shape` тот факт, что объекты являются *конкретными разновидностями* `Shape`, теряется. С точки зрения массива в нем хранятся просто объекты `Shape`.

При извлечении элемента из массива контейнер (который в действительности хранит все в формате `Object`) автоматически преобразует результат обратно в `Shape`. Это основная форма RTTI, поскольку все подобные преобразования в языке Java проверяются во время исполнения на корректность. Именно для этого и служит RTTI: во время выполнения программы проверяется истинный тип объекта.

В нашем случае определение типа происходит частично: тип `Object` преобразуется к базовому типу фигур `Shape`, а не к конкретным типам `Circle`, `Square` или `Triangle`. Просто потому, что в данный момент только нам *известно* то, что массив заполнен фигурами `Shape`. Во время компиляции правильность преобразований обеспечивается контейнером и системой обобщенных типов Java, но при выполнении требует явного преобразования типов.

После в действие вступает полиморфизм — для каждой фигуры `Shape` выполняемый код определяется в зависимости от того, окружность ли это (`Circle`), прямоугольник (`Square`) или треугольник (`Triangle`). И в основном именно так все и должно работать; основная часть вашего кода не должна заботиться о *фактическом* типе объекта, она оперирует с универсальным представлением целого семейства объектов (в нашем случае это фигура (`Shape`)). В результате программа легче пишется, а впоследствии проще читается и сопровождается, и самые сложные проекты проще реализовать, понять и изменить. По этой причине полиморфизм считается желательной целью при написании объектно-ориентированных программ.

Но что, если у вас имеется не совсем обычная задача, для успешного решения которой необходимо знать точный тип объекта, обладая ссылкой только базового типа? К примеру, пользователи программы с фигурами захотели подсветить определенные фигуры фиолетовым цветом. Таким образом они легко обнаружат на экране все треугольники — они будут отличаться от других фигур цветом. А может, метод должен выполнить поворот для фигур из списка, но поворачивать круги бессмысленно, и вы предпочитаете исключить их из обработки. Эту задачу помогает решить RTTI: вы

можете запросить точный тип объекта, на который указывает ссылка базового типа `Shape`, и отобразить объекты, требующие особого обращения.

Объект `Class`

Чтобы понять, как работает RTTI в Java, сначала необходимо узнать, каким образом хранится информация о типе во время выполнения программы. Делается это с помощью специального *объекта* типа `Class`, который и содержит информацию о классе. Более того, объект `Class` используется при создании всех «обыкновенных» объектов любой программы. Java выполняет операции RTTI с использованием объекта `Class` даже при выполнении явного преобразования типа. Класс `Class` также предоставляет ряд других возможностей использования RTTI.

Для каждого класса, использующегося вашей программой, существует свой объект `Class`. То есть каждый раз при написании и последующей компиляции нового класса для него создается объект `Class` (который затем сохраняется в одноименном файле с расширением `.class`). Чтобы создать объект этого класса, виртуальная машина Java (JVM) использует подсистему, называемую *загрузчиком классов*.

Подсистема загрузчика классов в действительности может включать цепочку загрузчиков, но существует всего один *первичный загрузчик классов*, который является частью реализации JVM. Первичный загрузчик классов загружает так называемые *доверенные классы*, включая классы Java API (обычно с локального диска). Как правило, включать дополнительные загрузчики в цепочку не обязательно, но в особых ситуациях (например, при загрузке классов для поддержки приложений веб-серверов или загрузке классов по сети) в вашем распоряжении имеется механизм подключения дополнительных загрузчиков классов.

Все классы загружаются в JVM динамически, при первом использовании класса. Это происходит при первом обращении к статическому члену класса. Оказывается, конструктор тоже является статическим членом класса, хотя ключевое слово `static` для конструктора не используется. Таким образом, создание нового объекта класса оператором `new` также считается обращением к статическому члену класса.

Итак, программа Java не загружается полностью в самом начале; ее фрагменты загружаются по мере необходимости. В этом отношении Java отличается от многих традиционных языков. Динамическая загрузка позволяет реализовать поведение, которое трудно или невозможно воспроизвести в языках со статической загрузкой (таких, как C++).

Загрузчик классов сначала проверяет, загружен ли объект `Class` для указанного типа. При отрицательном результате загрузчик по умолчанию ищет файл `.class` с соответствующим именем (внешний загрузчик, например, может прочитать байт-код из базы данных). В процессе загрузки исполнительная система *проверяет* байт-код класса, убеждаясь в том, что он не был поврежден и не содержит некорректного кода Java (одна из линий обеспечения безопасности в Java).

Как только объект `Class` для определенного типа окажется в памяти, в дальнейшем он используется при создании всех объектов этого типа. Следующая программа поясняет сказанное:


```
//: typeinfo/SweetShop.java
// Examination of the way the class loader works.
import static net.mindview.util.Print.*;

class Candy {
    static { print("Загрузка класса Candy"); }
}

class Gum {
    static { print("Загрузка класса Gum"); }
}

class Cookie {
    static { print("Загрузка класса Cookie"); }
}

public class SweetShop {
    public static void main(String[] args) {
        print("в методе main");
        new Candy();
        print("После создания объекта Candy");
        try {
            Class.forName("Gum");
        } catch(ClassNotFoundException e) {
            print("Не удалось найти Gum");
        }
        print("После вызова метода Class.forName(\"Gum\")");
        new Cookie();
        print("После создания объекта Cookie");
    }
} /* Output:
в методе main
Загрузка класса Candy
После создания объекта Candy
Загрузка класса Gum
После вызова метода Class.forName("Gum")
Загрузка класса Cookie
После создания объекта Cookie
*///:~
```

В каждом классе программы имеется статический блок, который отработывает один раз, при первой загрузке класса. При выполнении этого блока выводится сообщение, говорящее о том, какой класс загружается. В методе `main()` создание объектов классов `Candy`, `Gum` и `Cookie` чередуется с выводом на экран вспомогательных сообщений, чтобы вы правильно определили, в какой момент времени загружается класс.

Из результата работы программы видно, что каждый объект `Class` загружается, только если в этом есть прямая необходимость, а «статическая» инициализация производится при загрузке этого объекта.

Особенно интересна строка программы

```
Class.forName("Gum");
```

Это вызов статического метода класса `Class` (которому принадлежат все объекты `Class`). Объект `Class` ничем не хуже обычных объектов, поэтому вы можете создавать его и манипулировать ссылками на него. (Именно так и поступает загрузчик классов.)

Один из способов получить ссылку на объект `Class` — вызвать метод `forName()`, передав ему в качестве аргумента строку (`String`) с именем (следите за правильностью написания!) определенного класса. Этот метод возвращает ссылку на объект типа `Class`, которая нами здесь игнорировалась; вызов метода `Class.forName()` был произведен для получения побочного эффекта, частью которого является загрузка класса `Gum`, если он еще не в памяти. В процессе загрузки выполняется `static`-инициализатор класса `Gum`.

В рассмотренном примере, если бы метод `Class.forName()` завершился неудачей (не смог бы найти класс, который вы хотели загрузить), он возбудил бы исключение `ClassNotFoundException` (в идеале, название исключения скажет вам практически все, что нужно знать о возникшей ошибке¹). Здесь мы просто информируем о проблеме и двигаемся дальше, однако в более сложной программе можно было бы попытаться исправить ошибку в обработчике исключения.

Каждый раз, когда вы хотите использовать информацию о типе во время выполнения, сначала необходимо получить ссылку на соответствующий объект `Class`. Для этого удобно воспользоваться методом `Class.forName()`, потому что для получения ссылки на `Class` объект не нужен. Но если у вас уже имеется объект интересующего вас типа, для получения ссылки на `Class` можно вызвать метод, являющийся частью корневого класса `Object`: `getClass()`. Метод возвращает ссылку на объект `Class`, представляющий фактический тип объекта. Класс `Class` содержит много интересных методов; вот лишь несколько из них:

```
//: typeinfo/toys/ToyTest.java

// Тестирование класса Class.
package typeinfo.toys;
import static net.mindview.util.Print.*;

interface HasBatteries {}
interface Waterproof {}
interface Shoots {}

class Toy {
    // Закомментируйте следующий конструктор по умолчанию,
    // чтобы увидеть ошибку NoSuchMethodError из (*1*)
    Toy() {}
    Toy(int i) {}
}

class FancyToy extends Toy
implements HasBatteries, Waterproof, Shoots {
    FancyToy() { super(1); }
}

public class ToyTest {
    static void printInfo(Class cc) {
        print("Имя класса: " + cc.getName() +
            " является интерфейсом? [" + cc.isInterface() + "]");
        print("Простое имя: " + cc.getSimpleName());
        print("Каноническое имя : " + cc.getCanonicalName());
    }
}
```

¹ `ClassNotFoundException` — «не удалось найти класс». — *Примеч. перев.*

```
public static void main(String[] args) {
    Class c = null;
    try {
        c = Class.forName("typeinfo.toys.FancyToy");
    } catch(ClassNotFoundException e) {
        print("Не удастся найти FancyToy");
        System.exit(1);
    }
    printInfo(c);
    for(Class face : c.getInterfaces())
        printInfo(face);
    Class up = c.getSuperclass();
    Object obj = null;
    try {
        // Необходим конструктор по умолчанию:
        obj = up.newInstance();
    } catch(InstantiationException e) {
        print("Не удалось создать экземпляр");
        System.exit(1);
    } catch(IllegalAccessException e) {
        print("Отказано в доступе");
        System.exit(1);
    }
    printInfo(obj.getClass());
}
} /* Output:
Имя класса: typeinfo.toys.FancyToy является интерфейсом? [false]
Простое имя: FancyToy
Каноническое имя : typeinfo.toys.FancyToy
Имя класса: typeinfo.toys.HasBatteries является интерфейсом? [true]
Простое имя: HasBatteries
Каноническое имя : typeinfo.toys.HasBatteries
Имя класса: typeinfo.toys.Waterproof является интерфейсом? [true]
Простое имя: Waterproof
Каноническое имя : typeinfo.toys.Waterproof
Имя класса: typeinfo.toys.Shoots является интерфейсом? [true]
Простое имя: Shoots
Каноническое имя : typeinfo.toys.Shoots
Имя класса: typeinfo.toys.Toy является интерфейсом? [false]
Простое имя: Toy
Каноническое имя : typeinfo.toys.Toy
*///:~
```

FancyToy наследует от Toy и реализует интерфейсы HasBatteries, Waterproof и Shoots. В методе main() создается ссылка на Class, которая инициализируется информацией класса FancyToy с использованием метода forName() в подходящем блоке try. Обратите внимание: в строке, передаваемой forName(), должно указываться полное, то есть уточненное имя (включающее имя пакета).

Метод printInfo() использует getName() для получения полного имени класса, а методы getSimpleName() и getCanonicalName() (появившиеся в Java SE5) возвращают имя без пакета и полное имя соответственно. Как нетрудно догадаться по имени, метод isInterface() сообщает, представляет ли данный объект Class интерфейс. Таким образом, по объекту Class можно получить практически любую информацию о типе, которая только может понадобиться.

Метод `Class.getInterfaces()`, вызываемый в `main()`, возвращает массив объектов `Class`, представляющих интерфейсы, содержащиеся в заданном объекте `Class`.

У объекта `Class` также можно запросить информацию о непосредственном базовом классе методом `getSuperclass()`. Метод возвращает ссылку на `Class`, которая может использоваться для получения дополнительной информации. Таким образом, во время выполнения можно определить полную иерархию классов объекта.

Метод `newInstance()` класса `Class` может использоваться для реализации «виртуального конструктора», работающего по принципу: «Я не знаю точно, к какому типу ты относишься, но все равно создай себя так, как положено». В приведенном примере `up` представляет собой обычную ссылку на `Class` без дополнительной информации о типе, известной во время компиляции. При создании нового экземпляра возвращается ссылка на `Object`. Но в данном случае эта ссылка указывает на объект `Toy`. Конечно, прежде чем отправлять объекту какие-либо сообщения (кроме тех, которые принимает `Object`), необходимо получить информацию о нем и выполнить преобразование типа. Кроме того, класс, создаваемый вызовом `newInstance()`, должен иметь конструктор по умолчанию. Позже в этой главе вы увидите, как динамически создавать объекты с использованием любого конструктора через API отражения Java.

1. (1) В примере `ToyTest.java` закомментируйте конструктор `Toy` по умолчанию. Объясните, что при этом происходит.
2. (2) Встройте новый интерфейс в `ToyTest.java`. Убедитесь в том, что он обнаруживается, а информация о нем выводится программой.
3. (2) Добавьте класс `Rhomboid` в иерархию `Shapes.java`. Создайте объект `Rhomboid`, выполните восходящее преобразование его в `Shape`, а затем снова верните его к классу `Rhomboid` нисходящим преобразованием. Попробуйте выполнить нисходящее преобразование к `Circle` и объясните, что при этом происходит.
4. (3) Измените предыдущий пример так, чтобы перед нисходящим преобразованием в нем использовался оператор `instanceof` для проверки типа.
5. (3) Реализуйте в `Shapes.java` метод `rotate(Shape)`, который проверяет, не относится ли фигура к классу `Circle` (и если относится — не выполняет ли операцию).
6. (4) Измените пример `Shapes.java` так, чтобы он мог «выделять» (установкой внутреннего флага) все фигуры некоторого типа. Метод `toString()` каждого класса, производного от `Shape`, должен указывать, находится ли данная фигура в «выделенном» состоянии.
7. (3) Измените пример `SweetShop.java` так, чтобы тип создаваемых объектов определялся аргументом командной строки. Другими словами, если для запуска используется командная строка `java SweetShop Candy`, то создаются только объекты `Candy`. Обратите внимание на возможность управления тем, какие объекты `Class` загружаются программой, через аргумент командной строки.
8. (5) Напишите метод, который получает объект и рекурсивно выводит все классы в иерархии этого объекта.
9. (5) Измените предыдущее упражнение так, чтобы в нем использовался метод `Class.getDeclaredFields()` для вывода информации о полях класса.

- 10. (3)** Напишите программу, которая определяет, является ли массив `char` примитивным типом или «настоящим» объектом.

Литералы `class`

В Java есть еще один способ получить ссылку на объект `Class` — посредством *литерала* `class`. В рассмотренной программе мы бы получили ее таким образом:

```
Gum.class;
```

Это не только проще, но еще и безопаснее, поскольку проверка осуществляется еще во время компиляции. К тому же не нужно вызывать метод, а значит, такой подход более эффективен.

Литералы `class` работают со всеми обычными классами, так же как и с интерфейсами, массивами и даже с примитивами. Вдобавок во всех классах-оболочках для простейших типов имеется поле с именем `TYPE`. Это поле содержит ссылку на объект `Class` для ассоциированного с ним простейшего типа, что показано в табл. 14.1.

Таблица 14.1. Альтернативное обозначение объекта `Class` с помощью литералов

Литерал	Ссылка на объект <code>Class</code>
<code>boolean.class</code>	<code>Boolean.TYPE</code>
<code>char.class</code>	<code>Character.TYPE</code>
<code>Byte.class</code>	<code>Byte.TYPE</code>
<code>short.class</code>	<code>Short.TYPE</code>
<code>int.class</code>	<code>Integer.TYPE</code>
<code>long.class</code>	<code>Long.TYPE</code>
<code>float.class</code>	<code>Float.TYPE</code>
<code>double.class</code>	<code>Double.TYPE</code>
<code>void.class</code>	<code>Void.TYPE</code>

Лично я предпочитаю использовать версию `.class`, так как она лучше сочетается с обычными классами.

Интересно, что создание ссылки на объект `Class` с применением синтаксиса `.class` не приводит к автоматической инициализации объекта `Class`. В действительности подготовка класса к использованию состоит из трех шагов.

1. *Загрузка*, выполняемая загрузчиком классов. Загрузчик находит байт-код (обычно, но не обязательно, хранящийся на диске в каталогах `CLASSPATH`) и создает на его основе объект `Class`.
2. *Компоновка*. В фазе компоновки проверяется байт-код класса, выделяется память для статических полей, а при необходимости разрешаются все ссылки на другие классы, используемые текущим классом.
3. *Инициализация*. Если у класса имеется суперкласс, выполняется его инициализация. Также выполняются статические инициализаторы и блоки статической инициализации.

Инициализация откладывается до первого обращения к статическому методу (конструктор неявно является статическим) или не-константному статическому полю:

```
//: typeinfo/ClassInitialization.java
import java.util.*;

class Initable {
    static final int staticFinal = 47;
    static final int staticFinal2 =
        ClassInitialization.rand.nextInt(1000);
    static {
        System.out.println("Инициализация Initable");
    }
}

class Initable2 {
    static int staticNonFinal = 147;
    static {
        System.out.println("Инициализация Initable2");
    }
}

class Initable3 {
    static int staticNonFinal = 74;
    static {
        System.out.println("Инициализация Initable3");
    }
}

public class ClassInitialization {
    public static Random rand = new Random(47);
    public static void main(String[] args) throws Exception {
        Class initable = Initable.class;
        System.out.println("После создания ссылки на Initable");
        // Не запускает процесс инициализации:
        System.out.println(Initable.staticFinal);
        // Запускает процесс инициализации:
        System.out.println(Initable.staticFinal2);
        // Запускает процесс инициализации:
        System.out.println(Initable2.staticNonFinal);
        Class initable3 = Class.forName("Initable3");
        System.out.println("После создания ссылки на Initable3");
        System.out.println(Initable3.staticNonFinal);
    }
} /* Output:
After creating Initable ref
47
Инициализация Initable
258
Инициализация Initable2
147
Инициализация Initable3
После создания ссылки на Initable3
74
*///:~
```

По сути инициализация откладывается настолько, насколько возможно. Из создания ссылки `initable` мы видим, что простое использование синтаксиса `.class` для

получения ссылки на класс не приводит к инициализации. С другой стороны, вызов `Class.forName()` немедленно инициализирует класс для получения ссылки на `Class`, как мы видим на примере создания `initable3`.

Если статическое неизменное (`static final`) значение является «константой времени компиляции», как `Initable.staticFinal`, оно может читаться без инициализации класса `Initable`. Впрочем, объявление поля со спецификаторами `static` и `final` не гарантирует этого поведения; обращение к `Initable.staticFinal2` запускает инициализацию класса, потому что это значение не может быть константой времени компиляции.

Если `static`-поле не объявлено с ключевым словом `final`, то при обращении к нему чтение возможно только после проведения компоновки (для выделения памяти) и инициализации (для инициализации выделенной памяти), как мы видим на примере `Initable2.staticNonFinal`.

Ссылки на обобщенные классы

Ссылка на `Class` указывает на объект `Class`, который производит экземпляры классов и содержит весь код методов этих экземпляров. Кроме того, он содержит статические члены класса. Таким образом, ссылка на `Class` в действительности точно обозначает тип объекта, на который она указывает: это объект класса `Class`.

Однако проектировщики Java SE5 увидели возможность сделать ситуацию чуть более конкретной за счет ограничения типа объекта `Class`, на который указывает ссылка, при помощи синтаксиса обобщенных типов. В следующем примере оба варианта синтаксиса правильны:

```
///  
typeinfo/GenericClassReferences.java  
  
public class GenericClassReferences {  
    public static void main(String[] args) {  
        Class intClass = int.class;  
        Class<Integer> genericIntClass = int.class;  
        genericIntClass = Integer.class; // То же  
        intClass = double.class;  
        // genericIntClass = double.class; // Недопустимо  
    }  
} ///  
~
```

Как видите, обычную ссылку на `Class` можно связать с любым другим объектом `Class`, тогда как параметризованную ссылку можно связать только с объявленным типом. Использование синтаксиса обобщенных типов позволяет компилятору реализовать дополнительную проверку типа.

А если вы захотите немного ослабить ограничение? Казалось бы, можно использовать решение следующего вида:

```
Class<Number> genericNumberClass = int.class;
```

Выглядит разумно, потому что `Integer` наследует от `Number`. Однако такое решение не сработает, потому что объект `Class` для `Integer` не является субклассом объекта `Class` для `Number` (эта тема более подробно рассматривается в главе 15).

Для ослабления ограничений при использовании параметризованных ссылок на `Class` я применяю метасимвол `?`, который является частью обобщенных типов Java и означает «что угодно». Если добавить метасимвол `?` в обычную ссылку на `Class` в приведенном примере, результат останется неизменным:

```
//: typeinfo/WildcardClassReferences.java

public class WildcardClassReferences {
    public static void main(String[] args) {
        Class<?> intClass = int.class;
        intClass = double.class;
    }
} ///:~
```

В Java SE5 запись `Class<?>` считается предпочтительной по сравнению с `Class`, хотя эти две формы эквивалентны, а простое обозначение `Class`, как вы видели, не порождает предупреждения компилятора. Преимущество записи `Class<?>` заключается в том, что она ясно показывает: ссылка на неконкретный класс не используется случайно или по незнанию. Разработчик сознательно выбирает неконкретную версию.

Чтобы создать ссылку на `Class`, ограниченную типом *или любыми его подтипами*, метасимвол `?` следует объединить с ключевым словом `extends` для создания *привязки*. Таким образом, вместо `Class<Number>` используется запись следующего вида:

```
//: typeinfo/BoundedClassReferences.java

public class BoundedClassReferences {
    public static void main(String[] args) {
        Class<? extends Number> bounded = int.class;
        bounded = double.class;
        bounded = Number.class;
        // Или любой класс, производный от Number.
    }
} ///:~
```

Синтаксис параметризации был добавлен в ссылки на `Class` только для обеспечения проверки типов во время компиляции, чтобы вы могли раньше узнать о допущенных ошибках. С обычными ссылками на `Class` вы узнаете о них только на стадии выполнения, что может быть неудобно.

В следующем примере используется синтаксис параметризованных ссылок. Он сохраняет ссылку на класс, а потом создает контейнер `List` с объектами, сгенерированными методом `newInstance()`:

```
//: typeinfo/FilledList.java
import java.util.*;

class CountedInteger {
    private static long counter;
    private final long id = counter++;
    public String toString() { return Long.toString(id); }
}

public class FilledList<T> {
    private Class<T> type;
    public FilledList(Class<T> type) { this.type = type; }
```



```

public List<T> create(int nElements) {
    List<T> result = new ArrayList<T>();
    try {
        for(int i = 0; i < nElements; i++)
            result.add(type.newInstance());
    } catch(Exception e) {
        throw new RuntimeException(e);
    }
    return result;
}
public static void main(String[] args) {
    FilledList<CountedInteger> fl =
        new FilledList<CountedInteger>(CountedInteger.class);
    System.out.println(fl.create(15));
}
} /* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
*///:~

```

Этот класс предполагает, что любой тип, с которым он работает, имеет конструктор по умолчанию (вызываемый без аргументов); если это условие нарушится, возникает исключение. Компилятор не выдает никаких предупреждений в этой программе.

При использовании синтаксиса параметризации для объектов `Class` происходит нечто интересное: метод `newInstance()` возвращает точный тип объекта вместо базового типа `Object`, как в примере `ToyTest.java`. Впрочем, это поведение несколько ограничено:

```

//: typeinfo/toys/GenericToyTest.java
// Тестирование класса Class.
package typeinfo.toys;
public class GenericToyTest {
    public static void main(String[] args) throws Exception {
        Class<FancyToy> ftClass = FancyToy.class;
        // Создает точный тип:
        FancyToy fancyToy = ftClass.newInstance();
        Class<? super FancyToy> up = ftClass.getSuperclass();
        // Не компилируется:
        // Class<Toy> up2 = ftClass.getSuperclass();
        // Создает только Object:
        Object obj = up.newInstance();
    }
} ////:~

```

При получении объекта суперкласса компилятор только позволит сказать, что ссылка на суперкласс представляет «некий класс, являющийся суперклассом `FancyToy`» (выражение `Class<? super FancyToy>`). Он не примет объявление `Class<Toy>`. На первый взгляд это странно, потому что `getSuperclass()` возвращает базовый класс (не интерфейс), а компилятору этот класс известен во время компиляции — в данном случае `Toy.class`, а не «какой-то суперкласс `FancyToy`». В любом случае из-за неоднозначности `up.newInstance()` возвращает не точный тип, а просто `Object`.

Новый синтаксис приведения типа

В Java SE5 также был добавлен новый синтаксис приведения типа для ссылок на `Class` — метод `cast()`:

```
//: typeinfo/ClassCasts.java

class Building {}
class House extends Building {}

public class ClassCasts {
    public static void main(String[] args) {
        Building b = new House();
        Class<House> houseType = House.class;
        House h = houseType.cast(b);
        h = (House)b; // ... или так.
    }
} ///:~
```

Метод `cast()` получает аргумент-объект и преобразует его к типу ссылки на `Class`. Конечно, приведенный выше код вроде бы выполняет много лишней работы по сравнению с последней строкой `main()`, которая делает то же самое.

Новый синтаксис приведения типов полезен в ситуациях, в которых нельзя использовать обычное приведение типа. Обычно это происходит при написании обобщенного кода (см. главу 15) и сохранении ссылки на `Class`, которая будет использоваться для приведения типа позднее. Как выясняется, такие ситуации встречаются редко — я нашел только один пример использования `cast()` во всей библиотеке Java SE5 (в `com.sun.mirror.util.DeclarationFilter`).

Еще одна новая возможность, которая *не нашла* применения в библиотеке Java SE5, — метод `Class.asSubclass()`, позволяющий преобразовать объект класса к более конкретному типу.

Проверка перед приведением типов

Пока что мы рассмотрели следующие формы RTTI.

1. Классическое преобразование (например, `(Shape)`), использующее RTTI для проверки правильности преобразования. В случае неверного преобразования возбуждается исключение `ClassCastException`.
2. Объект `Class`, представляющий тип вашего объекта. У объекта `Class` можно запросить полезную информацию во время выполнения.

В языке C++ классическая форма `(Shape)` вообще *не задействует* RTTI. Она просто сообщает компилятору, что необходимо обращаться с объектом как с новым типом. В языке Java, который при приведении проверяет соответствие типов, такое преобразование часто называют «безопасным нисходящим приведением типов». Слово «нисходящее» используется в силу традиций, сложившихся в практике составления диаграмм наследования. Если приведение окружности `Circle` к фигуре `Shape` является восходящим, то приведение фигуры `Shape` к окружности `Circle` является соответственно нисходящим. При этом точно известно, что окружность есть фигура, поэтому компилятор позволяет использовать восходящее преобразование без дополнительных указаний, однако никто *не может* утверждать, что фигура `Shape` обязательно представляет собой окружность `Circle` (это также может быть `Shape` или подтип `Shape` — `Circle`, `Square`, `Triangle` или любой другой тип). Соответственно, компилятор не позволит выполнить

нисходящее присваивание без явного приведения типа, которое сообщает, что по имеющейся у вас дополнительной информации вы точно знаете, что объект относится к конкретному фактическому типу (компилятор проверяет корректность нисходящего преобразования и не разрешит выполнить приведение к типу, не являющемуся производным классом).

Также в Java существует и третья форма RTTI. Она реализуется ключевым словом `instanceof`, которое сообщает вам, принадлежит ли объект к некоторому определенному типу. Результат запроса имеет логический (`boolean`) тип, поэтому вы просто «задаете» вопрос в следующей форме:

```
if(x instanceof Dog)
    ((Dog)x).bark();
```

Это предложение `if` сначала проверяет, принадлежит ли объект `x` классу `Dog`, прежде чем выполнять приведение объекта к типу `Dog`. Настоятельно рекомендуется использовать ключевое слово `instanceof` перед проведением нисходящего преобразования, особенно при недостатке информации о точном типе объекта, иначе все кончится исключением `ClassCastException`.

Обычно проводится поиск одного определенного типа (например, поиск треугольников среди прочих фигур), но с помощью ключевого слова `instanceof` легко можно «подсчитать» все типы объекта. Предположим, что у нас есть семейство классов на основе класса `Pet`, описывающего домашних животных (и их владельцев — они пригодятся нам в следующем примере). У каждого объекта `Individual` в этой иерархии имеется идентификатор `id` и необязательное имя `name`. (В классе `Individual` существуют некоторые сложности, из-за которых этот код будет более подробно рассмотрен в главе 17.) Впрочем, сейчас подробно рассматривать код `Individual` не обязательно — достаточно знать, что объекты могут создаваться с именем и без него, а каждый объект `Individual` содержит метод `id()`, который возвращает уникальный идентификатор (создаваемый посредством подсчета объектов). Также имеется метод `toString()`; если при создании `Individual` имя не задано, метод `toString()` выдает только простое имя типа.

Итак, следующая иерархия классов наследует от `Individual`:

```
//: typeinfo/pets/Person.java
package typeinfo.pets;

public class Person extends Individual {
    public Person(String name) { super(name); }
} ///:~

//: typeinfo/pets/Pet.java
package typeinfo.pets;

public class Pet extends Individual {
    public Pet(String name) { super(name); }
    public Pet() { super(); }
} ///:~

//: typeinfo/pets/Dog.java
package typeinfo.pets;
```

```

public class Dog extends Pet {
    public Dog(String name) { super(name); }
    public Dog() { super(); }
} ///:~

//: typeinfo/pets/Mutt.java
package typeinfo.pets;

public class Mutt extends Dog {
    public Mutt(String name) { super(name); }
    public Mutt() { super(); }
} ///:~

//: typeinfo/pets/Pug.java
package typeinfo.pets;

public class Pug extends Dog {
    public Pug(String name) { super(name); }
    public Pug() { super(); }
} ///:~

//: typeinfo/pets/Cat.java
package typeinfo.pets;

public class Cat extends Pet {
    public Cat(String name) { super(name); }
    public Cat() { super(); }
} ///:~

//: typeinfo/pets/EgyptianMau.java
package typeinfo.pets;

public class EgyptianMau extends Cat {
    public EgyptianMau(String name) { super(name); }
    public EgyptianMau() { super(); }
} ///:~

//: typeinfo/pets/Manx.java
package typeinfo.pets;

public class Manx extends Cat {
    public Manx(String name) { super(name); }
    public Manx() { super(); }
} ///:~

//: typeinfo/pets/Cymric.java
package typeinfo.pets;

public class Cymric extends Manx {
    public Cymric(String name) { super(name); }
    public Cymric() { super(); }
} ///:~

//: typeinfo/pets/Rodent.java
package typeinfo.pets;

public class Rodent extends Pet {
    public Rodent(String name) { super(name); }

```

```
public Rodent() { super(); }
} ///:~

//: typeinfo/pets/Rat.java
package typeinfo.pets;

public class Rat extends Rodent {
    public Rat(String name) { super(name); }
    public Rat() { super(); }
} ///:~

//: typeinfo/pets/Mouse.java
package typeinfo.pets;

public class Mouse extends Rodent {
    public Mouse(String name) { super(name); }
    public Mouse() { super(); }
} ///:~

//: typeinfo/pets/Hamster.java
package typeinfo.pets;

public class Hamster extends Rodent {
    public Hamster(String name) { super(name); }
    public Hamster() { super(); }
} ///:~
```

Также нам понадобится инструмент для случайного создания разных объектов, производных от `Pet`, а также (для удобства) массивов и контейнеров `List` с элементами `Pet`. Чтобы этот инструмент мог пережить несколько разных реализаций, мы определим его в виде абстрактного класса:

```
//: typeinfo/pets/PetCreator.java
// Creates random sequences of Pets.
package typeinfo.pets;
import java.util.*;

public abstract class PetCreator {
    private Random rand = new Random(47);
    // Контейнер List с разными видами создаваемых объектов Pet :
    public abstract List<Class? extends Pet>> types();
    public Pet randomPet() { // Создание одного случайного объекта Pet
        int n = rand.nextInt(types().size());
        try {
            return types().get(n).newInstance();
        } catch (InstantiationException e) {
            throw new RuntimeException(e);
        } catch (IllegalAccessException e) {
            throw new RuntimeException(e);
        }
    }
    public Pet[] createArray(int size) {
        Pet[] result = new Pet[size];
        for(int i = 0; i < size; i++)
            result[i] = randomPet();
        return result;
    }
}
```

```

public ArrayList<Pet> arrayList(int size) {
    ArrayList<Pet> result = new ArrayList<Pet>();
    Collections.addAll(result, createArray(size));
    return result;
}
} ///:~

```

Абстрактный метод `getTypes()` обращается к производному классу для получения контейнера `List` объектов `Class` (вариация на тему паттерна проектирования «Шаблонный метод»). Учтите, что тип класса указан как «любой класс, производный от `Pet`», так что `newInstance()` создает `Pet`, не требуя приведения типа. Метод `randomPet()` осуществляет случайное индексирование `List` и использует выбранный объект `Class` для создания нового экземпляра этого класса методом `Class.newInstance()`. Метод `createArray()` использует `randomPet()` для заполнения массива, а метод `arrayList()` последовательно использует `createArray()`.

При вызове `newInstance()` могут возбуждаться два вида исключений. Они обрабатываются в блоках `catch`, следующих за блоком `try`. И снова имена исключений достаточно ясно говорят о том, что же именно пошло не так. (Исключение `IllegalAccessException` относится к нарушению системы безопасности Java — в данном случае, если конструктор по умолчанию объявлен закрытым.)

При создании класса, производного от `PetCreator`, необходимо передать только контейнер `List` с типами объектов, которые должны создаваться методом `randomPet()` и другими методами. Метод `getTypes()` обычно просто возвращает ссылку на статический контейнер `List`. В следующей реализации используется метод `forName()`:

```

//: typeinfo/pets/ForNameCreator.java
package typeinfo.pets;
import java.util.*;

public class ForNameCreator extends PetCreator {
    private static List<Class<? extends Pet>> types =
        new ArrayList<Class<? extends Pet>>();
    // Типы, которые должны создаваться случайным образом:
    private static String[] typeNames = {
        "typeinfo.pets.Mutt",
        "typeinfo.pets.Pug",
        "typeinfo.pets.EgyptianMau",
        "typeinfo.pets.Manx",
        "typeinfo.pets.Cymric",
        "typeinfo.pets.Rat",
        "typeinfo.pets.Mouse",
        "typeinfo.pets.Hamster"
    };
    @SuppressWarnings("unchecked")
    private static void loader() {
        try {
            for(String name : typeNames)
                types.add(
                    (Class<? extends Pet>)Class.forName(name));
        } catch(ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }
}

```

```
static { loader(); }
public List<Class<? extends Pet>> types() {return types;}
} ///:~
```

Метод `loader()` создает контейнер `List` с объектами `Class`, используя метод `Class.forName()`. При этом может быть возбуждено исключение `ClassNotFoundException`, что вполне логично, так как передаваемая строка не может проверяться во время компиляции. Так как объекты `Pet` находятся в пакете `typeinfo`, при обращениях к классам должно указываться имя пакета.

Для создания типизованного контейнера `List` с объектами `Class` необходимо приведение типа, из-за которого компилятор выдает предупреждение. Метод `loader()` определяется отдельно, а затем помещается в блок статической инициализации, потому что аннотация `@SuppressWarnings` не может размещаться непосредственно в блоке статической инициализации.

Для подсчета объектов `Pet` необходим инструмент, который позволяет хранить количество объектов для разных типов `Pet`. Контейнер `Map` идеально подходит для этой цели; ключами являются имена типов `Pet`, а значениями — количества `Pet`. Располагая такой информацией, можно легко получить ответ на вопрос: «Сколько создано объектов `Hamster`?» Для подсчета объектов `Pet` можно воспользоваться оператором `instanceof`:

```
//: typeinfo/PetCount.java
// Using instanceof.
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class PetCount {
    static class PetCounter extends HashMap<String,Integer> {
        public void count(String type) {
            Integer quantity = get(type);
            if(quantity == null)
                put(type, 1);
            else
                put(type, quantity + 1);
        }
    }
    public static void
    countPets(PetCreator creator) {
        PetCounter counter= new PetCounter();
        for(Pet pet : creator.createArray(20)) {
            // Подсчет объектов Pet:
            printnb(pet.getClass().getSimpleName() + " ");
            if(pet instanceof Pet)
                counter.count("Pet");
            if(pet instanceof Dog)
                counter.count("Dog");
            if(pet instanceof Mutt)
                counter.count("Mutt");
            if(pet instanceof Pug)
                counter.count("Pug");
            if(pet instanceof Cat)
                counter.count("Cat");
            if(pet instanceof Manx)
                counter.count("EgyptianMau");
        }
    }
}
```

```

    if(pet instanceof Manx)
        counter.count("Manx");
    if(pet instanceof Manx)
        counter.count("Cymric");
    if(pet instanceof Rodent)
        counter.count("Rodent");
    if(pet instanceof Rat)
        counter.count("Rat");
    if(pet instanceof Mouse)
        counter.count("Mouse");
    if(pet instanceof Hamster)
        counter.count("Hamster");
}
// Вывод счетчиков:
print();
print(counter);
}
public static void main(String[] args) {
    countPets(new ForNameCreator());
}
} /* Output:
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat
EgyptianMau Hamster EgyptianMau Mutt Mutt Cymric Mouse Pug
Mouse Cymric
{Pug=3, Cat=9, Hamster=1, Cymric=7, Mouse=2, Mutt=3,
Rodent=5, Pet=20, Manx=7, EgyptianMau=7, Dog=6, Rat=2}
*///:~

```

В методе `countPets()` массив случайным образом заполняется объектами `Pet` с использованием `PetCreator`. Затем каждый объект `Pet` в массиве проверяется и подсчитывается с использованием `instanceof`.

Однако у оператора `instanceof` имеется несколько серьезных ограничений: объект можно сравнивать только с именованным типом, но не с объектом `Class`. Глядя на предыдущий пример, вы наверняка подумали, какая же обуза — писать все эти предложения `instanceof`, и вы правы. Но возможности автоматизировать этот процесс нет — создать из объектов `Class` список `ArrayList` и сравнить объекты по очереди с каждым его элементом не получится (не огорчайтесь — альтернатива существует, но об этом позже). Впрочем, особенно горевать по этому поводу не стоит — если в вашей программе приходится нагромождать один оператор `instanceof` на другой, то, скорее всего, изъян кроется в ее архитектуре, и программу следует переписать.

Использование литералов `class`

Если переписать `PetCreator` с использованием литералов `class`, программа получается более компактной:

```

//: typeinfo/pets/LiteralPetCreator.java
// Использование литералов class.
package typeinfo.pets;
import java.util.*;

public class LiteralPetCreator extends PetCreator {
    // Блок try не нужен.
    @SuppressWarnings("unchecked")

```



```

public static final List<Class<? extends Pet>> allTypes =
    Collections.unmodifiableList(Arrays.asList(
        Pet.class, Dog.class, Cat.class, Rodent.class,
        Mutt.class, Pug.class, EgyptianMau.class, Manx.class,
        Cymric.class, Rat.class, Mouse.class, Hamster.class));
// Типы, которые должны создаваться случайным образом:
private static final List<Class<? extends Pet>> types =
    allTypes.subList(allTypes.indexOf(Mutt.class),
        allTypes.size());
public List<Class<? extends Pet>> types() {
    return types;
}
public static void main(String[] args) {
    System.out.println(types);
}
} /* Output:
[class typeinfo.pets.Mutt, class typeinfo.pets.Pug, class
typeinfo.pets.EgyptianMau, class typeinfo.pets.Manx, class
typeinfo.pets.Cymric, class typeinfo.pets.Rat, class
typeinfo.pets.Mouse, class typeinfo.pets.Hamster]
*///:~

```

В следующем примере `PetCount3.java` контейнер `Map` необходимо заранее заполнить всеми типами `Pet` (не только теми, которые будут генерироваться случайным образом), поэтому контейнер `allTypes` необходим. Список `types` содержит часть `allTypes` (созданную вызовом `List.subList()`), которая включает типы `Pet`, используемые для генерирования случайных объектов `Pet`.

На этот раз создание `types` не нужно заключать в блок `try`, потому что обработка осуществляется на стадии компиляции, а следовательно, не приведет к выдаче исключений (в отличие от `Class.forName()`).

Теперь библиотека `typeinfo.pets` содержит две реализации `PetCreator`. Чтобы вторая реализация использовалась по умолчанию, мы можем предоставить реализацию паттерна *Фасад*, использующую `LiteralPetCreator`:

```

//: typeinfo/pets/Pets.java
// Фасад для создания PetCreator по умолчанию.
package typeinfo.pets;
import java.util.*;

public class Pets {
    public static final PetCreator creator =
        new LiteralPetCreator();
    public static Pet randomPet() {
        return creator.randomPet();
    }
    public static Pet[] createArray(int size) {
        return creator.createArray(size);
    }
    public static ArrayList<Pet> arrayList(int size) {
        return creator.arrayList(size);
    }
} ////:~

```

Фасад также обеспечивает опосредованный доступ к `randomPet()`, `createArray()` и `arrayList()`.

Так как метод `PetCount.countPets()` получает аргумент `PetCreator`, мы можем легко протестировать `LiteralPetCreator` (через приведенный выше Фасад):

```
//: typeinfo/PetCount2.java
import typeinfo.pets.*;

public class PetCount2 {
    public static void main(String[] args) {
        PetCount.countPets(Pets.creator);
    }
} /* (Execute to see output) *///:~
```

Программа выводит те же результаты, что и `PetCount.java`.

Динамическая проверка типа

Метод `Class.isInstance()` предоставляет возможность динамической проверки типа объекта. С помощью этого метода в примере `PetCount` наконец-то можно будет избавиться от громоздких конструкций `instanceof`:

```
//: typeinfo/PetCount3.java
// Using isInstance()
import typeinfo.pets.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class PetCount3 {
    static class PetCounter
        extends LinkedHashMap<Class<? extends Pet>, Integer> {
        public PetCounter() {
            super(MapData.map(LiteralPetCreator.allTypes, 0));
        }
        public void count(Pet pet) {
            // Class.isInstance() избавляет от цепочки instanceof:
            for(Map.Entry<Class<? extends Pet>, Integer> pair
                : entrySet())
                if(pair.getKey().isInstance(pet))
                    put(pair.getKey(), pair.getValue() + 1);
        }
        public String toString() {
            StringBuilder result = new StringBuilder("{");
            for(Map.Entry<Class<? extends Pet>, Integer> pair
                : entrySet()) {
                result.append(pair.getKey().getSimpleName());
                result.append("=");
                result.append(pair.getValue());
                result.append(", ");
            }
            result.delete(result.length()-2, result.length());
            result.append("}");
            return result.toString();
        }
    }

    public static void main(String[] args) {
        PetCounter petCount = new PetCounter();
        for(Pet pet : Pets.createArray(20)) {
```

```

        println(pet.getClass().getSimpleName() + " ");
        petCount.count(pet);
    }
    print();
    print(petCount);
}
} /* Output:
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat
EgyptianMau Hamster EgyptianMau Mutt Mutt Cymric Mouse Pug
Mouse Cymric
{Pet=20, Dog=6, Cat=9, Rodent=5, Mutt=3, Pug=3,
EgyptianMau=2, Manx=7, Cymric=5, Rat=2, Mouse=2, Hamster=1}
*///:~

```

Для подсчета всех разновидностей Pet карта PetCounter заполняется типами из LiteralPetCreator.allTypes. При этом используется класс net.mindview.util.MapData, который получает Iterable (контейнер allTypes) и константу (нуль в данном случае) и заполняет карту ключами из allTypes, ассоциированными со значением 0. Без предварительного заполнения Map при подсчете учитывались бы только случайно сгенерированные типы, но не базовые типы вроде Pet и Cat.

Вы можете видеть, что метод isInstance() избавил нас от необходимости использовать громоздкие цепочки instanceof. Вдобавок, теперь в программу можно легко добавить новые типы — для этого следует просто изменить массив LiteralPetCreator.types; остальная часть программы не потребует правки (в отличие от реализации с instanceof).

Метод toString() был перегружен для получения более удобочитаемого результата.

Рекурсивный подсчет

Контейнер Map в примере PetCount3.PetCounter был заранее заполнен разными классами Pet. Вместо предварительного заполнения также можно воспользоваться методом Class.isAssignableFrom() и создать инструмент общего назначения, который не ограничивается подсчетом Pet:

```

//: net/mindview/util/TypeCounter.java
// Подсчет экземпляров семейства типов.
package net.mindview.util;
import java.util.*;

public class TypeCounter extends HashMap<Class<?>,Integer>{
    private Class<?> baseType;
    public TypeCounter(Class<?> baseType) {
        this.baseType = baseType;
    }
    public void count(Object obj) {
        Class<?> type = obj.getClass();
        if(!baseType.isAssignableFrom(type))
            throw new RuntimeException(obj + " неправильный тип: "
                + type + ", должен быть подтипом "
                + baseType);
        countClass(type);
    }
    private void countClass(Class<?> type) {
        Integer quantity = get(type);
        put(type, quantity == null ? 1 : quantity + 1);
    }
}

```

продолжение ➤

```

    Class<?> superClass = type.getSuperclass();
    if(superClass != null &&
        baseType.isAssignableFrom(superClass))
        countClass(superClass);
}
public String toString() {
    StringBuilder result = new StringBuilder("{}");
    for(Map.Entry<Class<?>, Integer> pair : entrySet()) {
        result.append(pair.getKey().getSimpleName());
        result.append("=");
        result.append(pair.getValue());
        result.append(", ");
    }
    result.delete(result.length()-2, result.length());
    result.append("}");
    return result.toString();
}
} ///:~

```

Метод `count()` получает объект `Class` аргумента и при помощи `isAssignableFrom()` проверяет во время выполнения, что переданный объект действительно принадлежит нужной иерархии.

`countClass()` сначала подсчитывает экземпляры реального типа класса. Затем, если `baseType` допускает присваивание из суперкласса, то `countClass()` вызывается рекурсивно для суперкласса:

```

//: typeinfo/PetCount4.java
import typeinfo.pets.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class PetCount4 {
    public static void main(String[] args) {
        TypeCounter counter = new TypeCounter(Pet.class);
        for(Pet pet : Pets.createArray(20)) {
            printnb(pet.getClass().getSimpleName() + " ");
            counter.count(pet);
        }
        print();
        print(counter);
    }
} /* Output: (Sample)
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat
EgyptianMau Hamster EgyptianMau Mutt Mutt Cymric Mouse Pug
Mouse Cymric
{Mouse=2, Dog=6, Manx=7, EgyptianMau=2, Rodent=5, Pug=3,
Mutt=3, Cymric=5, Cat=9, Hamster=1, Pet=20, Rat=2}
*///:~

```

Как видно из выходных данных, подсчитываются как базовые, так и реальные типы.

- 11.** (2) Добавьте класс `Gerbil` в библиотеку `typeinfo.pets` и измените все примеры этой главы так, чтобы в них использовался новый класс.
- 12.** (3) Используйте `TypeCounter` с классом `CoffeeGenerator.java` из главы 15.
- 13.** (3) Используйте `TypeCounter` с примером `RegisteredFactories.java` этой главы.

Зарегистрированные фабрики

У системы генерирования объектов иерархии `Pet` есть один недостаток: каждый раз, когда в иерархию добавляется новый тип `Pet`, вы должны добавить его данные в `LiteralPetCreator.java`. При регулярном добавлении новых классов это требование может создать проблемы.

Напрашивается мысль о добавлении в каждый субкласс статического инициализатора, который будет добавлять свой класс в некий список. К сожалению, статические инициализаторы вызываются только при первой загрузке класса, поэтому возникает классический «порочный круг»: класс отсутствует в списке генератора, в результате генератор не может создать объект этого класса и класс не будет загружен и включен в список.

По сути, вам придется создавать список самостоятельно, вручную (если только вам не захочется написать программу, которая просматривает и анализирует кодовую базу, а затем создает и компилирует список). Вероятно, лучшее, что можно сделать в такой ситуации, — разместить список в каком-то очевидном месте. Пожалуй, для этой цели лучше всего подойдет базовый класс иерархии.

Также мы внесем другое изменение — создание объекта делегируется самому классу с использованием паттерна «Фабричный метод». Фабричный метод может вызываться полиморфно и создает объект нужного типа. В этой очень простой версии «фабричным» является метод `create()` интерфейса `Factory`:

```
//: typeinfo/factory/Factory.java
package typeinfo.factory;
public interface Factory<T> { T create(); } ///:~
```

Параметр-тип `T` позволяет `create()` возвращать разные типы для разных реализаций `Factory`. При этом также используется ковариантность возвращаемых типов.

В этом примере базовый класс `Pet` содержит контейнер `List` с «фабричными» объектами. Фабрики типов, которые должны производиться методом `createRandom()`, «регистрируются» в базовом классе посредством добавления в список `partFactories`:

```
//: typeinfo/RegisteredFactories.java
// Регистрация фабрик класса в базовом классе.
import typeinfo.factory.*;
import java.util.*;

class Part {
    public String toString() {
        return getClass().getSimpleName();
    }
    static List<Factory<? extends Part>> partFactories =
        new ArrayList<Factory<? extends Part>>();
    static {
        // Для Collections.addAll() выдается предупреждение
        // "неконтролируемое создание обобщенного массива"
        partFactories.add(new FuelFilter.Factory());
        partFactories.add(new AirFilter.Factory());
        partFactories.add(new CabinAirFilter.Factory());
        partFactories.add(new OilFilter.Factory());
    }
}
```

продолжение ➤

```

    partFactories.add(new FanBelt.Factory());
    partFactories.add(new PowerSteeringBelt.Factory());
    partFactories.add(new GeneratorBelt.Factory());
}
private static Random rand = new Random(47);
public static Part createRandom() {
    int n = rand.nextInt(partFactories.size());
    return partFactories.get(n).create();
}
}

class Filter extends Part {}

class FuelFilter extends Filter {
    // Создание фабрики класса для каждого конкретного типа:
    public static class Factory
    implements typeinfo.factory.Factory<FuelFilter> {
        public FuelFilter create() { return new FuelFilter(); }
    }
}

class AirFilter extends Filter {
    public static class Factory
    implements typeinfo.factory.Factory<AirFilter> {
        public AirFilter create() { return new AirFilter(); }
    }
}

class CabinAirFilter extends Filter {
    public static class Factory
    implements typeinfo.factory.Factory<CabinAirFilter> {
        public CabinAirFilter create() {
            return new CabinAirFilter();
        }
    }
}

class OilFilter extends Filter {
    public static class Factory
    implements typeinfo.factory.Factory<OilFilter> {
        public OilFilter create() { return new OilFilter(); }
    }
}

class Belt extends Part {}

class FanBelt extends Belt {
    public static class Factory
    implements typeinfo.factory.Factory<FanBelt> {
        public FanBelt create() { return new FanBelt(); }
    }
}

class GeneratorBelt extends Belt {
    public static class Factory
    implements typeinfo.factory.Factory<GeneratorBelt> {
        public GeneratorBelt create() {
            return new GeneratorBelt();
        }
    }
}

```

```
    }  
  }  
}  
  
class PowerSteeringBelt extends Belt {  
    public static class Factory  
    implements typeinfo.factory.Factory<PowerSteeringBelt> {  
        public PowerSteeringBelt create() {  
            return new PowerSteeringBelt();  
        }  
    }  
}  
  
public class RegisteredFactories {  
    public static void main(String[] args) {  
        for(int i = 0; i < 10; i++)  
            System.out.println(Part.createRandom());  
    }  
} /* Output:  
GeneratorBelt  
CabinAirFilter  
GeneratorBelt  
AirFilter  
PowerSteeringBelt  
CabinAirFilter  
FuelFilter  
PowerSteeringBelt  
PowerSteeringBelt  
FuelFilter  
*///:~
```

В данном примере `Filter` и `Belt` существуют только для классификации, поэтому программа не должна создавать экземпляры этих классов, а только их субклассов. Если класс должен создаваться методом `createRandom()`, то он содержит внутренний класс `Factory`. Как видно из примера, имя `Factory` может использоваться только с уточнением `typeinfo.factory.Factory`.

И хотя метод `Collections.addAll()` может использоваться для добавления фабрик в список, компилятор выдает предупреждение о «создании обобщенного массива» (что вроде бы должно быть невозможно, как будет показано в главе 15), поэтому я вернулся к вызову `add()`. Метод `createRandom()` выбирает случайный объект-фабрику из `partFactories` и вызывает его метод `create()` для создания нового объекта `Part`.

- 14.** (4) Конструктор является разновидностью «Фабричного метода». Измените пример `RegisteredFactories.java` так, чтобы вместо использования явно заданной фабрики объект класса сохранялся в `List`, а для создания каждого объекта использовался метод `newInstance()`.
- 15.** (4) Реализуйте новую версию `PetCreator` с использованием регистрации фабрик. Измените фасад `Pets`, чтобы он мог использовать эту версию вместо двух других. Убедитесь в том, что остальные примеры, использующие `Pets.java`, продолжают работать правильно.
- 16.** (4) Измените иерархию `Coffee` в главе 15 так, чтобы в ней использовалась регистрация фабрик.

instanceof и сравнение объектов Class

При получении информации о типе объекта важно различать действие любой формы оператора instanceof (будь это сам оператор instanceof или метод `isInstance()` — они дают одинаковые результаты) и прямого сравнения объектов Class. Следующий пример показывает, чем они отличаются:

```

//: typeinfo/FamilyVsExactType.java
// Различия между instanceof и сравнением Class
package typeinfo;
import static net.mindview.util.Print.*;

class Base {}
class Derived extends Base {}

public class FamilyVsExactType {
    static void test(Object x) {
        print("Тестируем x типа " + x.getClass());
        print("x instanceof Base " + (x instanceof Base));
        print("x instanceof Derived " + (x instanceof Derived));
        print("Base.isInstance(x) " + Base.class.isInstance(x));
        print("Derived.isInstance(x) " +
            Derived.class.isInstance(x));
        print("x.getClass() == Base.class " +
            (x.getClass() == Base.class));
        print("x.getClass() == Derived.class " +
            (x.getClass() == Derived.class));
        print("x.getClass().equals(Base.class) " +
            (x.getClass().equals(Base.class)));
        print("x.getClass().equals(Derived.class) " +
            (x.getClass().equals(Derived.class)));
    }
    public static void main(String[] args) {
        test(new Base());
        test(new Derived());
    }
} /* Output:
Тестируем x типа class typeinfo.Base
x instanceof Base true
x instanceof Derived false
Base.isInstance(x) true
Derived.isInstance(x) false
x.getClass() == Base.class true
x.getClass() == Derived.class false
x.getClass().equals(Base.class) true
x.getClass().equals(Derived.class) false
Тестируем x типа class typeinfo.Derived
x instanceof Base true
x instanceof Derived true
Base.isInstance(x) true
Derived.isInstance(x) true
x.getClass() == Base.class false
x.getClass() == Derived.class true
x.getClass().equals(Base.class) false
x.getClass().equals(Derived.class) true
*///:~

```

Метод `test()` осуществляет проверку типов полученного объекта, используя для этого обе формы оператора instanceof. Затем он получает ссылку на объект Class и использует

операцию сравнения ссылок `==` и метод `equals()`, чтобы проверить объекты `Class` на равенство. Пример доказывает справедливость утверждения о том, что действие оператора `instanceof` и метода `isInstance()` одинаково. Совпадают и результаты работы операции сравнения `==` и метода `equals()`. Но по итогам проведенных проверок можно сделать разные выводы. Результат работы оператора `instanceof` является ответом на вопрос: «Ты принадлежишь этому классу или унаследован от него?» С другой стороны, сравнение объектов `Class` с оператором `==` не затрагивает наследования — типы либо совпадают, либо не совпадают.

Отражение: динамическая информация о классе

Если точный тип объекта неизвестен, RTTI сообщит вам его. Однако в этом случае существуют ограничения: тип должен быть известен еще во время компиляции программы, иначе определить его с помощью RTTI и сделать с этой информацией что-то полезное будет невозможно. Другими словами, компилятор должен располагать информацией обо всех классах, с которыми вы работаете.

Сначала кажется, что ограничение не такое уж серьезное, но предположим, что у вас появилась ссылка на объект, который не находится в пространстве вашей программы. Класс этого объекта недоступен во время ее компиляции. Например, вы получили последовательность байтов с диска или из сетевого соединения и вам сказали, что эта последовательность является некоторым классом. Но компилятор не ведал об этом классе, когда обрабатывал вашу программу, как же его можно использовать?

В традиционных средах программирования такая задача показалась бы надуманной. Однако с расширением границ мира программирования появляются важные ситуации, в которых эта задача требует решения. Во-первых, такие возможности требуются для компонентного программирования, которое служит основой для систем *быстрой разработки приложений* (Rapid Application Development, RAD). Это визуальная среда для создания программ, в которой значки, представляющие визуальные компоненты, перетаскиваются на форму. Затем эти компоненты настраиваются, для чего разработчик задает их свойства во время программирования. Чтобы настроить компоненты до выполнения программы, необходимо каким-то образом создать их экземпляры, просмотреть их содержимое, читать и записывать свойства. Вдобавок компоненты с поддержкой событий графического интерфейса должны как-то рассказать о них, чтобы система быстрой разработки приложений сумела помочь программисту реализовать поддержку этих событий. Механизм отражения предоставляет способы определить доступные методы и их имена. Такое компонентное программирование поддерживается и в Java с помощью технологии JavaBeans (она описана в главе 22).

Другая важная предпосылка поддержки динамической информации о классе — возможность создания и использования объектов на удаленных платформах. Такая возможность называется *удаленным вызовом методов* (Remote Method Invocation, RMI), она позволяет программе на Java распределять свои объекты по нескольким машинам. Удаленный вызов методов обусловлен рядом причин: например, при выполнении задачи с интенсивными вычислениями можно сбалансировать нагрузку по доступным компьютерам. В некоторых ситуациях необходимо разместить код, занимающийся

определенными задачами («бизнес-правила» в многоуровневой архитектуре «клиент—сервер»), на одной машине, чтобы она стала общим хранилищем этих задач, и при изменении кода на такой машине изменения автоматически затронут всех клиентов этого кода. (Интересная ситуация — машина существует исключительно для того, чтобы упростить изменение программного обеспечения!) Ко всему прочему, распределенное программирование также поддерживает удаленное специализированное оборудование, которое эффективно выполняет некоторые задачи — например, обращение матриц, — которые при решении их на локальной машине могут потребовать слишком много времени и ресурсов.

Класс `Class` (уже описанный в этой главе) вводит понятие *отражения* (reflection), для которого существует дополнительная библиотека `java.lang.reflect`, состоящая из классов `Field`, `Method` и `Constructor` (каждый реализует интерфейс `Member`). Объекты этих классов создаются JVM, чтобы представлять соответствующие члены неизвестного класса. Вы можете использовать объекты `Constructor` для создания новых объектов класса, методы `get()` и `set()` — для чтения и записи значений полей класса, представленных объектами `Field`, метод `invoke()` — для вызова метода, представленного объектом `Method`. Вдобавок, в классе `Class` имеются удобные методы `getFields()`, `getMethods()` и `getConstructors()`, которые возвращают массивы таких объектов, как поля класса, его методы и конструкторы. (Узнать о методах класса `Class` подробнее можно в интерактивной документации JDK.) Таким образом, информация о неизвестном объекте становится доступной прямо во время выполнения программы, а потребность в ее получении ко времени компиляции программы отпадает.

Важно понимать, что в механизме отражения нет ничего сверхъестественного. Когда вы используете отражение для общения с объектом неизвестного типа, виртуальная машина JVM рассматривает его и видит, что он принадлежит определенному классу (это делает и обычное RTTI), но перед тем как проводить с ним некоторые действия, необходимо загрузить соответствующий объекту файл `.class`. Таким образом, файл `.class` для класса этого объекта должен быть доступен JVM либо в сети, либо в локальной системе. Отсюда заключение: разница между традиционным RTTI и отражением в том, что при использовании RTTI файл `.class` открывается и изучается компилятором. Другими словами, методы объекта можно вызывать «обычным» способом. Когда вы работаете с отражением, файл `.class` во время компиляции недоступен; он открывается и обрабатывается системой выполнения.

Извлечение информации о методах класса

Отражение редко используется напрямую; оно существует в языке в основном для поддержки других возможностей, таких как сериализация объектов (глава 18), компоненты `JavaBeans` (глава 22). Однако существуют ситуации, в которых динамическая информация о классе просто незаменима.

Например, программа, выводящая на экран список методов определенного класса, будет очень полезна. Как уже упоминалось, при просмотре исходного текста класса или его документации напрямую видны только те методы, которые были определены или переопределены именно *в текущем классе*. Но в классе может быть еще множество методов, доступных из его базовых классов. Найти их достаточно сложно, и к тому

же это долго¹. К счастью, с помощью отражения нам под силу написать простой инструмент, который будет показывать полный интерфейс класса. Вот как он работает:

```
//: typeinfo/ShowMethods.java
// Использование отражения для вывода на экран всех методов
// класса, включая те, что были определены в базовых классах.
// {Args: ShowMethods}
import java.lang.reflect.*;
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class ShowMethods {
    private static String usage =
        "usage:\n" +
        "ShowMethods qualified.class.name\n" +
        "To show all methods in class or:\n" +
        "ShowMethods qualified.class.name word\n" +
        "To search for methods involving 'word'";
    private static Pattern p = Pattern.compile("\\w+\\.");
    public static void main(String[] args) {
        if(args.length < 1) {
            print(usage);
            System.exit(0);
        }
        int lines = 0;
        try {
            Class<> c = Class.forName(args[0]);
            Method[] methods = c.getMethods();
            Constructor[] ctors = c.getConstructors();
            if(args.length == 1) {
                for(Method method : methods)
                    print(
                        p.matcher(method.toString()).replaceAll(""));
                for(Constructor ctor : ctors)
                    print(p.matcher(ctor.toString()).replaceAll(""));
                lines = methods.length + ctors.length;
            } else {
                for(Method method : methods)
                    if(method.toString().indexOf(args[1]) != -1) {
                        print(
                            p.matcher(method.toString()).replaceAll(""));
                        lines++;
                    }
                for(Constructor ctor : ctors)
                    if(ctor.toString().indexOf(args[1]) != -1) {
                        print(p.matcher(
                            ctor.toString()).replaceAll(""));
                        lines++;
                    }
            }
        } catch(ClassNotFoundException e) {
            print("No such class: " + e);
        }
    }
}
```

продолжение ➤

¹ Сказанное относится в основном к документации ранних версий Java. Теперь фирма Sun значительно улучшила HTML-документацию Java и найти методы базовых классов стало проще.

```

} /* Output:
public static void main(String[])
public native int hashCode()
public final native Class getClass()
public final void wait(long,int) throws
InterruptedException
public final void wait() throws InterruptedException
public final native void wait(long) throws
InterruptedException
public boolean equals(Object)
public String toString()
public final native void notify()
public final native void notifyAll()
public ShowMethods()
*///:~

```

Методы класса `Class` `getMethods()` и `getConstructors()` возвращают массивы объектов `Method` и `Constructor`, которые представляют методы и конструкторы класса. В каждом из этих классов есть методы, позволяющие извлечь и проанализировать имена, аргументы и возвращаемые значения представляемых методов и конструкторов. Однако можно использовать простой метод `toString()`, как и сделано здесь, чтобы получить строку с полным именем метода. Остальная часть кода разбирает командную строку и определяет, подходит ли определенное выражение образцу для поиска (с использованием `indexOf()`), а после выделяет описатели имен классов с использованием регулярных выражений (см. главу 13).

Результат, возвращаемый методом `Class.forName()`, неизвестен на стадии компиляции, поэтому вся информация сигнатуры метода извлекается во время выполнения. Если вы тщательно изучите документацию отражения, входящую в поставку JDK, то увидите, что оно предоставляет достаточно возможностей, чтобы установить необходимые аргументы и вызвать метод объекта, о котором ничего не известно во время компиляции программы (чуть позже будут соответствующие примеры). Скорее всего, вам этого делать никогда не придется, но сама возможность весьма интересна.

Приведенный выше результат был получен при запуске программы командой

```
java ShowMethods ShowMethods
```

В результате мы видим, что класс содержит открытый (`public`) конструктор по умолчанию, хотя в тексте программы такой конструктор не определяется. Тот конструктор, что имеется теперь в классе, автоматически синтезирован компилятором. Если вы после этого сделаете класс `ShowMethods` не открытым (удалите из его определения спецификатор доступа `public`, то есть предоставите ему доступ в пределах пакета»), то синтезированный компилятором конструктор больше не появится в списке методов. Синтезируемый конструктор имеет тот же уровень доступа, что и его класс.

Также интересно может оказаться запустить программу командой

```
java ShowMethods java,lang.String
```

указав ей при этом дополнительные параметры `char`, `int`, `String` и т. п.

Эта программа может сэкономить немало времени при программировании, когда вы будете долго и мучительно вспоминать, есть ли у этого класса определенный метод.

Она придет на помощь, если вам не хочется рыскать по иерархиям классов в интерактивной документации или когда у вас возникнет желание узнать, например, о том, есть ли у некоторого класса методы, которые возвращают объекты `Color`.

В главе 22 создается еще одна версия этой программы, на этот раз с графическим интерфейсом (специально адаптированная для графических компонентов библиотеки `Swing`), что позволяет держать ее постоянно в памяти и при необходимости быстро разыскивать методы какого-либо класса.

17. (2) Измените регулярное выражение в программе `ShowMethods.java` так, чтобы оно дополнительно выделяло ключевые слова `native` и `final` (подсказка: используйте оператор ИЛИ — `|`).
18. (1) Отмените объявление `ShowMethods` как открытого (`public`) класса. Убедитесь в том, что синтезированный конструктор по умолчанию пропадает из результатов.
19. (4) В программе `ToyTest.java` используйте отражение для создания объекта `Toy` конструктором с аргументами.
20. (5) Найдите описание класса `java.lang.Class` в документации JDK (<http://java.sun.com>). Напишите программу, которая получает имя класса в параметре командной строки, а затем использует методы класса `Class` для вывода всей доступной информации о классе. Протестируйте программу на одном из классов стандартной библиотеки Java и на своем собственном классе.

Динамические заместители

Заместитель (проху) является одним из основных паттернов проектирования. Он представляет собой объект, который подставляется на место «настоящего» объекта для предоставления дополнительных или других операций — обычно подразумевающих взаимодействие с «настоящим» объектом, поэтому заместитель чаще всего выполняет функции посредника. Простейший пример, демонстрирующий структуру заместителя:

```
//: typeinfo/SimpleProxyDemo.java
import static net.mindview.util.Print.*;

interface Interface {
    void doSomething();
    void somethingElse(String arg);
}

class RealObject implements Interface {
    public void doSomething() { print("doSomething"); }
    public void somethingElse(String arg) {
        print("somethingElse " + arg);
    }
}

class SimpleProxy implements Interface {
    private Interface proxied;
    public SimpleProxy(Interface proxied) {
        this.proxied = proxied;
    }
}
```

```

    public void doSomething() {
        print("SimpleProxy doSomething");
        proxied.doSomething();
    }
    public void somethingElse(String arg) {
        print("SimpleProxy somethingElse " + arg);
        proxied.somethingElse(arg);
    }
}

class SimpleProxyDemo {
    public static void consumer(Interface iface) {
        iface.doSomething();
        iface.somethingElse("bonobo");
    }
    public static void main(String[] args) {
        consumer(new RealObject());
        consumer(new SimpleProxy(new RealObject()));
    }
} /* Output:
doSomething
somethingElse bonobo
SimpleProxy doSomething
doSomething
SimpleProxy somethingElse bonobo
somethingElse bonobo
*///:~

```

Так как метод `consumer()` получает `Interface`, он не может знать, получает ли он `RealObject` или `Proxy`, поскольку оба класса реализуют `Interface`. Объект `Proxy`, вставленный между клиентом и `RealObject`, выполняет операции и вызывает идентичный метод `RealObject`.

Заместитель может пригодиться в любой ситуации, в которой требуется включить дополнительные операции в любом месте, отличном от «настоящего» объекта, особенно если вы хотите иметь возможность легко переключаться между использованием и игнорированием дополнительных операций (одной из целей паттернов проектирования является инкапсуляция изменений; следовательно, при применении паттерна что-то должно изменяться). Что делать, если вы, допустим, хотите отслеживать вызовы методов `RealObject` или измерить затраты на такие вызовы? Вставлять такой код в свое приложение не стоит, а заместитель позволяет легко добавить и удалить его при необходимости.

Динамические заместители Java продвигают концепцию заместителя на один шаг вперед: объект заместителя создается динамически, а вызовы замещенных методов тоже обрабатываются динамически. Все вызовы, обращенные к динамическому заместителю, перенаправляются одному обработчику, который идентифицирует вызовы и решает, что с ними делать. Ниже приведен пример `SimpleProxyDemo.java`, переписанный для использования динамического заместителя:

```

//: typeinfo/SimpleDynamicProxy.java
import java.lang.reflect.*;

class DynamicProxyHandler implements InvocationHandler {
    private Object proxied;
    public DynamicProxyHandler(Object proxied) {

```

```

    this.proxied = proxied;
}
public Object
invoke(Object proxy, Method method, Object[] args)
throws Throwable {
    System.out.println("**** proxy: " + proxy.getClass() +
        ", method: " + method + ", args: " + args);
    if(args != null)
        for(Object arg : args)
            System.out.println(" " + arg);
    return method.invoke(proxied, args);
}
}

class SimpleDynamicProxy {
    public static void consumer(Interface iface) {
        iface.doSomething();
        iface.somethingElse("bonobo");
    }
    public static void main(String[] args) {
        RealObject real = new RealObject();
        consumer(real);
        // Insert a proxy and call again:
        Interface proxy = (Interface)Proxy.newProxyInstance(
            Interface.class.getClassLoader(),
            new Class[]{ Interface.class },
            new DynamicProxyHandler(real));
        consumer(proxy);
    }
} /* Output: (95% match)
doSomething
somethingElse bonobo
**** proxy: class $Proxy0, method: public abstract void
Interface.doSomething(), args: null
doSomething
**** proxy: class $Proxy0, method: public abstract void
Interface.somethingElse(java.lang.String), args:
[Ljava.lang.Object;@42e816
    bonobo
somethingElse bonobo
*///:~

```

Динамический заместитель создается вызовом статического метода `Proxy.newProxyInstance()`, которому передается загрузчик классов (как правило, можно просто передать загрузчик из уже загруженного объекта), список интерфейсов (не классов и не абстрактных классов), которые должны реализоваться заместителем, и обработчик вызовов (реализация интерфейса `InvocationHandler`). Динамический посредник перенаправляет все вызовы обработчику, так что конструктор обработчика вызовов обычно получает ссылку на «настоящий» объект для перенаправления запросов после выполнения его промежуточной операции.

Метод `invoke()` получает объект-заместитель на тот случай, если обработка должна зависеть от источника запроса, но чаще источник игнорируется. Впрочем, будьте внимательны при вызове методов заместителя в `invoke()`, потому что вызовы через интерфейс перенаправляются через заместителя.

В общем случае выполняется промежуточная операция, после чего вызов `Method.invoke()` перенаправляет запрос замещаемому объекту с передачей необходимых аргументов. На первый взгляд кажется, что это создает ограничения, поскольку вы можете выполнять только общие операции. Однако программа может отфильтровывать некоторые вызовы, пропуская другие:

```

//: typeinfo/SelectingMethods.java
// Поиск конкретных методов в динамическом заместителе.
import java.lang.reflect.*;
import static net.mindview.util.Print.*;

class MethodSelector implements InvocationHandler {
    private Object proxied;
    public MethodSelector(Object proxied) {
        this.proxied = proxied;
    }
    public Object
    invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
        if(method.getName().equals("interesting"))
            print("Proxy detected the interesting method");
        return method.invoke(proxied, args);
    }
}

interface SomeMethods {
    void boring1();
    void boring2();
    void interesting(String arg);
    void boring3();
}

class Implementation implements SomeMethods {
    public void boring1() { print("boring1"); }
    public void boring2() { print("boring2"); }
    public void interesting(String arg) {
        print("interesting " + arg);
    }
    public void boring3() { print("boring3"); }
}

class SelectingMethods {
    public static void main(String[] args) {
        SomeMethods proxy= (SomeMethods)Proxy.newProxyInstance(
            SomeMethods.class.getClassLoader(),
            new Class[]{ SomeMethods.class },
            new MethodSelector(new Implementation()));
        proxy.boring1();
        proxy.boring2();
        proxy.interesting("bonobo");
        proxy.boring3();
    }
} /* Output:
boring1
boring2
Proxy detected the interesting method
interesting bonobo
boring3
*///:~

```


В данном случае проверяются имена методов, но также можно было проверять другие аспекты сигнатуры и даже конкретные значения аргументов.

Динамический заместитель — не тот инструмент, который вы будете использовать ежедневно, но он очень хорошо подходит для решения некоторых видов задач. Паттерн «Заместитель» и другие паттерны проектирования представлены в книгах «Thinking in Patterns» (см. www.MindView.net) и «Приемы объектно-ориентированного проектирования» Эрика Гаммы и др. (Питер, 2013).

21. (3) Измените пример `SimpleProxyDemo.java`, чтобы в нем измерялось время вызова методов.
22. (3) Измените пример `SimpleDynamicProxy.java`, чтобы в нем измерялось время вызова методов.
23. (3) Внутри метода `invoke()` из примера `SimpleDynamicProxy.java` попробуйте вывести аргумент-заместитель. Объясните, что при этом происходит.

Проект¹. Напишите систему, использующую динамические заместители для реализации транзакций: заместитель закрепляет транзакцию, если опосредованный вызов выполнен успешно (т. е. не возбудил исключений) или выполняет отмену в случае неудачи. Закрепление и отмена должны работать для внешних текстовых файлов, что выходит за границы исключений Java. Уделите особое внимание атомарности операций.

Null-объекты

Если отсутствие объекта представляется встроенным значением `null`, то ссылку при каждом использовании приходится проверять на `null`. Многочисленные проверки утомительны, а код становится громоздким. Проблема в том, что `null` не имеет собственного поведения (если не считать возбуждения исключения `NullPointerException` при попытке выполнения операции). Иногда в таких ситуациях помогает концепция `null`-объекта², принимающего сообщения для объекта, который он «изображает», но возвращающего значения, указывающие на отсутствие «настоящего» объекта. Разработчик может считать, что все объекты действительны и ему не нужно тратить время на проверки `null` (а также чтение соответствующего кода).

И хотя было бы занятно представить язык программирования, который автоматически создает `null`-объекты за вас, на практике их повсеместное использование не имеет смысла — иногда проверки на `null` вполне достаточно, иногда можно с достаточным основанием полагать, что ссылка отлична от `null`, а иногда допустимы даже решения с выявлением аномальных ситуаций через `NullPointerException`. Похоже, `null`-объекты приносят наибольшую пользу «вблизи к данным» — для объектов, представляющих

¹ Проекты могут использоваться в процессе обучения в качестве курсовых работ.

² Этот паттерн, предложенный Бобби Вульфом и Брюсом Андерсоном, может рассматриваться как частный случай паттерна «Стратегия». Разновидностью паттерна «Null-объект» является паттерн «Null-итератор», который делает перебор узлов составной иерархии прозрачным с точки зрения клиента (клиент может использовать одну логику для перебора составных и листовых узлов).

сущности в пространстве задачи. Простой пример: во многих системах существует класс `Person`, представляющий человека. Если в коде информация о человеке недоступна (или недостаточна), традиционно программист использует `null` в качестве значения ссылки. Вместо этого также можно создать `null`-объект. Но даже при том, что `null`-объект реагирует на все сообщения, на которые реагирует «настоящий» объект, должен существовать какой-то способ проверки «`null`-объектности». Проще всего воспользоваться идентификационным интерфейсом:

```
//: net/mindview/util/Null.java
package net.mindview.util;
public interface Null {} ///:~
```

Такое решение позволяет использовать `instanceof` для обнаружения `null`-объектов и, что еще важнее, не требует включения метода `isNull` во все классы (который по сути будет моделировать RTTI — почему не воспользоваться встроенными средствами?).

```
//: typeinfo/Person.java
// Класс с null-объектом.
import net.mindview.util.*;

class Person {
    public final String first;
    public final String last;
    public final String address;
    // И т.д.
    public Person(String first, String last, String address){
        this.first = first;
        this.last = last;
        this.address = address;
    }
    public String toString() {
        return "Person: " + first + " " + last + " " + address;
    }
    public static class NullPerson
        extends Person implements Null {
        private NullPerson() { super("None", "None", "None"); }
        public String toString() { return "NullPerson"; }
    }
    public static final Person NULL = new NullPerson();
} ///:~
```

В общем случае `null`-объект будет одиночным объектом (синглетом), поэтому в данном примере он создается как статический неизменяемый (`static final`) экземпляр. Такой подход работает, потому что объект `Person` не может модифицироваться — вы можете задать значения в конструкторе и затем прочитать их, но не можете изменить (потому что объекты `String` по своей природе не модифицируются). Если вам потребуется изменить объект `NullPerson`, вы сможете только заменить его новым объектом `Person`. Обратите внимание: при использовании `instanceof` у вас имеется возможность обнаружения общего интерфейса `Null` или более конкретного класса `NullPerson`, но в ситуации с одиночным объектом также можно воспользоваться вызовом `equals()` или даже `==` для сравнения с `Person.NULL`.

Представьте, что вы вернулись в эпоху молодых и амбиционных интернет-компаний и выбили финансирование для своей Потрясающей Идеи. Теперь можно переходить

к набору персонала. Пока в штате остаются незаполненные вакансии, в качестве «заполнителя» каждой вакансии (Position) используется null-объект Person:

```
//: typeinfo/Position.java
```

```
class Position {
    private String title;
    private Person person;
    public Position(String jobTitle, Person employee) {
        title = jobTitle;
        person = employee;
        if(person == null)
            person = Person.NULL;
    }
    public Position(String jobTitle) {
        title = jobTitle;
        person = Person.NULL;
    }
    public String getTitle() { return title; }
    public void setTitle(String newTitle) {
        title = newTitle;
    }
    public Person getPerson() { return person; }
    public void setPerson(Person newPerson) {
        person = newPerson;
        if(person == null)
            person = Person.NULL;
    }
    public String toString() {
        return "Position: " + title + " " + person;
    }
} ///:~
```

Использовать null-объект для Position не нужно, поскольку существование Person.NULL подразумевает, что вакансия свободна (возможно, позднее выяснится необходимость в добавлении явной реализации null-объекта для Position, но принцип YAGNI¹ (You Aren't Going to Need It) требует использовать в прототипе «самое простое решение, которое может сработать», и добавлять дополнительные возможности лишь при появлении реальной необходимости).

Класс Staff теперь может проверять null-объекты при заполнении вакансий:

```
//: typeinfo/Staff.java
import java.util.*;
```

```
public class Staff extends ArrayList<Position> {
    public void add(String title, Person person) {
        add(new Position(title, person));
    }
    public void add(String... titles) {
        for(String title : titles)
            add(new Position(title));
    }
}
```

продолжение ➤

¹ Один из канонических принципов экстремального программирования.

```

public Staff(String... titles) { add(titles); }
public boolean positionAvailable(String title) {
    for(Position position : this)
        if(position.getTitle().equals(title) &&
            position.getPerson() == Person.NULL)
            return true;
    return false;
}
public void fillPosition(String title, Person hire) {
    for(Position position : this)
        if(position.getTitle().equals(title) &&
            position.getPerson() == Person.NULL) {
            position.setPerson(hire);
            return;
        }
    throw new RuntimeException(
        "Position " + title + " not available");
}
public static void main(String[] args) {
    Staff staff = new Staff("President", "CTO",
        "Marketing Manager", "Product Manager",
        "Project Lead", "Software Engineer",
        "Software Engineer", "Software Engineer",
        "Software Engineer", "Test Engineer",
        "Technical Writer");
    staff.fillPosition("President",
        new Person("Me", "Last", "The Top, Lonely At"));
    staff.fillPosition("Project Lead",
        new Person("Janet", "Planner", "The Burbs"));
    if(staff.positionAvailable("Software Engineer"))
        staff.fillPosition("Software Engineer",
            new Person("Bob", "Coder", "Bright Light City"));
    System.out.println(staff);
}
} /* Output:
[Position: President Person: Me Last The Top, Lonely At,
Position: CTO NullPerson, Position: Marketing Manager
NullPerson, Position: Product Manager NullPerson, Position:
Project Lead Person: Janet Planner The Burbs, Position:
Software Engineer Person: Bob Coder Bright Light City,
Position: Software Engineer NullPerson, Position: Software
Engineer NullPerson, Position: Software Engineer
NullPerson, Position: Test Engineer NullPerson, Position:
Technical Writer NullPerson]
*///:~

```

Обратите внимание: в некоторых местах по-прежнему приходится проверять null-объекты, что не так уж сильно отличается от проверки null-ссылок, но в других местах (например, в преобразованиях toString()) лишние проверки не нужны; вы можете просто предположить, что все ссылки на объекты являются действительными.

Если вы работаете с интерфейсами вместо конкретных классов, для автоматического создания null-объектов можно использовать динамический заместитель. Допустим, интерфейс Robot определяет название, модель и список List<Operation> с описаниями функций робота. Объект Operation содержит описание и команду (разновидность паттерна «Команда»):

```
//: typeinfo/Operation.java
```

```
public interface Operation {
    String description();
    void command();
} ///:~
```

Обращение к функциональности Robot осуществляется вызовом `operations()`:

```
//: typeinfo/Robot.java
```

```
import java.util.*;
```

```
import net.mindview.util.*;
```

```
public interface Robot {
    String name();
    String model();
    List<Operation> operations();
}

class Test {
    public static void test(Robot r) {
        if(r instanceof Null)
            System.out.println("[Null Robot]");
        System.out.println("Robot name: " + r.name());
        System.out.println("Robot model: " + r.model());
        for(Operation operation : r.operations()) {
            System.out.println(operation.description());
            operation.command();
        }
    }
}
} ///:~
```

В этом решении также задействован вложенный класс для выполнения тестов.

Теперь мы можем создать класс, представляющий робота для уборки снега:

```
//: typeinfo/SnowRemovalRobot.java
```

```
import java.util.*;
```

```
public class SnowRemovalRobot implements Robot {
    private String name;
    public SnowRemovalRobot(String name) { this.name = name;}
    public String name() { return name; }
    public String model() { return "SnowBot Series 11"; }
    public List<Operation> operations() {
        return Arrays.asList(
            new Operation() {
                public String description() {
                    return name + " can shovel snow";
                }
                public void command() {
                    System.out.println(name + " shoveling snow");
                }
            },
            new Operation() {
                public String description() {
                    return name + " can chip ice";
                }
                public void command() {

```

продолжение ➤

```

        System.out.println(name + " chipping ice");
    }
},
new Operation() {
    public String description() {
        return name + " can clear the roof";
    }
    public void command() {
        System.out.println(name + " clearing roof");
    }
}
);
}
public static void main(String[] args) {
    Robot.Test.test(new SnowRemovalRobot("Slusher"));
}
} /* Output:
Robot name: Slusher
Robot model: SnowBot Series 11
Slusher can shovel snow
Slusher shoveling snow
Slusher can chip ice
Slusher chipping ice
Slusher can clear the roof
Slusher clearing roof
*///:~

```

Можно предположить, что в программе будет использоваться много разновидностей `Robot`, и `null`-объект должен делать что-то особенное для каждого типа `Robot` — в данном случае это означает включение информации о точном типе `Robot`. Эта информация сохраняется динамическим заместителем:

```

//: typeinfo/NullRobot.java
// Использование динамического заместителя для создания null-объекта.
import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;

class NullRobotProxyHandler implements InvocationHandler {
    private String nullName;
    private Robot proxied = new NRobot();
    NullRobotProxyHandler(Class<? extends Robot> type) {
        nullName = type.getSimpleName() + " NullRobot";
    }
    private class NRobot implements Null, Robot {
        public String name() { return nullName; }
        public String model() { return nullName; }
        public List<Operation> operations() {
            return Collections.emptyList();
        }
    }
    public Object
    invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
        return method.invoke(proxied, args);
    }
}

```

```

public class NullRobot {
    public static Robot
    newNullRobot(Class<? extends Robot> type) {
        return (Robot)Proxy.newProxyInstance(
            NullRobot.class.getClassLoader(),
            new Class[]{ Null.class, Robot.class },
            new NullRobotProxyHandler(type));
    }
    public static void main(String[] args) {
        Robot[] bots = {
            new SnowRemovalRobot("SnowBee"),
            new NullRobot(SnowRemovalRobot.class)
        };
        for(Robot bot : bots)
            Robot.Test.test(bot);
    }
} /* Output:
Robot name: SnowBee
Robot model: SnowBot Series 11
SnowBee can shovel snow
SnowBee shoveling snow
SnowBee can chip ice
SnowBee chipping ice
SnowBee can clear the roof
SnowBee clearing roof
[Null Robot]
Robot name: SnowRemovalRobot NullRobot
Robot model: SnowRemovalRobot NullRobot
*///:~

```

Когда вам понадобится null-объект Robot, вы просто вызовете `newNullRobot()`, передавая тип Robot, для которого нужен заместитель. Заместитель выполняет требования к интерфейсам Robot и Null и предоставляет конкретное имя замещаемого типа.

Фиктивные объекты и заглушки

.Логические вариации на тему null-объекта — фиктивный объект (Mock Object) и заглушка (Stub). Как и null-объект, обе модели являются «дублерами» для настоящего объекта, который будет использоваться в программе. Однако и фиктивные объекты, и заглушки «притворяются» полноценными объектами, которые поставляют реальную информацию (в отличие от интеллектуальных суррогатов для null).

Различия между фиктивными объектами и заглушками скорее количественные, а не качественные. Фиктивные объекты обычно легковесны, самотестируемы, и обычно для разных ситуаций тестирования в программе создается несколько таких объектов. Заглушки просто возвращают представляемые данные, «вешают» больше и повторно используются между тестами. Заглушки можно настроить так, чтобы их поведение изменялось в зависимости от способа вызова. Таким образом, заглушка представляет собой сложный объект, который реализует много разных функций, тогда как для выполнения разнообразных функций обычно создается много простых, небольших фиктивных объектов.

24. (4) Добавьте null-объекты в `RegisteredFactories.java`.

Интерфейсы и информация типов

Одной из важных целей ключевого слова `interface` является возможность изоляции компонентов, снижающая *связность* системы. Программирование на уровне интерфейсов позволяет достичь этой цели, но информация типа позволяет обойти это ограничение — интерфейсы не обеспечивают стопроцентной гарантии снижения связности. Следующий пример начинается с определения интерфейса:

```
//: typeinfo/interfacea/A.java
package typeinfo.interfacea;
```

```
public interface A {
    void f();
} ///:~
```

Затем интерфейс реализуется, но разработчик при желании может добраться до фактического типа реализации:

```
//: typeinfo/InterfaceViolation.java
// Программирование в обход интерфейса.
import typeinfo.interfacea.*;

class B implements A {
    public void f() {}
    public void g() {}
}

public class InterfaceViolation {
    public static void main(String[] args) {
        A a = new B();
        a.f();
        // a.g(); // Ошибка компиляции
        System.out.println(a.getClass().getName());
        if(a instanceof B) {
            B b = (B)a;
            b.g();
        }
    }
} /* Output:
B
*///:~
```

Используя RTTI, мы обнаруживаем, что интерфейс `a` реализуется классом `B`. Выполняя приведение типа `k B`, мы можем вызвать метод, отсутствующий в `A`. Все это абсолютно законно и допустимо, но возможно, вам бы не хотелось предоставлять такую возможность программистам-клиентам; ведь тем самым в программе образуется более высокий уровень связности, чем вам хотелось бы. Другими словами, ключевое слово `interface` на самом деле ни от чего не защищает, а факт использования `B` для реализации `A` в данном случае по сути является обычной декларацией¹.

¹ Самый знаменитый пример — операционная система Windows с программным интерфейсом (API), который должен использоваться программистами, и видимым набором недокументированных функций, которые программист при желании может обнаружить и вызвать. Программисты использовали недокументированные функции API, из-за чего компании Microsoft пришлось сопровождать их так, словно они являются частью доступного API. Это обернулось для Microsoft значительными затратами.

Возможное решение проблемы — просто заявить, что если программист решает использовать фактический класс вместо интерфейса, то он действует на свой страх и риск. Вероятно, это разумно во многих случаях, но если «вероятно» вас не устраивает, можно применить более жесткие меры.

Проще всего использовать для реализации доступ на уровне пакета, чтобы внешние клиенты не могли видеть ее:

```
//: typeinfo/packageaccess/HiddenC.java
package typeinfo.packageaccess;
import typeinfo.interfacea.*;
import static net.mindview.util.Print.*;

class C implements A {
    public void f() { print("public C.f()"); }
    public void g() { print("public C.g()"); }
    void u() { print("package C.u()"); }
    protected void v() { print("protected C.v()"); }
    private void w() { print("private C.w()"); }
}

public class HiddenC {
    public static A makeA() { return new C(); }
} ///:~
```

Единственная открытая (`public`) часть этого пакета `HiddenC` при вызове создает интерфейс `A`. Здесь интересно то, что даже если `makeA()` будет возвращать `C`, за пределами пакета не удастся использовать ничего, кроме `A`, так как имя `C` вне пакета будет недоступно.

Теперь при попытке нисходящего преобразования к `C` ничего не выйдет, потому что за пределами пакета тип `C` недоступен:

```
//: typeinfo/HiddenImplementation.java
// Sneaking around package access.
import typeinfo.interfacea.*;
import typeinfo.packageaccess.*;
import java.lang.reflect.*;

public class HiddenImplementation {
    public static void main(String[] args) throws Exception {
        A a = HiddenC.makeA();
        a.f();
        System.out.println(a.getClass().getName());
        // Ошибка компиляции: не удастся найти имя 'C':
        /* if(a instanceof C) {
            C c = (C)a;
            c.g();
        } */
        // Сюрприз! Отражение позволяет вызвать g():
        callHiddenMethod(a, "g");
        // И даже менее доступные методы!
        callHiddenMethod(a, "u");
        callHiddenMethod(a, "v");
        callHiddenMethod(a, "w");
    }
}
```

```

static void callHiddenMethod(Object a, String methodName)
throws Exception {
    Method g = a.getClass().getDeclaredMethod(methodName);
    g.setAccessible(true);
    g.invoke(a);
}
} /* Output:
public C.f()
typeinfo.packageaccess.C
public C.g()
package C.u()
protected C.v()
private C.w()
*///:~

```

Как видите, рефлексия позволяет добраться до любых методов и вызвать даже закрытые методы! Зная имя метода, можно вызвать `setAccessible(true)` для объекта `Method` (как это делается в `callHiddenMethod()`).

Возможно, вы думаете, что проблема решается распространением только откомпилированного кода, но это не так. Достаточно запустить `javap` — декомпилятор, входящий в поставку JDK. Командная строка выглядит так:

```
javap -private C
```

Флаг `-private` означает, что программа должна выводить все члены, даже закрытые. Результат выглядит так:

```

class typeinfo.packageaccess.C extends
java.lang.Object implements typeinfo.interfacea.A {
    typeinfo.packageaccess.C();
    public void f();
    public void g();
    void u();
    protected void v();
    private void w();
}

```

Итак, кто угодно может узнать имена и сигнатуры ваших самых закрытых методов и вызвать их.

А если реализовать интерфейс как закрытый внутренний класс? Вот как это выглядит:

```

//: typeinfo/InnerImplementation.java
// Закрытые внутренние классы не скрыты от отражения.
import typeinfo.interfacea.*;
import static net.mindview.util.Print.*;

class InnerA {
    private static class C implements A {
        public void f() { print("public C.f()"); }
        public void g() { print("public C.g()"); }
        void u() { print("package C.u()"); }
        protected void v() { print("protected C.v()"); }
        private void w() { print("private C.w()"); }
    }
    public static A makeA() { return new C(); }
}

```

```

public class InnerImplementation {
    public static void main(String[] args) throws Exception {
        A a = InnerA.makeA();
        a.f();
        System.out.println(a.getClass().getName());
        // Reflection still gets into the private class:
        HiddenImplementation.callHiddenMethod(a, "g");
        HiddenImplementation.callHiddenMethod(a, "u");
        HiddenImplementation.callHiddenMethod(a, "v");
        HiddenImplementation.callHiddenMethod(a, "w");
    }
} /* Output:
public C.f()
InnerA$C
public C.g()
package C.u()
protected C.v()
private C.w()
*///:~

```

Ничего скрыть от отражения не удалось. Как насчет анонимного класса?

```

//: typeinfo/AnonymousImplementation.java
// Анонимные внутренние классы не скрыты от отражения.
import typeinfo.interfacea.*;
import static net.mindview.util.Print.*;

class AnonymousA {
    public static A makeA() {
        return new A() {
            public void f() { print("public C.f()"); }
            public void g() { print("public C.g()"); }
            void u() { print("package C.u()"); }
            protected void v() { print("protected C.v()"); }
            private void w() { print("private C.w()"); }
        };
    }
}

public class AnonymousImplementation {
    public static void main(String[] args) throws Exception {
        A a = AnonymousA.makeA();
        a.f();
        System.out.println(a.getClass().getName());
        // Отражение позволяет получить доступ к анонимному классу:
        HiddenImplementation.callHiddenMethod(a, "g");
        HiddenImplementation.callHiddenMethod(a, "u");
        HiddenImplementation.callHiddenMethod(a, "v");
        HiddenImplementation.callHiddenMethod(a, "w");
    }
} /* Output:
AnonymousA$1
public C.g()
package C.u()
protected C.v()
private C.w()
*///:~

```

Похоже, не существует способа помешать отражению получить доступ и вызвать методы с ограниченным доступом. Это относится и к полям, даже к закрытым:

```
//: typeinfo/ModifyingPrivateFields.java
import java.lang.reflect.*;

class WithPrivateFinalField {
    private int i = 1;
    private final String s = "I'm totally safe";
    private String s2 = "Am I safe?";
    public String toString() {
        return "i = " + i + ", " + s + ", " + s2;
    }
}

public class ModifyingPrivateFields {
    public static void main(String[] args) throws Exception {
        WithPrivateFinalField pf = new WithPrivateFinalField();
        System.out.println(pf);
        Field f = pf.getClass().getDeclaredField("i");
        f.setAccessible(true);
        System.out.println("f.getInt(pf): " + f.getInt(pf));
        f.setInt(pf, 47);
        System.out.println(pf);
        f = pf.getClass().getDeclaredField("s");
        f.setAccessible(true);
        System.out.println("f.get(pf): " + f.get(pf));
        f.set(pf, "No, you're not!");
        System.out.println(pf);
        f = pf.getClass().getDeclaredField("s2");
        f.setAccessible(true);
        System.out.println("f.get(pf): " + f.get(pf));
        f.set(pf, "No, you're not!");
        System.out.println(pf);
    }
} /* Output:
i = 1, I'm totally safe, Am I safe?
f.getInt(pf): 1
i = 47, I'm totally safe, Am I safe?
f.get(pf): I'm totally safe
i = 47, I'm totally safe, Am I safe?
f.get(pf): Am I safe?
i = 47, I'm totally safe, No, you're not!
*///:~
```

Впрочем, `final`-поля действительно защищены от изменения. Исполнительная система без возражений принимает любые попытки изменения, но при этом ничего не происходит.

В общем случае все эти нарушения доступа не так уж страшны. Если кто-то использует подобные средства для вызова методов, помеченных как закрытые или ограниченные доступом в пределах пакета (что явно указывает на то, что эти методы вызываться не должны), вряд ли он станет жаловаться на то, что вы измените некоторые аспекты этих методов. С другой стороны, наличие «черного хода» к классу позволяет решать некоторые проблемы, которые трудно или невозможно решить другими способами, а польза от отражения вообще неоспорима.

- 25.** (2) Создайте класс, содержащий методы с разными уровнями доступа: закрытым (`private`), защищенным (`protected`), с доступом в пределах пакета. Напишите код для вызова этих методов из-за пределов пакета класса.

Резюме

Динамическое определение типов (RTTI) позволяет вам получить информацию о точном типе объекта тогда, когда у вас имеется лишь ссылка на базовый класс. Таким образом, оно может неправильно использоваться новичком, который еще не понял и не успел оценить всю мощь полиморфизма. У многих людей, ранее работавших с процедурными языками, возникает сильное желание превратить свою программу в одну (или несколько) большую конструкцию `switch`. Они могут это сделать с помощью RTTI, из-за чего не дают в полной мере воспользоваться преимуществами полиморфизма. В Java рекомендуется использовать именно полиморфные методы везде, где можно, а к RTTI прибегать только при крайней необходимости.

Впрочем, при использовании полиморфных методов требуется полный контроль над базовым классом, поскольку в некоторой точке программы, после наследования очередного класса, вы можете обнаружить, что базовый класс не содержит нужного вам метода, и тогда на помощь приходит RTTI: при наследовании вы расширяете интерфейс класса, добавляя в него новые методы. Особенно верно это при использовании в качестве базовых классов библиотек, которые вы не можете изменить. Далее в вашем коде в подходящий момент вы узнаете свой новый тип и вызываете для него его собственный метод. Такой подход не противоречит основам полиморфизма и расширяемости программы, так как добавление в программу нового типа не требует прерывания бесчисленного множества предложений `switch`. Однако, чтобы извлечь пользу из дополнительной функциональности нового класса, придется использовать RTTI.

Добавление некоторого специфического метода в базовый класс будет выгодно только одному производному классу, который действительно реализует его, но все остальные производные классы будут вынуждены использовать для этого метода какую-либо бесполезную «заглушку». Это «загрязняет» интерфейс базового класса и раздражает тех, кому приходится переопределять ненужные абстрактные методы при наследовании от базового класса. Например, рассмотрим иерархию классов, представляющих музыкальные инструменты. Предположим, что вы хотите прочистить мундштуки духовых инструментов своего оркестра. Конечно, можно поместить в базовый класс `Instrument` (общее представление музыкального инструмента) еще один метод `clearSpitValve()` (прочистка мундштуков), но тогда получится, что и у синтезатора, и у барабана есть мундштук! С помощью RTTI можно получить гораздо более верное решение данной задачи, поскольку этот метод уместно поместить в более конкретный класс (например, в класс `Wind`, базовый для всех духовых инструментов). Однако еще более разумным стало бы включение в класс `Instrument` метода `prepareInstrument()` (подготовить инструмент к игре), который подошел бы всем инструментам без исключения, но при решении задачи вы могли не заметить этого и ошибочно решить, что в данном случае без RTTI не обойтись.

Наконец, иногда RTTI решает проблемы производительности. Если ваш код использует полиморфизм по всем правилам, но один из объектов чрезвычайно непродуктивно

обрабатывается кодом, предназначенным для базового типа, то для этого объекта можно сделать исключение, определить его точный тип с помощью RTTI и работать с ним более производительно. Однако ни в коем случае не следует слишком рано беспокоиться об эффективности программ, как бы соблазнительно это ни выглядело. Сначала надо получить *работающую* программу и только после этого решать, достаточно ли быстро она работает, и разобраться с проблемами быстрогодействия, вооружившись инструментами для проверки скорости исполнения (см. приложение <http://MindView.net/Books/BetterJava>).

Вы также узнали, что отражение открывает совершенно новый круг возможностей, основанных на более динамичном стиле программирования. Некоторых разработчиков динамичность отражения беспокоит. Для того, кто привык к безопасности статической проверки типов, возможность выполнения операций, которые могут проверяться только во время выполнения, а обо всех ошибках сообщают посредством исключений, кажется шагом в неверном направлении. Доходит до того, что, по мнению некоторых разработчиков, сама возможность выдачи исключений времени выполнения говорит о том, что такого кода следует избегать. На мой взгляд, это иллюзия; во время выполнения всегда могут возникнуть какие-нибудь проблемы, порождающие исключения, — даже в программе без блоков `try` и спецификаций исключений.

Также я полагаю, что существование последовательной модели получения информации об ошибках позволяет писать динамический код средствами отражения. Конечно, лучше писать код, который может проверяться статически... там, где это возможно. Но я полагаю, что динамический код — один из важнейших аспектов, отделяющих Java от таких языков, как C++.

26. (3) Реализуйте `clearSpitValve()` так, как описано выше.

15

Обобщенные типы

Обычные классы и методы работают с конкретными типами: примитивами или классами. Если вы пишете код, рассчитанный на работу с разнообразными типами, подобная жесткость может создавать нежелательные ограничения¹.

Один из механизмов обобщения в объектно-ориентированных языках основан на полиморфизме. Разработчик пишет (к примеру) метод, получающий в аргументе объект базового класса, после чего использует этот метод для любых классов, производных от базового. Метод стал чуть более общим и может использоваться в большем количестве мест. Этот принцип действует и в отношении классов — в любом месте, где может использоваться конкретный тип, можно использовать базовый тип для повышения гибкости. Конечно, все классы (кроме объявленных `final`²) могут расширяться, поэтому указанная гибкость в большинстве случаев достигается автоматически.

Иногда ограничение одной иерархией оказывается слишком жестким. Если аргументом метода является интерфейс (вместо класса), то ограничения ослабляются до любой реализации этого интерфейса — включая классы, которые еще не были созданы. У программиста-клиента появляется возможность реализовать интерфейс для того, чтобы соответствовать требованиям класса или метода. Таким образом, интерфейсы позволяют расширить обобщение между иерархиями классов — при условии, что разработчик может создать новый класс.

В некоторых ситуациях даже вариант с интерфейсом оказывается недостаточно гибким. Код все равно должен работать с конкретным интерфейсом. Чтобы написать еще более общий код, необходимо иметь возможность указать, что код работает с неким «условным типом» (в отличие от конкретного интерфейса или класса).

¹ Неоценимую помощь в подготовке этой главы оказал документ «Java Generics FAQ» (см. www.langer.camelot.de), написанный Анжеликой Лангер, а также ее другие работы (совместно с Клаусом Крефтом).

² Или классов с закрытым конструктором.

В этом заключается суть механизма *обобщенных типов*, или *обобщений* (generics), — одного из самых значительных изменений в Java SE5. Обобщения реализуют концепцию *параметризованных типов*, позволяющих создавать компоненты (прежде всего контейнеры), которые удобно использовать с разными типами. Под термином «обобщение» следует понимать «применимость к большой группе классов». Обобщенные типы создавались в языках программирования прежде всего для достижения максимальной выразительности при написании классов или методов за счет ослабления ограничений на типы, с которыми работают эти классы или методы. Как будет показано в этой главе, реализация обобщений в Java не настолько универсальна — возможно, сам термин «обобщение» для этой возможности подходит не лучшим образом.

Если вы никогда не сталкивались с механизмами параметризации типов, обобщения Java покажутся удобным дополнением к языку. Когда вы создаете экземпляр параметризованного типа, все приведения типов выполняются за вас, а правильность типов проверяется во время компиляции. Вроде бы улучшения очевидны.

Но разработчики с опытом использования механизма параметризации (например, в языке C++) увидят, что обобщения Java не всегда оправдывают ожидания. Использовать готовый обобщенный тип несложно, но при создании собственных обобщенных типов вас ждут сюрпризы. Я постараюсь объяснить, почему эта возможность была реализована именно так, а не иначе.

Это не значит, что обобщения Java бесполезны — достаточно часто они делают код более прямолинейным и даже элегантным. Но если у вас есть опыт работы на языке с более «чистой» реализацией обобщений, возможно, вы будете разочарованы. В этой главе будут рассмотрены как достоинства, так и недостатки обобщений Java, что позволит вам более эффективно использовать новую возможность.

Сравнение с C++

Создатели Java утверждают, что источником вдохновения их работы был язык C++. Несмотря на это, Java можно изучать без знания C++; я решил именно так и поступить — кроме тех случаев, когда сравнение поможет глубже понять материал.

Обобщения требуют сравнения с C++ по двум причинам. Во-первых, понимание некоторых особенностей шаблонов C++ (основного прототипа обобщений, включая базовый синтаксис) поможет разобраться в основах самой концепции, а также (что очень важно) в том, на что обобщения Java не способны — и почему. Я постараюсь четко объяснить, где проходят эти границы, потому что, по моему опыту, понимание этих границ повышает квалификацию программиста. Зная, что вы сделать не можете, вы будете более эффективно использовать доступные возможности (отчасти потому, что вы не тратите время, выбираясь из очередного тупика).

Вторая причина заключается в том, что в сообществе Java недостаточно хорошо понимают шаблоны C++, и это недопонимание мешает понять цели обобщений.

Итак, хотя в этой главе приведены примеры шаблонов C++, я постараюсь свести их количество к минимуму.

Простые обобщения

Одна из самых убедительных причин для применения обобщений — классы контейнеров, представленные в главе 11 (а также более подробно рассматриваемые в главе 17). Контейнер представляет собой хранилище для объектов, с которыми работает программа. И хотя то же самое можно сказать и о массивах, контейнеры обладают большей гибкостью и отличаются от простых массивов своими характеристиками. Практически в любой программе возникает необходимость хранения используемых объектов, поэтому контейнеры относятся к числу наиболее часто используемых библиотек классов.

Рассмотрим класс, предназначенный для хранения одного объекта. Разумеется, в этом классе должен быть указан точный тип хранимого объекта:

```
//: generics/Holder1.java

class Automobile {}

public class Holder1 {
    private Automobile a;
    public Holder1(Automobile a) { this.a = a; }
    Automobile get() { return a; }
} ///:~
```

Но этот класс нельзя назвать универсальным, потому что никакие другие данные в нем храниться не могут. Писать новый класс для каждого возможного типа не хотелось бы. До выхода Java SE5 можно было бы просто хранить поле `Object`:

```
//: generics/Holder2.java

public class Holder2 {
    private Object a;
    public Holder2(Object a) { this.a = a; }
    public void set(Object a) { this.a = a; }
    public Object get() { return a; }
    public static void main(String[] args) {
        Holder2 h2 = new Holder2(new Automobile());
        Automobile a = (Automobile)h2.get();
        h2.set("Not an Automobile");
        String s = (String)h2.get();
        h2.set(1); // Автоматическая упаковка в Integer
        Integer x = (Integer)h2.get();
    }
} ///:~
```

Теперь в `Holder2` можно хранить что угодно — в приведенном примере один класс `holder2` хранит объекты трех разных типов.

Иногда бывает нужно, чтобы в контейнере хранились объекты разных типов, но чаще в контейнере размещаются объекты только одного типа. Одной из главных причин для введения обобщений была возможность указать, какой тип объектов должен храниться в контейнере, и чтобы эта информация контролировалась компилятором.

Итак, вместо `Object` было бы желательно использовать «условный» тип, с которым программист может определиться позднее. Для этого параметр-тип указывается в угловых

скобках после имени класса, а при использовании класса заменяется фактическим типом. В следующем примере параметр-тип обозначается идентификатором `T`:

```
//: generics/Holder3.java

public class Holder3<T> {
    private T a;
    public Holder3(T a) { this.a = a; }
    public void set(T a) { this.a = a; }
    public T get() { return a; }
    public static void main(String[] args) {
        Holder3<Automobile> h3 =
            new Holder3<Automobile>(new Automobile());
        Automobile a = h3.get(); // Преобразование не требуется
        // h3.set("Not an Automobile"); // Ошибка
        // h3.set(1); // Ошибка
    }
} ///:~
```

При создании `Holder3` необходимо указать тип включаемых объектов с использованием того же синтаксиса с угловыми скобками, который мы видим в `main()`. В объекте можно размещать только объекты указанного типа (или его подтипа, потому что принцип подстановки работает и для обобщенных типов). Получаемое значение автоматически относится к правильному типу.

В этом заключается основной принцип обобщенных типов Java: вы сообщаете, какой тип хотите использовать, а обобщение берет на себя все подробности.

В общем случае вы работаете с обобщениями точно так же, как с любым другим типом, — просто у обобщений имеются параметры-типы. Но как будет вскоре показано, обобщения можно использовать простым указанием их имен со списком аргументов-типов.

1. (1) Используйте `Holder3` с библиотекой `typeinfo.pets` и продемонстрируйте, что объект `Holder3`, объявленный с базовым типом, также может хранить производный тип.
2. (1) Создайте класс для хранения трех объектов одного типа, с методами сохранения и выборки этих объектов и конструктором для инициализации всех трех объектов.

Библиотека кортежей

На практике при вызове метода часто возникает необходимость вернуть несколько объектов. Команда `return` позволяет вернуть только один объект, поэтому задачу приходится решать созданием объекта, содержащего несколько возвращаемых объектов. Конечно, можно писать специальный класс каждый раз, когда возникнет такая ситуация, но обобщения позволяют решить задачу один раз и сэкономить время в будущем. В то же время такое решение гарантирует безопасность типов во время компиляции.

Группа объектов, «завернутых» в один объект, называется *кортежем* (*tuple*). Получатель объекта может читать элементы, но не может добавлять новые (эта концепция также называется «объектом передачи данных»).

Обычно кортеж может иметь произвольную длину, но все объекты кортежа могут относиться к разным типам. Однако мы хотим задать тип каждого объекта и убедиться в том, что при чтении значения получатель получит правильный тип. Для решения

проблемы с разной длиной можно создать несколько разных кортежей. Следующая версия содержит два объекта:

```
//: net/mindview/util/TwoTuple.java
package net.mindview.util;

public class TwoTuple<A,B> {
    public final A first;
    public final B second;
    public TwoTuple(A a, B b) { first = a; second = b; }
    public String toString() {
        return "(" + first + ", " + second + ")";
    }
} ///:~
```

Конструктор сохраняет объект в кортеже, а вспомогательный метод `toString()` выводит значения из списка. Учтите, что кортеж неявно поддерживает порядок элементов.

На первый взгляд можно подумать, что такое решение нарушает основные принципы безопасности при программировании на Java. Разве поля `first` и `second` не следует объявить закрытыми, а для обращения к ним использовать методы `getFirst()` и `getSecond()`? Подумайте, какая безопасность достигается в данном случае: клиент может прочитать объект и сделать с ним то, что захочет, но ему не удастся присвоить `first` или `second` другое значение. Объявление `final` обеспечивает тот же уровень безопасности, но приведенная форма короче и проще.

Возможно, вы захотите, чтобы клиент мог связать `first` или `second` с другим объектом. Однако безопаснее оставить решение в этой форме и заставить клиента создать новый объект `TwoTuple`, если ему потребуется изменить его элементы.

Для создания кортежей большей длины можно применить наследование. Как показывает следующий пример, добавить новые параметры-типы несложно:

```
//: net/mindview/util/ThreeTuple.java
package net.mindview.util;

public class ThreeTuple<A,B,C> extends TwoTuple<A,B> {
    public final C third;
    public ThreeTuple(A a, B b, C c) {
        super(a, b);
        third = c;
    }
    public String toString() {
        return "(" + first + ", " + second + ", " + third + ")";
    }
} ///:~

//: net/mindview/util/FourTuple.java
package net.mindview.util;

public class FourTuple<A,B,C,D> extends ThreeTuple<A,B,C> {
    public final D fourth;
    public FourTuple(A a, B b, C c, D d) {
        super(a, b, c);
        fourth = d;
    }
}
```

```

    public String toString() {
        return "(" + first + ", " + second + ", " +
            third + ", " + fourth + ")";
    }
} ///:~

//: net/mindview/util/FiveTuple.java
package net.mindview.util;

public class FiveTuple<A,B,C,D,E>
extends FourTuple<A,B,C,D> {
    public final E fifth;
    public FiveTuple(A a, B b, C c, D d, E e) {
        super(a, b, c, d);
        fifth = e;
    }
    public String toString() {
        return "(" + first + ", " + second + ", " +
            third + ", " + fourth + ", " + fifth + ")";
    }
} ///:~

```

Чтобы использовать кортеж, вы просто определяете объект с нужной длиной как возвращаемое значение функции, а затем создаете и возвращаете его в команде `return`:

```

//: generics/TupleTest.java
import net.mindview.util.*;

class Amphibian {}
class Vehicle {}

public class TupleTest {
    static TwoTuple<String,Integer> f() {
        // Автоматическая упаковка преобразует int в Integer:
        return new TwoTuple<String,Integer>("hi", 47);
    }
    static ThreeTuple<Amphibian,String,Integer> g() {
        return new ThreeTuple<Amphibian, String, Integer>(
            new Amphibian(), "hi", 47);
    }
    static
    FourTuple<Vehicle,Amphibian,String,Integer> h() {
        return
            new FourTuple<Vehicle,Amphibian,String,Integer>(
                new Vehicle(), new Amphibian(), "hi", 47);
    }
    static
    FiveTuple<Vehicle,Amphibian,String,Integer,Double> k() {
        return new
            FiveTuple<Vehicle,Amphibian,String,Integer,Double>(
                new Vehicle(), new Amphibian(), "hi", 47, 11.1);
    }
    public static void main(String[] args) {
        TwoTuple<String,Integer> ttsi = f();
        System.out.println(ttsi);
        // ttsi.first = "there"; // Ошибка компиляции: final
        System.out.println(g());
        System.out.println(h());
    }
}

```

```

    System.out.println(k());
}
} /* Output: (80% match)
(hi, 47)
(Amphibian@1f6a7b9, hi, 47)
(Vehicle@35ce36, Amphibian@757aef, hi, 47)
(Vehicle@9cab16, Amphibian@1a46e30, hi, 47, 11.1)
*///:~

```

Благодаря обобщениям вы можете легко создать произвольный кортеж, возвращающий любую группу типов, просто написав нужное выражение.

Как видите, спецификация `final` для открытых (`public`) полей предотвращает повторное присваивание после создания; на это указывает ошибка при обработке команды `ttsi.first = "there"`.

Новые выражения получаются немного громоздкими. Позднее в этой главе вы увидите, как упростить их за счет использования обобщенных методов.

3. (1) Создайте и протестируйте обобщенный тип `SixTuple` для шести элементов.

4. (3) «Обобщите» пример `innerclasses/Sequence.java`.

Класс стека

Рассмотрим нечто менее тривиальное: традиционный стек. В главе 11 была представлена реализация стека на базе `LinkedList` — класс `net.mindview.util.Stack`. В том примере класс `LinkedList` уже содержал необходимые методы для создания стека. Стек конструировался посредством композиции одного обобщенного класса (`Stack<T>`) с другим обобщенным классом (`LinkedList<T>`). Обобщенный тип вел себя как самый обычный тип (за несколькими исключениями, которые будут рассмотрены ниже).

Вместо использования `LinkedList` мы также могли реализовать собственный механизм внутреннего хранения данных.

```

//: generics/LinkedStack.java
// Стек, реализованный на базе связанного списка.

```

```

public class LinkedStack<T> {
    private static class Node<U> {
        U item;
        Node<U> next;
        Node() { item = null; next = null; }
        Node(U item, Node<U> next) {
            this.item = item;
            this.next = next;
        }
        boolean end() { return item == null && next == null; }
    }
    private Node<T> top = new Node<T>(); // Стопж
    public void push(T item) {
        top = new Node<T>(item, top);
    }
    public T pop() {
        T result = top.item;

```

```

        if(!top.end())
            top = top.next;
        return result;
    }
    public static void main(String[] args) {
        LinkedStack<String> lss = new LinkedStack<String>();
        for(String s : "Phasers on stun!".split(" "))
            lss.push(s);
        String s;
        while((s = lss.pop()) != null)
            System.out.println(s);
    }
} /* Output:
stun!
on
Phasers
*///:~

```

Внутренний класс `Node` тоже является обобщенным и имеет собственный параметр-тип.

В этом примере для определения пустого стека используется *сторож* (end sentinel). Сторож создается при конструировании `LinkedStack`, и при каждом вызове `push()` новый узел `Node<T>` создается и связывается с предыдущим узлом `Node<T>`. При вызове `pop()` всегда возвращается `top.item`, после чего текущий узел `Node<T>` удаляется и происходит переход к следующему узлу; но при достижении сторожа перемещение не выполняется. Если клиент будет и дальше вызывать `pop()`, он будет получать `null` (признак пустого стека).

5. (2) Удалите параметр-тип класса `Node` и измените остальной код `LinkedStack.java` так, чтобы показать, что для внутреннего класса доступны обобщенные параметры-типы внешнего класса.

RandomList

Рассмотрим другой пример: допустим, мы хотим создать особую разновидность списка, которая случайным образом выбирает один из своих элементов при каждом вызове `select()`. Чтобы реализация такого списка работала с любыми объектами, следует использовать обобщения:

```

//: generics/RandomList.java
import java.util.*;

public class RandomList<T> {
    private ArrayList<T> storage = new ArrayList<T>();
    private Random rand = new Random(47);
    public void add(T item) { storage.add(item); }
    public T select() {
        return storage.get(rand.nextInt(storage.size()));
    }
    public static void main(String[] args) {
        RandomList<String> rs = new RandomList<String>();
        for(String s: ("The quick brown fox jumped over " +
            "the lazy brown dog").split(" "))
            rs.add(s);
        for(int i = 0; i < 11; i++)

```

```

        System.out.print(rs.select() + " ");
    }
} /* Output:
brown over fox quick quick dog brown The brown lazy brown
*///:~

```

6. (1) Используйте RandomList еще с двумя типами (кроме представленного в main()).

Обобщенные интерфейсы

Обобщения также работают с интерфейсами. Например, *генератор* представляет собой класс для создания объектов. В действительности он является разновидностью паттерна проектирования «Фабричный метод», но запрашивая у генератора новый объект, вы не передаете ему никаких аргументов, тогда как фабричный метод обычно вызывается с аргументами. Генератор умеет создавать новые объекты без дополнительной информации.

Как правило, генератор определяет всего один метод, создающий новые объекты. Мы назовем его `next()` и включим в стандартный инструментарий:

```

//: net/mindview/util/Generator.java
// Обобщенный интерфейс.
package net.mindview.util;
public interface Generator<T> { T next(); } ///:~

```

Возвращаемый тип `next()` параметризуется по `T`. Как видите, с интерфейсами обобщения работают почти так же, как с классами.

Чтобы продемонстрировать реализацию `Generator`, нам понадобятся классы. Следующая иерархия моделирует разные виды кофе:

```

//: generics/coffee/Coffee.java
package generics.coffee;

public class Coffee {
    private static long counter = 0;
    private final long id = counter++;
    public String toString() {
        return getClass().getSimpleName() + " " + id;
    }
} ///:~

```

```

//: generics/coffee/Latte.java
package generics.coffee;
public class Latte extends Coffee {} ///:~

```

```

//: generics/coffee/Mocha.java
package generics.coffee;
public class Mocha extends Coffee {} ///:~

```

```

//: generics/coffee/Cappuccino.java
package generics.coffee;
public class Cappuccino extends Coffee {} ///:~

```

```
//: generics/coffee/Americano.java
package generics.coffee;
public classAmericano extends Coffee {} ///:~
```

```
//: generics/coffee/Breve.java
package generics.coffee;
public class Breve extends Coffee {} ///:~
```

Теперь мы можем реализовать интерфейс `Generator<Coffee>` для создания случайных типов объектов `Coffee`:

```
//: generics/coffee/CoffeeGenerator.java
// Генерирование разных типов Coffee:
package generics.coffee;
import java.util.*;
import net.mindview.util.*;

public class CoffeeGenerator
implements Generator<Coffee>, Iterable<Coffee> {
    private Class[] types = { Latte.class, Mocha.class,
        Cappuccino.class,Americano.class, Breve.class, };
    private static Random rand = new Random(47);
    public CoffeeGenerator() {}
    // Для перебора:
    private int size = 0;
    public CoffeeGenerator(int sz) { size = sz; }
    public Coffee next() {
        try {
            return (Coffee)
                types[rand.nextInt(types.length)].newInstance();
            // Сообщить программисту об ошибках во время выполнения:
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    class CoffeeIterator implements Iterator<Coffee> {
        int count = size;
        public boolean hasNext() { return count > 0; }
        public Coffee next() {
            count--;
            return CoffeeGenerator.this.next();
        }
        public void remove() { // Не реализован
            throw new UnsupportedOperationException();
        }
    };
    public Iterator<Coffee> iterator() {
        return new CoffeeIterator();
    }
    public static void main(String[] args) {
        CoffeeGenerator gen = new CoffeeGenerator();
        for(int i = 0; i < 5; i++)
            System.out.println(gen.next());
        for(Coffee c : new CoffeeGenerator(5))
            System.out.println(c);
    }
} /* Output:
Americano 0
Latte 1
Americano 2
```



```
Mocha 3
Mocha 4
Breve 5
Americano 6
Latte 7
Cappuccino 8
Cappuccino 9
*///:~
```

Параметризованный интерфейс `Generator` гарантирует, что `next()` вернет параметр-тип. `CoffeeGenerator` также реализует интерфейс `Iterable`, что позволяет использовать его в конструкции `foreach`. Однако ему необходим «сторож» для обнаружения завершения перебора, который создается во втором конструкторе.

Вторая реализация `Generator<T>` — на этот раз предназначенная для генерирования чисел Фибоначчи:

```
//: generics/Fibonacci.java
// Генерирование последовательности Фибоначчи.
import net.mindview.util.*;

public class Fibonacci implements Generator<Integer> {
    private int count = 0;
    public Integer next() { return fib(count++); }
    private int fib(int n) {
        if(n < 2) return 1;
        return fib(n-2) + fib(n-1);
    }
    public static void main(String[] args) {
        Fibonacci gen = new Fibonacci();
        for(int i = 0; i < 18; i++)
            System.out.print(gen.next() + " ");
    }
} /* Output:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
*///:~
```

Хотя и внутри, и вне класса мы работаем с `int`, в параметре-типе указывается `Integer`. Здесь проявляется одно из ограничений механизма обобщений Java: примитивы не могут использоваться как параметры-типы. Однако в Java SE5 появились удобные средства автоматической упаковки и распаковки для перехода от примитивных типов к объектным «оберткам» и обратно. Эффект проявляется в этом примере, так как значения `int` прозрачно используются и производятся классом.

Можно пойти еще дальше и создать генератор чисел Фибоначчи, реализующий `Iterable`. Один из вариантов заключается в повторной реализации класса с добавлением интерфейса `Iterable`, но исходный код не всегда доступен, и переписывать код без необходимости нежелательно. Вместо этого мы создадим *адаптер*, предоставляющий нужный интерфейс (этот паттерн уже встречался ранее).

Существуют разные способы реализации адаптеров. Например, для создания адаптируемого класса можно применить наследование:

```
//: generics/IterableFibonacci.java
// Адаптация класса Fibonacci для поддержки Iterable.
import java.util.*;
```

продолжение ➤

```

public class IterableFibonacci
extends Fibonacci implements Iterable<Integer> {
    private int n;
    public IterableFibonacci(int count) { n = count; }
    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            public boolean hasNext() { return n > 0; }
            public Integer next() {
                n--;
                return IterableFibonacci.this.next();
            }
            public void remove() { // Не реализован
                throw new UnsupportedOperationException();
            }
        };
    }
    public static void main(String[] args) {
        for(int i : new IterableFibonacci(18))
            System.out.print(i + " ");
    }
} /* Output:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
*///:~

```

Чтобы использовать `IterableFibonacci` в конструкции `foreach`, мы передаем конструктору границу, чтобы метод `hasNext()` знал, когда возвращать `false`.

7. (2) Используйте композицию вместо наследования при адаптации `Fibonacci`, обеспечивающей поддержку `Iterable`.
8. (2) По образцу `Coffee` создайте иерархию персонажей своего любимого фильма `StoryCharacters`, разделив их на положительных (`GoodGuys`) и отрицательных (`BadGuys`). Создайте генератор для `StoryCharacters` по образцу `CoffeeGenerator`.

Обобщенные методы

До настоящего момента мы рассматривали параметризацию целых классов. Также возможна параметризация методов внутри класса. Сам класс при этом может быть обобщенным, а может и не быть — это не зависит от наличия обобщенных методов.

Обобщенный метод может изменяться независимо от класса. Как правило, применять обобщенные методы следует там, где только возможно, — иначе говоря, если существует возможность сделать обобщенным только метод вместо целого класса, вероятно, такое решение будет менее громоздким.

Чтобы определить обобщенный метод, следует поместить список параметров-типов перед возвращаемым значением:

```

//: generics/GenericMethods.java

public class GenericMethods {
    public <T> void f(T x) {
        System.out.println(x.getClass().getName());
    }
}

```

```
public static void main(String[] args) {
    GenericMethods gm = new GenericMethods();
    gm.f("");
    gm.f(1);
    gm.f(1.0);
    gm.f(1.0F);
    gm.f('c');
    gm.f(gm);
}
} /* Output:
java.lang.String
java.lang.Integer
java.lang.Double
java.lang.Float
java.lang.Character
GenericMethods
*///:~
```

Класс `GenericMethods` не параметризован, хотя и класс, и его методы могут быть параметризованы одновременно. Но в данном случае параметр-тип имеется только у метода `f()` (на что указывает список перед возвращаемым типом метода).

Обратите внимание: в обобщенных классах параметры-типы должны указываться при создании экземпляра класса. При использовании обобщенных методов указывать параметры-типы обычно не обязательно, потому что компилятор может вычислить их за вас. Таким образом, вызовы `f()` выглядят как обычные вызовы методов, и внешне все выглядит так, словно `f()` существует в бесконечном количестве перегруженных версий. Он даже может получить аргумент типа `GenericMethods`.

Для вызовов `f()`, использующих примитивные типы, в дело вступает *автоматическая упаковка* — автоматическое преобразование примитивных типов в соответствующие объекты. Более того, обобщенные методы и автоматическая упаковка иногда позволяют убрать код, который раньше приходилось обрабатывать вручную.

9. (1) Измените пример `GenericMethods.java`, чтобы метод `f()` получал три аргумента, относящихся к разным параметризованным типам.
10. (1) Измените предыдущее упражнение так, чтобы один из аргументов `f()` был непараметризованным.

Использование автоматического определения аргументов-типов

Разработчики жалуются, что обобщения часто приводят к увеличению объема кода. Возьмем пример `holding/MapOfList.java` из главы 11. Создание контейнера `Map` с элементами `List` выглядит так:

```
Map<Person, List<? extends Pet>> petPeople =
    new HashMap<Person, List<? extends Pet>>();
```

(ключевое слово `extends` и вопросительные знаки будут описаны позже в этой главе). Создается впечатление, что мы повторяемся, а компилятор мог бы определить один из списков обобщенных аргументов по другому. К сожалению, он этого сделать не может, но механизм автоматического определения аргументов-типов в обобщенных методах

способен привести к некоторому упрощению. Например, можно создать вспомогательный класс с различными статическими методами, который предоставляет наиболее часто используемые реализации различных контейнеров:

```
//: net/mindview/util/New.java
// Упрощенное создание обобщенных контейнеров
// за счет использования автоматического определения
// аргументов-типов.
package net.mindview.util;
import java.util.*;

public class New {
    public static <K,V> Map<K,V> map() {
        return new HashMap<K,V>();
    }
    public static <T> List<T> list() {
        return new ArrayList<T>();
    }
    public static <T> LinkedList<T> llist() {
        return new LinkedList<T>();
    }
    public static <T> Set<T> set() {
        return new HashSet<T>();
    }
    public static <T> Queue<T> queue() {
        return new LinkedList<T>();
    }
    // Примеры:
    public static void main(String[] args) {
        Map<String, List<String>> sls = New.map();
        List<String> ls = New.list();
        LinkedList<String> lls = New.llist();
        Set<String> ss = New.set();
        Queue<String> qs = New.queue();
    }
} ///:~
```

В методе `main()` встречаются примеры использования — автоматическое определение аргументов-типов устраняет необходимость в повторении списка параметров обобщения. Применим новый класс в `holding/MapOfList.java`:

```
//: generics/SimplerPets.java
import typeinfo.pets.*;
import java.util.*;
import net.mindview.util.*;

public class SimplerPets {
    public static void main(String[] args) {
        Map<Person, List<? extends Pet>> petPeople = New.map();
        // Остальной код остается без изменений...
    }
} ///:~
```

И хотя этот пример интересен, трудно сказать, насколько реальна приносимая им польза. Человеку, читающему код, придется разобраться в дополнительной библиотеке и ее нюансах, поэтому с таким же успехом можно было оставить исходное определение (содержащее повторения) — как ни парадоксально, ради простоты. Однако, если в стандартной библиотеке Java появится некий аналог `New.java`, его стоит использовать.

Автоматическое определение аргументов-типов работает *только* при присваивании. Если передать результат вызова метода (например, `New.map()`) в аргументе другого метода, компилятор не пытается вычислять тип. Вместо этого он интерпретирует вызов метода так, словно возвращаемое значение присваивается переменной типа `Object`. При компиляции следующего примера произойдет ошибка:

```
//: generics/LimitsOfInference.java
import typeinfo.pets.*;
import java.util.*;

public class LimitsOfInference {
    static void
    f(Map<Person, List<? extends Pet>> petPeople) {}
    public static void main(String[] args) {
        // f(New.map()); // Не компилируется
    }
} ///:~
```

- 11.** (1) Протестируйте пример `New.java`; создайте собственные классы и убедитесь в том, что класс `New` правильно работает с ними.

Явное указание типа

Тип для обобщенного метода можно задать явно, хотя необходимость в таком синтаксисе возникает редко. Для этого тип указывается в угловых скобках после точки и непосредственно перед именем метода. При вызове метода из того же класса перед точкой необходимо использовать `this`, а для статических методов перед точкой должно находиться имя класса. Следующий синтаксис решает проблему, продемонстрированную в `LimitsOfInference.java`:

```
//: generics/ExplicitTypeSpecification.java
import typeinfo.pets.*;
import java.util.*;
import net.mindview.util.*;

public class ExplicitTypeSpecification {
    static void f(Map<Person, List<Pet>> petPeople) {}
    public static void main(String[] args) {
        f(New.<Person, List<Pet>>map());
    }
} ///:~
```

Конечно, при этом теряются преимущества использования класса `New` для сокращения объема вводимого текста, но дополнительный синтаксис необходим только вне команд присваивания.

- 12.** (1) Повторите предыдущее упражнение с явным указанием типа.

Списки аргументов переменной длины и обобщенные методы

Обобщенные методы нормально сосуществуют со списками аргументов переменной длины:

```
//: generics/GenericVarargs.java
import java.util.*;
```

```

public class GenericVarargs {
    public static <T> List<T> makeList(T... args) {
        List<T> result = new ArrayList<T>();
        for(T item : args)
            result.add(item);
        return result;
    }
    public static void main(String[] args) {
        List<String> ls = makeList("A");
        System.out.println(ls);
        ls = makeList("A", "B", "C");
        System.out.println(ls);
        ls = makeList("ABCDEFFHIJKLMNOPQRSTUVWXYZ".split(""));
        System.out.println(ls);
    }
} /* Output:
[A]
[A, B, C]
[, A, B, C, D, E, F, F, H, I, J, K, L, M, N, O, P, Q, R, S,
T, U, V, W, X, Y, Z]
*///:~

```

Метод `makeList()` реализует ту же функциональность, что и метод `java.util.Arrays.asList()` стандартной библиотеки.

Обобщенный метод для использования с генераторами

Для заполнения коллекций удобно использовать генератор, причем эту операцию логично обобщить:

```

//: generics/Generators.java
// Вспомогательный класс для работы с Generator.
import generics.coffee.*;
import java.util.*;
import net.mindview.util.*;

public class Generators {
    public static <T> Collection<T>
    fill(Collection<T> coll, Generator<T> gen, int n) {
        for(int i = 0; i < n; i++)
            coll.add(gen.next());
        return coll;
    }
    public static void main(String[] args) {
        Collection<Coffee> coffee = fill(
            new ArrayList<Coffee>(), new CoffeeGenerator(), 4);
        for(Coffee c : coffee)
            System.out.println(c);
        Collection<Integer> fnumbers = fill(
            new ArrayList<Integer>(), new Fibonacci(), 12);
        for(int i : fnumbers)
            System.out.print(i + ", ");
    }
} /* Output:
Americano 0
Latte 1
Americano 2

```

Mocha 3

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
*///:~
```

Обратите внимание на прозрачность применения обобщенного метода `fill()` к контейнерам и генераторам `Coffee` и `Integer`.

- 13.** (4) Перегрузите метод `fill()` так, чтобы аргументы и возвращаемые типы были конкретными подтипами `Collection`: `List`, `Queue` и `Set`. При этом информация о типе контейнера не теряется. Сможет ли перегруженная версия различать `List` и `LinkedList`?

Генератор общего назначения

Следующий класс создает `Generator` для любого класса, имеющего конструктор по умолчанию. Для сокращения объема кода он также включает обобщенный метод для создания `BasicGenerator`:

```
//: net/mindview/util/BasicGenerator.java
// Автоматическое создание Generator для класса
// с конструктором по умолчанию (без аргументов).
package net.mindview.util;

public class BasicGenerator<T> implements Generator<T> {
    private Class<T> type;
    public BasicGenerator(Class<T> type){ this.type = type; }
    public T next() {
        try {
            // Предполагается, что type - открытый класс:
            return type.newInstance();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    // Создание генератора по умолчанию для заданного type:
    public static <T> Generator<T> create(Class<T> type) {
        return new BasicGenerator<T>(type);
    }
}
///:~
```

Этот класс предоставляет базовую реализацию для создания объекта класса, который (1) является открытым (так как `BasicGenerator` находится в отдельном пакете, соответствующий класс должен иметь открытый уровень доступа, а не доступ в границах пакета), и (2) имеет конструктор по умолчанию (вызываемый без аргументов). Чтобы создать один из объектов `BasicGenerator`, следует вызвать метод `create()` и передать ему маркер генерируемого типа. Обобщенный метод `create()` позволяет использовать запись `BasicGenerator.create(MyType.class)` вместо более громоздкой записи `new BasicGenerator<MyType>(MyType.class)`.

Например, следующий простой класс содержит конструктор по умолчанию:

```
//: generics/CountedObject.java

public class CountedObject {
    private static long counter = 0;
```

продолжение ➤

```
private final long id = counter++;
public long id() { return id; }
public String toString() { return "CountedObject " + id;}
} ///:~
```

Класс `CountedObject` отслеживает количество созданных экземпляров и выводит его в `toString()`.

При помощи `BasicGenerator` можно легко создать `Generator` для `CountedObject`:

```
//: generics/BasicGeneratorDemo.java
import net.mindview.util.*;

public class BasicGeneratorDemo {
    public static void main(String[] args) {
        Generator<CountedObject> gen =
            BasicGenerator.create(CountedObject.class);
        for(int i = 0; i < 5; i++)
            System.out.println(gen.next());
    }
} /* Output:
CountedObject 0
CountedObject 1
CountedObject 2
CountedObject 3
CountedObject 4
*///:~
```

Вы видите, как обобщенный метод сокращает объем кода, необходимого для создания объекта `Generator`. Обобщения Java все равно заставляют передать объект `Class`, так что его можно с таким же успехом использовать для автоматического определения типов в методе `create()`.

- 14.** (1) Измените пример `BasicGeneratorDemo.java`, чтобы в нем использовалась явная форма создания `Generator` (то есть используйте явный вызов конструктора вместо обобщенного метода `create()`).

Упрощение использования кортежей

Автоматическое определение аргументов-типов в сочетании со статическим импортированием позволяет оформить кортежи, с которыми мы встречались ранее, в виде библиотеки общего назначения. Для создания кортежей может использоваться перегруженный статический метод:

```
//: net/mindview/util/Tuple.java
// Библиотека кортежей с использованием
// автоматического определения аргументов-типов.
package net.mindview.util;

public class Tuple {
    public static <A,B> TwoTuple<A,B> tuple(A a, B b) {
        return new TwoTuple<A,B>(a, b);
    }
    public static <A,B,C> ThreeTuple<A,B,C>
    tuple(A a, B b, C c) {
        return new ThreeTuple<A,B,C>(a, b, c);
    }
}
```



```

public static <A,B,C,D> FourTuple<A,B,C,D>
tuple(A a, B b, C c, D d) {
    return new FourTuple<A,B,C,D>(a, b, c, d);
}
public static <A,B,C,D,E>
FiveTuple<A,B,C,D,E> tuple(A a, B b, C c, D d, E e) {
    return new FiveTuple<A,B,C,D,E>(a, b, c, d, e);
}
} ///:~

```

Модифицированная версия TupleTest.java для тестирования Tuple.java:

```

//: generics/TupleTest2.java
import net.mindview.util.*;
import static net.mindview.util.Tuple.*;

public class TupleTest2 {
    static TwoTuple<String,Integer> f() {
        return tuple("hi", 47);
    }
    static TwoTuple f2() { return tuple("hi", 47); }
    static ThreeTuple<Amphibian,String,Integer> g() {
        return tuple(new Amphibian(), "hi", 47);
    }
    static
    FourTuple<Vehicle,Amphibian,String,Integer> h() {
        return tuple(new Vehicle(), new Amphibian(), "hi", 47);
    }
    static
    FiveTuple<Vehicle,Amphibian,String,Integer,Double> k() {
        return tuple(new Vehicle(), new Amphibian(),
            "hi", 47, 11.1);
    }
    public static void main(String[] args) {
        TwoTuple<String,Integer> ttsi = f();
        System.out.println(ttsi);
        System.out.println(f2());
        System.out.println(g());
        System.out.println(h());
        System.out.println(k());
    }
} /* Output: (80% match)
(hi, 47)
(hi, 47)
(Amphibian@7d772e, hi, 47)
(Vehicle@757aef, Amphibian@d9f9c3, hi, 47)
(Vehicle@1a46e30, Amphibian@3e25a5, hi, 47, 11.1)
*///:~

```

Обратите внимание: `f()` возвращает параметризованный объект `TwoTuple`, тогда как `f2()` возвращает непараметризованный объект `TwoTuple`. В данном случае компилятор не выдает предупреждение о `f2()`, потому что возвращаемое значение не используется «параметризованным образом»; в некотором смысле оно превращается в непараметризованную версию `TwoTuple` восходящим преобразованием. Но при попытке сохранить результат `f2()` в параметризованном объекте `TwoTuple` компилятор выдаст предупреждение.

15. (1) Проверьте предыдущее утверждение.

16. (2) Добавьте класс `SixTuple` в `Tuple.java` и протестируйте его в программе `TupleTest2.java`.

Операции с множествами

Рассмотрим еще один пример использования обобщенных методов: математические отношения, выражаемые с использованием множеств (*Set*). Их удобно определить в виде обобщенных методов, которые могут использоваться с любыми типами:

```
//: net/mindview/util/Sets.java
package net.mindview.util;
import java.util.*;

public class Sets {
    public static <T> Set<T> union(Set<T> a, Set<T> b) {
        Set<T> result = new HashSet<T>(a);
        result.addAll(b);
        return result;
    }
    public static <T>
    Set<T> intersection(Set<T> a, Set<T> b) {
        Set<T> result = new HashSet<T>(a);
        result.retainAll(b);
        return result;
    }
    // Вычитание подмножества из надмножества:
    public static <T> Set<T>
    difference(Set<T> superset, Set<T> subset) {
        Set<T> result = new HashSet<T>(superset);
        result.removeAll(subset);
        return result;
    }
    // Рефлексивность -- все, что не входит в пересечение:
    public static <T> Set<T> complement(Set<T> a, Set<T> b) {
        return difference(union(a, b), intersection(a, b));
    }
} ///:~
```

Первые три метода дублируют первый аргумент, копируют его ссылки в новый объект *HashSet*, так что аргументы *Set* напрямую не изменяются. Таким образом, возвращаемое значение представляет собой новый объект *Set*.

Четыре метода представляют математические операции с множествами: *union()* возвращает объект *Set* с объединением двух аргументов; *intersection()* возвращает объект *Set*, содержащий общие элементы двух аргументов (пересечение); *difference()* вычитает элементы подмножества из надмножества; *complement()* возвращает объект *Set* со всеми элементами, не входящими в пересечение. Чтобы создать простой пример, демонстрирующий результат применения этих методов, мы используем перечисление с различными названиями красок:

```
//: generics/watercolors/Watercolors.java
package generics.watercolors;

public enum Watercolors {
    ZINC, LEMON_YELLOW, MEDIUM_YELLOW, DEEP_YELLOW, ORANGE,
    BRILLIANT_RED, CRIMSON, MAGENTA, ROSE_MADDER, VIOLET,
    CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE,
    COBALT_BLUE_HUE, PERMANENT_GREEN, VIRIDIAN_HUE,
    SAP_GREEN, YELLOW_OCHRE, BURNT_SIENNA, RAW_UMBER,
    BURNT_UMBER, PAYNES_GRAY, IVORY_BLACK
} ///:~
```

Для удобства (чтобы имена не надо было уточнять) в следующем примере используется статическое импортирование. В примере используется EnumSet — инструмент Java SE5 для простого создания объектов Set на базе перечислений (дополнительная информация о EnumSet приведена в главе 19). В данном случае статический метод EnumSet.range() получает первый и последний элементы диапазона, создаваемого в результирующем объекте Set:

```
//: generics/WatercolorSets.java
import generics.watercolors.*;
import java.util.*;
import static net.mindview.util.Print.*;
import static net.mindview.util.Sets.*;
import static generics.watercolors.Watercolors.*;

public class WatercolorSets {
    public static void main(String[] args) {
        Set<Watercolors> set1 =
            EnumSet.range(BRILLIANT_RED, VIRIDIAN_HUE);
        Set<Watercolors> set2 =
            EnumSet.range(CERULEAN_BLUE_HUE, BURNT_UMBER);
        print("set1: " + set1);
        print("set2: " + set2);
        print("union(set1, set2): " + union(set1, set2));
        Set<Watercolors> subset = intersection(set1, set2);
        print("intersection(set1, set2): " + subset);
        print("difference(set1, subset): " +
            difference(set1, subset));
        print("difference(set2, subset): " +
            difference(set2, subset));
        print("complement(set1, set2): " +
            complement(set1, set2));
    }
} /* Output: (Sample)
set1: [BRILLIANT_RED, CRIMSON, MAGENTA, ROSE_MADDER,
VIOLET, CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE,
COBALT_BLUE_HUE, PERMANENT_GREEN, VIRIDIAN_HUE]
set2: [CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE,
COBALT_BLUE_HUE, PERMANENT_GREEN, VIRIDIAN_HUE, SAP_GREEN,
YELLOW_OCHRE, BURNT_SIENNA, RAW_UMBER, BURNT_UMBER]
union(set1, set2): [SAP_GREEN, ROSE_MADDER, YELLOW_OCHRE,
PERMANENT_GREEN, BURNT_UMBER, COBALT_BLUE_HUE, VIOLET,
BRILLIANT_RED, RAW_UMBER, ULTRAMARINE, BURNT_SIENNA,
CRIMSON, CERULEAN_BLUE_HUE, PHTHALO_BLUE, MAGENTA,
VIRIDIAN_HUE]
intersection(set1, set2): [ULTRAMARINE, PERMANENT_GREEN,
COBALT_BLUE_HUE, PHTHALO_BLUE, CERULEAN_BLUE_HUE,
VIRIDIAN_HUE]
difference(set1, subset): [ROSE_MADDER, CRIMSON, VIOLET,
MAGENTA, BRILLIANT_RED]
difference(set2, subset): [RAW_UMBER, SAP_GREEN,
YELLOW_OCHRE, BURNT_SIENNA, BURNT_UMBER]
complement(set1, set2): [SAP_GREEN, ROSE_MADDER,
YELLOW_OCHRE, BURNT_UMBER, VIOLET, BRILLIANT_RED,
RAW_UMBER, BURNT_SIENNA, CRIMSON, MAGENTA]
*///:~
```

В выходных данных приведены результаты выполнения всех операций.

В следующем примере метод `Sets.difference()` демонстрирует различия в методах различных классов `Collection` и `Map` из `java.util`:

```
//: net/mindview/util/ContainerMethodDifferences.java
package net.mindview.util;
import java.lang.reflect.*;
import java.util.*;

public class ContainerMethodDifferences {
    static Set<String> methodSet(Class<?> type) {
        Set<String> result = new TreeSet<String>();
        for(Method m : type.getMethods())
            result.add(m.getName());
        return result;
    }
    static void interfaces(Class<?> type) {
        System.out.print("Interfaces in " +
            type.getSimpleName() + ": ");
        List<String> result = new ArrayList<String>();
        for(Class<?> c : type.getInterfaces())
            result.add(c.getSimpleName());
        System.out.println(result);
    }
    static Set<String> object = methodSet(Object.class);
    static { object.add("clone"); }
    static void
    difference(Class<?> superset, Class<?> subset) {
        System.out.print(superset.getSimpleName() +
            " extends " + subset.getSimpleName() + ", adds: ");
        Set<String> comp = Sets.difference(
            methodSet(superset), methodSet(subset));
        comp.removeAll(object); // Не показывать методы 'Object'
        System.out.println(comp);
        interfaces(superset);
    }
    public static void main(String[] args) {
        System.out.println("Collection: " +
            methodSet(Collection.class));
        interfaces(Collection.class);
        difference(Set.class, Collection.class);
        difference(HashSet.class, Set.class);
        difference(LinkedHashSet.class, HashSet.class);
        difference(TreeSet.class, Set.class);
        difference(List.class, Collection.class);
        difference(ArrayList.class, List.class);
        difference(LinkedList.class, List.class);
        difference(Queue.class, Collection.class);
        difference(PriorityQueue.class, Queue.class);
        System.out.println("Map: " + methodSet(Map.class));
        difference(HashMap.class, Map.class);
        difference(LinkedHashMap.class, HashMap.class);
        difference(SortedMap.class, Map.class);
        difference(TreeMap.class, Map.class);
    }
} ///:~
```

Результаты работы этой программы использовались в разделе «Резюме» главы 11.

17. (4) Изучите документацию JDK по EnumSet и найдите в ней метод clone(). Однако для ссылки на интерфейс Set, передаваемой в Sets.java, метод clone() вызываться не может. Удастся ли вам изменить код Sets.java таким образом, чтобы он обрабатывал как общий случай интерфейса Set, так и особый случай EnumSet, используя clone() вместо создания нового объекта HashSet?

Анонимные внутренние классы

Обобщения также могут использоваться с внутренними классами и анонимными внутренними классами. В следующем примере интерфейс Generator реализуется с использованием анонимных внутренних классов:

```
//: generics/BankTeller.java
// Очень простая модель кассира.
import java.util.*;
import net.mindview.util.*;

class Customer {
    private static long counter = 1;
    private final long id = counter++;
    private Customer() {}
    public String toString() { return "Customer " + id; }
    // Метод для получения объектов Generator:
    public static Generator<Customer> generator() {
        return new Generator<Customer>() {
            public Customer next() { return new Customer(); }
        };
    }
}

class Teller {
    private static long counter = 1;
    private final long id = counter++;
    private Teller() {}
    public String toString() { return "Teller " + id; }
    // Объект Generator:
    public static Generator<Teller> generator =
        new Generator<Teller>() {
            public Teller next() { return new Teller(); }
        };
}

public class BankTeller {
    public static void serve(Teller t, Customer c) {
        System.out.println(t + " serves " + c);
    }
    public static void main(String[] args) {
        Random rand = new Random(47);
        Queue<Customer> line = new LinkedList<Customer>();
        Generators.fill(line, Customer.generator(), 15);
        List<Teller> tellers = new ArrayList<Teller>();
        Generators.fill(tellers, Teller.generator, 4);
        for(Customer c : line)
            serve(tellers.get(rand.nextInt(tellers.size())), c);
    }
}
```

```

} /* Output:
Teller 3 serves Customer 1
Teller 2 serves Customer 2
Teller 3 serves Customer 3
Teller 1 serves Customer 4
Teller 1 serves Customer 5
Teller 3 serves Customer 6
Teller 1 serves Customer 7
Teller 2 serves Customer 8
Teller 3 serves Customer 9
Teller 3 serves Customer 10
Teller 2 serves Customer 11
Teller 4 serves Customer 12
Teller 2 serves Customer 13
Teller 1 serves Customer 14
Teller 1 serves Customer 15
*///:~

```

И `Customer`, и `Teller` имеют закрытые конструкторы, поэтому для создания их экземпляров приходится использовать объекты `Generator`. Класс `Customer` содержит метод `generator()`, который создает новый объект `Generator<Customer>` при каждом вызове, так как множественные объекты `Generator` не понадобятся, и `Teller` создает один открытый объект `generator`. Оба решения продемонстрированы в методах `fill()` внутри `main()`.

Так как и метод `generator()` в `Customer`, и объект `Generator` в `Teller` объявлены статическими, они не могут быть частью интерфейса, так что «обобщить» эту конкретную идиому не удастся. Несмотря на это, в методе `fill()` она работает достаточно хорошо.

Другие разновидности этой задачи рассматриваются в главе 21.

- 18.** (3) По образцу `BankTeller.java` создайте пример, в котором в океане (`Ocean`) большие рыбы (`BigFish`) едят маленьких рыб (`LittleFish`).

Построение сложных моделей

Одним из важных преимуществ обобщений является возможность простого и безопасного создания сложных моделей. Например, можно легко создать контейнер `List` для кортежей:

```

//: generics/TupleList.java
// Создание сложных обобщенных типов
// посредством объединения обобщенных типов.
import java.util.*;
import net.mindview.util.*;

public class TupleList<A,B,C,D>
extends ArrayList<FourTuple<A,B,C,D>> {
    public static void main(String[] args) {
        TupleList<Vehicle, Amphibian, String, Integer> tl =
            new TupleList<Vehicle, Amphibian, String, Integer>();
        tl.add(TupleTest.h());
        tl.add(TupleTest.h());
        for(FourTuple<Vehicle, Amphibian, String, Integer> i: tl)
            System.out.println(i);
    }
}

```

```

} /* Output: (75% match)
(Vehicle@11b86e7, Amphibian@35ce36, hi, 47)
(Vehicle@757aef, Amphibian@d9f9c3, hi, 47)
*///:~

```

И хотя код получается немного пространным (особенно создание итератора), в результате мы получаем достаточно мощную структуру данных при достаточно компактной реализации.

Следующий пример также показывает, как легко строятся сложные модели с применением обобщенных типов. Несмотря на то что каждый отдельный класс создается как «строительный блок», общая модель состоит из множества частей. В данном случае моделируется магазин с рядами (Aisle), полками (Shelf) и товарами (Product):

```

//: generics/Store.java
// Построение сложной модели с использованием
// обобщенных контейнеров.
import java.util.*;
import net.mindview.util.*;

class Product {
    private final int id;
    private String description;
    private double price;
    public Product(int IDnumber, String descr, double price){
        id = IDnumber;
        description = descr;
        this.price = price;
        System.out.println(toString());
    }
    public String toString() {
        return id + ": " + description + ", price: $" + price;
    }
    public void priceChange(double change) {
        price += change;
    }
    public static Generator<Product> generator =
        new Generator<Product>() {
            private Random rand = new Random(47);
            public Product next() {
                return new Product(rand.nextInt(1000), "Test",
                    Math.round(rand.nextDouble() * 1000.0) + 0.99);
            }
        };
}

class Shelf extends ArrayList<Product> {
    public Shelf(int nProducts) {
        Generators.fill(this, Product.generator, nProducts);
    }
}

class Aisle extends ArrayList<Shelf> {
    public Aisle(int nShelves, int nProducts) {
        for(int i = 0; i < nShelves; i++)
            add(new Shelf(nProducts));
    }
}

```

```

class CheckoutStand {}
class Office {}

public class Store extends ArrayList<Aisle> {
    private ArrayList<CheckoutStand> checkouts =
        new ArrayList<CheckoutStand>();
    private Office office = new Office();
    public Store(int nAisles, int nShelves, int nProducts) {
        for(int i = 0; i < nAisles; i++)
            add(new Aisle(nShelves, nProducts));
    }
    public String toString() {
        StringBuilder result = new StringBuilder();
        for(Aisle a : this)
            for(Shelf s : a)
                for(Product p : s) {
                    result.append(p);
                    result.append("\n");
                }
        return result.toString();
    }
    public static void main(String[] args) {
        System.out.println(new Store(14, 5, 10));
    }
} /* Output:
258: Test, price: $400.99
861: Test, price: $160.99
868: Test, price: $417.99
207: Test, price: $268.99
551: Test, price: $114.99
278: Test, price: $804.99
520: Test, price: $554.99
140: Test, price: $530.99
...
*///:~

```

Как видно из `Store.toString()`, в результате создается многоуровневая структура контейнеров, которая тем не менее остается безопасной по отношению к типам и управляемой. Интересно и то, что построение такой модели не требует особых интеллектуальных усилий.

19. (2) По образцу `Store.java` постройте модель грузового судна-контейнеровоза.

Загадка стирания

По мере более основательного изучения обобщений проявляются некоторые аспекты, которые, на первый взгляд, выглядят бессмысленно. Скажем, хотя вы можете использовать конструкцию `ArrayList.class`, запись `ArrayList<Integer>.class` недопустима. Также рассмотрим следующий пример:

```

//: generics/ErasedTypeEquivalence.java
import java.util.*;

public class ErasedTypeEquivalence {

```



```

public static void main(String[] args) {
    Class c1 = new ArrayList<String>().getClass();
    Class c2 = new ArrayList<Integer>().getClass();
    System.out.println(c1 == c2);
}
} /* Output:
true
*///:~

```

Логика подсказывает, что `ArrayList<String>` и `ArrayList<Integer>` являются разными типами. Разные типы ведут себя по-разному; так, при попытке поместить `Integer` в `ArrayList<String>` вы получите другое поведение (ошибка), чем при помещении `Integer` в `ArrayList<Integer>` (успех). И все же из приведенной программы следует, что они относятся к одному типу.

В следующем примере ситуация становится еще более загадочной:

```

//: generics/LostInformation.java
import java.util.*;

class Frob {}
class Fnorkle {}
class Quark<Q> {}
class Particle<POSITION,MOMENTUM> {}

public class LostInformation {
    public static void main(String[] args) {
        List<Frob> list = new ArrayList<Frob>();
        Map<Frob,Fnorkle> map = new HashMap<Frob,Fnorkle>();
        Quark<Fnorkle> quark = new Quark<Fnorkle>();
        Particle<Long,Double> p = new Particle<Long,Double>();
        System.out.println(Arrays.toString(
            list.getClass().getTypeParameters()));
        System.out.println(Arrays.toString(
            map.getClass().getTypeParameters()));
        System.out.println(Arrays.toString(
            quark.getClass().getTypeParameters()));
        System.out.println(Arrays.toString(
            p.getClass().getTypeParameters()));
    }
} /* Output:
[E]
[K, V]
[Q]
[POSITION, MOMENTUM]
*///:~

```

Согласно документации JDK, метод `Class.getTypeParameters()` «возвращает массив объектов `TypeVariable`, представляющих переменные типов, объявленные в объявлении обобщения...» Это наводит на мысль, что вы можете получить информацию о параметрах-типах. Но как видно из результатов, метод возвращает только идентификаторы, представляющие параметры, а эта информация интереса не представляет.

Итак, жестокая правда:

В обобщенном коде информация о параметрах-типах обобщения недоступна.

Вы можете узнать такие аспекты, как идентификатор параметра-типа и ограничения обобщенного типа — но не фактические параметры-типы, использованные для создания конкретного экземпляра. Это обстоятельство, особенно раздражающее программистов с опытом C++, является самой фундаментальной проблемой, с которой вы сталкиваетесь при работе с обобщениями Java.

Обобщения Java реализуются с использованием *стирания* (erasure). Иначе говоря, при использовании обобщения любая конкретная информация о типе теряется. В обобщении известно лишь то, что вы используете объект. Таким образом, `List<String>` и `List<Integer>` во время выполнения по сути являются одним типом. Обе формы «стираются» до «низкоуровневого» типа `List`. Если вы поймете, как работает стирание и как справиться с присущими ему проблемами, вы преодолеете одно из самых серьезных препятствий на пути изучения обобщений Java. В следующем разделе мы займемся этой темой.

Подход C++

Ниже приведен пример использования *шаблонов* C++. Как видите, синтаксис параметризованных типов очень похож, потому что источником вдохновения для создателей Java был язык C++:

```
//: generics/Templates.cpp
#include <iostream>
using namespace std;

template<class T> class Manipulator {
    T obj;
public:
    Manipulator(T x) { obj = x; }
    void manipulate() { obj.f(); }
};

class HasF {
public:
    void f() { cout << "HasF::f()" << endl; }
};

int main() {
    HasF hf;
    Manipulator<HasF> manipulator(hf);
    manipulator.manipulate();
} /* Output:
HasF::f()
///:~
```

Класс `Manipulator` хранит объект типа `T`. Нас здесь интересует метод `manipulate()`, который вызывает метод `f()` для `obj`. Как он может определить, что у параметра-типа `T` существует метод `f()`? Компилятор C++ осуществляет проверку при создании экземпляра шаблона, так что в точке создания экземпляра `Manipulator<HasF>` он убеждается в том, что `HasF` содержит метод `f()`. Если это условие не выполняется, компилятор выдает сообщение об ошибке; таким образом обеспечивается безопасность типов.

На C++ такой код пишется тривиально, потому что при создании шаблона код шаблона знает тип своих параметров. С обобщениями Java дело обстоит иначе. Ниже приведен аналог HasF:

```
//: generics/HasF.java
```

```
public class HasF {  
    public void f() { System.out.println("HasF.f()"); }  
} ///:~
```

Если взять остальной код примера и перевести его на Java, он не откомпилируется:

```
//: generics/Manipulation.java  
// {CompileTimeError} (не компилируется)
```

```
class Manipulator<T> {  
    private T obj;  
    public Manipulator(T x) { obj = x; }  
    // Ошибка: не удастся найти метод f():  
    public void manipulate() { obj.f(); }  
}  
  
public class Manipulation {  
    public static void main(String[] args) {  
        HasF hf = new HasF();  
        Manipulator<HasF> manipulator =  
            new Manipulator<HasF>(hf);  
        manipulator.manipulate();  
    }  
} ///:~
```

Из-за стирания компилятор Java не может связать требование, согласно которому метод `manipulate()` должен быть способен вызвать `f()` для `obj`, с тем фактом, что класс `HasF` содержит метод `f()`. Чтобы вызвать `f()`, необходимо помочь обобщенному классу — передать ему *ограничение*, которое сообщает компилятору, что он должен принимать только типы, удовлетворяющие этому ограничению. Для этой цели снова используется ключевое слово `extends`. Благодаря введению ограничения следующий код успешно компилируется:

```
//: generics/Manipulator2.java
```

```
class Manipulator2<T extends HasF> {  
    private T obj;  
    public Manipulator2(T x) { obj = x; }    public void manipulate() { obj.f(); }  
} ///:~
```

Ограничение `<T extends HasF>` гласит, что `T` должен быть типом `HasF` или типом, производным от `HasF`. Если условие выполняется, то вызов `f()` для `obj` безопасен.

Говорят, что параметр обобщенного типа *стирается до первого ограничения* (как вы вскоре увидите, ограничений может быть несколько). Также используется выражение «стирание параметра-типа». Компилятор заменяет параметр-тип его стиранием, так что в предыдущем примере `T` стирается до `HasF`; это то же самое, что замена `T` на `HasF` в теле класса.

Возможно, вы заметили, что в `Manipulation2.java` обобщения ничего не дают. С таким же успехом вы могли провести стирание самостоятельно и создать класс без обобщения:

```
//: generics/Manipulator3.java
```

```
class Manipulator3 {
    private HasF obj;
    public Manipulator3(HasF x) { obj = x; }
    public void manipulate() { obj.f(); }
} ///:~
```

Мы подошли к важному моменту: обобщения приносят пользу только тогда, когда вы хотите использовать параметры-типы, более «универсальные», чем конкретный тип (и все его подтипы), — то есть если код должен работать в границах нескольких классов. В результате параметры-типы и их применение в практическом обобщенном коде обычно сложнее простой замены класса. Впрочем, это не значит, что любая конструкция вида `<T extends HasF>` бесполезна. Например, если класс содержит метод, который возвращает `T`, обобщения приносят пользу, потому что они возвращают точный тип:

```
//: generics/ReturnGenericType.java
```

```
class ReturnGenericType<T extends HasF> {
    private T obj;
    public ReturnGenericType(T x) { obj = x; }
    public T get() { return obj; }
} ///:~
```

Разработчик должен проанализировать свой код и понять, является ли он «достаточно сложным» для применения обобщений.

Ограничения будут более подробно рассмотрены позже в этой главе.

- 20.** (1) Создайте интерфейс с двумя методами и класс, который реализует этот интерфейс и добавляет еще один метод. В другом классе создайте обобщенный метод с аргументом-типом, ограниченным по интерфейсу; покажите, что методы интерфейса могут вызываться в этом обобщенном методе. В `main()` передайте экземпляр реализующего класса обобщенному методу.

Миграционная совместимость

Чтобы устранить все возможные недоразумения по поводу стирания, необходимо четко понимать, что оно не является особенностью языка. Это компромисс в реализации обобщений Java, неизбежный из-за того, что обобщения не были частью языка с самого начала. Этот компромисс принесет вам немало неприятностей, поэтому постарайтесь пораньше привыкнуть к нему и понять, почему он появился.

Если бы обобщения были частью Java 1.0, то они не были бы реализованы на базе стирания — создатели языка воспользовались бы *конкретизацией* (reification) для поддержания параметров-типов на уровне полноправных сущностей, чтобы с ними можно было выполнять типизованные языковые и рефлексивные операции. Позже в этой главе будет показано, что стирания понижают уровень «обобщенности». Обобщения Java все равно полезны — но не настолько, насколько могли бы, и причиной этому является стирание.

В реализации со стиранием обобщенные типы рассматриваются как «второсортные» типы, которые не могут использоваться в некоторых важных контекстах. Обобщенные

типы присутствуют только во время статической проверки типов, после которой каждый обобщенный тип в программе стирается, то есть заменяется необобщенным верхним ограничением. Например, обозначение вида `List<T>` стирается до `List`, а переменные обычных типов стираются до `Object` (если не задано ограничение).

Основная причина для применения стирания заключается в том, что оно позволяет использовать обобщенный клиентский код с необобщенными библиотеками, и наоборот. Это обстоятельство часто называется «миграционной совместимостью». В идеальном мире когда-нибудь наступит день, когда весь код будет обобщен одновременно. В реальности, даже если программисты будут писать только обобщенный код, им придется иметь дело с необобщенными библиотеками, написанными до Java SE5. Возможно, авторы этих библиотек не видят причин заниматься обобщением своего кода или просто не торопятся.

Итак, обобщения Java должны обеспечивать не только обратную совместимость (существующий код и файлы классов остаются действительными, а их смысл не изменился), но и миграционную совместимость, чтобы авторы библиотеки могли заниматься их обобщением в нужном темпе, а когда библиотека становится обобщенной — она не нарушает работоспособности кода и приложений, которые от нее зависят. Определившись с целями, проектировщики Java и различные группы, работавшие над задачей, решили, что стирание является единственным приемлемым решением. Стирание обеспечивает возможность миграции к обобщению, позволяя необобщенному коду сосуществовать с обобщенным.

Допустим, приложение использует две библиотеки, X и Y, и Y использует библиотеку Z. С выходом Java SE5 создатели этого приложения и библиотек с большой вероятностью захотят перевести свой код на обобщения, но все они будут руководствоваться собственными соображениями и ограничениями в отношении того, когда эта миграция произойдет. Для достижения миграционной совместимости каждая библиотека и приложение не должны зависеть от других в отношении использования обобщений. А следовательно, они не должны иметь возможность обнаружить, использует ли другой код обобщения или нет. Таким образом, все признаки использования обобщений конкретной библиотекой должны быть «стерты».

Без миграционного пути все библиотеки, построенные со временем, могли быть потеряны для разработчиков, решивших перейти на обобщения Java. Пожалуй, из всех составляющих языка библиотеки оказывают наибольшее влияние на производительность, так что этот вариант неприемлем. Было ли стирание лучшим решением проблемы или только возможным миграционным путем? Только время покажет.

Проблема стирания

Итак, главной причиной для реализации стирания является процесс перехода от необобщенного кода к обобщенному и возможность встраивания обобщений в язык без нарушения работоспособности существующих библиотек. Благодаря стиранию существующий необобщенный клиентский код используется без изменений, пока клиенты не будут готовы переписать код для поддержки обобщений. Это стоящая цель, потому что она предотвращает внезапное нарушение работоспособности всего существующего кода.

За стирание приходится заплатить значительную цену. Обобщенные типы не могут использоваться в операциях, в которых явно задействуются типы времени выполнения — приведения типов, операции `instanceof` и выражения `new`. Так как вся информация о типе параметров теряется, при написании обобщенного кода вы должны постоянно напоминать себе, что наличие информации о типе — не более чем видимость. Таким образом, когда вы пишете фрагмент кода вида:

```
class Foo<T> {
    T var;
}
```

может показаться, что при создании экземпляра `Foo`:

```
Foo<Cat> f = new Foo<Cat>();
```

код класса `Foo` должен знать, что теперь он работает с `Cat`. Синтаксис создает впечатление, что тип `T` автоматически подставляется везде, где он используется в классе. Но это не так, и при написании кода класса вы должны постоянно напоминать себе: «Нет, это всего лишь `Object`».

Кроме того, стирание и миграционная совместимость означают, что правильность использования обобщений не обеспечивается там, где это было бы желательно:

```
//: generics/ErasureAndInheritance.java

class GenericBase<T> {
    private T element;
    public void set(T arg) { arg = element; }
    public T get() { return element; }
}

class Derived1<T> extends GenericBase<T> {}

class Derived2 extends GenericBase {} // Без предупреждений

// class Derived3 extends GenericBase<?> {}
// Странная ошибка:
// найден непредвиденный тип : ?
// требуется : класс или интерфейс без ограничений

public class ErasureAndInheritance {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        Derived2 d2 = new Derived2();
        Object obj = d2.get();
        d2.set(obj); // Предупреждение!
    }
} ///:~
```

`Derived2` наследует от `GenericBase` без параметров, и компилятор не предупреждает об этом. Предупреждение не выдается до момента вызова `set()`.

Для отключения предупреждений в Java существуют *аннотации*; пример встречается в этом листинге (данная аннотация не поддерживалась в более ранних версиях Java SE5):

```
@SuppressWarnings("unchecked")
```

Обратите внимание: эта строка включается в метод, генерирующий предупреждение, а не в класс в целом. При подавлении предупреждений следует действовать как можно более избирательно, чтобы массовое отключение предупреждений не скрыло настоящую проблему.

Предполагается, что ошибка, выданная для `Derived3`, означает, что компилятор рассчитывает получить низкоуровневый базовый класс.

Добавьте к этому дополнительный труд по управлению ограничениями, когда вы хотите интерпретировать параметр-тип как нечто большее, чем просто `Object`, — и в результате вы получаете существенно больше усилий при меньшей пользе, чем при использовании параметризованных типов в таких языках, как C++, Ada или Eiffel. Это означает не то, что эти языки в целом работают эффективнее Java для большинства задач программирования, а лишь то, что поддержка параметризованных типов в этих языках обладает большей гибкостью и мощностью, чем в Java.

Граничные ситуации

На мой взгляд, из-за стирания самый нелогичный аспект обобщений заключается в возможности представления того, что не имеет смысла. Пример:

```
//: generics/ArrayMaker.java
import java.lang.reflect.*;
import java.util.*;

public class ArrayMaker<T> {
    private Class<T> kind;
    public ArrayMaker(Class<T> kind) { this.kind = kind; }
    @SuppressWarnings("unchecked")
    T[] create(int size) {
        return (T[])Array.newInstance(kind, size);
    }
    public static void main(String[] args) {
        ArrayMaker<String> stringMaker =
            new ArrayMaker<String>(String.class);
        String[] stringArray = stringMaker.create(9);
        System.out.println(Arrays.toString(stringArray));
    }
} /* Output:
[null, null, null, null, null, null, null, null, null]
*///:~
```

При том, что `kind` хранится в виде `Class<T>`, механизм стирания означает, что в действительности хранится только `Class` без параметра. Таким образом, при выполнении любой операции (например, создании массива) `Array.newInstance()` не располагает информацией типа, подразумеваемой для `kind`; следовательно, метод не может выдать конкретный результат, и возникает необходимость в приведении типа с выдачей предупреждения.

Метод `Array.newInstance()` рекомендуется использовать для создания массивов в обобщениях.

При создании контейнера вместо массива ситуация изменяется.

```

//: generics/ListMaker.java
import java.util.*;

public class ListMaker<T> {
    List<T> create() { return new ArrayList<T>(); }
    public static void main(String[] args) {
        ListMaker<String> stringMaker= new ListMaker<String>();
        List<String> stringList = stringMaker.create();
    }
} ///:~

```

Компилятор не выдает предупреждений, хотя мы знаем (из-за стирания), что `<T>` в `new ArrayList<T>()` внутри `create()` исключается — во время выполнения в классе нет `<T>`, что выглядит бессмысленно. Но если заменить выражение на `new ArrayList()`, компилятор выдает предупреждение.

Но так ли уж бессмысленно происходящее? А если вы поместите объекты в `list` перед тем, как возвращать его, как в следующем примере?

```

//: generics/FilledListMaker.java
import java.util.*;

public class FilledListMaker<T> {
    List<T> create(T t, int n) {
        List<T> result = new ArrayList<T>();
        for(int i = 0; i < n; i++)
            result.add(t);
        return result;
    }
    public static void main(String[] args) {
        FilledListMaker<String> stringMaker =
            new FilledListMaker<String>();
        List<String> list = stringMaker.create("Hello", 4);
        System.out.println(list);
    }
} /* Output:
[Hello, Hello, Hello, Hello]
*///:~

```

Хотя компилятор не может ничего знать о `T` внутри `create()`, он все равно может проследить (во время компиляции) за тем, что значения, помещаемые в `result`, относятся к типу `T`, и поэтому соглашается с `ArrayList<T>`. Итак, хотя стирание удаляет информацию о фактическом типе внутри метода или класса, компилятор все равно может обеспечить внутреннюю целостность использования типа в методе или классе.

Так как стирание удаляет информацию о типе в теле метода, во время выполнения важны *границные точки*, в которых объекты входят в метод и покидают его. В этих точках компилятор выполняет проверку типа во время компиляции и вставляет код приведения типа. Рассмотрим следующий необобщенный пример:

```

//: generics/SimpleHolder.java

public class SimpleHolder {
    private Object obj;
    public void set(Object obj) { this.obj = obj; }
    public Object get() { return obj; }
}

```



```

public static void main(String[] args) {
    SimpleHolder holder = new SimpleHolder();
    holder.set("Item");
    String s = (String)holder.get();
}
} ///:~

```

При декомпиляции результата командой `javap -c SimpleHolder` будет получен следующий результат (после редактирования):

```

public void set(java.lang.Object);
    0:    aload_0
    1:    aload_1
    2:    putfield #2; //Field obj:Object;
    5:    return

public java.lang.Object get();
    0:    aload_0
    1:    getfield #2; //Field obj:Object;
    4:    areturn

public static void main(java.lang.String[]);
    0:    new #3; //class SimpleHolder
    3:    dup
    4:    invokespecial #4; //Method "<init>":()V
    7:    astore_1
    8:    aload_1
    9:    ldc #5; //String Item
   11:    invokevirtual #6; //Method set:(Object;)V
   14:    aload_1
   15:    invokevirtual #7; //Method get:()Object;
   18:    checkcast #8; //class java/lang/String
   21:    astore_2
   22:    return

```

Методы `set()` и `get()` просто сохраняют значение и выдают его, а приведение типа проверяется в точке вызова `get()`.

Теперь вставим в этот код поддержку обобщений:

```

//: generics/GenericHolder.java

public class GenericHolder<T> {
    private T obj;
    public void set(T obj) { this.obj = obj; }
    public T get() { return obj; }
    public static void main(String[] args) {
        GenericHolder<String> holder =
            new GenericHolder<String>();
        holder.set("Item");
        String s = holder.get();
    }
} ///:~

```

Необходимость в приведении типа из `get()` отпала, но мы также знаем, что значение, передаваемое `set()`, проходит проверку типа во время компиляции. Соответствующий байт-код выглядит так:

```

public void set(java.lang.Object);
    0:    aload_0
    1:    aload_1
    2:    putfield #2; //Field obj:Object;
    5:    return

public java.lang.Object get();
    0:    aload_0
    1:    getfield #2; //Field obj:Object;
    4:    areturn

public static void main(java.lang.String[]);
    0:    new #3; //class GenericHolder
    3:    dup
    4:    invokespecial #4; //Method "<init>":()V
    7:    astore_1
    8:    aload_1
    9:    ldc #5; //String Item
   11:    invokevirtual #6; //Method set:(Object;)V
   14:    aload_1
   15:    invokevirtual #7; //Method get:()Object;
   18:    checkcast #8; //class java/lang/String
   21:    astore_2
   22:    return

```

Итоговый код идентичен предыдущему. Дополнительная работа по проверке входного типа в `set()` обходится бесплатно, потому что она выполняется компилятором. А приведение исходящего значения `get()` никуда не делось, но оно автоматически вставляется компилятором, так что код, который вы пишете (и читаете), получается более «чистым».

Так как методы `get()` и `set()` генерируют одинаковый байт-код, все действие происходит в граничных точках — дополнительная проверка входных значений во время компиляции и вставляемое приведение типа для выходных значений. Для прояснения путаницы со стиранием полезно запомнить, что «все реальные действия происходят в граничных точках».

Компенсация стирания

Как мы видели, стирание приводит к невозможности выполнения некоторых операций в обобщенном коде. Любые операции, требующие знания точного типа во время выполнения, работать не будут:

```

//: generics/Erased.java
// {CompileTimeError} (не компилируется)

public class Erased<T> {
    private final int SIZE = 100;
    public static void f(Object arg) {
        if(arg instanceof T) {} // Ошибка
        T var = new T(); // Ошибка
        T[] array = new T[SIZE]; // Ошибка
        T[] array = (T)new Object[SIZE]; // Неконтролируемое предупреждение
    }
} ///:~

```

Иногда эти проблемы можно обойти на программном уровне, но в отдельных случаях стирание приходится компенсировать введением *метки типа*. Это означает явную передачу объекта `Class` для своего типа, чтобы его можно было использовать в выражениях типов.

Например, попытка использования `instanceof` в предыдущей программе завершается неудачей, потому что информация типа была стерта. При введении метки типа можно использовать динамический вызов `isInstance()`:

```
//: generics/ClassTypeCapture.java

class Building {}
class House extends Building {}

public class ClassTypeCapture<T> {
    Class<T> kind;
    public ClassTypeCapture(Class<T> kind) {
        this.kind = kind;
    }
    public boolean f(Object arg) {
        return kind.isInstance(arg);
    }
    public static void main(String[] args) {
        ClassTypeCapture<Building> ctt1 =
            new ClassTypeCapture<Building>(Building.class);
        System.out.println(ctt1.f(new Building()));
        System.out.println(ctt1.f(new House()));
        ClassTypeCapture<House> ctt2 =
            new ClassTypeCapture<House>(House.class);
        System.out.println(ctt2.f(new Building()));
        System.out.println(ctt2.f(new House()));
    }
} /* Output:
true
true
false
true
*///:~
```

Компилятор следит за тем, чтобы метка типа соответствовала обобщенному аргументу.

- 21.** (4) Измените пример `ClassTypeCapture.java`: добавьте в него контейнер `Map<String, Class<?>>`, метод `addType(String typename, Class<?> kind)` и метод `createNew(String typename)`. Метод `createNew()` либо создает новый экземпляр класса, связанный с аргументом-строкой, либо выводит сообщение об ошибке.

Создание экземпляров типов

Попытка создания экземпляра `new T()` в `Erased.java` завершилась неудачей — отчасти из-за стирания, отчасти из-за того, что компилятор не может убедиться в наличии у `T` конструктора по умолчанию (без аргументов). Но в C++ эта операция выполняется естественно, прямолинейно и безопасно (с проверкой во время компиляции):

```
//: generics/InstantiateGenericType.cpp
// C++, не Java!
```

продолжение ➤

```

template<class T> class Foo {
    T x; // Создание поля типа T
    T* y; // Указатель на T
public:
    // Инициализация указателя:
    Foo() { y = new T(); }
};

class Bar {};

int main() {
    Foo<Bar> fb;
    Foo<int> fi; // ...и работает с примитивами
} ///:~

```

В Java проблема решается передачей объекта-фабрики и использованием его для создания нового экземпляра. В качестве фабрики удобно использовать объект `Class`, так что при использовании метки типа для создания нового объекта этого типа можно воспользоваться методом `newInstance()`:

```

//: generics/InstantiateGenericType.java
import static net.mindview.util.Print.*;

class ClassAsFactory<T> {
    T x;
    public ClassAsFactory(Class<T> kind) {
        try {
            x = kind.newInstance();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

class Employee {}

public class InstantiateGenericType {
    public static void main(String[] args) {
        ClassAsFactory<Employee> fe =
            new ClassAsFactory<Employee>(Employee.class);
        print("ClassAsFactory<Employee> succeeded");
        try {
            ClassAsFactory<Integer> fi =
                new ClassAsFactory<Integer>(Integer.class);
        } catch (Exception e) {
            print("ClassAsFactory<Integer> failed");
        }
    }
} /* Output:
ClassAsFactory<Employee> succeeded
ClassAsFactory<Integer> failed
*///:~

```

Программа компилируется, но при выполнении `ClassAsFactory<Integer>` происходит ошибка, потому что `Integer` не имеет конструктора по умолчанию. Так как ошибка не обнаруживается во время компиляции, программисты Sun к таким решениям относятся неодобрительно. Вместо этого они рекомендуют использовать явную фабрику и ограничивать тип, чтобы принимался только класс, реализующий эту фабрику:

```

//: generics/FactoryConstraint.java

interface FactoryI<T> {
    T create();
}

class Foo2<T> {
    private T x;
    public <F extends FactoryI<T>> Foo2(F factory) {
        x = factory.create();
    }
    // ...
}

class IntegerFactory implements FactoryI<Integer> {
    public Integer create() {
        return new Integer(0);
    }
}

class Widget {
    public static class Factory implements FactoryI<Widget> {
        public Widget create() {
            return new Widget();
        }
    }
}

public class FactoryConstraint {
    public static void main(String[] args) {
        new Foo2<Integer>(new IntegerFactory());
        new Foo2<Widget>(new Widget.Factory());
    }
} ///:~

```

Это решение — всего лишь разновидность передачи `Class<T>`. В обоих решениях используются фабрики; просто `Class<T>` — встроенный объект-фабрика, а в приведенном решении фабрика создается явно. Но в этом случае вы можете пользоваться преимуществами проверки на стадии компиляции.

Другое возможное решение основано на паттерне проектирования «Шаблонный метод». В следующем примере `get()` является шаблонным методом, а `create()` определяется в субклассе для создания объекта соответствующего типа:

```

//: generics/CreatorGeneric.java

abstract class GenericWithCreate<T> {
    final T element;
    GenericWithCreate() { element = create(); }
    abstract T create();
}

class X {}

class Creator extends GenericWithCreate<X> {
    X create() { return new X(); }
    void f() {

```

продолжение ↗

```

        System.out.println(element.getClass().getSimpleName());
    }
}
public class CreatorGeneric {
    public static void main(String[] args) {
        Creator c = new Creator();
        c.f();
    }
} /* Output:
X
*///:~

```

22. (6) Используя метку типа в сочетании с отражением, создайте метод, использующий версию `newInstance()` с аргументами для создания объекта класса, конструктор которого получает аргументы.
23. (1) Измените пример `FactoryConstraint.java`, чтобы метод `create()` получал аргумент.
24. (3) Измените упражнение 21 так, чтобы объекты-фабрики хранились в `Map` вместо `Class<?>`.

Массивы обобщений

Как было показано в `Erased.java`, создать массив обобщений невозможно. Как правило, проблема решается использованием `ArrayList` везде, где возникает потребность в создании массива обобщений:

```

//: generics/ListOfGenerics.java
import java.util.*;

public class ListOfGenerics<T> {
    private List<T> array = new ArrayList<T>();
    public void add(T item) { array.add(item); }
    public T get(int index) { return array.get(index); }
} ///:~

```

Здесь мы получаем поведение массива, но с безопасностью типов времени компиляции, которую предоставляют обобщения.

Время от времени все равно появляется необходимость создания массива обобщенных типов (например, массивы используются во внутренней реализации `ArrayList`). Интересно, что *ссылку* можно определить так, чтобы компилятор не жаловался, например:

```

//: generics/ArrayOfGenericReference.java

class Generic<T> {}
public class ArrayOfGenericReference {
    static Generic<Integer>[] gia;
} ///:~

```

Компилятор принимает этот фрагмент, не выдавая никаких предупреждений. Но вам никогда не удастся создать массив этого конкретного типа (включая параметры-типы), так что это обстоятельство немного сбивает с толку. Так как все массивы имеют одинаковую структуру (размер каждой ячейки массива и размещение в памяти) независимо от хранимого типа, казалось бы, вы можете создать массив `Object` и привести его

к нужному типу. Такое решение компилируется, но не работает, выдавая исключение `ClassCastException`:

```
//: generics/ArrayOfGeneric.java
```

```
public class ArrayOfGeneric {
    static final int SIZE = 100;
    static Generic<Integer>[] gia;
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        // Компилируется; выдает ClassCastException:
        ///! gia = (Generic<Integer>[])new Object[SIZE];
        // Тип времени выполнения - низкоуровневый (стертый) тип:
        gia = (Generic<Integer>[])new Generic[SIZE];
        System.out.println(gia.getClass().getSimpleName());
        gia[0] = new Generic<Integer>();
        ///! gia[1] = new Object(); // Ошибка времени компиляции
        // Обнаруживается несоответствие типов:
        ///! gia[2] = new Generic<Double>();
    }
} /* Output:
Generic[]
*///:~
```

Проблема в том, что массивы хранят свой фактический тип, который устанавливается в момент создания массива. Итак, хотя ссылка `gia` была приведена к `Generic<Integer>[]`, эта информация существует только во время компиляции (и без аннотации `@SuppressWarnings` вы получите предупреждение). Во время выполнения она остается массивом `Object`, и это создает проблемы. Единственный успешный способ создания массива обобщенного типа — создание нового массива «стертого» типа с последующим приведением типа.

Рассмотрим чуть более сложный пример. Допустим, существует простая обобщенная обертка для массива:

```
//: generics/GenericArray.java
```

```
public class GenericArray<T> {
    private T[] array;
    @SuppressWarnings("unchecked")
    public GenericArray(int sz) {
        array = (T[])new Object[sz];
    }
    public void put(int index, T item) {
        array[index] = item;
    }
    public T get(int index) { return array[index]; }
    // Предоставляет нижележащее представление:
    public T[] rep() { return array; }
    public static void main(String[] args) {
        GenericArray<Integer> gai =
            new GenericArray<Integer>(10);
        // Приводит к возбуждению ClassCastException:
        ///! Integer[] ia = gai.rep();
        // А это допустимо:
        Object[] oa = gai.rep();
    }
} ///:~
```

Как и прежде, использовать конструкцию `T[] array = new T[sz]` нельзя, так что мы создаем массив объектов и выполняем приведение типа.

Метод `rep()` возвращает `T[]`, что для `gal` в `main()` соответствует `Integer[]`, но при попытке вызвать его и сохранить результат по ссылке на `Integer[]` выдается исключение `ClassCastException` — снова потому, что фактическим типом времени выполнения является `Object[]`.

Если откомпилировать пример `GenericArray.java` с закомментированной аннотацией `@SuppressWarnings`, компилятор выдает предупреждение:

Внимание: В `GenericArray.java` используются неконтролируемые или небезопасные операции.
Внимание: Перекомпилируйте с ключом `-Xlint:unchecked` для получения дополнительной информации.

В данном случае выдается одно предупреждение — вероятно, связанное с приведением типа. Но для полной уверенности следует откомпилировать программу с флагом `-Xlint:unchecked`:

```
GenericArray.java:7: warning: [unchecked] unchecked cast
found   : java.lang.Object[]
required: T[]
    array = (T[])new Object[sz];
              ^
```

1 warning

Действительно, компилятор недоволен приведением типов. Так как предупреждения увеличивают объем выводимой информации, вероятно, лучшее, что можно сделать после проверки, — отключить предупреждение аннотацией `@SuppressWarnings`. Когда компилятор выводит предупреждение, мы проверяем его.

Из-за стирания типом массива во время выполнения может быть только `Object[]`. Если немедленно привести его к `T[]`, то при компиляции фактический тип массива теряется, и компилятор может упустить некоторые потенциальные проверки ошибок. По этой причине в коллекции лучше использовать `Object[]`, а выполнять приведение к `T` при использовании элемента массива. Посмотрим, как это выглядит в примере `GenericArray.java`:

```
//: generics/GenericArray2.java

public class GenericArray2<T> {
    private Object[] array;
    public GenericArray2(int sz) {
        array = new Object[sz];
    }
    public void put(int index, T item) {
        array[index] = item;
    }
    @SuppressWarnings("unchecked")
    public T get(int index) { return (T)array[index]; }
    @SuppressWarnings("unchecked")
    public T[] rep() {
        return (T[])array; // Предупреждение: неконтролируемое приведение типа
    }
    public static void main(String[] args) {
```



```

GenericArray2<Integer> gai =
    new GenericArray2<Integer>(10);
for(int i = 0; i < 10; i++)
    gai.put(i, i);
for(int i = 0; i < 10; i++)
    System.out.print(gai.get(i) + " ");
System.out.println();
try {
    Integer[] ia = gai.rep();
} catch(Exception e) { System.out.println(e); }
}
} /* Output: (Sample)
0 1 2 3 4 5 6 7 8 9
java.lang.ClassCastException: [Ljava.lang.Object; cannot be
cast to [Ljava.lang.Integer;
*///:~

```

На первый взгляд почти ничего не изменилось — разве что переместилось приведение типа. Без аннотаций `@SuppressWarnings` будут выдаваться «неконтролируемые» преобразования. Тем не менее внутренним представлением теперь является `Object[]`, а не `T[]`. При вызове `get()` метод преобразует объект к `T`, т. е. правильному типу, так что операция безопасна. Однако при вызове `rep()` снова совершается попытка преобразования `Object[]` в `T[]` — по-прежнему некорректная, поэтому во время компиляции выдается предупреждение, а во время выполнения возбуждается исключение. «Подделать» тип нижележащего массива невозможно — им может быть только `Object[]`. Внутреннее представление массива в форме `Object[]` вместо `T[]` снижает опасность того, что вы забудете тип массива на стадии выполнения и случайно внесете ошибку (хотя большинство таких ошибок будет быстро обнаружено во время выполнения).

В новом коде следует передать маркер типа. В таком случае класс `GenericArray` принимает следующий вид:

```

//: generics/GenericArrayWithTypeToken.java
import java.lang.reflect.*;

public class GenericArrayWithTypeToken<T> {
    private T[] array;
    @SuppressWarnings("unchecked")
    public GenericArrayWithTypeToken(Class<T> type, int sz) {
        array = (T[])Array.newInstance(type, sz);
    }
    public void put(int index, T item) {
        array[index] = item;
    }
    public T get(int index) { return array[index]; }
    // Предоставляет нижележащее представление:
    public T[] rep() { return array; }
    public static void main(String[] args) {
        GenericArrayWithTypeToken<Integer> gai =
            new GenericArrayWithTypeToken<Integer>(
                Integer.class, 10);
        // Теперь работает:
        Integer[] ia = gai.rep();
    }
} //:~

```

Маркер типа `Class<T>` передается конструктору для восстановления последствий стирания, чтобы мы могли создать фактический тип нужного массива, хотя предупреждения от приведения типа приходится подавлять аннотацией `@SuppressWarnings`. После получения фактического типа можно вернуть его и получить нужные результаты, как показано в `main()`. Типом массива на стадии выполнения является точный тип `T[]`.

К сожалению, в исходном коде стандартных библиотек Java SE5 приведения массивов `Object` к параметризованным типам встречаются сплошь и рядом. Например, вот как выглядит конструктор `ArrayList` по данным `Collection` после небольшой чистки и упрощения:

```
public ArrayList(Collection c) {
    size = c.size();
    elementData = (E[])new Object[size];
    c.toArray(elementData);
}
```

Просмотрев код `ArrayList.java`, вы найдете много таких преобразований. И что произойдет при его компиляции?

Внимание: В `ArrayList.java` используются неконтролируемые или небезопасные операции.
Внимание: Перекомпилируйте с ключом `-Xlint:unchecked` для получения дополнительной информации.

Безусловно, стандартные библиотеки выдают множество предупреждений. Если вы работали на языке C (особенно до выхода ANSI C), то вы помните особенность предупреждений: когда вы понимаете, что предупреждения можно проигнорировать, вы так и делаете. По этой причине лучше не провоцировать никакие сообщения от компилятора, если программист не должен что-то сделать с ними.

В своем блоге¹ Нил Гафтер (один из ведущих разработчиков Java SE5) сообщает, что он поленился при переписывании библиотек Java и что мы не должны делать то, что сделал он. Нил также указывает, что ему не удалось исправить некоторый библиотечный код Java без нарушения существующего интерфейса. Итак, даже если какие-то идиомы присутствуют в исходном коде библиотек Java, это не означает, что это правильное решение. Не стоит полагать, что код библиотек — это образец, которому стоит следовать в вашем коде.

Ограничения

Ограничения были кратко представлены ранее в этой главе. Они сужают диапазон параметров-типов, которые могут использоваться с обобщениями. И хотя это позволяет вам задавать свои правила относительно типов, к которым могут применяться обобщения, более важным может оказаться другой эффект: возможность вызова методов из ограниченных типов.

Так как стирание удаляет информацию типа, для неограничиваемых параметров обобщений можно вызывать только методы, доступные для `Object`. Но при ограничении параметра подмножеством типов вы сможете вызывать методы этого подмножества.

¹ <http://gafter.blogspot.com/2004/09/puzzling-through-erasure-answer.html>

Для установления этого ограничения обобщения Java используют ключевое слово `extends`. Важно понимать, что в контексте обобщений `extends` имеет совершенно иной смысл, чем обычно. Следующий пример демонстрирует основные принципы использования ограничений:

```
//: generics/BasicBounds.java

interface HasColor { java.awt.Color getColor(); }

class Colored<T extends HasColor> {
    T item;
    Colored(T item) { this.item = item; }
    T getItem() { return item; }
    // Ограничение позволяет вызывать метод:
    java.awt.Color color() { return item.getColor(); }
}

class Dimension { public int x, y, z; }

// Не работает -- сначала должен быть указан класс, затем интерфейсы:
// class ColoredDimension<T extends HasColor & Dimension> {

// Множественные ограничения:
class ColoredDimension<T extends Dimension & HasColor> {
    T item;
    ColoredDimension(T item) { this.item = item; }
    T getItem() { return item; }
    java.awt.Color color() { return item.getColor(); }
    int getX() { return item.x; }
    int getY() { return item.y; }
    int getZ() { return item.z; }
}

interface Weight { int weight(); }

// Как и в случае с наследованием, можно указать только один
// конкретный класс, но несколько интерфейсов:
class Solid<T extends Dimension & HasColor & Weight> {
    T item;
    Solid(T item) { this.item = item; }
    T getItem() { return item; }
    java.awt.Color color() { return item.getColor(); }
    int getX() { return item.x; }
    int getY() { return item.y; }
    int getZ() { return item.z; }
    int weight() { return item.weight(); }
}

class Bounded
extends Dimension implements HasColor, Weight {
    public java.awt.Color getColor() { return null; }
    public int weight() { return 0; }
}

public class BasicBounds {
    public static void main(String[] args) {
        Solid<Bounded> solid =
```

продолжение ➤

```

        new Solid<Bounded>(new Bounded());
        solid.color();
        solid.getY();
        solid.weight();
    }
} ///:~

```

Возможно, вы заметили, что пример `BasicBounds.java` содержит избыточность, которую можно устранить посредством наследования. В следующем примере видно, что каждый уровень наследования также добавляет ограничения:

```
///: generics/InheritBounds.java
```

```

class HoldItem<T> {
    T item;
    HoldItem(T item) { this.item = item; }
    T getItem() { return item; }
}

class Colored2<T extends HasColor> extends HoldItem<T> {
    Colored2(T item) { super(item); }
    java.awt.Color color() { return item.getColor(); }
}

class ColoredDimension2<T extends Dimension & HasColor>
extends Colored2<T> {
    ColoredDimension2(T item) { super(item); }
    int getX() { return item.x; }
    int getY() { return item.y; }
    int getZ() { return item.z; }
}

class Solid2<T extends Dimension & HasColor & Weight>
extends ColoredDimension2<T> {
    Solid2(T item) { super(item); }
    int weight() { return item.weight(); }
}

public class InheritBounds {
    public static void main(String[] args) {
        Solid2<Bounded> solid2 =
            new Solid2<Bounded>(new Bounded());
        solid2.color();
        solid2.getY();
        solid2.weight();
    }
} ///:~

```

`HoldItem` просто хранит объект; это поведение наследуется классом `Colored2`, который также требует, чтобы его параметр реализовал `HasColor`. `ColoredDimension2` и `Solid2` расширяют иерархию и добавляют ограничения на каждом уровне. Теперь методы наследуются и их не нужно повторять для каждого класса.

Пример с большим количеством уровней:

```

///: generics/EpicBattle.java
// Demonstrating bounds in Java generics.
import java.util.*;

```

```

interface SuperPower {}
interface XRayVision extends SuperPower {
    void seeThroughWalls();
}
interface SuperHearing extends SuperPower {
    void hearSubtleNoises();
}
interface SuperSmell extends SuperPower {
    void trackBySmell();
}

class SuperHero<POWER> extends SuperPower {
    POWER power;
    SuperHero(POWER power) { this.power = power; }
    POWER getPower() { return power; }
}

class SuperSleuth<POWER> extends XRayVision
extends SuperHero<POWER> {
    SuperSleuth(POWER power) { super(power); }
    void see() { power.seeThroughWalls(); }
}

class CanineHero<POWER> extends SuperHearing & SuperSmell
extends SuperHero<POWER> {
    CanineHero(POWER power) { super(power); }
    void hear() { power.hearSubtleNoises(); }
    void smell() { power.trackBySmell(); }
}

class SuperHearSmell implements SuperHearing, SuperSmell {
    public void hearSubtleNoises() {}
    public void trackBySmell() {}
}

class DogBoy extends CanineHero<SuperHearSmell> {
    DogBoy() { super(new SuperHearSmell()); }
}

public class EpicBattle {
    // Ограничения в обобщенных методах:
    static <POWER extends SuperHearing>
    void useSuperHearing(SuperHero<POWER> hero) {
        hero.getPower().hearSubtleNoises();
    }
    static <POWER extends SuperHearing & SuperSmell>
    void superFind(SuperHero<POWER> hero) {
        hero.getPower().hearSubtleNoises();
        hero.getPower().trackBySmell();
    }
    public static void main(String[] args) {
        DogBoy dogBoy = new DogBoy();
        useSuperHearing(dogBoy);
        superFind(dogBoy);
        // Так поступить можно:
        List<? extends SuperHearing> audioBoys;
        // А так нельзя:
        // List<? extends SuperHearing & SuperSmell> dogBoys;
    }
} ///:~

```

Заметьте, что маски (которые будут рассмотрены в следующем разделе) действуют только в одном ограничении.

- 25.** (2) Создайте два интерфейса и класс, реализующий оба интерфейса. Создайте два обобщенных метода; у одного параметр-тип ограничивается первым интерфейсом, а у другого — вторым интерфейсом. Создайте экземпляр класса, который реализует оба интерфейса, и покажите, что он может использоваться с обоими обобщенными методами.

Маски

Мы уже видели простые примеры использования масок (вопросительные знаки в выражениях обобщенных аргументов) в главах 11 и 14. В этом разделе тема рассматривается более подробно.

Начнем с примера, демонстрирующего особое поведение массивов: массив производного типа можно присвоить ссылке на массив базового типа:

```
//: generics/CovariantArrays.java

class Fruit {}
class Apple extends Fruit {}
class Jonathan extends Apple {}
class Orange extends Fruit {}

public class CovariantArrays {
    public static void main(String[] args) {
        Fruit[] fruit = new Apple[10];
        fruit[0] = new Apple(); // OK
        fruit[1] = new Jonathan(); // OK
        // Тип времени выполнения – Apple[], а не Fruit[] и не Orange[]:
        try {
            // Компилятор позволяет добавить Fruit:
            fruit[0] = new Fruit(); // ArrayStoreException
        } catch (Exception e) { System.out.println(e); }
        try {
            // Компьютер позволяет добавлять Orange:
            fruit[0] = new Orange(); // ArrayStoreException
        } catch (Exception e) { System.out.println(e); }
    }
} /* Output:
java.lang.ArrayStoreException: Fruit
java.lang.ArrayStoreException: Orange
*///:~
```

Первая строка `main()` создает массив `Apple` и присваивает его ссылке на массив `Fruit`. Это разумно — яблоко (`Apple`) является разновидностью фрукта (`Fruit`), так что массив элементов `Apple` также может рассматриваться как массив `Fruit`.

Но если фактическим типом массива является `Apple[]`, то в массив могут помещаться только объекты типа `Apple` или типов, производных от `Apple`, что фактически работает как во время компиляции, так и во время выполнения. Но обратите внимание: компилятор позволяет поместить объект `Fruit` в массив. Для компилятора это выглядит

логично, потому что он работает со ссылкой `Fruit[]`, — так почему бы не разрешить включение в массив объекта `Fruit` или типа, производного от `Fruit` (например, `Orange`)? Во время компиляции это допустимо. Однако ядро времени выполнения знает, что оно имеет дело с `Apple[]`, и выдает исключение при помещении в массив постороннего типа.

Пожалуй, термин «восходящее преобразование» здесь не совсем точен — в действительности происходит присваивание одного массива другому. Хотя массив содержит другие объекты, из-за возможности восходящего преобразования очевидно, что объект массива может обеспечить выполнение требований к типу содержащихся в нем объектов. Массив «знает», что в нем хранится, так что между проверками времени компиляции и проверками времени выполнения никакие злоупотребления невозможны.

Для массивов такая ситуация не столь ужасна, потому что вы сможете узнать во время выполнения о вставке неподходящего типа. Но одной из основных целей обобщений является обнаружение подобных ошибок во время компиляции. Что же происходит при попытке использования обобщенных контейнеров вместо массивов?

```
//: generics/NonCovariantGenerics.java
// {CompileTimeError} (не компилируется)
import java.util.*;

public class NonCovariantGenerics {
    // Ошибка компиляции: несовместимые типы:
    List<Fruit> flist = new ArrayList<Apple>();
} ///:~
```

И хотя на первый взгляд это читается как «Контейнер с элементами `Apple` нельзя присвоить контейнеру `Fruit`», помните, что применение обобщений не сводится к контейнерам. По сути говорится другое: «Обобщение с `Apple` не может быть присвоено обобщению с `Fruit`». Если (как в случае с массивами) компилятор знает о коде достаточно, чтобы понять, что в нем задействованы контейнеры, возможно, он мог бы немного смягчить требования. Но компилятор такой информацией не располагает и поэтому отказывается разрешать «восходящее преобразование». Впрочем, на деле это не является «восходящим преобразованием» — контейнер `List` с элементами `Apple` не является `List` с элементами `Fruit`. В контейнере `List` с элементами `Apple` могут храниться объекты типа `Apple` и типов, производных от `Apple`, а в контейнере `List` с элементами `Fruit` могут храниться любые разновидности `Fruit`. Да, в том числе и `Apple`, но от этого контейнер не становится контейнером `List` с элементами `Apple`; он остается контейнером `List` с элементами `Fruit`. Контейнер `List` с элементами `Apple` не эквивалентен по типу контейнеру `List` с элементами `Fruit`, несмотря на то что `Apple` является разновидностью `Fruit`.

Настоящая проблема заключается в том, что речь идет о типе *контейнера*, а не о типе элементов, хранящихся в контейнере. В отличие от массивов обобщения не обладают встроенной ковариантностью. Это связано с тем, что массивы полностью определяются в языке, а следовательно, для них реализованы встроенные проверки как на стадии компиляции, так и на стадии выполнения, тогда как с обобщениями компилятор и исполнительная система не знают, что вы хотите делать со своими типами и какие при этом должны использоваться правила.

Однако в некоторых ситуациях между двумя типами желательно установить некую разновидность отношений восходящего преобразования. Для этой цели и используются *маски* (wildcards).

```
//: generics/GenericsAndCovariance.java
import java.util.*;

public class GenericsAndCovariance {
    public static void main(String[] args) {
        // Маски обеспечивают ковариантность:
        List<? extends Fruit> flist = new ArrayList<Apple>();
        // Ошибка компиляции: добавление объектов невозможно:
        // flist.add(new Apple());
        // flist.add(new Fruit());
        // flist.add(new Object());
        flist.add(null); // Допустимо, но неинтересно
        // Мы знаем, что возвращается как минимум Fruit:
        Fruit f = flist.get(0);
    }
} ///:~
```

Теперь `flist` имеет тип `List<? extends Fruit>`, что можно прочесть как «список с элементами любого типа, производного от `Fruit`». Однако это не означает, что в `List` можно будет хранить любые виды `Fruit`. Маска относится к определенному типу, так что это означает «некоторый конкретный тип, не указанный для ссылки `flist`». Итак, присваиваемый объект `List` должен содержать некоторый указанный тип (например, `Fruit` или `Apple`), но для восходящего преобразования к `flist` этот тип несущественен.

Если единственное ограничение состоит в том, что `List` может содержать объекты `Fruit` или типа, производного от `Fruit`, но для вас несущественно, что это за тип, то что можно сделать с таким контейнером `List`? Если вы не знаете, какой тип хранится в `List`, как безопасно добавить объект? Как и в случае с «восходящим преобразованием» массива в `CovariantArrays.java`, это невозможно — только на этот раз запрет исходит от компилятора, а не от исполнительной системы. Проблема обнаруживается быстрее.

Кто-то скажет, что это уже перебор, потому что теперь мы не можем даже добавить `Apple` в контейнер `List`, в котором, как только что было сказано, будут храниться объекты `Apple`. Да, но ведь компилятор этого не знает! `List<? extends Fruit>` может на законных основаниях указывать на `List<Orange>`. После такого «восходящего преобразования» теряется способность передачи чего-либо, даже `Object`.

С другой стороны, при вызове метода, возвращающего `Fruit`, такой подход безопасен; известно, что все объекты в списке должны относиться как минимум к типу `Fruit`, поэтому компилятор не возражает.

26. (2) Продемонстрируйте ковариантность массивов на примере `Number` и `Integer`.

27. (2) Покажите, что ковариантность не работает с `List` на примере `Number` и `Integer`, затем добавьте шаблоны.

Насколько умен компилятор?

Создается впечатление, что компилятор не позволит вызывать методы, получающие аргументы, но возьмем следующий пример:


```
//: generics/CompilerIntelligence.java
import java.util.*;

public class CompilerIntelligence {
    public static void main(String[] args) {
        List<? extends Fruit> flist =
            Arrays.asList(new Apple());
        Apple a = (Apple)flist.get(0); // Без предупреждений
        flist.contains(new Apple()); // Аргумент 'Object'
        flist.indexOf(new Apple());  // Аргумент 'Object'
    }
} ///:~
```

В примере вызываются методы `contains()` и `indexOf()`, получающие объекты `Apple` в аргументах, и все проходит нормально. Означает ли это, что компилятор в действительности анализирует код и проверяет, изменяет ли конкретный метод свой объект?

Просматривая документацию `ArrayList`, мы видим, что компилятор не настолько умен. Если `add()` получает аргумент с параметром-типом обобщения, `contains()` и `indexOf()` получают аргументы типа `Object`. Таким образом, с объявлением `ArrayList<? extends Fruit>` аргумент `add()` превращается в `? extends Fruit`. По этому описанию компилятор не может узнать, какой конкретный подтип `Fruit` здесь нужен, и поэтому не принимает никакие типы `Fruit`. Даже если вы предварительно выполнили восходящее преобразование `Apple` к `Fruit`, компилятор просто отказывается вызывать метод (такой, как `add()`), если в списке аргументов присутствует маска.

У `contains()` и `indexOf()` аргументы относятся к типу `Object`, так что маски не задействованы и компиляторы разрешают вызов. Это означает, что проектировщик обобщенного класса сам решает, какие вызовы «безопасны», и использует типы `Object` для их аргументов. Чтобы запретить вызов при использовании типа с масками, используйте параметр-тип в списке аргументов.

Для примера рассмотрим очень простой класс `Holder`:

```
//: generics/Holder.java

public class Holder<T> {
    private T value;
    public Holder() {}
    public Holder(T val) { value = val; }
    public void set(T val) { value = val; }
    public T get() { return value; }
    public boolean equals(Object obj) {
        return value.equals(obj);
    }
    public static void main(String[] args) {
        Holder<Apple> Apple = new Holder<Apple>(new Apple());
        Apple d = Apple.get();
        Apple.set(d);
        // Holder<Fruit> Fruit = Apple; // Не может выполнить
        //                               // восходящее преобразование
        Holder<? extends Fruit> fruit = Apple; // OK
        Fruit p = fruit.get();
        d = (Apple)fruit.get(); // Возвращает 'Object'
        try {
```

продолжение ➤

```

    Orange c = (Orange)fruit.get(); // Без предупреждений
} catch(Exception e) { System.out.println(e); }
// fruit.set(new Apple()); // Вызов set() невозможен
// fruit.set(new Fruit()); // Вызов set() невозможен
System.out.println(fruit.equals(d)); // OK
}
} /* Output: (Sample)
java.lang.ClassCastException: Apple cannot be cast to
Orange
true
*///:~

```

Класс `Holder` содержит метод `set()`, который получает `T`; метод `get()`, который получает `T`, и метод `equals()`, который получает `Object`. Как вы уже видели, при создании объекта `Holder<Apple>` вы не сможете выполнить его восходящее преобразование в `Holder<Fruit>`, но можно повысить до `Holder<? extends Fruit>`. Если вызвать `get()`, метод вернет `Fruit` — это максимальная информация, которой он располагает при наличии ограничения «любой класс, расширяющий `Fruit`». Если у вас больше информации, вы можете преобразовать его к конкретному подтипу `Fruit` без выдачи предупреждений, но возникает исключение `ClassCastException`. Метод `set()` не будет работать с `Apple` или `Fruit`, потому что аргумент `set()` тоже объявлен как «`? Extends Fruit`»; это означает, что им может быть что угодно, а компилятор не может проверить безопасность типов для «чего угодно».

Однако метод `equals()` работает нормально, потому что он получает аргумент `Object` вместо `T`. Таким образом, компилятор только обращает внимание на типы передаваемых и возвращаемых объектов. Он не анализирует код, чтобы проверить, выполняются ли в нем фактические операции чтения или записи.

Контравариантность

Также можно пойти другим путем и использовать *ограничения супертипа*. В этом случае разработчик устанавливает ограничение по любому базовому классу некоторого класса конструкцией `<? super MyClass>` или даже параметру-типу: `<? super T>` (хотя вы не сможете задать ограничение супертипа для обобщенного параметра, то есть конструкция `<T super MyClass>` запрещена). Это позволяет безопасно передать типизованный объект обобщенному типу. Таким образом, ограничение супертипа делает возможной запись в `Collection`:

```

//: generics/SuperTypeWildcards.java
import java.util.*;

public class SuperTypeWildcards {
    static void writeTo(List<? super Apple> apples) {
        apples.add(new Apple());
        apples.add(new Jonathan());
        // apples.add(new Fruit()); // Ошибка
    }
} ///:~

```

Аргумент `apples` содержит `List` типа, являющегося базовым для `Apple`; соответственно, мы знаем, что добавление `Apple` или подтипа `Apple` безопасно. Так как `Apple` определяет «нижнюю границу», мы не знаем, безопасно ли добавить `Fruit` в такой контейнер `List`,

потому что это откроет `List` для добавления типов, отличных от `Apple`, и приведет к нарушению статической безопасности типов.

Таким образом, ограничения подтипа и супертипа можно рассматривать в контексте возможности «записи» (передачи методу) в обобщенный тип и «чтения» (возвращения из метода) из обобщенного типа.

Ограничения супертипа ослабляют требования к тому, что может передаваться методу:

```
//: generics/GenericWriting.java
import java.util.*;

public class GenericWriting {
    static <T> void writeExact(List<T> list, T item) {
        list.add(item);
    }
    static List<Apple> apples = new ArrayList<Apple>();
    static List<Fruit> fruit = new ArrayList<Fruit>();
    static void f1() {
        writeExact(apples, new Apple());
        // writeExact(fruit, new Apple()); // Ошибка:
        // Несовместимые типы: обнаружен Fruit, требуется Apple
    }
    static <T> void
    writeWithWildcard(List<? super T> list, T item) {
        list.add(item);
    }
    static void f2() {
        writeWithWildcard(apples, new Apple());
        writeWithWildcard(fruit, new Apple());
    }
    public static void main(String[] args) { f1(); f2(); }
} ///:~
```

Метод `writeExact()` использует точный параметр-тип (без масок). В `f1()` мы видим, что вызов работает нормально — при условии, что в `List<Apple>` помещаются только объекты `Apple`. Однако метод `writeExact()` не позволяет поместить `Apple` в `List<Fruit>`, хотя вы и знаете, что это возможно.

В методе `writeWithWildcard()` аргумент имеет тип `List<? super T>`, так что `List` содержит конкретный тип, производный от `T`; следовательно, в аргументе методов `List` можно безопасно передать `T` или любой тип, производный от `T`. В этом можно убедиться в `f2()`, где тип `Apple`, как и прежде, можно поместить в `List<Apple>`, но также появилась возможность поместить `Apple` в `List<Fruit>`.

Аналогичный анализ можно провести для изучения ковариантности и ограничений:

```
//: generics/GenericReading.java
import java.util.*;

public class GenericReading {
    static <T> T readExact(List<T> list) {
        return list.get(0);
    }
    static List<Apple> apples = Arrays.asList(new Apple());
    static List<Fruit> fruit = Arrays.asList(new Fruit());
    // Статический метод адаптируется к каждому вызову:
```

продолжение ➤

```

static void f1() {
    Apple a = readExact(apples);
    Fruit f = readExact(fruit);
    f = readExact(apples);
}
// Но если речь идет о классе, тип устанавливается
// при создании экземпляра класса:
static class Reader<T> {
    T readExact(List<T> list) { return list.get(0); }
}
static void f2() {
    Reader<Fruit> fruitReader = new Reader<Fruit>();
    Fruit f = fruitReader.readExact(fruit);
    // Fruit a = fruitReader.readExact(apples); // Ошибка:
    // readExact(List<Fruit>) не может
    // применяться к (List<Apple>).
}
static class CovariantReader<T> {
    T readCovariant(List<? extends T> list) {
        return list.get(0);
    }
}
static void f3() {
    CovariantReader<Fruit> fruitReader =
        new CovariantReader<Fruit>();
    Fruit f = fruitReader.readCovariant(fruit);
    Fruit a = fruitReader.readCovariant(apples);
}
public static void main(String[] args) {
    f1(); f2(); f3();
}
} ///:~

```

Как и прежде, первый метод `readExact()` использует точный тип. Итак, если используется точный тип без масок, вы сможете читать и записывать этот точный тип в `List`. Кроме того, для возвращаемого значения статический обобщенный метод `readExact()` фактически «адаптируется» к каждому вызову и возвращает `Apple` из `List<Apple>` и `Fruit` из `List<Fruit>`, как показано в `f1()`. Таким образом, при использовании статического обобщенного метода ковариантность нужна не всегда.

Но если вы работаете с обобщенным классом, параметр задается для класса при создании экземпляра этого класса. Как показано в `f2()`, экземпляр `fruitReader` может прочитать `Fruit` из `List<Fruit>`, поскольку это его точный тип. Но контейнер `List<Apple>` также должен производить объекты `Fruit`, а `fruitReader` этого не позволяет.

Для решения проблемы метод `CovariantReader.readCovariant()` получает `List<? extends T>`, поэтому читать `T` из этого списка безопасно (все объекты в списке как минимум имеют тип `T`, а могут относиться к типу, производному от `T`). В методе `f3()` мы видим, что теперь тип `Fruit` можно прочитать из `List<Apple>`.

28. (4) Создайте обобщенный класс `Generic1<T>` с одним методом, получающим аргумент типа `T`. Создайте второй обобщенный класс `Generic2<T>` с одним методом, возвращающим аргумент типа `T`. Напишите обобщенный метод с контравариантным аргументом первого обобщенного класса, который вызывает его метод. Протестируйте на типах библиотеки `typeinfo.pets`.

Неограниченные маски

Неограниченная маска `<?>` означает «что угодно», так что может показаться, что использование неограниченной маски эквивалентно использованию самого типа. Действительно, компилятор на первый взгляд соглашается с этим предположением:

```
//: generics/UnboundedWildcards1.java
import java.util.*;

public class UnboundedWildcards1 {
    static List list1;
    static List<?> list2;
    static List<? extends Object> list3;
    static void assign1(List list) {
        list1 = list;
        list2 = list;
        // list3 = list; // Предупреждение: неконтролируемое преобразование
        // Обнаружен: List, требуется: List<? extends Object>
    }
    static void assign2(List<?> list) {
        list1 = list;
        list2 = list;
        list3 = list;
    }
    static void assign3(List<? extends Object> list) {
        list1 = list;
        list2 = list;
        list3 = list;
    }
    public static void main(String[] args) {
        assign1(new ArrayList());
        assign2(new ArrayList());
        // assign3(new ArrayList()); // Предупреждение:
        // Неконтролируемое преобразование. Обнаружен: ArrayList
        // Требуется: List<? extends Object>
        assign1(new ArrayList<String>());
        assign2(new ArrayList<String>());
        assign3(new ArrayList<String>());
        // Обе формы допустимы как List<?>:
        List<?> wildList = new ArrayList();
        wildList = new ArrayList<String>();
        assign1(wildList);
        assign2(wildList);
        assign3(wildList);
    }
} ///:~
```

Во многих ситуациях компилятор действительно совершенно не интересуется, используете ли вы сам тип или `<?>`. В таких случаях `<?>` можно рассматривать как обычное украшение. Тем не менее маска все же полезна, потому что она фактически говорит: «Этот код написан с учетом обобщений Java, и здесь имеется в виду, что в обобщенном параметре может передаваться любой тип».

Второй пример демонстрирует одно важное применение неограниченных масок. С несколькими обобщенными параметрами иногда бывает важно разрешить для одного параметра любые типы и установить для другого параметра конкретный тип:

```

//: generics/UnboundedWildcards2.java
import java.util.*;

public class UnboundedWildcards2 {
    static Map map1;
    static Map<?,?> map2;
    static Map<String,?> map3;
    static void assign1(Map map) { map1 = map; }
    static void assign2(Map<?,?> map) { map2 = map; }
    static void assign3(Map<String,?> map) { map3 = map; }
    public static void main(String[] args) {
        assign1(new HashMap());
        assign2(new HashMap());
        // assign3(new HashMap()); // Предупреждение:
        // Неконтролируемое преобразование. Обнаружен: HashMap
        // Требуется: Map<String,?>
        assign1(new HashMap<String,Integer>());
        assign2(new HashMap<String,Integer>());
        assign3(new HashMap<String,Integer>());
    }
} ///:~

```

Если обобщение состоит только из неограниченных масок, как `Map<?,?>`, компилятор, похоже, не отличает его от «обычного» типа `Map`. Кроме того, `UnboundedWildcards1.java` показывает, что компилятор по-разному рассматривает `List<?>` и `List<? extends Object>`.

При этом компилятор не всегда обращает внимание на различия между, допустим, `List` и `List<?>`; создается впечатление, что это одно и то же, а это может породить путаницу. В самом деле, поскольку обобщенный аргумент стирается до первого ограничения, `List<?>` кажется эквивалентом `List<Object>`, а `List` по сути тоже является `List<Object>` — впрочем, оба эти утверждения не совсем точны. `List` в действительности означает «неспециализированный контейнер `List`, в котором могут храниться любые объекты `Object`», тогда как `List<?>` означает «специализация `List` для конкретного типа, который нам неизвестен».

Когда же компилятор различает несспециализированные (raw) типы и типы с неограниченными масками? В следующем примере используется определенный ранее класс `Holder<T>`. Он содержит методы, которые получают аргумент `Holder`, но в разных формах: несспециализированный тип, с конкретным параметром-типом и с неограниченной маской:

```

//: generics/Wildcards.java
// Изучение смысла масок.

public class Wildcards {
    // Неспециализированный аргумент:
    static void rawArgs(Holder holder, Object arg) {
        // holder.set(arg); // Предупреждение:
        // Неконтролируемый вызов set(T) как члена
        // несспециализированного типа Holder
        // holder.set(new Wildcards()); // То же предупреждение

        // Невозможно - 'T' отсутствует:
        // T t = holder.get();

        // Допустимо, но информация типа теряется:

```

```
    Object obj = holder.get();
}
// Аналогично rawArgs(), но с выдачей ошибок вместо предупреждений:
static void unboundedArg(Holder<?> holder, Object arg) {
    // holder.set(arg); // Ошибка:
    //   set(capture of ?) в Holder<capture of ?>
    //   не может применяться к (Object)
    // holder.set(new Wildcards()); // Та же ошибка

    // Невозможно - 'T' отсутствует:
    // T t = holder.get();

    // Допустимо, но информация типа теряется:
    Object obj = holder.get();
}
static <T> T exact1(Holder<T> holder) {
    T t = holder.get();
    return t;
}
static <T> T exact2(Holder<T> holder, T arg) {
    holder.set(arg);
    T t = holder.get();
    return t;
}
static <T>
T wildSubtype(Holder<? extends T> holder, T arg) {
    // holder.set(arg); // Ошибка:
    //   set(capture of ? extends T) в
    //   Holder<capture of ? extends T>
    //   не может применяться к (T)
    T t = holder.get();
    return t;
}
static <T>
void wildSupertype(Holder<? super T> holder, T arg) {
    holder.set(arg);
    // T t = holder.get(); // Ошибка:
    //   Несовместимые типы : обнаружен Object, требуется T

    // Допустимо, но информация типа теряется:
    Object obj = holder.get();
}
public static void main(String[] args) {
    Holder raw = new Holder<Long>();
    // Или:
    raw = new Holder();
    Holder<Long> qualified = new Holder<Long>();
    Holder<?> unbounded = new Holder<Long>();
    Holder<? extends Long> bounded = new Holder<Long>();
    Long lng = 1L;

    rawArgs(raw, lng);
    rawArgs(qualified, lng);
    rawArgs(unbounded, lng);
    rawArgs(bounded, lng);

    unboundedArg(raw, lng);
    unboundedArg(qualified, lng);
}
```

```

unboundedArg(unbounded, lng);
unboundedArg(bounded, lng);

// Object r1 = exact1(raw); // Предупреждения:
// Неконтролируемое преобразование Holder в Holder<T>
// Неконтролируемый вызов метода: exact1(Holder<T>)
// применяется к (Holder)
Long r2 = exact1(qualified);
Object r3 = exact1(unbounded); // Должен вернуть Object
Long r4 = exact1(bounded);

// Long r5 = exact2(raw, lng); // Предупреждения:
// Неконтролируемое преобразование Holder в Holder<Long>
// Неконтролируемый вызов метода: exact2(Holder<T>,T)
// применяется к (Holder,Long)
Long r6 = exact2(qualified, lng);
// Long r7 = exact2(unbounded, lng); // Ошибка:
// exact2(Holder<T>,T) не может применяться
// к (Holder<capture of ?>,Long)
// Long r8 = exact2(bounded, lng); // Ошибка:
// exact2(Holder<T>,T) не может применяться
// к (Holder<capture of ? extends Long>,Long)

// Long r9 = wildSubtype(raw, lng); // Предупреждения:
// Неконтролируемое преобразование Holder
// в Holder<? extends Long>
// Неконтролируемый вызов метода:
// wildSubtype(Holder<? extends T>,T)
// применяется к (Holder,Long)
Long r10 = wildSubtype(qualified, lng);
// Допустимо, но может возвращать только Object:
Object r11 = wildSubtype(unbounded, lng);
Long r12 = wildSubtype(bounded, lng);

// wildSupertype(raw, lng); // Предупреждения:
// Неконтролируемое преобразование Holder
// в Holder<? super Long>
// Неконтролируемый вызов метода:
// wildSupertype(Holder<? super T>,T)
// применяется к (Holder,Long)
wildSupertype(qualified, lng);
// wildSupertype(unbounded, lng); // Ошибка:
// wildSupertype(Holder<? super T>,T) не может
// применяться к (Holder<capture of ?>,Long)
// wildSupertype(bounded, lng); // Ошибка:
// wildSupertype(Holder<? super T>,T) не может
// применяться к (Holder<capture of ? extends Long>,Long)
}
} ///:~

```

В `rawArgs()` компилятор знает, что `Holder` является обобщенным типом, поэтому даже несмотря на то, что здесь он выражается неспециализированным типом, компилятор знает, что передача `Object` методу `set()` небезопасна. Так как это неспециализированный тип, методу `set()` можно передать объект любого типа, и этот объект будет приведен к `Object` восходящим преобразованием. Итак, каждый раз, когда вы используете неспециализированный тип, вы отказываетесь от проверки на стадии компиляции. Вызов `get()` демонстрирует ту же проблему: при отсутствии `T` результатом может быть только `Object`.

Легко прийти к мнению, что неспециализированные типы `Holder` и `Holder<?>` обозначают приблизительно одно и то же. Но метод `unboundedArg()` подчеркивает различия между ними — он выявляет те же проблемы, но сообщает о них как об ошибках, а не как о предупреждениях, потому что неспециализированный тип `Holder` может содержать комбинацию любых типов, а `Holder<?>` содержит однородную коллекцию *некоторого конкретного типа*, а следовательно, передать `Object` просто не получится.

В `exact1()` и `exact2()` используются точные параметры обобщений — без масок. Как видно по дополнительному аргументу, `exact2()` имеет другие ограничения, чем `exact1()`.

В `wildSubtype()` ограничения для типа `Holder` ослабляются до `Holder` для любых типов, расширяющих `T`. И снова это означает, что типом `T` может быть `Fruit`, тогда как `holder` может содержать `Holder<Apple>`. Чтобы предотвратить помещение `Orange` в `Holder<Apple>`, вызов `set()` (или любого другого метода, получающего параметр-тип) запрещен. Но при этом вы знаете, что все, что выходит из `Holder<? extends Fruit>`, будет по меньшей мере `Fruit`, поэтому вызов `get()` (или любого метода, возвращающего параметр-тип) разрешен.

Применение маски супертипа продемонстрировано в методе `wildSupertype()`, поведение которого обратно поведению `wildSubtype()`: `holder` может содержать контейнер с элементами любого типа, который является разновидностью базового класса `T`. Таким образом, `set()` может принять `T`, так как все, что работает с базовым типом, будет полиморфно работать с производным типом (отсюда `T`). Однако попытка вызова `get()` пользы не принесет, потому что тип, хранимый в `holder`, может быть абсолютно любым супертипом, поэтому единственным безопасным вариантом является `Object`. Этот пример также показывает ограниченные возможности выполнения операций с неограниченным параметром в `unbounded()`: вы не можете вызвать `get()` или `set()` для `T`, потому что у вас нет `T`.

В методе `main()` мы видим, какие из этих методов могут получать те или иные типы аргументов без ошибок и предупреждений. Для обеспечения миграционной совместимости `rawArgs()` будет получать все разновидности `Holder` без выдачи предупреждений. Метод `unboundedArg()` также принимает все типы, хотя, как упоминалось ранее, в теле метода он работает с ними иначе.

Если передать неспециализированную ссылку `Holder` методу, который получает «точный» обобщенный тип (без маски), вы получите предупреждение, потому что точный аргумент ожидает получить информацию, не существующую в неспециализированном типе. И если передать неограниченную ссылку `exact1()`, то у компилятора не будет информации о типе для установления возвращаемого типа.

Мы видим, что метод `exact2()` имеет больше всего ограничений, поскольку он должен получать `Holder<T>` и аргумент типа `T`; не получив точных аргументов, он выдает ошибки и предупреждения. Иногда такой подход приемлем, но если он создает неудобства, вы можете использовать маски в зависимости от того, хотите ли вы получать возвращаемые значения из своего обобщенного аргумента (как в `wildSubtype()`) или передавать в нем типизованные аргументы (как в `wildSupertype()`).

Таким образом, преимущество использования точных типов вместо масок заключается в том, что оно позволяет больше сделать с обобщенными параметрами. С другой

стороны, использование масок позволяет принять в аргументах более широкий диапазон параметризованных типов. Вы должны решить, какой из вариантов лучше подходит для ваших потребностей, в каждом конкретном случае.

Фиксация

Есть ситуация, в которой использование `<?>` вместо неспециализированного типа особенно принципиально. Если передать неспециализированный тип методу, использующему `<?>`, компилятор может автоматически вычислить фактический параметр-тип и вызвать другой метод, использующий точный тип. Следующий пример демонстрирует работу этого механизма, который называется *фиксацией* (capture conversion), потому что неуказанный тип маски фиксируется и преобразуется к точному типу. В этом случае комментарии о предупреждениях действуют только при удалении аннотации `@SuppressWarnings`:

//: generics/CaptureConversion.java

```
public class CaptureConversion {
    static <T> void f1(Holder<T> holder) {
        T t = holder.get();
        System.out.println(t.getClass().getSimpleName());
    }
    static void f2(Holder<?> holder) {
        f1(holder); // Вызов с зафиксированным типом
    }
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        Holder raw = new Holder<Integer>(1);
        // f1(raw); // Выдает предупреждения
        f2(raw);    // Без предупреждений
        Holder rawBasic = new Holder();
        rawBasic.set(new Object()); // Предупреждение
        f2(rawBasic); // Без предупреждений
        // Восходящее преобразование к Holder<?>,
        // тип все равно определяется правильно:
        Holder<?> wildcarded = new Holder<Double>(1.0);
        f2(wildcarded);
    }
} /* Output:
Integer
Object
Double
*///:~
```

Все параметры-типы `f1()` являются точными, не содержат масок или ограничений. В `f2()` параметр `holder` является неограниченной маской, так что на первый взгляд он фактически неизвестен. Однако в `f2()` вызывается метод `f1()`, которому требуется известный параметр. Здесь параметр-тип фиксируется в процессе вызова `f2()` и может использоваться при вызове `f1()`.

Нельзя ли использовать эту методику для записи? Для этого вместе с `Holder<?>` необходимо передавать конкретный тип. Фиксация работает только в ситуациях, при которых внутри метода необходимо работать с точным типом. Учтите, что вы не можете вернуть

т из `f2()`, потому что тип `T` неизвестен для `f2()`. Фиксация — интересный механизм, но его возможности невелики.

- 29.** (5) Создайте обобщенный метод, который получает аргумент `Holder<List<?>>`. Определите, какие методы могут или не могут вызываться для `Holder` и для `List`. Повторите для аргумента `List<Holder<?>>`.

Проблемы

В этом разделе рассматриваются разнообразные проблемы, встречающиеся при использовании обобщений Java.

Примитивы не могут использоваться как параметры-типы

Как упоминалось ранее в этой главе, один из недостатков обобщений Java заключается в том, что примитивы не могут использоваться в качестве параметров-типов. Таким образом, например, вы не сможете создать `ArrayList<int>`.

Проблема решается использованием примитивных классов-«оберток» в сочетании с автоматической упаковкой Java SE. Если вы создадите `ArrayList<Integer>` и используете примитивные значения `int` с этим контейнером, вы обнаружите, что автоматическая упаковка выполняет преобразование к `Integer` и обратно автоматически — все выглядит так, словно вы работаете с `ArrayList<int>`:

```
//: generics/ListOfInt.java
// Автоматическая упаковка компенсирует невозможность
// использования примитивов с обобщениями.
import java.util.*;

public class ListOfInt {
    public static void main(String[] args) {
        List<Integer> li = new ArrayList<Integer>();
        for(int i = 0; i < 5; i++)
            li.add(i);
        for(int i : li)
            System.out.print(i + " ");
    }
} /* Output:
0 1 2 3 4
*///:~
```

Автоматическая упаковка даже позволяет использовать синтаксис `foreach` для производства `int`. В общем случае такое решение работает нормально — ваш код может успешно сохранять и читать `int`. При этом выполняются некоторые преобразования типа, но они остаются скрытыми от вас. Впрочем, если лишние преобразования создают проблемы с производительностью, вы можете использовать специализированную версию контейнеров, приспособленную для примитивных типов; одна из реализаций с открытым кодом доступна по адресу org.apache.commons.collections.primitives.

Ниже представлено другое решение, в котором создается множество (`Set`) элементов `Byte`:

```

//: generics/ByteSet.java
import java.util.*;

public class ByteSet {
    Byte[] possibles = { 1,2,3,4,5,6,7,8,9 };
    Set<Byte> mySet =
        new HashSet<Byte>(Arrays.asList(possibles));
    // Но такая запись невозможна:
    // Set<Byte> mySet2 = new HashSet<Byte>(
    //     Arrays.<Byte>asList(1,2,3,4,5,6,7,8,9));
} ///:~

```

Автоматическая упаковка решает некоторые проблемы, но не все. В следующем примере представлен обобщенный интерфейс `Generator` с методом `next()`, возвращающим объект параметра-типа. Класс `FArray` содержит обобщенный метод, использующий генератор для заполнения массива объектами (переход к обобщенному классу в данном случае не работает, потому что метод является статическим). Реализации `Generator` взяты из главы 16, а в методе `main()` продемонстрировано использование `FArray.fill()` для заполнения массива объектами:

```

//: generics/PrimitiveGenericTest.java
import net.mindview.util.*;

// Заполнение массива с использованием генератора:
class FArray {
    public static <T> T[] fill(T[] a, Generator<T> gen) {
        for(int i = 0; i < a.length; i++)
            a[i] = gen.next();
        return a;
    }
}

public class PrimitiveGenericTest {
    public static void main(String[] args) {
        String[] strings = FArray.fill(
            new String[7], new RandomGenerator.String(10));
        for(String s : strings)
            System.out.println(s);
        Integer[] integers = FArray.fill(
            new Integer[7], new RandomGenerator.Integer());
        for(int i: integers)
            System.out.println(i);
        // Автоматическая упаковка не спасет -
        // команда не компилируется:
        // int[] b =
        //     FArray.fill(new int[7], new RandIntGenerator());
    }
} /* Output:
YNzbrnyGcF
OWZnTcQrGs
eGZMmJMRoE
suEcUOneOE
dLsmwHLGEa
hKcxrEqUCB
bkInaMesbt
7052
6665

```

```
2654
3909
5202
2209
5458
*///:~
```

Так как `RandomGenerator.Integer` реализует `Generator<Integer>`, я надеялся, что автоматическая упаковка преобразует значение `next()` из `Integer` в `int`. Однако автоматическая упаковка не применяется к массивам, поэтому решение не работает.

- 30.** (2) Создайте объект `Holder` для каждой «обертки» примитивного типа. Продемонстрируйте, что автоматическая упаковка и распаковка работает для методов `set()` и `get()` каждого экземпляра.

Реализация параметризованных интерфейсов

Класс не может реализовать две разновидности одного обобщенного интерфейса. Из-за стирания они превратятся в один интерфейс. Пример ситуации, в которой происходит такой конфликт:

```
//: generics/MultipleInterfaceVariants.java
// {CompileTimeError} (не компилируется)
```

```
interface Payable<T> {}
```

```
class Employee implements Payable<Employee> {}
class Hourly extends Employee
    implements Payable<Hourly> {} ///:~
```

Класс `Hourly` не компилируется, потому что стирание сокращает `Payable<Employee>` и `Payable<Hourly>` до одного класса `Payable`, а приведенный выше код будет означать, что один интерфейс реализуется дважды. Интересно, что при удалении обобщенных параметров из обоих мест использования `Payable` (как это делает компилятор при стирании) код откомпилируется.

Эта проблема проявляется при работе с некоторыми фундаментальными интерфейсами Java (такими, как `Comparable<T>`), как будет показано позже в этом разделе.

- 31.** (1) Удалите все обобщения из `MultipleInterfaceVariants.java` и измените код, чтобы пример компилировался.

Приведения типа и предупреждения

Использование приведения типов и `instanceof` с обобщенным параметром-типом ни к чему не приводит. Следующий контейнер хранит значения во внутреннем представлении как `Object` и преобразует их к типу `T` при чтении:

```
//: generics/GenericCast.java
```

```
class FixedSizeStack<T> {
    private int index = 0;
    private Object[] storage;
```

продолжение ➤

```

public FixedSizeStack(int size) {
    storage = new Object[size];
}
public void push(T item) { storage[index++] = item; }
@SuppressWarnings("unchecked")
public T pop() { return (T)storage[--index]; }
}

public class GenericCast {
    public static final int SIZE = 10;
    public static void main(String[] args) {
        FixedSizeStack<String> strings =
            new FixedSizeStack<String>(SIZE);
        for(String s : "A B C D E F G H I J".split(" "))
            strings.push(s);
        for(int i = 0; i < SIZE; i++) {
            String s = strings.pop();
            System.out.print(s + " ");
        }
    }
} /* Output:
J I H G F E D C B A
*///:~

```

Без аннотации `@SuppressWarnings` компилятор выдает для `pop()` предупреждение «неконтролируемое приведение типа». Из-за стирания он не знает, безопасно ли приведение типа, и метод `pop()` не выполняет никакого реального приведения. `T` стирается по первому ограничению, которым по умолчанию является `Object`, так что `pop()` фактически просто преобразует `Object` в `Object`.

В некоторых ситуациях обобщения не отменяют необходимости в приведении типа, и компилятор выдает нежелательное предупреждение. Пример:

```

//: generics/NeedCasting.java
import java.io.*;
import java.util.*;

public class NeedCasting {
    @SuppressWarnings("unchecked")
    public void f(String[] args) throws Exception {
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream(args[0]));
        List<Widget> shapes = (List<Widget>)in.readObject();
    }
} ///:~

```

Как вы узнаете в следующей главе, метод `readObject()` не может знать, что он читает, поэтому возвращает объект, к которому должно быть применено преобразование типа. Но если закомментировать аннотацию `@SuppressWarnings` и откомпилировать программу, вы получите предупреждение:

Внимание: В `NeedCasting.java` используются неконтролируемые или небезопасные операции.
Внимание: Перекомпилируйте с ключом `-Xlint:unchecked` для получения дополнительной информации.

А если последовать инструкциям и перекомпилировать программу с ключом `-Xlint:unchecked`, результат будет выглядеть так:

```
NeedCasting.java:12: предупреждение: [неконтролируемое] неконтролируемое приведение типа
обнаружено : java.lang.Object
требуется: java.util.List<Widget>
    List<Shape> shapes = (List<Widget>)in.readObject();
```

Вы обязаны выполнить приведение типа, но при этом вам подсказывают, что делать этого не следует. Для решения следует применить новую форму приведения типа, появившуюся в Java SE5, — приведение через обобщенный класс:

```
//: generics/ClassCasting.java
import java.io.*;
import java.util.*;

public class ClassCasting {
    @SuppressWarnings("unchecked")
    public void f(String[] args) throws Exception {
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream(args[0]));
        // Не компилируется:
        // List<Widget> lw1 =
        // List<Widget>.class.cast(in.readObject());
        List<Widget> lw2 = List.class.cast(in.readObject());
    }
} ///:~
```

Однако приведение к фактическому типу (`List<Widget>`) невозможно. Иначе говоря, вы не можете использовать вызов

```
List<Widget>.class.cast(in.readObject())
```

и даже при добавлении дополнительного приведения типа:

```
(List<Widget>)List.class.cast(in.readObject())
```

все равно будет выдано предупреждение.

- 32.** (1) Убедитесь в том, что `FixedSizeStack` в `GenericCast.java` генерирует предупреждения при попытке выхода за границы. Означает ли это, что код проверки границ не обязателен?
- 33.** (3) Исправьте ошибку в `GenericCast.java` с использованием `ArrayList`.

Перегрузка

Следующая программа не откомпилируется, хотя на первый взгляд выглядит логично:

```
//: generics/UseList.java
// {CompileTimeError} (не компилируется)
import java.util.*;

public class UseList<W,T> {
    void f(List<T> v) {}
    void f(List<W> v) {}
} ///:~
```

Перегрузка метода порождает идентичную сигнатуру типа из-за стирания.

Вместо этого необходимо явно указать имена методов, если стерты аргументы не позволяют получить уникальный список аргументов:

```
//: generics/UseList2.java
import java.util.*;

public class UseList2<W,T> {
    void f1(List<T> v) {}
    void f2(List<W> v) {}
} ///:~
```

К счастью, подобные проблемы обнаруживаются компилятором.

Перехват интерфейса базовым классом

Предположим, имеется класс `Pet`, реализующий `Comparable` для сравнения с другими объектами `Pet`:

```
//: generics/ComparablePet.java

public class ComparablePet
implements Comparable<ComparablePet> {
    public int compareTo(ComparablePet arg) { return 0; }
} ///:~
```

Есть смысл сузить тип, с которым может сравниваться subclass `ComparablePet`. Например, `Cat` может сравниваться только с другими объектами `Cat`:

```
//: generics/HijackedInterface.java
// {CompileTimeError} (не компилируется)

class Cat extends ComparablePet implements Comparable<Cat>{
    // Ошибка: Comparable не может наследоваться
    // с разными аргументами: <Cat> and <Pet>
    public int compareTo(Cat arg) { return 0; }
} ///:~
```

К сожалению, это решение не работает. Когда для `Comparable` устанавливается аргумент `ComparablePet`, ни один реализующий класс не может сравниваться ни с чем, кроме `ComparablePet`:

```
//: generics/RestrictedComparablePets.java

class Hamster extends ComparablePet
implements Comparable<ComparablePet> {
    public int compareTo(ComparablePet arg) { return 0; }
}

// Или просто:

class Gecko extends ComparablePet {
    public int compareTo(ComparablePet arg) { return 0; }
} ///:~
```

Класс `Hamster` показывает, что возможно повторно реализовать интерфейс, присутствующий в `ComparablePet`, — при условии его точного совпадения, включая параметры-типы. Однако это ничем не отличается от переопределения методов базового класса, как видно на примере `Gecko`.

Самоограничиваемые типы

Существует одна довольно заумная идиома, которая периодически встречается в обобщениях Java. Вот как она выглядит:

```
class SelfBounded<T extends SelfBounded<T>> { // ...
```

Напоминает два зеркала, обращенных друг к другу; своего рода бесконечное отражение. Класс `SelfBounded` получает обобщенный аргумент `T`; `T` ограничивается по `SelfBounded`, получающего аргумент `T`.

Когда вы впервые сталкиваетесь с этой идиомой, понять ее довольно трудно. Это лишний раз указывает, что ключевое слово `extends` в ограничениях играет совершенно иную роль, чем при создании субклассов.

Необычные рекурсивные обобщения

Чтобы понять, что обозначает самоограничиваемый тип, для начала рассмотрим упрощенную версию идиомы, в которой нет самоограничения.

Прямое наследование от обобщенного параметра невозможно. Тем не менее наследование можно применить к классу, который использует обобщенный параметр в своем определении. Другими словами, следующий фрагмент возможен:

```
//: generics/CuriouslyRecurringGeneric.java
```

```
class GenericType<T> {}

public class CuriouslyRecurringGeneric
    extends GenericType<CuriouslyRecurringGeneric> {} ///:~
```

Происходящее можно назвать «необычным рекурсивным обобщением» (*curiously recurring generics*, CRG) по названию паттерна «Необычный рекурсивный шаблон» для C++, описанного Джимом Коплином. Под «необычной рекурсией» имеется в виду то, что класс несколько необычно появляется в своем базовом классе.

Чтобы понять, что это значит, попробуйте произнести вслух: «Я создаю новый класс, производный от обобщенного типа, который получает имя моего класса как параметр». Что может сделать обобщенный базовый тип при получении имени производного класса? Основной смысл обобщений в Java проявляется в передаче аргументов и возвращаемых типов, так что он может создать базовый класс, использующий производный тип в типах своих аргументов и возвращаемого значения. Также производный тип может использоваться для типов полей, несмотря на то что они будут стерты до `Object`. Следующий обобщенный класс выражает эту мысль:

```
//: generics/BasicHolder.java
```

```
public class BasicHolder<T> {
    T element;
    void set(T arg) { element = arg; }
    T get() { return element; }
    void f() {
        System.out.println(element.getClass().getSimpleName());
    }
} ///:~
```

Это вполне рядовой обобщенный тип с методами, которые получают и возвращают объекты параметра-типа, а также методом, который работает с полем (хотя и выполняет с ним только операции `Object`).

`BasicHolder` можно использовать в необычном рекурсивном обобщении:

```
//: generics/CRGWithBasicHolder.java
```

```
class Subtype extends BasicHolder<Subtype> {}

public class CRGWithBasicHolder {
    public static void main(String[] args) {
        Subtype st1 = new Subtype(), st2 = new Subtype();
        st1.set(st2);
        Subtype st3 = st1.get();
        st1.f();
    }
} /* Output:
Subtype
*///:~
```

Обратите внимание на важную подробность: новый класс `Subtype` получает аргументы и возвращает значения типа `Subtype`, а не базового класса `BasicHolder`. В этом проявляется сущность «необычных рекурсивных обобщений»: базовый класс заменяет свои параметры производным классом. Это означает, что обобщенный базовый класс становится своего рода шаблоном общей функциональности для всех производных классов, но эта функциональность будет использовать производный тип для всех своих аргументов и возвращаемых значений. Иначе говоря, в полученном классе будет использоваться точный тип вместо базового типа. Таким образом, в `Subtype` и аргументы `set()`, и возвращаемое значение `get()` относятся к точному типу `Subtype`.

Самоограничение

`BasicHolder` может использовать в своем обобщенном параметре любой тип, как показывает следующий пример:

```
//: generics/Unconstrained.java
```

```
class Other {}
class BasicOther extends BasicHolder<Other> {}

public class Unconstrained {
    public static void main(String[] args) {
        BasicOther b = new BasicOther(), b2 = new BasicOther();
        b.set(new Other());
        Other other = b.get();
        b.f();
    }
} /* Output:
Other
*///:~
```

Самоограничение делает следующий шаг: оно *заставляет* обобщение использовать самого себя в качестве аргумента ограничения. Посмотрим, как может (и не может) использоваться полученный класс:

```

//: generics/SelfBounding.java

class SelfBounded<T extends SelfBounded<T>> {
    T element;
    SelfBounded<T> set(T arg) {
        element = arg;
        return this;
    }
    T get() { return element; }
}

class A extends SelfBounded<A> {}
class B extends SelfBounded<A> {} // Также допустимо

class C extends SelfBounded<C> {
    C setAndGet(C arg) { set(arg); return get(); }
}

class D {}
// Невозможно:
// class E extends SelfBounded<D> {}
// Ошибка компиляции: параметр-тип D не соответствует ограничению

// Увы, это возможно, так что принудить к выполнению
// идиомы не удастся:
class F extends SelfBounded {}
public class SelfBounding {
    public static void main(String[] args) {
        A a = new A();
        a.set(new A());
        a = a.set(new A()).get();
        a = a.get();
        C c = new C();
        c = c.setAndGet(new C());
    }
} ///:~

```

По сути самоограничение требует использования класса в отношениях наследования следующего вида:

```
class A extends SelfBounded<A> {}
```

Это заставляет вас передать определяемый класс в параметре базового класса.

Какую пользу приносит самоограничение? Параметр-тип должен совпадать с определяемым классом. Как видно из определения класса `B`, также возможно наследование от `SelfBounded`, в параметре которого снова используется `SelfBounded`, хотя на практике обычно преобладает использование, продемонстрированное для класса `A`. Попытка определить `E` показывает, что вы не можете использовать параметр-тип, который не является `SelfBounded`.

К сожалению, код `F` компилируется без предупреждений, так что идиома самоограничения не является обязательной к исполнению. Если для вас это так важно, возможно, вам придется использовать внешние инструменты, которые будут убеждать в том, что неспециализированные типы не используются вместо параметризованных типов. Обратите внимание: ограничение можно снять, и все по-прежнему будут компилироваться, но тогда также откомпилируется и класс `E`:

```
//: generics/NotSelfBounded.java
```

```
public class NotSelfBounded<T> {
    T element;
    NotSelfBounded<T> set(T arg) {
        element = arg;
        return this;
    }
    T get() { return element; }
}

class A2 extends NotSelfBounded<A2> {}
class B2 extends NotSelfBounded<A2> {}

class C2 extends NotSelfBounded<C2> {
    C2 setAndGet(C2 arg) { set(arg); return get(); }
}

class D2 {}
// Теперь возможно:
class E2 extends NotSelfBounded<D2> {} ///:~
```

Итак, самоограничение служит только для принудительного соблюдения отношений наследования. Используя самоограничение, вы знаете, что параметр-тип, используемый классом, будет относиться к тому же базовому типу, что и класс, использующий этот параметр. Оно заставляет всех пользователей этого класса следовать указанному образцу.

Самоограничение также можно использовать для обобщенных методов:

```
//: generics/SelfBoundingMethods.java
```

```
public class SelfBoundingMethods {
    static <T extends SelfBounded<T>> T f(T arg) {
        return arg.set(arg).get();
    }
    public static void main(String[] args) {
        A a = f(new A());
    }
} ///:~
```

Тем самым вы запрещаете применять метод к чему-либо, кроме самоограниченного аргумента представленной формы.

Ковариантность аргументов

Полезность самоограничиваемых аргументов заключается в том, что они производят *ковариантные типы аргументов* — типы аргументов методов меняются в соответствии с субклассами.

И хотя самоограничиваемые типы также производят возвращаемые типы, соответствующие типу субкласса, это не так важно, потому что ковариантные возвращаемые типы были представлены в Java SE5:

```
//: generics/CovariantReturnTypes.java
```

```
class Base {}
class Derived extends Base {}
```

```

interface OrdinaryGetter {
    Base get();
}

interface DerivedGetter extends OrdinaryGetter {
    // Возвращаемый тип переопределенного метода может изменяться:
    Derived get();
}

public class CovariantReturnTypes {
    void test(DerivedGetter d) {
        Derived d2 = d.get();
    }
} ///:~

```

Метод `get()` в `DerivedGetter` переопределяет `get()` из `OrdinaryGetter` и возвращает тип, производный от типа, возвращаемого `OrdinaryGetter.get()`. И хотя это абсолютно логично — метод производного типа должен быть способен вернуть более конкретный тип, чем переопределяемый метод базового типа, — в предыдущих версиях Java это было недопустимо.

Самоограничиваемое обобщение действительно производит для возвращаемого значения точный производный тип, как видно из `get()`:

```

//: generics/GenericsAndReturnTypes.java

interface GenericGetter<T> extends GenericGetter<T>> {
    T get();
}

interface Getter extends GenericGetter<Getter> {}

public class GenericsAndReturnTypes {
    void test(Getter g) {
        Getter result = g.get();
        GenericGetter gg = g.get(); // Также базовый тип
    }
} ///:~

```

Учтите, что этот код не компилировался бы без включения в Java SE5 ковариантных возвращаемых типов.

Впрочем, в необобщенном коде типы аргументов нельзя заставить изменяться с под-типами:

```

//: generics/OrdinaryArguments.java

class OrdinarySetter {
    void set(Base base) {
        System.out.println("OrdinarySetter.set(Base)");
    }
}

class DerivedSetter extends OrdinarySetter {
    void set(Derived derived) {
        System.out.println("DerivedSetter.set(Derived)");
    }
}

```

продолжение ➤

```

public class OrdinaryArguments {
    public static void main(String[] args) {
        Base base = new Base();
        Derived derived = new Derived();
        DerivedSetter ds = new DerivedSetter();
        ds.set(derived);
        ds.set(base); // Компилируется: перегрузка, не переопределение!
    }
} /* Output:
DerivedSetter.set(Derived)
OrdinarySetter.set(Base)
*///:~

```

Оба вызова — `set(derived)` и `set(base)` — допустимы, так что `DerivedSetter.set()` не переопределяет `OrdinarySetter.set()`, а *перегружает* этот метод. Из выходных данных видно, что в `DerivedSetter` содержатся два метода, так что версия базового класса остается доступной; это подтверждает факт перегрузки.

Однако с самоограничиваемыми типами в производном классе содержится только один метод, в аргументе которого передается производный, а не базовый тип:

```

//: generics/SelfBoundingAndCovariantArguments.java

interface SelfBoundSetter<T extends SelfBoundSetter<T>> {
    void set(T arg);
}

interface Setter extends SelfBoundSetter<Setter> {}

public class SelfBoundingAndCovariantArguments {
    void testA(Setter s1, Setter s2, SelfBoundSetter sbs) {
        s1.set(s2);
        // s1.set(sbs); // Ошибка:
        // set(Setter) в SelfBoundSetter<Setter>
        // не может применяться к (SelfBoundSetter)
    }
} ///:~

```

Компилятор не распознает попытку передачи в аргументе `set()` базового типа, потому что метода с такой сигнатурой не существует. Фактически аргумент был переопределен.

Без самоограничения вступает в силу обычный механизм наследования, и происходит перегрузка, как и в случае с необобщенным случаем:

```

//: generics/PlainGenericInheritance.java

class GenericSetter<T> { // Без самоограничения
    void set(T arg){
        System.out.println("GenericSetter.set(Base)");
    }
}

class DerivedGS extends GenericSetter<Base> {
    void set(Derived derived){
        System.out.println("DerivedGS.set(Derived)");
    }
}

```

```

public class PlainGenericInheritance {
    public static void main(String[] args) {
        Base base = new Base();
        Derived derived = new Derived();
        DerivedGS dgs = new DerivedGS();
        dgs.set(derived);
        dgs.set(base); // Компилируется: перегрузка, не переопределение!
    }
} /* Output:
DerivedGS.set(Derived)
GenericSetter.set(Base)
*///:~

```

Этот код имитирует `OrdinaryArguments.java`; в этом примере `DerivedSetter` наследует от класса `OrdinarySetter`, содержащего метод `set(Base)`. Здесь класс `DerivedGS` наследует от `GenericSetter<Base>`, который также содержит метод `set(Base)`, созданный обобщением. Как и в случае `OrdinaryArguments.java`, из результатов видно, что `DerivedGS` содержит две перегруженные версии `set()`. Без самоограничения перегрузка осуществляется по типам аргументов. Если вы используете самоограничение, остается только одна версия метода, получающая точный тип аргумента.

- 34.** (4) Создайте самоограничиваемый обобщенный тип с абстрактным методом, который получает обобщенный параметр-тип и возвращает обобщенный параметр-тип. В неабстрактном методе класса вызовите абстрактный метод и верните его результат. Используйте наследование от самоограничиваемого типа и протестируйте полученный класс.

Динамическая безопасность типов

Так как обобщенные контейнеры могут передаваться коду, написанному до Java SE5, все еще остается опасность повреждения контейнеров старым кодом. В Java SE5 входит библиотека `java.util.Collections` с набором инструментов для решения проблем проверки типов в подобных ситуациях: статические методы `checkedCollection()`, `checkedList()`, `checkedMap()`, `checkedSet()`, `checkedSortedMap()` и `checkedSortedSet()`. Каждый метод в первом аргументе получает контейнер, который вы хотите динамически проверять, а во втором — тип, который требуется выдержать.

Контролируемый контейнер возбуждает исключение `ClassCastException` при попытке вставки неподходящего объекта (в отличие от неспециализированного контейнера, который сообщит о проблеме при извлечении объекта). В последнем случае вы знаете, что проблема существует, но не знаете ее причины; с контролируемыми контейнерами вы будете знать, кто попытался вставить недопустимый объект).

Рассмотрим проблему «включения кошки в список собак» с применением контролируемого контейнера. Здесь `oldStyleMethod()` представляет старый код, потому что он получает неспециализированный тип `List`, а для подавления предупреждения необходимо включить аннотацию `@SuppressWarnings("unchecked")`:

```

//: generics/CheckedList.java
// Using Collection.checkedList().
import typeinfo.pets.*;

```

продолжение ➤

```
import java.util.*;

public class CheckedList {
    @SuppressWarnings("unchecked")
    static void oldStyleMethod(List probablyDogs) {
        probablyDogs.add(new Cat());
    }
    public static void main(String[] args) {
        List<Dog> dogs1 = new ArrayList<Dog>();
        oldStyleMethod(dogs1); // Принимает Cat без возражений
        List<Dog> dogs2 = Collections.checkedList(
            new ArrayList<Dog>(), Dog.class);
        try {
            oldStyleMethod(dogs2); // Возбуждает исключение
        } catch (Exception e) {
            System.out.println(e);
        }
        // Производные типы работают нормально:
        List<Pet> pets = Collections.checkedList(
            new ArrayList<Pet>(), Pet.class);
        pets.add(new Dog());
        pets.add(new Cat());
    }
} /* Output:
java.lang.ClassCastException: Attempt to insert class
typeinfo.pets.Cat element into collection with element type
class typeinfo.pets.Dog
*///:~
```

Запустив программу, вы увидите, что вставка Cat проходит незамеченной для `dogs1`, а `dogs2` немедленно возбуждает исключение при вставке неправильного типа. Также видно, что объекты производного типа могут помещаться в контейнер с проверкой базового типа.

- 35.** (1) Измените пример `CheckedList.java` так, чтобы в нем использовались классы `Coffee`, определенные в этой главе.

Исключения

Из-за стирания возможности использования обобщений с исключениями сильно ограничены. Блок `catch` не может перехватывать исключение обобщенного типа, потому что тип исключения должен быть известен и во время компиляции, и во время выполнения. Кроме того, обобщенный класс не может прямо или косвенно наследовать от `Throwable` (это препятствует определению обобщенных исключений, которые невозможно перехватить).

Однако параметры-типы могут использоваться в секции `throws` объявления метода. Это позволяет писать обобщенный код, изменяемый по типу контролируемого исключения:

```
//: generics/ThrowGenericException.java
import java.util.*;

interface Processor<T,E extends Exception> {
    void process(List<T> resultCollector) throws E;
}
```



```
class ProcessRunner<T,E extends Exception>
extends ArrayList<Processor<T,E>> {
    List<T> processAll() throws E {
        List<T> resultCollector = new ArrayList<T>();
        for(Processor<T,E> processor : this)
            processor.process(resultCollector);
        return resultCollector;
    }
}

class Failure1 extends Exception {}

class Processor1 implements Processor<String,Failure1> {
    static int count = 3;
    public void
    process(List<String> resultCollector) throws Failure1 {
        if(count-- > 1)
            resultCollector.add("Hepl");
        else
            resultCollector.add("Hol");
        if(count < 0)
            throw new Failure1();
    }
}

class Failure2 extends Exception {}

class Processor2 implements Processor<Integer,Failure2> {
    static int count = 2;
    public void
    process(List<Integer> resultCollector) throws Failure2 {
        if(count-- == 0)
            resultCollector.add(47);
        else {
            resultCollector.add(11);
        }
        if(count < 0)
            throw new Failure2();
    }
}

public class ThrowGenericException {
    public static void main(String[] args) {
        ProcessRunner<String,Failure1> runner =
            new ProcessRunner<String,Failure1>();
        for(int i = 0; i < 3; i++)
            runner.add(new Processor1());
        try {
            System.out.println(runner.processAll());
        } catch(Failure1 e) {
            System.out.println(e);
        }

        ProcessRunner<Integer,Failure2> runner2 =
            new ProcessRunner<Integer,Failure2>();
        for(int i = 0; i < 3; i++)
            runner2.add(new Processor2());
        try {
            System.out.println(runner2.processAll());
        }
```

```

    } catch(Failure2 e) {
        System.out.println(e);
    }
}
} ///:~

```

Интерфейс `Processor` выполняет `process()` и может возбудить исключение типа `E`. Результат `process()` сохраняется в `List<T> resultCollector`. `ProcessRunner` содержит метод `processAll()`, который выполняет каждый содержащийся в нем объект `Process` и возвращает `resultCollector`.

- 36.** (2) Добавьте в класс `Processor` второе параметризованное исключение и продемонстрируйте возможность независимого изменения исключений.

Примеси

Похоже, со временем термин «*примесь*» (*mixin*) обрел много разных значений, но основной смысл подразумевает смешение функциональности нескольких классов для получения итогового класса, представляющего все типы примесей. Часто это делается в последнюю минуту, что позволяет легко и быстро объединять классы.

Одна из полезных особенностей примесей заключается в том, что они единообразно применяют характеристики и поведение нескольких классов. Кроме того, если вы захотите что-то изменить в классе примеси, эти изменения распространяются по всем классам, к которым применяется примесь. По этой причине примеси в определенном смысле связаны с аспектно-ориентированным программированием (АОП), а аспекты часто рекомендуют для решения проблемы примесей.

Примеси в C++

Именно использование примесей было одним из самых сильных аргументов в пользу множественного наследования в C++. Однако более интересный и элегантный подход к реализации примесей основан на использовании параметризованных типов, где примесь представляется классом, производным от параметра-типа. В C++ можно легко создавать примеси, потому что C++ запоминает тип параметров своих шаблонов.

В следующем примере C++ представлены два типа примесей: один «подмешивает» свойство временной метки, а другой — серийный номер каждого экземпляра:

```

//: generics/Mixins.cpp
#include <string>
#include <ctime>
#include <iostream>
using namespace std;

template<class T> class TimeStamped : public T {
    long timeStamp;
public:
    TimeStamped() { timeStamp = time(0); }
    long getStamp() { return timeStamp; }
};

template<class T> class SerialNumbered : public T {
    long serialNumber;

```

```

    static long counter;
public:
    SerialNumbered() { serialNumber = counter++; }
    long getSerialNumber() { return serialNumber; }
};

/ Определение и инициализация статического хранилища:
template<class T> long SerialNumbered<T>::counter = 1;

class Basic {
    string value;
public:
    void set(string val) { value = val; }
    string get() { return value; }
};

int main() {
    TimeStamped<SerialNumbered<Basic> > mixin1, mixin2;
    mixin1.set("test string 1");
    mixin2.set("test string 2");
    cout << mixin1.get() << " " << mixin1.getStamp() <<
        " " << mixin1.getSerialNumber() << endl;
    cout << mixin2.get() << " " << mixin2.getStamp() <<
        " " << mixin2.getSerialNumber() << endl;
} /* Output: (Sample)
test string 1 1129840250 1
test string 2 1129840250 2
*///:~

```

В `main()` итоговые типы `mixin1` и `mixin2` содержат все методы примесных типов. Примесь можно рассматривать как функцию, отображающую существующие классы на новые субклассы. Обратите внимание, насколько тривиально создаются примеси этим способом; по сути вы просто говорите: «Вот что мне нужно», — а остальное происходит автоматически:

```
TimeStamped<SerialNumbered<Basic> > mixin1, mixin2;
```

К сожалению, обобщения Java такого не позволяют. Стирание «забывает» тип базового класса, поэтому обобщенный класс не может напрямую наследовать от обобщенного параметра.

Примеси с использованием интерфейсов

Обычно для достижения эффекта примесей рекомендуется использовать интерфейсы:

```

//: generics/Mixins.java
import java.util.*;

interface TimeStamped { long getStamp(); }

class TimeStampedImp implements TimeStamped {
    private final long timeStamp;
    public TimeStampedImp() {
        timeStamp = new Date().getTime();
    }
    public long getStamp() { return timeStamp; }
}

```

продолжение ➤

```

interface SerialNumbered { long getSerialNumber(); }
class SerialNumberedImp implements SerialNumbered {
    private static long counter = 1;
    private final long serialNumber = counter++;
    public long getSerialNumber() { return serialNumber; }
}

interface Basic {
    public void set(String val);
    public String get();
}

class BasicImp implements Basic {
    private String value;
    public void set(String val) { value = val; }
    public String get() { return value; }
}

class Mixin extends BasicImp
implements TimeStamped, SerialNumbered {
    private TimeStamped timeStamp = new TimeStampedImp();
    private SerialNumbered serialNumber =
        new SerialNumberedImp();
    public long getStamp() { return timeStamp.getStamp(); }
    public long getSerialNumber() {
        return serialNumber.getSerialNumber();
    }
}

public class Mixins {
    public static void main(String[] args) {
        Mixin mixin1 = new Mixin(), mixin2 = new Mixin();
        mixin1.set("test string 1");
        mixin2.set("test string 2");
        System.out.println(mixin1.get() + " " +
            mixin1.getStamp() + " " + mixin1.getSerialNumber());
        System.out.println(mixin2.get() + " " +
            mixin2.getStamp() + " " + mixin2.getSerialNumber());
    }
} /* Output: (Sample)
test string 1 1132437151359 1
test string 2 1132437151359 2
*///:~

```

Класс `Mixin` фактически использует *делегирование*, так что каждый примесный тип требует поля в `Mixin`, и вы должны написать все необходимые методы в `Mixin` для передачи вызовов соответствующим объектам. В этом примере используются тривиальные классы, но при более сложных примесях код быстро разрастается¹.

- 37.** (2) Добавьте в `Mixins.java` новый класс примеси `Colored`. Подмешайте его в `Mixin` и покажите, что программа работает.

¹ Некоторые среды разработки — такие, как Eclipse и IntelliJ Idea, — автоматически генерируют код делегирования.

Использование паттерна «Декоратор»

Если посмотреть на способ использования, концепция примеси кажется тесно связанной с паттерном проектирования «Декоратор»¹. Декораторы часто используются в ситуациях, когда для обеспечения всех возможных комбинаций простое субклассирование порождает столько классов, что становится непрактичным.

Паттерн «Декоратор» использует иерархию объектов для динамического и прозрачного добавления обязанностей в отдельные объекты. Он определяет, что все объекты, являющиеся «обертками» для вашего исходного объекта, обладают единым базовым интерфейсом. Имеется декорируемый объект, а вы наращиваете его функциональность посредством добавления «оберток». Это делает использование декораторов прозрачным — имеется общий набор сообщений, которые могут отправляться объекту независимо от того, был он декорирован или нет. Декорирующий класс может добавлять методы, но, как вы вскоре увидите, в этом его возможности ограничены.

Декораторы реализуются с использованием композиции и формальных структур (иерархия декорируемый объект/декоратор), тогда как примеси основываются на наследовании. Таким образом, примеси, основанные на параметризованном типе, можно рассматривать как обобщенный механизм декораторов, не требующий структуры наследования паттерна «Декоратор».

Предыдущий пример можно реализовать с применением паттерна «Декоратор»:

```
//: generics/decorator/Decoration.java
package generics.decorator;
import java.util.*;

class Basic {
    private String value;
    public void set(String val) { value = val; }
    public String get() { return value; }
}

class Decorator extends Basic {
    protected Basic basic;
    public Decorator(Basic basic) { this.basic = basic; }
    public void set(String val) { basic.set(val); }
    public String get() { return basic.get(); }
}

class TimeStamped extends Decorator {
    private final long timeStamp;
    public TimeStamped(Basic basic) {
        super(basic);
        timeStamp = new Date().getTime();
    }
    public long getStamp() { return timeStamp; }
}
```

продолжение ➤

¹ Тема паттернов рассматривается в книге «Thinking in Patterns (with Java)», которую можно найти на сайте www.MindView.net. Также см. «Приемы объектно-ориентированного проектирования», Э. Гаммы и др. (Питер, 2013).

```

class SerialNumbered extends Decorator {
    private static long counter = 1;
    private final long serialNumber = counter++;
    public SerialNumbered(Basic basic) { super(basic); }
    public long getSerialNumber() { return serialNumber; }
}

public class Decoration {
    public static void main(String[] args) {
        TimeStamped t = new TimeStamped(new Basic());
        TimeStamped t2 = new TimeStamped(
            new SerialNumbered(new Basic()));
        //! t2.getSerialNumber(); // Недоступно
        SerialNumbered s = new SerialNumbered(new Basic());
        SerialNumbered s2 = new SerialNumbered(
            new TimeStamped(new Basic()));
        //! s2.getStamp(); // Недоступно
    }
} ///:~

```

Класс, полученный в результате применения примесей, содержит все нужные методы, но типом объекта, полученного от использования декораторов, будет последний тип, которым он был декорирован. Итак, хотя вы можете добавить более одного уровня, фактический тип будет определяться последним уровнем, так что видны только методы последнего уровня, тогда как тип примеси образуется *всеми* смешиваемыми типами. Следовательно, существенный недостаток паттерна «Декоратор» заключается в том, что он фактически работает с одним уровнем декорирования (последним), а решения с примесями, пожалуй, выглядит более естественно. Таким образом, паттерн «Декоратор» может считаться ограниченным решением задачи при помощи примесей.

- 38.** (4) Создайте простую систему декораторов: начните с базового типа *Coffee*, а затем добавьте декораторы для разных кофейных напитков: капучино, с декоративной пенкой, с шоколадом, с карамелью и взбитыми сливками.

Примеси и динамические заместители

Динамический заместитель позволяет создать механизм, который более точно моделирует примеси, чем паттерн «Декоратор» (за информацией о динамических заместителях Java обращайтесь к главе 14). С динамическим заместителем динамический тип полученного класса образуется объединением всех смешиваемых типов.

Из-за ограничений динамических посредников каждый добавляемый класс должен быть реализацией интерфейса:

```

//: generics/DynamicProxyMixin.java
import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Tuple.*;

class MixinProxy implements InvocationHandler {
    Map<String, Object> delegatesByMethod;
    public MixinProxy(TwoTuple<Object, Class<?>>... pairs) {

```

```

    delegatesByMethod = new HashMap<String, Object>();
    for(TwoTuple<Object, Class<?>> pair : pairs) {
        for(Method method : pair.second.getMethods()) {
            String methodName = method.getName();
            // Первый интерфейс в карте реализует метод.
            if (!delegatesByMethod.containsKey(methodName))
                delegatesByMethod.put(methodName, pair.first);
        }
    }
}

public Object invoke(Object proxy, Method method,
    Object[] args) throws Throwable {
    String methodName = method.getName();
    Object delegate = delegatesByMethod.get(methodName);
    return method.invoke(delegate, args);
}

@SuppressWarnings("unchecked")
public static Object newInstance(TwoTuple... pairs) {
    Class[] interfaces = new Class[pairs.length];
    for(int i = 0; i < pairs.length; i++) {
        interfaces[i] = (Class)pairs[i].second;
    }
    ClassLoader cl =
        pairs[0].first.getClass().getClassLoader();
    return Proxy.newProxyInstance(
        cl, interfaces, new MixinProxy(pairs));
}
}

public class DynamicProxyMixin {
    public static void main(String[] args) {
        Object mixin = MixinProxy.newInstance(
            tuple(new BasicImp(), Basic.class),
            tuple(new TimeStampedImp(), TimeStamped.class),
            tuple(new SerialNumberedImp(), SerialNumbered.class));
        Basic b = (Basic)mixin;
        TimeStamped t = (TimeStamped)mixin;
        SerialNumbered s = (SerialNumbered)mixin;
        b.set("Hello");
        System.out.println(b.get());
        System.out.println(t.getStamp());
        System.out.println(s.getSerialNumber());
    }
} /* Output: (Sample)
Hello
1132519137015
1
*///:~

```

Так как все типы-примеси включают только динамический, а не статический тип, это решение все равно хуже подхода C++, потому что вам приходится выполнять нисходящее преобразование к соответствующему типу, прежде чем вызывать для него методы. Несмотря на это, оно намного ближе к «настоящим» примесям.

В направлении поддержки примесей в Java ведется значительная работа, включающая создание как минимум одного расширения — языка Jam, предназначенного специально для поддержки примесей.

39. (1) Добавьте в `DynamicProxyMixin.java` новый класс примеси `Colored`. Продемонстрируйте, что он работает.

Латентная типизация

В начале этой главы была представлена концепция написания кода, который может применяться как можно шире. Для этого необходимы механизмы ослабления ограничений на типы, с которыми может работать код, без потери преимуществ статической проверки типов. Тогда мы сможем писать код, используемый в большем количестве ситуаций без изменений, то есть более «обобщенный» код.

Обобщения Java вроде бы делают шаг в нужном направлении. Когда вы пишете или используете обобщение, которое просто хранит объекты, этот код будет работать с любым типом (кроме примитивов, но как мы видели, автоматическая упаковка сглаживает эту проблему). Для кода несущественно, с каким типом он работает; такой код может применяться везде, а следовательно, является достаточно универсальным.

Проблема возникает при выполнении манипуляций с обобщенными типами (помимо вызова методов `Object`), потому что из-за стирания для успешного вызова конкретных методов обобщенных объектов в вашем коде необходимо задать ограничения обобщенных типов. Это существенно снижает полезность концепции «обобщений», потому что вам приходится ограничивать свои обобщенные типы так, чтобы они наследовали от конкретных классов или реализовали конкретные интерфейсы. В некоторых ситуациях это может привести к использованию обычного класса или интерфейса, потому что ограниченное обобщение может не отличаться от него по сути.

В некоторых языках программирования эта проблема решается при помощи *латентной* (latent), или структурной, типизации. Также встречается экзотическое выражение «утиная типизация», означающее: «Если нечто плавает как утка и крякает как утка, то с ним можно обходиться как с уткой». Термин «утиная типизация» стал довольно популярным, потому что он, в отличие от других терминов, не несет лишней исторической нагрузки.

Обобщенный код обычно вызывает небольшое количество методов обобщенного типа. Язык с латентной типизацией ослабляет это ограничение (и порождает более универсальный код), требуя лишь реализации подмножества методов, а не конкретного класса или интерфейса. По этой причине латентная типизация позволяет охватывать иерархии классов и вызывать методы, которые не являются частью общего интерфейса. Таким образом, фрагмент кода может по сути означать: «Меня не интересует, к какому типу ты относишься, — главное, что ты можешь вызвать `speak()` и `sit()`». Отказ от требования конкретного типа повышает универсальность их кода.

Латентная типизация представляет собой механизм структурирования и повторного использования кода. Фрагмент кода, написанный с применением латентной типизации, использовать заново будет проще, чем без нее. Структура и повторное использование кода — краеугольные камни всего программирования: написать код один раз, использовать много раз, хранить код в одном месте. Поскольку я не обязан указывать имя точного интерфейса, с которым работает мой код, с латентной типизацией я смогу написать меньший объем кода, который будет проще применять в разных местах.

Латентная типизация поддерживается в языках Python (свободно загружается на сайте www.Python.org) и C++¹. В Python используется динамическая типизация (практически вся проверка типов происходит во время выполнения), а C++ является языком со статической типизацией (проверка типов выполняется во время компиляции), так что для латентной типизации не является обязательной ни статическая, ни динамическая проверка типов.

Если взять приведенное выше описание и выразить его на Python, оно будет выглядеть так:

```
#: generics/DogsAndRobots.py
```

```
class Dog:
    def speak(self):
        print "Arf!"
    def sit(self):
        print "Sitting"
    def reproduce(self):
        pass
```

```
class Robot:
    def speak(self):
        print "Click!"
    def sit(self):
        print "Clank!"
    def oilChange(self):
        pass
def perform(anything):
    anything.speak()
    anything.sit()
```

```
a = Dog()
b = Robot()
perform(a)
perform(b)
#::~~
```

Python использует отступы для определения области действия (таким образом, фигурные скобки не обязательны), а новая область действия обозначается двоеточием. Символ # обозначает комментарий до конца строки, по аналогии с // в Java. Методы класса явно включают эквивалент ссылки `this` в первом аргументе, по правилам называемом `self`. При вызовах конструктора ключевое слово `new` не используется. Кроме того, в Python допускаются обычные (не являющиеся членами классов) функции, как показывает `perform()`. В методе `perform(anything)` параметр `anything` не имеет типа, это всего лишь идентификатор. Он должен быть способен выполнить операции, которые запрашивает `perform()`, так что здесь подразумевается интерфейс. Но вам не приходится явно определять этот интерфейс — он является *латентным*. Для метода `perform()` тип его аргумента не важен; ему можно передать любой объект, поддерживающий методы `speak()` и `sit()`. Если передаваемый `perform()` объект не поддерживает эти операции, происходит исключение времени выполнения.

¹ Языки Ruby и Smalltalk также поддерживают латентную типизацию.

Аналогичного эффекта можно добиться на C++:

```
//: generics/DogsAndRobots.cpp

class Dog {
public:
    void speak() {}
    void sit() {}
    void reproduce() {}
};

class Robot {
public:
    void speak() {}
    void sit() {}
    void oilChange() {}
};

template<class T> void perform(T anything) {
    anything.speak();
    anything.sit();
}

int main() {
    Dog d;
    Robot r;
    perform(d);
    perform(r);
} ///:~
```

В Python и C++ типы Dog и Robot не имеют ничего общего, кроме того, что они содержат два метода с одинаковыми сигнатурами. С точки зрения типов это совершенно разные типы. Однако метод `perform()` не обращает внимания на конкретный тип своего аргумента, а латентная типизация позволяет ему получать объекты обоих типов.

C++ убеждается, что он действительно может отправлять эти сообщения. При попытке передачи неправильного типа компилятор выдает сообщение об ошибке (эти сообщения традиционно были очень длинными и страшными — главная причина, по которой шаблоны C++ пользовались дурной славой). И хотя они делают это в разное время — C++ во время компиляции, Python во время выполнения, — оба языка следят за тем, чтобы типы использовались правильно, а следовательно, считаются сильно типизованными¹. Латентная типизация не нарушает сильную типизацию.

Поскольку обобщения были добавлены в Java на поздней стадии, ни малейшей возможности реализовать латентную типизацию не было, поэтому в Java данная возможность не поддерживается. В результате на первый взгляд кажется, что механизм обобщений Java «менее обобщен», чем в языках с поддержкой латентной типизации². Например, при попытке реализовать приведенный пример на Java придется использовать класс или интерфейс и указать его в выражении ограничения:

¹ Из-за возможности приведения типов, которое фактически подавляет систему типов, некоторые специалисты считают C++ языком со слабой типизацией, но это преувеличение. Пожалуй, правильнее сказать, что C++ обладает «сильной типизацией с лазейкой».

² Обобщения Java на базе стирания иногда называются *второклассными обобщениями*.

```
//: generics/Performs.java

public interface Performs {
    void speak();
    void sit();
} ///:~

//: generics/DogsAndRobots.java
// В Java нет латентной типизации
import typeinfo.pets.*;
import static net.mindview.util.Print.*;

class PerformingDog extends Dog implements Performs {
    public void speak() { print("Woof!"); }
    public void sit() { print("Sitting"); }
    public void reproduce() {}
}

class Robot implements Performs {
    public void speak() { print("Click!"); }
    public void sit() { print("Clank!"); }
    public void oilChange() {}
}

class Communicate {
    public static <T extends Performs>
    void perform(T performer) {
        performer.speak();
        performer.sit();
    }
}

public class DogsAndRobots {
    public static void main(String[] args) {
        PerformingDog d = new PerformingDog();
        Robot r = new Robot();
        Communicate.perform(d);
        Communicate.perform(r);
    }
} /* Output:
Woof!
Sitting
Click!
Clank!
*///:~
```

Однако следует заметить, что для работы `perform()` не обязательно использовать обобщения. Можно просто передать объект `Performs`:

```
//: generics/SimpleDogsAndRobots.java
// Removing the generic; code still works.

class CommunicateSimply {
    static void perform(Performs performer) {
        performer.speak();
        performer.sit();
    }
}
```

```

public class SimpleDogsAndRobots {
    public static void main(String[] args) {
        CommunicateSimply.perform(new PerformingDog());
        CommunicateSimply.perform(new Robot());
    }
} /* Output:
Woof!
Sitting
Click!
Clank!
*///:~

```

В данном случае обобщения просто не нужны, поскольку классы уже вынуждены реализовать интерфейс `Performs`.

Компенсация отсутствия латентной типизации

Хотя в Java латентная типизация не поддерживается, это не значит, что ограниченный обобщенный код не может применяться между разными иерархиями типов. Написать полностью обобщенный код возможно, но для этого придется дополнительно потрудиться.

Отражение

Первый способ основан на использовании отражения. Вот как выглядит метод `perform()`, использующий латентную типизацию:

```

//: generics/LatentReflection.java
// Использование отражения для моделирования латентной типизации.
import java.lang.reflect.*;
import static net.mindview.util.Print.*;

// Не реализует Performs:
class Mime {
    public void walkAgainstTheWind() {}
    public void sit() { print("Pretending to sit"); }
    public void pushInvisibleWalls() {}
    public String toString() { return "Mime"; }
}

// Не реализует Performs:
class SmartDog {
    public void speak() { print("Woof!"); }
    public void sit() { print("Sitting"); }
    public void reproduce() {}
}

class CommunicateReflectively {
    public static void perform(Object speaker) {
        Class<?> spkr = speaker.getClass();
        try {
            try {
                Method speak = spkr.getMethod("speak");
                speak.invoke(speaker);
            }

```

```

    } catch(NoSuchMethodException e) {
        print(speaker + " cannot speak");
    }
    try {
        Method sit = spkr.getMethod("sit");
        sit.invoke(speaker);
    } catch(NoSuchMethodException e) {
        print(speaker + " cannot sit");
    }
} catch(Exception e) {
    throw new RuntimeException(speaker.toString(), e);
}
}
}

```

```

public class LatentReflection {
    public static void main(String[] args) {
        CommunicateReflectively.perform(new SmartDog());
        CommunicateReflectively.perform(new Robot());
        CommunicateReflectively.perform(new Mime());
    }
}

```

/* Output:

Woof!

Sitting

Click!

Clank!

Mime cannot speak

Pretending to sit

*/~

Здесь классы полностью разъединены и не имеют общих базовых классов (кроме `Object`) или интерфейсов. Используя отражение, метод `CommunicateReflectively.perform()` может динамически проверить доступность нужных методов и вызвать их. Он даже может обработать ситуацию, в которой `Mime` содержит только один из необходимых методов, и частично решает свою задачу.

Применение метода к последовательности

Отражение открывает интересные возможности, но передает всю проверку типов на стадию выполнения, что нежелательно во многих ситуациях. Если проверку типов можно обеспечить на стадии компиляции, обычно этот вариант более желателен. Но возможно ли совместить проверку типов во время компиляции с латентной типизацией?

Рассмотрим пример, поясняющий суть проблемы. Допустим, требуется создать метод `apply()`, который будет применять произвольный метод к каждому объекту в последовательности. Похоже, в этой ситуации интерфейсы не подходят: интерфейсы слишком сильно ограничивают возможности описания «произвольного метода». Как сделать это в Java?

Прежде всего задачу можно решить посредством отражения; решение получается достаточно элегантным благодаря спискам аргументов переменной длины Java SE5:

```

//: generics/Apply.java
// {main: ApplyTest}
import java.lang.reflect.*;

```

продолжение ➤

```

import java.util.*;
import static net.mindview.util.Print.*;

public class Apply {
    public static <T, S extends Iterable<? extends T>>
    void apply(S seq, Method f, Object... args) {
        try {
            for(T t: seq)
                f.invoke(t, args);
        } catch(Exception e) {
            // Сбои являются ошибками программиста
            throw new RuntimeException(e);
        }
    }
}

class Shape {
    public void rotate() { print(this + " rotate"); }
    public void resize(int newSize) {
        print(this + " resize " + newSize);
    }
}

class Square extends Shape {}

class FilledList<T> extends ArrayList<T> {
    public FilledList(Class<? extends T> type, int size) {
        try {
            for(int i = 0; i < size; i++)
                // Предполагается конструктор по умолчанию:
                add(type.newInstance());
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
}

class ApplyTest {
    public static void main(String[] args) throws Exception {
        List<Shape> shapes = new ArrayList<Shape>();
        for(int i = 0; i < 10; i++)
            shapes.add(new Shape());
        Apply.apply(shapes, Shape.class.getMethod("rotate"));
        Apply.apply(shapes,
            Shape.class.getMethod("resize", int.class), 5);
        List<Square> squares = new ArrayList<Square>();
        for(int i = 0; i < 10; i++)
            squares.add(new Square());
        Apply.apply(squares, Shape.class.getMethod("rotate"));
        Apply.apply(squares,
            Shape.class.getMethod("resize", int.class), 5);

        Apply.apply(new FilledList<Shape>(Shape.class, 10),
            Shape.class.getMethod("rotate"));
        Apply.apply(new FilledList<Shape>(Square.class, 10),
            Shape.class.getMethod("rotate"));
    }
}

```

```

SimpleQueue<Shape> shapeQ = new SimpleQueue<Shape>();
for(int i = 0; i < 5; i++) {
    shapeQ.add(new Shape());
    shapeQ.add(new Square());
}
Apply.apply(shapeQ, Shape.class.getMethod("rotate"));
}
} /* (Execute to see output) *///:~

```

В методе `Apply` нам повезло, потому что в Java имеется встроенный интерфейс `Iterable`, который используется библиотекой контейнеров Java. По этой причине метод `apply()` может принять любой объект, реализующий интерфейс `Iterable`; к этой категории относятся все классы `Collection` (такие, как `List`). Но он также может принять любой другой объект при условии, что он будет реализовывать `Iterable`, — например, класс `SimpleQueue`, определяемый здесь и используемый в методе `main()` (см. выше):

```

//: generics/SimpleQueue.java
// Другой вид контейнера, реализующий Iterable
import java.util.*;

public class SimpleQueue<T> implements Iterable<T> {
    private LinkedList<T> storage = new LinkedList<T>();
    public void add(T t) { storage.offer(t); }
    public T get() { return storage.poll(); }
    public Iterator<T> iterator() {
        return storage.iterator();
    }
} ///:~

```

В методе `Apply.java` исключения преобразуются в `RuntimeException`, потому что продолжить работу программы после исключений будет затруднительно — в данном случае они представляют ошибки программиста.

Обратите внимание: мне пришлось включить ограничения и шаблоны, чтобы `Apply` и `FieldList` работали во всех нужных ситуациях. Попробуйте ради эксперимента убрать их; вы увидите, что в некоторых случаях `Apply` и `FilledList` работать не будут.

С `FilledList` возникают некоторые затруднения. Чтобы тип мог использоваться, он должен иметь конструктор по умолчанию (без аргументов). В Java не существует средств для такой проверки во время компиляции, поэтому проблема выявляется во время выполнения. Обычно для проведения проверки на стадии компиляции рекомендуется определить фабричный интерфейс, содержащий метод для генерирования объектов; затем `FilledList` передается этот интерфейс вместо «низкоуровневой фабрики» маркера типа. Проблема в том, что все классы, используемые в `FilledList`, должны реализовать фабричный интерфейс. К сожалению, большинство классов создается без информации о вашем интерфейсе и потому не реализует его. Позже я представлю решение, основанное на использовании адаптеров.

Однако решение с маркером типа, возможно, является разумным компромиссом (или по крайней мере прототипом решения). С этим подходом использовать `FilledList` достаточно просто. Конечно, поскольку информация об ошибках выдается во время выполнения, вы должны быть уверены в том, что эти ошибки будут выявлены на достаточно ранней стадии процесса разработки.

Решение с маркером типа рекомендуется в литературе по Java — например, в статье Гилада Брача «Generics in the Java Programming Language¹», где он замечает: «Например, эта идиома широко используется в новых API для управления аннотациями». Но я заметил, что не все разработчики достаточно уверенно пользуются этим подходом; некоторые предпочитают решение с фабриками, представленное ранее в этой главе.

Кроме того, при всей элегантности этого решения необходимо заметить, что отражение (хотя оно было значительно усовершенствовано в последних версиях Java) работает относительно медленно из-за большого объема работы во время выполнения. Это обстоятельство не должно стать препятствием для использования решения (по крайней мере в прототипах — не забывайте, что говорилось ранее о поспешной оптимизации), но оно безусловно отличает эти два подхода.

- 40.** (3) Добавьте метод `speak()` во все классы иерархии `typeinfo.pets`. Измените пример `Apply.java`, чтобы метод `speak()` вызывался в нем для разнородной коллекции `Pet`.

Если нужный интерфейс отсутствует

В предыдущем примере был доступен встроенный интерфейс `Iterable`, который делал в точности то, что требовалось. Но что делать в общем случае, если нет готового интерфейса, который подходит для ваших целей?

Для примера мы обобщим идею `FilledList` и создадим параметризованный метод `fill()`, который получает последовательность и заполняет ее при помощи `Generator`. Попытка запрограммировать это решение на Java сталкивается с проблемой: не существует удобного интерфейса `Addable` (сходного с интерфейсом `Iterable` из предыдущего примера). Итак, вместо формулировки «любой объект, для которого можно вызвать `add()`» необходимо использовать формулировку «подтип `Collection`». Полученный код не является полностью обобщенным, потому что он должен ограничиваться для работы с реализациями `Collection`. При попытке использования класса, не реализующего `Collection`, наш обобщенный код работать не будет. Вот как это выглядит:

```
//: generics/Fill.java
// Обобщение идеи FilledList
// {main: FillTest}
import java.util.*;

// Не работает с "любым объектом, содержащим add()". Интерфейса
// "Addable" не существует, так что приходится использовать
// Collection. В данном случае полноценное обобщение невозможно.

public class Fill {
    public static <T> void fill(Collection<T> collection,
        Class<? extends T> classToken, int size) {
        for(int i = 0; i < size; i++)
            // Предполагается наличие конструктора по умолчанию:
            try {
                collection.add(classToken.newInstance());
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
    }
}
```

¹ См. ссылку в конце главы.


```

    }
}

class Contract {
    private static long counter = 0;
    private final long id = counter++;
    public String toString() {
        return getClass().getName() + " " + id;
    }
}

class TitleTransfer extends Contract {}

class FillTest {
    public static void main(String[] args) {
        List<Contract> contracts = new ArrayList<Contract>();
        Fill.fill(contracts, Contract.class, 3);
        Fill.fill(contracts, TitleTransfer.class, 2);
        for(Contract c: contracts)
            System.out.println(c);
        SimpleQueue<Contract> contractQueue =
            new SimpleQueue<Contract>();
        // Не работает. Метод fill() недостаточно обобщен:
        // Fill.fill(contractQueue, Contract.class, 3);
    }
} /* Output:
Contract 0
Contract 1
Contract 2
TitleTransfer 3
TitleTransfer 4
*///:~

```

В этой ситуации механизм параметризованных типов с латентной типизацией будет полезен, потому что вы не зависите от прошлых решений, принятых создателем конкретной библиотеки, и вам не придется переписывать свой код каждый раз, когда встретится новая библиотека, не учитывающая эту ситуацию (то есть код действительно «обобщен»). В приведенном выше случае, поскольку проектировщики Java (по понятным причинам) не видели необходимости в интерфейсе `Addable`, мы ограничиваемся иерархией `Collection`, а класс `SimpleQueue`, несмотря на наличие у него метода `add()`, не подойдет. Код, ограниченный использованием `Collection`, вряд ли можно считать полностью обобщенным. С латентной типизацией такой проблемы не будет.

Моделирование латентной типизации с использованием адаптеров

Обобщения Java не имеют латентной типизации, а для написания кода, применимого между границами классов (то есть действительно «обобщенного» кода), необходимо некое подобие латентной типизации. Можно ли как-то обойти это ограничение?

Что здесь дает латентная типизация? По сути вы пишете код, который означает: «Неважно, какой тип я здесь использую, при условии, что он содержит эти методы». Латентная типизация в некотором смысле создает *неявный интерфейс*, содержащий нужные методы. Получается, что если мы напишем необходимый интерфейс вручную (поскольку Java не сделает это за нас), это должно решить проблему.

Написание кода для предоставления нужного интерфейса по имеющемуся интерфейсу — пример паттерна проектирования «Адаптер». Адаптеры приспособляют существующие классы для предоставления нужного интерфейса при относительно небольшом объеме кода. Следующее решение, в котором используется ранее определенная иерархия Coffee, демонстрирует разные способы написания адаптеров:

```
//: generics/Fill2.java
// Использование адаптеров для моделирования латентной типизации.
// {main: Fill2Test}
import generics.coffee.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

interface Addable<T> { void add(T t); }

public class Fill2 {
    // Версия с маркером:
    public static <T> void fill(Addable<T> addable,
        Class<? extends T> classToken, int size) {
        for(int i = 0; i < size; i++)
            try {
                addable.add(classToken.newInstance());
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
    }
    // Версия с генератором:
    public static <T> void fill(Addable<T> addable,
        Generator<T> generator, int size) {
        for(int i = 0; i < size; i++)
            addable.add(generator.next());
    }
}

// Для адаптации базового типа необходимо использовать композицию.
// Включение поддержки Addable в произвольный контейнер Collection
// с использованием композиции:
class AddableCollectionAdapter<T> implements Addable<T> {
    private Collection<T> c;
    public AddableCollectionAdapter(Collection<T> c) {
        this.c = c;
    }
    public void add(T item) { c.add(item); }
}

// Вспомогательный класс для автоматической фиксации типа:
class Adapter {
    public static <T>
        Addable<T> collectionAdapter(Collection<T> c) {
        return new AddableCollectionAdapter<T>(c);
    }
}

// Для адаптации конкретного типа можно применить наследование.
// Включение Addable в SimpleQueue Addable посредством наследования:
```

```

class AddableSimpleQueue<T>
extends SimpleQueue<T> implements Addable<T> {
    public void add(T item) { super.add(item); }
}

class Fill2Test {
    public static void main(String[] args) {
        // Адаптация Collection:
        List<Coffee> carrier = new ArrayList<Coffee>();
        Fill2.fill(
            new AddableCollectionAdapter<Coffee>(carrier),
            Coffee.class, 3);
        // Вспомогательный метод фиксирует тип:
        Fill2.fill(Adapter.collectionAdapter(carrier),
            Latte.class, 2);
        for(Coffee c: carrier)
            print(c);
        print("-----");
        // Использование адаптированного класса:
        AddableSimpleQueue<Coffee> coffeeQueue =
            new AddableSimpleQueue<Coffee>();
        Fill2.fill(coffeeQueue, Mocha.class, 4);
        Fill2.fill(coffeeQueue, Latte.class, 1);
        for(Coffee c: coffeeQueue)
            print(c);
    }
} /* Output:
Coffee 0
Coffee 1
Coffee 2
Latte 3
Latte 4
-----
Mocha 5
Mocha 6
Mocha 7
Mocha 8
Latte 9
*///:~

```

Класс `Fill2` не требует `Collection`, в отличие от `Fill`. Вместо этого ему требуется любой объект, реализующий `Addable`, а интерфейс `Addable` был написан только для `Fill` — это проявление латентного типа, который компилятор должен был создать для меня.

В этой версии также был добавлен перегруженный метод `fill()`, получающий `Generator` вместо маркера типа. Тип `Generator` безопасен на стадии компиляции: компилятор следит за тем, чтобы передавался правильный объект `Generator`, так что исключения возбуждаться не будут.

Первый адаптер `AddableCollectionAdapter` работает с базовым типом `Collection`; это означает, что может использоваться любая реализация `Collection`. Эта версия просто сохраняет ссылку на `Collection` и использует ее для реализации `add()`.

Если вместо базового класса иерархии используется конкретный тип, это позволяет несколько сократить объем кода при создании адаптера с применением наследования, как видно на примере `AddableSimpleQueue`.

В `Fill2Test.main()` продемонстрировано использование разных типов адаптеров. Сначала тип `Collection` адаптируется классом `AddableCollectionAdapter`. Вторая версия использует обобщенный вспомогательный метод; вы видите, как обобщенный метод фиксирует тип, чтобы его не приходилось записывать явно, — удобный трюк, который делает код более элегантным.

Затем используется заранее адаптированный класс `AddableSimpleQueue`. Обратите внимание: в обоих случаях адаптеры позволяют использовать классы, которые ранее не реализовали `Addable`, с `Fill2.fill()`.

Такое использование адаптеров вроде бы компенсирует отсутствие латентной типизации, а следовательно, позволяет писать по-настоящему обобщенный код. Однако это лишний шаг, который должен быть понятен как создателю библиотеки, так и ее потребителю, а у менее опытного программиста могут возникнуть трудности с пониманием этой концепции. Латентная типизация устраняет этот лишний шаг и делает обобщенный код более понятным; этим она и полезна.

- 41.** (1) Измените пример `Fill2.java`, чтобы вместо классов `Coffee` в нем использовались классы из `typeinfo.pets`.

Использование объектов функций как стратегий

В последнем примере мы создадим полноценный обобщенный код с использованием методологии адаптеров из предыдущего раздела. Пример начинался с попытки вычисления суммы по последовательности элементов (любого типа, поддерживающего суммирование), но поднялся до выполнения общих операций с использованием функционального стиля программирования.

Если просто рассмотреть попытки сложения объектов, становится понятно, что перед нами одна из ситуаций с наличием в классах общих операций, которые не могут быть представлены в базовом классе, — в одних случаях можно будет использовать оператор `+`, в других может применяться некая разновидность оператора метода `add()`. Обычно подобные ситуации встречаются при попытке написания обобщенного кода, потому что вы хотите, чтобы код мог применяться между классами, — особенно, как в нашем случае, несколькими уже существующими классами, которые мы не в состоянии «подправить». Даже если сузить постановку задачи до субклассов `Number`, этот суперкласс все равно не включает никаких аспектов, относящихся к «суммируемости».

Задача решается использованием паттерна проектирования «Стратегия», который порождает более элегантный код, полностью изолирующий «то, что изменяется» внутри объекта функции¹. Объект функции — объект, который в какой-то мере ведет себя как функция, — как правило, он содержит всего один метод, представляющий интерес (в языках, поддерживающих перегрузку операторов, вызов этого метода может выглядеть как обычное применение оператора). Полезность объектов функций связана с

¹ Иногда эти объекты называются *функторами*. Я буду использовать термин «объект функции», потому что в математике термин «функтор» имеет вполне конкретное и совершенно другое значение.

тем, что, в отличие от обычных методов, они могут передаваться при вызовах, а также могут иметь состояние, поддерживаемое между вызовами. Конечно, нечто подобное можно сделать при помощи любого метода класса, но объекты функций (как и любые паттерны проектирования), прежде всего, отличаются по своему назначению. В данном случае они предназначены для создания чего-то, что выглядит как метод, но может передаваться при вызовах; следовательно, это «что-то» тесно связано с паттерном проектирования «Стратегия» (а иногда неотличимо от него).

Как я обнаружил для многих паттернов проектирования, границы между паттернами здесь размываются: мы создаем объекты функций, которые выполняют адаптацию, и эти объекты передаются методам для использования как стратегии.

Выбрав этот путь, я добавил разнообразные обобщенные методы, которые изначально намеревался создать, и несколько других. Результат выглядит так:

```
//: generics/Functional.java
import java.math.*;
import java.util.concurrent.atomic.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Разные типы объектов функций:
interface Combiner<T> { T combine(T x, T y); }
interface UnaryFunction<R,T> { R function(T x); }
interface Collector<T> extends UnaryFunction<T,T> {
    T result(); // Извлечение результата из параметра-накопителя
}
interface UnaryPredicate<T> { boolean test(T x); }

public class Functional {
    // Вызывает объект Combiner для каждого элемента,
    // чтобы объединить его с накапливаемым результатом,
    // который в конечном итоге возвращается:
    public static <T> T
    reduce(Iterable<T> seq, Combiner<T> combiner) {
        Iterator<T> it = seq.iterator();
        if(it.hasNext()) {
            T result = it.next();
            while(it.hasNext())
                result = combiner.combine(result, it.next());
            return result;
        }
        // Если seq - пустой список:
        return null; // Или возбудить исключение
    }
    // Получает объект функции и вызывает его для каждого объекта
    // в списке, игнорируя возвращаемое значение. Объект
    // функции может действовать как параметр-накопитель,
    // поэтому он возвращается в конце.
    public static <T> Collector<T>
    forEach(Iterable<T> seq, Collector<T> func) {
        for(T t : seq)
            func.function(t);
        return func;
    }
    // Создает список результатов, вызывая объект функции
    // для каждого объекта в списке:
```

продолжение ➤

```

public static <R,T> List<R>
transform(Iterable<T> seq, UnaryFunction<R,T> func) {
    List<R> result = new ArrayList<R>();
    for(T t : seq)
        result.add(func.function(t));
    return result;
}
// Применяет унарный предикат к каждому элементу последовательности
// и возвращает список элементов, для которых
// было получено значение "true":
public static <T> List<T>
filter(Iterable<T> seq, UnaryPredicate<T> pred) {
    List<T> result = new ArrayList<T>();
    for(T t : seq)
        if(pred.test(t))
            result.add(t);
    return result;
}
// Для использования приведенных выше обобщенных методов
// необходимо создать объекты функций для их адаптации
// к нашим конкретным потребностям:
static class IntegerAdder implements Combiner<Integer> {
    public Integer combine(Integer x, Integer y) {
        return x + y;
    }
}
static class
IntegerSubtracter implements Combiner<Integer> {
    public Integer combine(Integer x, Integer y) {
        return x - y;
    }
}
static class
BigDecimalAdder implements Combiner<BigDecimal> {
    public BigDecimal combine(BigDecimal x, BigDecimal y) {
        return x.add(y);
    }
}
static class
BigIntegerAdder implements Combiner<BigInteger> {
    public BigInteger combine(BigInteger x, BigInteger y) {
        return x.add(y);
    }
}
static class
AtomicLongAdder implements Combiner<AtomicLong> {
    public AtomicLong combine(AtomicLong x, AtomicLong y) {
        // Неясно, насколько это осмысленно:
        return new AtomicLong(x.addAndGet(y.get()));
    }
}
static class BigDecimalUlp
implements UnaryFunction<BigDecimal, BigDecimal> {
    public BigDecimal function(BigDecimal x) {
        return x.ulp();
    }
}
static class GreaterThan<T extends Comparable<T>>
implements UnaryPredicate<T> {

```

```
private T bound;
public GreaterThan(T bound) { this.bound = bound; }
public boolean test(T x) {
    return x.compareTo(bound) > 0;
}
}
static class MultiplyingIntegerCollector
implements Collector<Integer> {
    private Integer val = 1;
    public Integer function(Integer x) {
        val *= x;
        return val;
    }
    public Integer result() { return val; }
}
public static void main(String[] args) {
    // Обобщения, списки аргументов переменной длины
    // и упаковка работают совместно:
    List<Integer> li = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
    Integer result = reduce(li, new IntegerAdder());
    print(result);

    result = reduce(li, new IntegerSubtracter());
    print(result);

    print(filter(li, new GreaterThan<Integer>(4)));

    print(forEach(li,
        new MultiplyingIntegerCollector()).result());

    print(forEach(filter(li, new GreaterThan<Integer>(4)),
        new MultiplyingIntegerCollector()).result());

    MathContext mc = new MathContext(7);
    List<BigDecimal> lbd = Arrays.asList(
        new BigDecimal(1.1, mc), new BigDecimal(2.2, mc),
        new BigDecimal(3.3, mc), new BigDecimal(4.4, mc));
    BigDecimal rbd = reduce(lbd, new BigDecimalAdder());
    print(rbd);

    print(filter(lbd,
        new GreaterThan<BigDecimal>(new BigDecimal(3))));

    // Использование средств генерирования простых чисел BigInteger:
    List<BigInteger> lbi = new ArrayList<BigInteger>();
    BigInteger bi = BigInteger.valueOf(11);
    for(int i = 0; i < 11; i++) {
        lbi.add(bi);
        bi = bi.nextProbablePrime();
    }
    print(lbi);
    BigInteger rbi = reduce(lbi, new BigIntegerAdder());
    print(rbi);
    // Сумма этого списка простых чисел также является простым числом:
    print(rbi.isProbablePrime(5));

    List<AtomicLong> lal = Arrays.asList(
        new AtomicLong(11), new AtomicLong(47),
        new AtomicLong(74), new AtomicLong(133));
```

```

    AtomicLong ral = reduce(lal, new AtomicLongAdder());
    print(ral);

    print(transform(lbd, new BigDecimalUlp()));
}
} /* Output:
28
-26
[5, 6, 7]
5040
210
11.000000
[3.300000, 4.400000]
[11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
311
true
265
[0.000001, 0.000001, 0.000001, 0.000001]
*///:~

```

Я начинаю с определения интерфейсов для разных типов объектов функций. Они создавались по мере надобности, пока я разрабатывал разные методы и находил применение для каждого из них. Класс `Combiner` был предложен анонимным читателям одной из статей, опубликованных на моем сайте. Он абстрагируется от подробностей попытки сложения двух объектов и просто говорит, что они каким-то образом объединяются. В результате и `IntegerAdder`, и `IntegerSubtractor` могут быть типами `Combiner`.

`UnaryFunction` получает один аргумент и производит результат; аргумент и результат не обязаны относиться к одному типу. `Collector` используется в качестве «параметра-накопителя», и вы можете извлечь результат при завершении. `UnaryPredicate` производит логический (`boolean`) результат. Также можно определить другие объекты функций, но для демонстрации достаточно и этих.

Класс `Functional` содержит набор обобщенных методов, применяющих объекты функций к последовательностям. Метод `reduce()` применяет функцию в `Combiner` к каждому элементу последовательности для получения единого результата.

Метод `forEach()` получает объект `Collector` и применяет его функцию к каждому элементу, игнорируя результат каждого вызова функции. Такая возможность может применяться просто ради побочных эффектов (что не соответствует «функциональному» стилю программирования, но все равно может быть полезно), или же `Collector` может поддерживать внутреннее состояние и стать параметром-накопителем, как в приведенном примере.

Метод `transform()` создает список, вызывая `UnaryFunction` для каждого объекта в последовательности и фиксируя результат.

Наконец, метод `filter()` применяет `UnaryPredicate` к каждому объекту в последовательности, сохраняет элементы, для которых было получено значение `true`, в контейнере `List` и возвращает его.

При желании вы можете определить дополнительные обобщенные функции. Библиотека C++ STL содержит множество примеров такого рода. Проблема также была решена в некоторых библиотеках с открытым кодом — таких, как JGA (Generic Algorithms for Java).

В C++ латентная типизация обеспечивает правильный подбор операций при вызове функций, а на Java необходимо писать объекты функций, адаптирующие обобщенные методы к конкретным потребностям. Соответственно в следующей части класса приводятся разные реализации объектов функций. Обратите внимание, например, что `IntegerAdder` и `BigDecimalAdder` решают одну задачу (сложение двух объектов), вызывая операции, соответствующие их конкретному типу. Таким образом, паттерн «Адаптер» здесь сочетается с паттерном «Стратегия».

В методе `main()` мы видим, что при каждом вызове метода последовательность передается вместе с подходящим объектом функции. Кроме того, некоторые выражения могут стать довольно сложными, как в следующем примере:

```
forEach(filter(li, new GreaterThan(4)),
        new MultiplyingIntegerCollector()).result()
```

Этот вызов создает список из всех элементов `li`, больших 4, после чего применяет `MultiplyingIntegerCollector()` к полученному списку и извлекает `result()`. Я не стану подробно объяснять остальной код — вероятно, вы сможете разобраться в нем самостоятельно.

- 42.** (5) Создайте два класса, никак не связанных друг с другом. Каждый класс должен содержать значение и по меньшей мере методы для получения и модификации этого значения. Измените пример `Functional.java` так, чтобы он выполнял функциональные операции с коллекциями классов (эти операции не обязаны быть арифметическими, как в `Functional.java`).

Резюме

Мне приходилось объяснять, как работают шаблоны C++, с самого момента их появления, и вероятно, я выдвигал следующий аргумент дольше, чем кто-либо. Лишь недавно я начал задумываться над тем, насколько актуален этот аргумент — так ли часто проблема, которую я собираюсь описать, проникает в программы?

Этот аргумент выглядит так. Одной из самых привлекательных областей для использования механизма обобщенных типов являются классы контейнеров: `List`, `Set`, `Map` и т. д. — те, которые рассматривались в главе 11 и будут более подробно рассматриваться в главе 17. До Java SE5 при помещении объекта в контейнер происходило восходящее преобразование в `Object`, приводящее к потере информации. Когда объект извлекался из контейнера для выполнения каких-либо операций, его приходилось преобразовывать обратно к фактическому типу. В качестве примера я приводил контейнер `List` с элементами `Cat` (вариация на эту тему с объектами `Apple` и `Orange` приводилась в начале главы 11).⁵ До появления обобщенных версий этих контейнеров Java SE5 в контейнер заносился тип `Object` и извлекался тип `Object`, поэтому в контейнер с элементами `Cat` можно было легко поместить объект `Dog`.

Однако язык Java до появления обобщений не допускал некорректного использования объектов, помещенных в контейнер. Если вы заносили в контейнер `Cat` объект `Dog`, а потом пытались интерпретировать все элементы контейнера как `Cat`, при извлечении ссылки на `Dog` и попытки ее преобразования к `Cat` происходило исключение

`RuntimeException`. Проблема все равно обнаруживалась, но это происходило во время выполнения, а не во время компиляции.

В предыдущих изданиях книги я писал:

«Это не просто неприятность, а возможная причина трудноуловимых ошибок. Если одна часть (или несколько частей) программы вставляет объекты в контейнер, а вы в совершенно другой части программы узнаете о помещении в контейнер неправильного объекта при помощи исключения, вам необходимо узнать, где была выполнена некорректная вставка».

Но после некоторых размышлений я начал сомневаться. Во-первых, насколько часто это случается? Я не помню, чтобы в моих программах происходило что-то подобное, а когда я начал опрашивать народ на конференциях, никто так и не сказал, что сталкивался с такой ошибкой. В другой книге в качестве примера приводился список `files`, который содержал объекты `String`, — в этом примере добавление объекта `File` в список `files` выглядит абсолютно естественно, так что объект лучше было назвать `fileNames`. Какую бы проверку типов ни предоставлял компилятор Java, написать невразумительную программу все равно возможно, а плохо написанная, но компилирующаяся программа так и остается плохо написанной. Вероятно, большинство разработчиков выбирает более подходящие имена контейнеров — например, имя `cats` послужит визуальным предупреждением для программиста, который попытается добавить в контейнер объект `Dog`. И даже если это случится, как долго такая ошибка будет оставаться скрытой? Вероятно, стоит вам начать проводить тесты с реальными данными, и исключение возникнет достаточно быстро. Впрочем, один автор утверждал, что такие ошибки «могут оставаться скрытыми годами».

Но я не припоминаю потока отчетов об ошибках от людей, у которых возникали трудности с выявлением ошибок «`Dog` в списке `Cat`» (или хотя бы с их частым появлением). В главе 21 вы увидите, что в многопоточном программировании очень просто допустить ошибку, которая проявляется крайне редко и дает лишь отдаленное представление о причинах происходящего. Так могло ли быть так, что аргумент «`Dog` в списке `Cat`» стал причиной для включения в Java этой очень важной и относительно сложной возможности?

Я думаю, что языковая возможность общего назначения, называемая «обобщенными типами» (не обязательно ее конкретная реализация в Java), была введена по соображениям выразительности, а не только для создания контейнеров, безопасных по отношению к типам. Типовая безопасность контейнеров является побочным эффектом возможности написания более универсального кода.

Итак, хотя аргумент «`Dog` в списке `Cat`» часто используется для «оправдания» обобщений, это утверждение спорно. И как было сказано в начале главы, я не думаю, что концепция обобщений была введена в язык по этой причине. Скорее суть обобщений напрямую следует из их названия — это способ написания более «обобщенного» кода, в меньшей степени ограниченного типами, с которыми он может работать, чтобы один блок кода мог применяться ко многим типам. В этой главе вы уже убедились, что написать обобщенный класс хранения данных (а контейнеры Java относятся к этой категории) относительно несложно, но написание обобщенного кода, работающего со своими обобщенными типами, потребует дополнительных усилий со стороны как

создателя, так и потребителя класса, который должен понимать концепцию и реализацию паттерна проектирования «Адаптер». Дополнительные усилия затрудняют использование этой функции и снижают ее пригодность в тех ситуациях, в которых она могла бы принести пользу.

Также следует учесть, что поскольку обобщения были встроены в Java на поздней стадии (а не проектировались как составная часть языка с самого начала), некоторые контейнеры невозможно сделать настолько надежными, насколько нам хотелось бы. Для примера возьмем контейнер `Map`, а конкретно методы `containsKey(Object key)` и `get(Object key)`. Если бы эти классы проектировались для уже существующих обобщений, эти методы использовали бы параметризованные типы вместо `Object`, с которыми была бы возможна проверка времени компиляции. Например, в картах C++ тип ключа всегда проверяется во время компиляции.

Одно можно сказать совершенно определенно: введение любых механизмов обобщения в более поздних версиях языка, после того как этот язык стал широко использоваться, очень, очень хлопотная затея, которая не может быть реализована безболезненно. В C++ шаблоны входили в исходную ISO-версию языка (причем даже это не обошлось без проблем, поскольку существовала более ранняя версия без шаблонов, вышедшая до появления первого Стандарта C++), так что фактически шаблоны всегда были частью языка. В Java обобщения были введены лишь через 10 лет после выхода первой версии, так что переход на обобщения был сопряжен с изрядными проблемами, которые существенно повлияли на проектирование обобщений. В результате программисты страдают из-за близорукости, проявленной проектировщиками Java при создании версии 1.0. Конечно, при создании первой версии Java проектировщики знали о шаблонах C++ и даже рассматривали возможность их включения в язык, но по каким-то причинам решили отказаться от них (по некоторым признакам они сильно торопились). В результате пострадали как язык, так и использующие его программисты. Только время покажет, как подход Java к реализации обобщений повлиял на язык.

Некоторые языки, прежде всего Nice (см. <http://nice.sourceforge.net>; этот язык генерирует байт-код Java и работает с существующими библиотеками Java) и NextGen (см. <http://japan.cs.rice.edu/nextgen>), содержат более чистую и менее проблемную реализацию параметризованных типов. Теоретически даже можно представить, как такой язык становится наследником Java, поскольку именно так поступил C++ по отношению к C: взять то, что уже существует, и усовершенствовать его.

Дополнительная литература

Вводная информация по обобщениям Java представлена в учебнике Гилада Брача «Generics in the Java Programming Language» (<http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>).

Еще один чрезвычайно полезный ресурс — «Java Generics FAQ» Анжелики Лангер (www.langer.camelot.de/GenericsFAQ/JavaGenericsFAQ.html).

Дополнительную информацию о шаблонах можно найти в статье Торгерсона, Эрнста, Хансена, фон дер Ахе, Брача и Гафтера «Adding Wildcards to the Java Programming Language» (www.jot.fm/issues/issue_2004_12/article5).

16

Массивы

В конце главы 5 вы научились определять и инициализировать массивы.

Как мы обычно представляем себе массивы? Их можно создать и заполнить данными, можно выбрать элементы по целочисленным индексам, и они не изменяют размер. В большинстве случаев ничего больше знать не нужно, но иногда с массивами требуется выполнять более сложные операции, и разработчику приходится выбирать между массивом и более гибким контейнером. В этой главе вы получите более глубокое представление о массивах.

Особое отношение к массивам

Существует немало других способов хранения объектов, что же выделяет массивы из всех прочих?

Есть три аспекта, отличающих массивы от остальных типов контейнеров: эффективность, тип и способность хранить примитивы. Массив в Java — наиболее эффективный способ организации размещения последовательности объектов (на самом деле ссылок на объекты) и произвольного доступа к ним. Массив — это простая линейная последовательность, что делает доступ к ее элементам быстрым, но за такую скорость приходится платить: размер создаваемого массива постоянен и не меняется в процессе его использования. Также существует класс `ArrayList` (см. главу 11), способный автоматически выделять дополнительную память с созданием нового массива и перемещения ссылок из старого массива в новый. Хотя в общем случае класс `ArrayList` предпочтительнее обычных массивов, за эту гибкость приходится расплачиваться, и эффективность работы с классом `ArrayList` заметно ниже, чем у массивов.

И массивы и контейнеры защищены от злоупотреблений. При использовании как массива, так и контейнера при выходе за границу (свидетельствующем об ошибке программиста) возбуждается исключение `RuntimeException`.

До появления обобщенных типов другие классы контейнеров работали с объектами так, словно те не имеют конкретного типа. Иначе говоря, они рассматривали их как

Объекты типа `Object` — корневого для всех классов в Java. Массивы предпочтительнее старых контейнеров, потому что массив создается для хранения конкретного типа. Это означает, что во время компиляции выполняется проверка, предотвращающая вставку недопустимого типа или неправильную интерпретацию извлекаемого типа. Конечно, Java все равно не позволит отправить неподходящее сообщение объекту во время компиляции или выполнения. Таким образом, нельзя сказать, что тот или иной способ безопаснее другого; просто выявление ошибок компилятором более удобно и снижает вероятность того, что конечный пользователь будет удивлен полученным исключением.

В массивах могут храниться примитивы, тогда как в старых контейнерах они храниться не могли. Однако контейнеры, использующие обобщения, могут задавать и проверять тип хранимых объектов, а благодаря автоматической упаковке контейнеры работают так, как если бы в них могли храниться примитивы (необходимые преобразования выполняются автоматически). В следующем примере массивы сравниваются с обобщенными контейнерами:

```
//: arrays/ContainerComparison.java
import java.util.*;
import static net.mindview.util.Print.*;

class BerylliumSphere {
    private static long counter;
    private final long id = counter++;
    public String toString() { return "Sphere " + id; }
}

public class ContainerComparison {
    public static void main(String[] args) {
        BerylliumSphere[] spheres = new BerylliumSphere[10];
        for(int i = 0; i < 5; i++)
            spheres[i] = new BerylliumSphere();
        print(Arrays.toString(spheres));
        print(spheres[4]);

        List<BerylliumSphere> sphereList =
            new ArrayList<BerylliumSphere>();
        for(int i = 0; i < 5; i++)
            sphereList.add(new BerylliumSphere());
        print(sphereList);
        print(sphereList.get(4));

        int[] integers = { 0, 1, 2, 3, 4, 5 };
        print(Arrays.toString(integers));
        print(integers[4]);

        List<Integer> intList = new ArrayList<Integer>(
            Arrays.asList(0, 1, 2, 3, 4, 5));
        intList.add(97);
        print(intList);
        print(intList.get(4));
    }
} /* Output:
```

```
[Sphere 0, Sphere 1, Sphere 2, Sphere 3, Sphere 4, null,
null, null, null, null]
Sphere 4
[Sphere 5, Sphere 6, Sphere 7, Sphere 8, Sphere 9]
Sphere 9
[0, 1, 2, 3, 4, 5]
4
[0, 1, 2, 3, 4, 5, 97]
4
*///:~
```

Оба способа хранения объектов обеспечивают проверку типов. Единственное очевидное отличие состоит в том, что массивы используют для обращения к элементам конструкции `[]`, а `List` — такие методы, как `add()` и `get()`. Сходство между массивами и `ArrayList` не случайно; это было сделано так, чтобы разработчику было концептуально проще переключаться между ними. Но как было показано в главе 11, по функциональности контейнеры заметно превосходят массивы.

С введением автоматической упаковки контейнеры стали почти такими же простыми в использовании с примитивами, как массивы. Единственным оставшимся преимуществом массивов является их эффективность. Однако при решении более общих задач массивы могут устанавливать слишком жесткие ограничения, и в таких случаях можно использовать класс контейнера.

Массивы как полноценные объекты

С каким бы типом массива вы ни работали, его идентификатор — всего лишь ссылка на реальный объект, созданный в куче. Это объект, хранящий ссылки на другие объекты, и он может быть создан неявно, в синтаксисе инициализации массива или же непосредственно выражением `new`. Обязательная часть объекта массива (фактически, поле или метод, к которому вы можете получить доступ) — доступный только для чтения элемент `length`, говорящий, сколько объектов можно хранить в массиве. Запись в форме `[]` — единственный способ обращения к данным массива.

Следующий пример демонстрирует разные способы инициализации массивов и то, как ссылки на массивы связываются с разными объектами массивов. Также демонстрируется, что массив простейших типов и массив объектов практически идентичны в использовании. Единственное отличие состоит в том, что массив объектов содержит ссылки на них, в то время как в массиве примитивов хранятся реальные значения.

```
//: arrays/ArrayOptions.java
// Инициализация и повторное присваивание массивов.
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayOptions {
    public static void main(String[] args) {
        // Массив объектов:
        BerylliumSphere[] a; // Локальная неинициализированная переменная
        BerylliumSphere[] b = new BerylliumSphere[5];
        // Ссылки в массиве автоматически
        // инициализируются null:
        print("b: " + Arrays.toString(b));
```

```
BerylliumSphere[] c = new BerylliumSphere[4];
for(int i = 0; i < c.length; i++)
    if(c[i] == null) // Можно проверить ссылку на null
        c[i] = new BerylliumSphere();
// Групповая инициализация:
BerylliumSphere[] d = { new BerylliumSphere(),
    new BerylliumSphere(), new BerylliumSphere()
};
// Динамическая групповая инициализация:
a = new BerylliumSphere[]{
    new BerylliumSphere(), new BerylliumSphere(),
};
// (Завершающая запятая в обоих случаях не обязательна)
print("a.length = " + a.length);
print("b.length = " + b.length);
print("c.length = " + c.length);
print("d.length = " + d.length);
a = d;
print("a.length = " + a.length);

// Массивы примитивов:
int[] e; // Null-ссылка
int[] f = new int[5];
// Примитивы в массиве автоматически
// инициализируются нулями:
print("f: " + Arrays.toString(f));
int[] g = new int[4];
for(int i = 0; i < g.length; i++)
    g[i] = i*i;
int[] h = { 11, 47, 93 };
// Ошибка компиляции: переменная e не инициализирована:
//!print("e.length = " + e.length);
print("f.length = " + f.length);
print("g.length = " + g.length);
print("h.length = " + h.length);
e = h;
print("e.length = " + e.length);
e = new int[]{ 1, 2 };
print("e.length = " + e.length);
}
} /* Output:
b: [null, null, null, null]
a.length = 2
b.length = 5
c.length = 4
d.length = 3
a.length = 3
f: [0, 0, 0, 0, 0]
f.length = 5
g.length = 4
h.length = 3
e.length = 3
e.length = 2
*///:~
```

Массив `a` изначально является обычной неинициализированной переменной, поэтому компилятор не разрешает выполнять с ней какие-либо действия до того, как она будет

инициализирована. Массив `b` инициализируется в месте определения ссылками на объекты `BerylliumSphere`, хотя реальных объектов `BerylliumSphere` в массив не помещается. Но вы все равно в силах узнать размер массива, так как `b` указывает на «законный» объект. Здесь возникает небольшое неудобство: нельзя выяснить, сколько объектов фактически *находится* в массиве, поскольку `length` сообщает только то, сколько элементов можно *разместить* в массиве; то есть указывает размер объекта массива, а не количество содержащихся в нем элементов. Но известно, что при создании объекта массива его ссылки инициализируются значениями `null`, поэтому легко проверить, имеется ли в определенной ячейке массива объект — надо просто посмотреть, не находится ли там `null`. Аналогично, массивы примитивов заполняются нулями для числовых типов, `(char)0` для `char` и значением «ложь» (`false`) для логического типа `boolean`.

Массив `c` демонстрирует создание объекта массива с последующим присоединением объектов `BerylliumSphere` к каждой ячейке массива. На примере массива `d` показан синтаксис групповой инициализации, которая также дает в результате объект массива (неявно вызывается оператор `new`, как и для массива `c`), в котором все ячейки инициализированы объектами `BerylliumSphere` одной командой.

Следующий за этим процесс инициализации массива можно назвать динамической групповой инициализацией. Примененная к массиву `d`, она должна проводиться в точке определения `d`, но при использовании второй формы записи можно создать и инициализировать объект массива в произвольном месте. Например, предположим, что имеется метод `hide()`, берущий в качестве аргумента массив объектов `BerylliumSphere`. Его можно вызвать следующим образом:

```
hide(d);
```

но также существует возможность динамически создать новый массив, который затем передается в качестве аргумента:

```
hide(new BerylliumSphere[] { new BerylliumSphere(), new BerylliumSphere() });
```

В большинстве ситуаций эта новая форма записи более удобна для написания кода.

Выражение

```
a = d;
```

показывает, как можно взять ссылку, присоединенную к одному объекту массива, и присвоить ее другому объекту массива; так же, как и в случае с любым другим типом объекта. Теперь как `a`, так и `d` указывают на один и тот же массив в общей куче.

Вторая часть программы `ArrayOptions.java` показывает, что работа с массивами примитивов практически ничем не отличается от работы с массивами объектов, *за одним исключением*: массивы примитивов хранят реальные значения.

1. (2) Создайте метод, получающий в аргументе массив объектов `BerylliumSphere`. Вызовите метод с динамическим созданием аргумента. Продемонстрируйте, что обычная групповая инициализация массива в этом случае не работает. Найдите ситуации, в которых обычная групповая инициализация массива работает, а динамическая групповая инициализация оказывается излишней.

Возврат массива

Предположим, что вы пишете метод и не хотите возвращать из него что-то одно, а собираетесь вернуть целое множество. В языках типа С и С++ осуществить подобное сложно, так как в них нельзя отдавать назад массив, разрешается возвращать лишь ссылку на него. Это сразу ведет к затруднениям, поскольку становится сложно определить жизненный цикл такого массива, что легко может стать причиной «утечки памяти».

В Java используется похожий подход, но вы просто «возвращаете массив». В Java вам не приходится беспокоиться о жизненном цикле массива — он будет доступен столько, сколько вам понадобится, а когда станет не нужен, об его удалении позаботится уборщик мусора.

В качестве примера рассмотрим возврат массива строк (String):

```
//: arrays/IceCream.java
// Возвращение массивов из методов.
import java.util.*;

public class IceCream {
    private static Random rand = new Random(47);
    static final String[] FLAVORS = {
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
        "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };
    public static String[] flavorSet(int n) {
        if(n > FLAVORS.length)
            throw new IllegalArgumentException("Set too big");
        String[] results = new String[n];
        boolean[] picked = new boolean[FLAVORS.length];
        for(int i = 0; i < n; i++) {
            int t;
            do
                t = rand.nextInt(FLAVORS.length);
            while(picked[t]);
            results[i] = FLAVORS[t];
            picked[t] = true;
        }
        return results;
    }
    public static void main(String[] args) {
        for(int i = 0; i < 7; i++)
            System.out.println(Arrays.toString(flavorSet(3)));
    }
} /* Output:
[Rum Raisin, Mint Chip, Mocha Almond Fudge]
[Chocolate, Strawberry, Mocha Almond Fudge]
[Strawberry, Mint Chip, Mocha Almond Fudge]
[Rum Raisin, Vanilla Fudge Swirl, Mud Pie]
[Vanilla Fudge Swirl, Chocolate, Mocha Almond Fudge]
[Praline Cream, Strawberry, Mocha Almond Fudge]
[Mocha Almond Fudge, Strawberry, Mint Chip]
*///:~
```

Метод `flavorSet()` создает массив строк `String` с именем `results`. Размер массива устанавливается аргументом метода `n`. Далее метод случайным образом выбирает из массива

FLAVORS сорта мороженого и помещает их названия в массив `results`, который и возвращает. Возврат массива сходен с возвратом любого другого объекта — это ссылка. Неважно, был ли массив создан внутри метода `flavorSet()` или в любом другом месте. Уборщик мусора возьмет на себя удаление массива после того, как вы завершите с ним работу, а до этого массив может существовать сколь угодно долго, пока в нем есть надобность.

Кстати, обратите внимание на уже сказанное, метод `flavorSet()` выбирает названия мороженого случайным образом и при этом проверяет, не повторилось ли это название. Все это делается в цикле `do`, в котором вероятностный выбор проводится до тех пор, пока не будет найдено название, еще не помеченное как использованное в массиве `picked`. (Конечно, можно было проводить сравнения строк, проверяя, есть ли уже соответствие в массиве `results`.) Если такое находится, в массив добавляется новый элемент, а затем поиск продолжается (параметр цикла `i` увеличивается на единицу).

Из выходных данных `flavorSet()` видно, что названия мороженого выбираются случайным образом.

2. (1) Напишите метод, который получает аргумент `int` и возвращает массив указанного размера, заполненный объектами `BerylliumSphere`.

Многомерные массивы

Java позволяет легко создавать многомерные массивы. Для многомерного массива примитивов каждый вектор заключается в фигурные скобки:

```
//: arrays/MultidimensionalPrimitiveArray.java
// Создание многомерных массивов.
import java.util.*;

public class MultidimensionalPrimitiveArray {
    public static void main(String[] args) {
        int[][] a = {
            { 1, 2, 3, },
            { 4, 5, 6, },
        };
        System.out.println(Arrays.deepToString(a));
    }
} /* Output:
[[1, 2, 3], [4, 5, 6]]
*///:~
```

Каждая вложенная пара фигурных скобок осуществляет переход на следующий уровень массива.

В этом примере используется метод `Java SE5 Arrays.deepToString()`, который преобразует многомерные массивы в `String` (результат его работы виден из выходных данных).

Также для создания массива можно воспользоваться ключевым словом `new`. Пример создания трехмерного массива в выражении `new`:

```
//: arrays/ThreeDWithNew.java
import java.util.*;
```

```

public class ThreeDWithNew {
    public static void main(String[] args) {
        // 3-D array with fixed length:
        int[][][] a = new int[2][2][4];
        System.out.println(Arrays.deepToString(a));
    }
} /* Output:
[[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0,
0]]]
*///:~

```

Как видите, если явное значение инициализации не указано, примитивные значения в массивах инициализируются автоматически. Массивы объектов инициализируются значениями `null`.

Векторы массивов, образующих матрицу, могут иметь произвольную длину (так называемые *ступенчатые массивы*):

```

//: arrays/RaggedArray.java
import java.util.*;

public class RaggedArray {
    public static void main(String[] args) {
        Random rand = new Random(47);
        // Трехмерный массив с векторами переменной длины:
        int[][][] a = new int[rand.nextInt(7)][][];
        for(int i = 0; i < a.length; i++) {
            a[i] = new int[rand.nextInt(5)][];
            for(int j = 0; j < a[i].length; j++)
                a[i][j] = new int[rand.nextInt(5)];
        }
        System.out.println(Arrays.deepToString(a));
    }
} /* Output:
[[], [[0], [0], [0, 0, 0, 0]], [[], [0, 0], [0, 0]], [[0,
0, 0], [0], [0, 0, 0, 0]], [[0, 0, 0], [0, 0, 0], [0], []],
[[0], [], [0]]]
*///:~

```

Первая конструкция `new` создает массив, у которого первый элемент имеет случайную длину, а остальные не определены. Вторая конструкция `new` в цикле `for` заполняет элементы, но оставляет третий индекс неопределенным до третьего `new`.

Аналогичным образом можно поступать с массивами непримитивных объектов. Следующий пример показывает, как объединить несколько выражений `new` в фигурных скобках:

```

//: arrays/MultidimensionalObjectArrays.java
import java.util.*;

public class MultidimensionalObjectArrays {
    public static void main(String[] args) {
        BerylliumSphere[][] spheres = {
            { new BerylliumSphere(), new BerylliumSphere() },
            { new BerylliumSphere(), new BerylliumSphere(),
              new BerylliumSphere(), new BerylliumSphere() },
            { new BerylliumSphere(), new BerylliumSphere(),

```

продолжение ➤

```

        new BerylliumSphere(), new BerylliumSphere(),
        new BerylliumSphere(), new BerylliumSphere(),
        new BerylliumSphere(), new BerylliumSphere() },
    };
    System.out.println(Arrays.deepToString(spheres));
}
} /* Output:
[[Sphere 0, Sphere 1], [Sphere 2, Sphere 3, Sphere 4,
Sphere 5], [Sphere 6, Sphere 7, Sphere 8, Sphere 9, Sphere
10, Sphere 11, Sphere 12, Sphere 13]]
*///:~

```

Как видите, `spheres` — еще один ступенчатый массив с разными длинами списков объектов.

Автоматическая упаковка также работает с инициализаторами массивов:

```

//: arrays/AutoBoxingArrays.java
import java.util.*;

public class AutoBoxingArrays {
    public static void main(String[] args) {
        Integer[][] a = { // AutoBoxing:
            { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 },
            { 21, 22, 23, 24, 25, 26, 27, 28, 29, 30 },
            { 51, 52, 53, 54, 55, 56, 57, 58, 59, 60 },
            { 71, 72, 73, 74, 75, 76, 77, 78, 79, 80 },
        };
        System.out.println(Arrays.deepToString(a));
    }
} /* Output:
[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [21, 22, 23, 24, 25, 26,
27, 28, 29, 30], [51, 52, 53, 54, 55, 56, 57, 58, 59, 60],
[71, 72, 73, 74, 75, 76, 77, 78, 79, 80]]
*///:~

```

А вот как происходит последовательное построение непримитивных объектов:

```

//: arrays/AssemblingMultidimensionalArrays.java
// Создание многомерных массивов.
import java.util.*;

public class AssemblingMultidimensionalArrays {
    public static void main(String[] args) {
        Integer[][] a;
        a = new Integer[3][];
        for(int i = 0; i < a.length; i++) {
            a[i] = new Integer[3];
            for(int j = 0; j < a[i].length; j++)
                a[i][j] = i * j; // Автоматическая упаковка
        }
        System.out.println(Arrays.deepToString(a));
    }
} /* Output:
[[0, 0, 0], [0, 1, 2], [0, 2, 4]]
*///:~

```

Выражение `i*j` используется только для помещения нетривиального значения в `Integer`.

Метод `Arrays.deepToString()` работает как с массивами примитивов, так и с массивами объектов:

```
//: arrays/MultiDimWrapperArray.java
// Многомерные массивы объектов-обертки.
import java.util.*;

public class MultiDimWrapperArray {
    public static void main(String[] args) {
        Integer[][] a1 = { // Автоматическая упаковка
            { 1, 2, 3, },
            { 4, 5, 6, },
        };
        Double[][][] a2 = { // Автоматическая упаковка
            { { 1.1, 2.2 }, { 3.3, 4.4 } },
            { { 5.5, 6.6 }, { 7.7, 8.8 } },
            { { 9.9, 1.2 }, { 2.3, 3.4 } },
        };
        String[][] a3 = {
            { "The", "Quick", "Sly", "Fox" },
            { "Jumped", "Over" },
            { "The", "Lazy", "Brown", "Dog", "and", "friend" },
        };
        System.out.println("a1: " + Arrays.deepToString(a1));
        System.out.println("a2: " + Arrays.deepToString(a2));
        System.out.println("a3: " + Arrays.deepToString(a3));
    }
}

/* Output:
a1: [[1, 2, 3], [4, 5, 6]]
a2: [[[1.1, 2.2], [3.3, 4.4]], [[5.5, 6.6], [7.7, 8.8]],
[[9.9, 1.2], [2.3, 3.4]]]
a3: [[The, Quick, Sly, Fox], [Jumped, Over], [The, Lazy,
Brown, Dog, and, friend]]
*///:~
```

И снова для массивов `Integer` и `Double` автоматическая упаковка Java SE5 создает объекты-«обертки» за вас.

3. (4) Напишите метод, который создает и инициализирует двумерный массив `double`. Размер массива определяется аргументами метода, а значения инициализации лежат в диапазоне, определяемом начальным и конечным индексами, также передаваемыми в аргументах метода. Создайте второй метод для вывода массива, сгенерированного первым методом. Протестируйте методы в `main()` — создайте и выведите массивы нескольких разных размеров.
4. (2) Повторите предыдущее управление для трехмерного массива.
5. (1) Продемонстрируйте, что многомерные массивы непримитивных типов автоматически инициализируются `null`.
6. (1) Напишите метод, который получает два аргумента `int`, определяющие два размера двумерного массива. Метод должен создавать и заполнять двумерный массив объектов `BerylliumSphere`, соответствующий переданным аргументам.
7. (1) Повторите предыдущее упражнение для трехмерного массива.

Массивы и обобщения

В общем случае массивы и обобщения плохо сочетаются друг с другом. Например, вы не сможете создать массивы параметризованных типов:

```
Peel<Banana>[] peels = new Peel<Banana>[10]; // Недопустимо
```

Стирание уничтожает информацию параметра-типа, а для обеспечения безопасности типа массив должен знать точный тип хранящихся в нем объектов.

Однако сам тип массива может быть параметризован:

```
//: arrays/ParameterizedArrayType.java

class ClassParameter<T> {
    public T[] f(T[] arg) { return arg; }
}

class MethodParameter {
    public static <T> T[] f(T[] arg) { return arg; }
}

public class ParameterizedArrayType {
    public static void main(String[] args) {
        Integer[] ints = { 1, 2, 3, 4, 5 };
        Double[] doubles = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        Integer[] ints2 =
            new ClassParameter<Integer>().f(ints);
        Double[] doubles2 =
            new ClassParameter<Double>().f(doubles);
        ints2 = MethodParameter.f(ints);
        doubles2 = MethodParameter.f(doubles);
    }
} ///:~
```

Обратите внимание на удобство использования параметризованного метода вместо параметризованного класса: вам не нужно создавать экземпляр класса с параметром для всех типов, к которым он применяется, и его можно сделать статическим. Конечно, параметризованный метод не всегда должен выбираться вместо параметризованного класса, но часто он оказывается предпочтительным.

Оказывается, утверждение о том, что создать массивы обобщенных типов невозможно, не совсем точно. Правда, компилятор не позволяет создать *экземпляр* массива обобщенного типа, однако он позволяет создать ссылку на такой массив, например:

```
List<String>[] ls;
```

Эта строка проходит компиляцию без жалоб. И хотя вы не сможете создать объект массива, содержащего обобщения, можно создать массив необобщенного типа и выполнить преобразование типа:

```
//: arrays/ArrayOfGenerics.java
// Возможность создания массива обобщений.
import java.util.*;
```

```

public class ArrayOfGenerics {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        List<String>[] ls;
        List[] la = new List[10];
        ls = (List<String>[])la; // "Неконтролируемое" предупреждение
        ls[0] = new ArrayList<String>();
        // Проверка во время компиляции приводит к ошибке:
        //! ls[1] = new ArrayList<Integer>();

        // Проблема: List<String> является подтипом Object
        Object[] objects = ls; // Значит, присваивание допустимо
        // Компилируется и выполняется без ошибок:
        objects[1] = new ArrayList<Integer>();

        // Однако при простейших потребностях можно создать массив
        // обобщений, хотя и с "неконтролируемым" предупреждением:
        List<BerylliumSphere>[] spheres =
            (List<BerylliumSphere>[])new List[10];
        for(int i = 0; i < spheres.length; i++)
            spheres[i] = new ArrayList<BerylliumSphere>();
    }
} ///:~

```

Как видите, после создания ссылки на `List<String>[]` обеспечивается частичная проверка времени компиляции. Проблема в том, что массивы ковариантны, поэтому `List<String>[]` также является `Object[]`, и вы можете использовать это обстоятельство для присваивания `ArrayList<Integer>` массиву без ошибок во время компиляции или выполнения.

Если вы знаете, что не собираетесь выполнять восходящее преобразование, а ваши потребности относительно просты, существует возможность создания массива обобщений, обеспечивающего базовую проверку типов во время компиляции. Однако обобщенный контейнер почти всегда будет предпочтительнее массива обобщений.

В общем случае, обобщения эффективны на границах класса или метода. Во внутренней реализации обобщения обычно становятся бесполезными из-за стирания. Например, вы не можете создать массив обобщенного типа:

```

//: arrays/ArrayOfGenericType.java
// Массивы обобщенных типов не компилируются.

```

```

public class ArrayOfGenericType<T> {
    T[] array; // OK
    @SuppressWarnings("unchecked")
    public ArrayOfGenericType(int size) {
        //! array = new T[size]; // Illegal
        array = (T[])new Object[size]; // "Неконтролируемое" предупреждение
    }
    // Недопустимо:
    //! public <U> U[] makeArray() { return new U[10]; }
} ///:~

```

Стирание снова встает на пути — этот пример пытается создать массивы типов, которые были стерты, а следовательно, стали неизвестными. Обратите внимание: вы можете создать массив `Object` и преобразовать его, но без аннотации `@SuppressWarnings` во время

компиляции будет выдано предупреждение, потому что массив в действительности не содержит тип `T` и не осуществляет динамической проверки. Другими словами, если создать массив `String[]`, Java как во время компиляции, так и во время выполнения будет следить за тем, чтобы в массив помещались только объекты `String`. Но если создать массив `Object[]`, в него можно будет поместить что угодно, кроме примитивных типов.

8. (1) Продемонстрируйте истинность утверждений из предыдущего абзаца.
9. (3) Создайте классы, необходимые для примера `Peel<Banana>`, и покажите, что компилятор их не принимает. Исправьте ошибку при помощи `ArrayList`.
10. (2) Измените пример `ArrayOfGenerics.java`, чтобы вместо массивов в нем использовались контейнеры. Покажите, что предупреждения компилятора можно устранить.

Создание тестовых данных

При экспериментах с массивами и при программировании часто требуется заполнить массив тестовыми данными. Примеры, представленные в этом разделе, заполняют массив значениями или объектами.

`Arrays.fill()`

Класс `Arrays` из стандартной библиотеки Java содержит довольно незамысловатый метод `fill()` — одно значение просто копируется во все ячейки массива, или, в случае с объектами, копируется одна и та же ссылка. Пример:

```
//: arrays/FillingArrays.java
// Использование метода Arrays.fill()
import java.util.*;
import static net.mindview.util.Print.*;

public class FillingArrays {
    public static void main(String[] args) {
        int size = 6;
        boolean[] a1 = new boolean[size];
        byte[] a2 = new byte[size];
        char[] a3 = new char[size];
        short[] a4 = new short[size];
        int[] a5 = new int[size];
        long[] a6 = new long[size];
        float[] a7 = new float[size];
        double[] a8 = new double[size];
        String[] a9 = new String[size];
        Arrays.fill(a1, true);
        print("a1 = " + Arrays.toString(a1));
        Arrays.fill(a2, (byte)11);
        print("a2 = " + Arrays.toString(a2));
        Arrays.fill(a3, 'x');
        print("a3 = " + Arrays.toString(a3));
        Arrays.fill(a4, (short)17);
        print("a4 = " + Arrays.toString(a4));
        Arrays.fill(a5, 19);
```



```
print("a5 = " + Arrays.toString(a5));
Arrays.fill(a6, 23);
print("a6 = " + Arrays.toString(a6));
Arrays.fill(a7, 29);
print("a7 = " + Arrays.toString(a7));
Arrays.fill(a8, 47);
print("a8 = " + Arrays.toString(a8));
Arrays.fill(a9, "Hello");
print("a9 = " + Arrays.toString(a9));
// Операции с диапазонами:
Arrays.fill(a9, 3, 5, "World");
print("a9 = " + Arrays.toString(a9));
}
} /* Output:
a1 = [true, true, true, true, true, true]
a2 = [11, 11, 11, 11, 11, 11]
a3 = [x, x, x, x, x, x]
a4 = [17, 17, 17, 17, 17, 17]
a5 = [19, 19, 19, 19, 19, 19]
a6 = [23, 23, 23, 23, 23, 23]
a7 = [29.0, 29.0, 29.0, 29.0, 29.0, 29.0]
a8 = [47.0, 47.0, 47.0, 47.0, 47.0, 47.0]
a9 = [Hello, Hello, Hello, Hello, Hello, Hello]
a9 = [Hello, Hello, Hello, World, World, Hello]
*///:~
```

Заполнять можно как целый массив, так и диапазон элементов (как в двух последних командах). Но поскольку `Arrays.fill()` может вызываться только для одного значения данных, такая возможность не особенно полезна.

Генераторы данных

Для создания более интересных массивов данных более гибкими средствами мы воспользуемся концепцией генератора, представленной в главе 15. Программа, использующая интерфейс `Generator`, может создавать любые данные, выбирая соответствующий генератор (пример паттерна проектирования «Стратегия» — разные генераторы представляют разные стратегии¹).

В этом разделе будут представлены некоторые разновидности генераторов, и как вы уже видели, вы можете легко определять собственные генераторы.

Начнем с базового набора генераторов-счетчиков для всех «оберток» примитивных типов и `String`. Классы генераторов вложены в класс `CountingGenerator`, чтобы они могли использовать имя типа, совпадающее с именем генерируемого типа; например, генератор, создающий объекты `Integer`, будет создаваться выражением `new CountingGenerator.Integer()`:

```
//: net/mindview/util/CountingGenerator.java
// Простые реализации генераторов.
package net.mindview.util;
```

продолжение ➤

¹ Здесь возникает некоторая неоднозначность: также можно утверждать, что генератор представляет паттерн «Команда». Однако я считаю, что задачей является заполнение массива, а генератор решает часть этой задачи, поэтому к «Стратегии» он ближе, чем к «Команде».

```

public class CountingGenerator {
    public static class
    Boolean implements Generator<java.lang.Boolean> {
        private boolean value = false;
        public java.lang.Boolean next() {
            value = !value; // Простое переключение
            return value;
        }
    }
    public static class
    Byte implements Generator<java.lang.Byte> {
        private byte value = 0;
        public java.lang.Byte next() { return value++; }
    }
    static char[] chars = ("abcdefghijklmnopqrstuvwxyz" +
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ").toCharArray();
    public static class
    Character implements Generator<java.lang.Character> {
        int index = -1;
        public java.lang.Character next() {
            index = (index + 1) % chars.length;
            return chars[index];
        }
    }
    public static class
    String implements Generator<java.lang.String> {
        private int length = 7;
        Generator<java.lang.Character> cg = new Character();
        public String() {}
        public String(int length) { this.length = length; }
        public java.lang.String next() {
            char[] buf = new char[length];
            for(int i = 0; i < length; i++)
                buf[i] = cg.next();
            return new java.lang.String(buf);
        }
    }
    public static class
    Short implements Generator<java.lang.Short> {
        private short value = 0;
        public java.lang.Short next() { return value++; }
    }
    public static class
    Integer implements Generator<java.lang.Integer> {
        private int value = 0;
        public java.lang.Integer next() { return value++; }
    }
    public static class
    Long implements Generator<java.lang.Long> {
        private long value = 0;
        public java.lang.Long next() { return value++; }
    }
    public static class
    Float implements Generator<java.lang.Float> {
        private float value = 0;
        public java.lang.Float next() {
            float result = value;
            value += 1.0;
        }
    }
}

```

```

        return result;
    }
}
public static class
Double implements Generator<java.lang.Double> {
    private double value = 0.0;
    public java.lang.Double next() {
        double result = value;
        value += 1.0;
        return result;
    }
}
} ///:~

```

Каждый класс реализует некоторое понятие «счетчика». В случае `CountingGenerator.Character` это простое повторение букв верхнего и нижнего регистра. Класс `CountingGenerator.String` использует `CountingGenerator.Character` для заполнения массива символов, который затем преобразуется в `String`. Размер массива определяется аргументом конструктора. Обратите внимание: `CountingGenerator.String` использует базовую версию `Generator<java.lang.Character>` вместо конкретной ссылки на `CountingGenerator.Character`. Позднее этот генератор может быть заменен для создания `RandomGenerator.String` в программе `RandomGenerator.java`.

Следующая тестовая программа использует отражение с идиомой вложенного генератора, что позволяет использовать ее для тестирования любого набора генераторов, построенных по определенной форме:

```

//: arrays/GeneratorsTest.java
import net.mindview.util.*;

public class GeneratorsTest {
    public static int size = 10;
    public static void test(Class<?> surroundingClass) {
        for(Class<?> type : surroundingClass.getClasses()) {
            System.out.print(type.getSimpleName() + ": ");
            try {
                Generator<?> g = (Generator<?>)type.newInstance();
                for(int i = 0; i < size; i++)
                    System.out.printf(g.next() + " ");
                System.out.println();
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
    public static void main(String[] args) {
        test(CountingGenerator.class);
    }
} /* Output:
Double: 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
Float: 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
Long: 0 1 2 3 4 5 6 7 8 9
Integer: 0 1 2 3 4 5 6 7 8 9
Short: 0 1 2 3 4 5 6 7 8 9
String: abcdefg hijklmn opqrstu vwxyzAB CDEFGHI JKLMNOP
QRSTUvw XYZabcd efghijk lmnopqr

```

продолжение ➤

```

Character: a b c d e f g h i j
Byte: 0 1 2 3 4 5 6 7 8 9
Boolean: true false true false true false true false true
false
*///:~

```

Предполагается, что тестируемый класс содержит набор вложенных объектов `Generator`, каждый из которых имеет конструктор по умолчанию (без аргументов). Метод отражения `getClasses()` выдает информацию обо всех вложенных классах. Метод `test()` создает экземпляр каждого из этих генераторов и выводит результат, полученный при десятикратном вызове `next()`.

Следующий набор `Generator` использует генератор случайных чисел. Так как конструктор `Random` инициализируется константой, результат повторяется при каждом выполнении программы с использованием одного из этих генераторов:

```

//: net/mindview/util/RandomGenerator.java
// Генераторы, выдающие случайные значения.
package net.mindview.util;
import java.util.*;

public class RandomGenerator {
    private static Random r = new Random(47);
    public static class
    Boolean implements Generator<java.lang.Boolean> {
        public java.lang.Boolean next() {
            return r.nextBoolean();
        }
    }
    public static class
    Byte implements Generator<java.lang.Byte> {
        public java.lang.Byte next() {
            return (byte)r.nextInt();
        }
    }
    public static class
    Character implements Generator<java.lang.Character> {
        public java.lang.Character next() {
            return CountingGenerator.chars[
                r.nextInt(CountingGenerator.chars.length)];
        }
    }
    public static class
    String extends CountingGenerator.String {
        // Подключение генератора случайных символов:
        { cg = new Character(); } // Инициализация экземпляра
        public String() {}
        public String(int length) { super(length); }
    }
    public static class
    Short implements Generator<java.lang.Short> {
        public java.lang.Short next() {
            return (short)r.nextInt();
        }
    }
    public static class
    Integer implements Generator<java.lang.Integer> {

```

```

    private int mod = 10000;
    public Integer() {}
    public Integer(int modulo) { mod = modulo; }
    public java.lang.Integer next() {
        return r.nextInt(mod);
    }
}
public static class
Long implements Generator<java.lang.Long> {
    private int mod = 10000;
    public Long() {}
    public Long(int modulo) { mod = modulo; }
    public java.lang.Long next() {
        return new java.lang.Long(r.nextInt(mod));
    }
}
public static class
Float implements Generator<java.lang.Float> {
    public java.lang.Float next() {
        // Усечение дробной части до двух цифр:
        int trimmed = Math.round(r.nextFloat() * 100);
        return ((float)trimmed) / 100;
    }
}
public static class
Double implements Generator<java.lang.Double> {
    public java.lang.Double next() {
        long trimmed = Math.round(r.nextDouble() * 100);
        return ((double)trimmed) / 100;
    }
}
} ///:~

```

`RandomGenerator.String` наследует от `CountingGenerator.String` и просто подключает новый генератор `Character`.

Чтобы генерируемые числа не были слишком большими, `RandomGenerator.Integer` по умолчанию использует коэффициент 10 000, но перегруженный конструктор позволяет выбрать меньшее значение. Похожий подход используется для `RandomGenerator.Long`. Для генераторов `Float` и `Double` дробная часть усекается.

Мы можем повторно использовать класс `GeneratorsTest` для тестирования `RandomGenerator`:

```

//: arrays/RandomGeneratorsTest.java
import net.mindview.util.*;

public class RandomGeneratorsTest {
    public static void main(String[] args) {
        GeneratorsTest.test(RandomGenerator.class);
    }
} /* Output:
Double: 0.73 0.53 0.16 0.19 0.52 0.27 0.26 0.05 0.8 0.76
Float: 0.53 0.16 0.53 0.4 0.49 0.25 0.8 0.11 0.02 0.8
Long: 7674 8804 8950 7826 4322 896 8033 2984 2344 5810
Integer: 8303 3141 7138 6012 9966 8689 7185 6992 5746 3976
Short: 3358 20592 284 26791 12834 -8092 13656 29324 -1423
5327

```

продолжение ➤

```
String: bkInaMe sbtWHkj UrUkZPg wsqPzDy CyRFJQA HxxHvHq
XumcXZJ oogoYWM NvqeuTp nXsgqia
Character: x x E A J J m z M s
Byte: -60 -17 55 -14 -5 115 39 -37 79 115
Boolean: false true false false true true true true
true
*///:~
```

Чтобы изменить количество генерируемых значений, измените открытое значение `GeneratorsTest.size`.

Применение генераторов для создания массивов

Чтобы взять `Generator` и создать массив, нам понадобятся два преобразователя. Первый использует произвольный генератор для создания массива подтипов `Object`. Для решения проблемы с примитивами второй получает произвольный массив «оберток» примитивных типов и создает соответствующий массив примитивов.

Первый преобразователь, представленный перегруженным статическим методом `array()`, обладает двумя методами. Первая версия метода получает существующий массив и заполняет его с использованием генератора, а вторая версия получает объект `Class`, `Generator` и нужное количество элементов, после чего создает новый массив, снова заполняя его при помощи генератора. Учтите, что программа создает только массивы подтипов `Object` и не может создавать массивы примитивов:

```
//: net/mindview/util/Generated.java
package net.mindview.util;
import java.util.*;

public class Generated {
    // Заполнение существующего массива:
    public static <T> T[] array(T[] a, Generator<T> gen) {
        return new CollectionData<T>(gen, a.length).toArray(a);
    }
    // Создание нового массива:
    @SuppressWarnings("unchecked")
    public static <T> T[] array(Class<T> type,
        Generator<T> gen, int size) {
        T[] a =
            (T[])java.lang.reflect.Array.newInstance(type, size);
        return new CollectionData<T>(gen, size).toArray(a);
    }
} ///:~
```

Класс `CollectionData` будет определен в главе 17. Он создает объект `Collection`, заполненный элементами, созданными генератором `Generator gen`. Количество элементов определяется вторым аргументом конструктора. Все подтипы `Collection` должны содержать метод `toArray()`, который заполняет аргумент-массив элементами `Collection`.

Второй метод использует отражение для динамического создания нового массива подходящего типа и размера. Затем он заполняется таким же способом, как и в первом методе.

Для тестирования `Generated` можно воспользоваться одним из классов `CountingGenerator`, определенных в предыдущем разделе:

```
//: arrays/TestGenerated.java
import java.util.*;
import net.mindview.util.*;

public class TestGenerated {
    public static void main(String[] args) {
        Integer[] a = { 9, 8, 7, 6 };
        System.out.println(Arrays.toString(a));
        a = Generated.array(a, new CountingGenerator.Integer());
        System.out.println(Arrays.toString(a));
        Integer[] b = Generated.array(Integer.class,
            new CountingGenerator.Integer(), 15);
        System.out.println(Arrays.toString(b));
    }
} /* Output:
[9, 8, 7, 6]
[0, 1, 2, 3]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
*///:~
```

Хотя массив `a` инициализирован, эти значения заменяются при передаче через метод `Generated.array()`, который оставляет исходный массив на месте. Инициализация `b` показывает, как создать заполненный массив «с нуля».

Обобщения не работают с примитивами, а мы хотим использовать генераторы для заполнения примитивных массивов. Для решения проблемы мы создадим преобразователь, который получает произвольный массив объектов-«оберток» и преобразует его в массив соответствующих примитивных типов. Без этого нам пришлось бы создавать специализированные генераторы для всех примитивов.

```
//: net/mindview/util/ConvertTo.java
package net.mindview.util;

public class ConvertTo {
    public static boolean[] primitive(Boolean[] in) {
        boolean[] result = new boolean[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i]; // Автоматическая распаковка
        return result;
    }
    public static char[] primitive(Character[] in) {
        char[] result = new char[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i];
        return result;
    }
    public static byte[] primitive(Byte[] in) {
        byte[] result = new byte[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i];
        return result;
    }
    public static short[] primitive(Short[] in) {
        short[] result = new short[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i];
        return result;
    }
}
```

продолжение ➤

```

    for(int i = 0; i < in.length; i++)
        result[i] = in[i];
    return result;
}
public static int[] primitive(Integer[] in) {
    int[] result = new int[in.length];
    for(int i = 0; i < in.length; i++)
        result[i] = in[i];
    return result;
}
public static long[] primitive(Long[] in) {
    long[] result = new long[in.length];
    for(int i = 0; i < in.length; i++)
        result[i] = in[i];
    return result;
}
public static float[] primitive(Float[] in) {
    float[] result = new float[in.length];
    for(int i = 0; i < in.length; i++)
        result[i] = in[i];
    return result;
}
public static double[] primitive(Double[] in) {
    double[] result = new double[in.length];
    for(int i = 0; i < in.length; i++)
        result[i] = in[i];
    return result;
}
} ///:~

```

Каждая версия `primitive()` создает соответствующий массив примитивов нужной длины, а затем копирует элементы из массива типов-«оберток». Обратите внимание на автоматическую распаковку в выражении:

```
result[i] = in[i];
```

Следующий пример показывает, как использовать `ConvertTo` для обеих версий `Generated.array()`:

```

//: arrays/PrimitiveConversionDemonstration.java
import java.util.*;
import net.mindview.util.*;

public class PrimitiveConversionDemonstration {
    public static void main(String[] args) {
        Integer[] a = Generated.array(Integer.class,
            new CountingGenerator.Integer(), 15);
        int[] b = ConvertTo.primitive(a);
        System.out.println(Arrays.toString(b));
        boolean[] c = ConvertTo.primitive(
            Generated.array(Boolean.class,
                new CountingGenerator.Boolean(), 7));
        System.out.println(Arrays.toString(c));
    }
} /* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
[true, false, true, false, true, false, true]
*///:~

```


Наконец, следующая программа тестирует средства генерирования массивов с использованием классов `RandomGenerator`:

```
/// arrays/TestArrayGeneration.java
// Тестирование вспомогательных средств заполнения
// массивов с использованием генераторов.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class TestArrayGeneration {
    public static void main(String[] args) {
        int size = 6;
        boolean[] a1 = ConvertTo.primitive(Generated.array(
            Boolean.class, new RandomGenerator.Boolean(), size));
        print("a1 = " + Arrays.toString(a1));
        byte[] a2 = ConvertTo.primitive(Generated.array(
            Byte.class, new RandomGenerator.Byte(), size));
        print("a2 = " + Arrays.toString(a2));
        char[] a3 = ConvertTo.primitive(Generated.array(
            Character.class,
            new RandomGenerator.Character(), size));
        print("a3 = " + Arrays.toString(a3));
        short[] a4 = ConvertTo.primitive(Generated.array(
            Short.class, new RandomGenerator.Short(), size));
        print("a4 = " + Arrays.toString(a4));
        int[] a5 = ConvertTo.primitive(Generated.array(
            Integer.class, new RandomGenerator.Integer(), size));
        print("a5 = " + Arrays.toString(a5));
        long[] a6 = ConvertTo.primitive(Generated.array(
            Long.class, new RandomGenerator.Long(), size));
        print("a6 = " + Arrays.toString(a6));
        float[] a7 = ConvertTo.primitive(Generated.array(
            Float.class, new RandomGenerator.Float(), size));
        print("a7 = " + Arrays.toString(a7));
        double[] a8 = ConvertTo.primitive(Generated.array(
            Double.class, new RandomGenerator.Double(), size));
        print("a8 = " + Arrays.toString(a8));
    }
}

/* Output:
a1 = [true, false, true, false, false, true]
a2 = [104, -79, -76, 126, 33, -64]
a3 = [Z, n, T, c, Q, r]
a4 = [-13408, 22612, 15401, 15161, -28466, -12603]
a5 = [7704, 7383, 7706, 575, 8410, 6342]
a6 = [7674, 8804, 8950, 7826, 4322, 896]
a7 = [0.01, 0.2, 0.4, 0.79, 0.27, 0.45]
a8 = [0.16, 0.87, 0.7, 0.66, 0.87, 0.59]
*///:~
```

Из результатов также видно, что все версии `ConvertTo.primitive()` работают правильно.

11. (2) Покажите, что автоматическая упаковка не работает с массивами.
12. (1) Создайте инициализированный массив `double` с использованием `CountingGenerator`. Выведите результаты.
13. (2) Заполните объект `String` с использованием `CountingGenerator.Character`.

14. (6) Создайте массив каждого из примитивных типов. Заполните каждый массив с использованием `CountingGenerator`. Выведите содержимое каждого массива.
15. (2) Измените пример `ContainerComparison.java`: создайте `Generator` для `BerylliumSphere` и измените метод `main()`, чтобы он использовал `Generator` с `Generated.array()`.
16. (3) Начав с `CountingGenerator.java`, создайте класс `SkipGenerator`, который создает новые значения посредством приращения, определяемого аргументом конструктора. Измените пример `TestArrayGeneration.java` и покажите, что новый класс работает правильно.
17. (5) Создайте и протестируйте `Generator` для `BigDecimal`. Убедитесь в том, что он работает с методами `Generated`.

Класс Arrays

В пакете `java.util` вы обнаружите класс `Arrays`, который содержит набор статических методов для выполнения нескольких, часто используемых операций над массивами. В их числе есть шесть основных методов: `equals()` для проверки двух массивов на равенство (и `deepEquals()` для многомерных массивов); `fill()`, который мы уже видели ранее; `sort()` для сортировки массива; `binarySearch()` для поиска элементов в отсортированном массиве; `toString()` для получения строкового представления массива; `hashCode()` для вычисления хеш-кода (подробнее см. главу 17). Все эти методы перегружены для всех простейших типов и класса `Object`. Вдобавок существует один дополнительный метод `asList()`, который преобразует любой массив в контейнер `List`, — этот метод был описан в главе 11.

Прежде чем рассматривать методы `Arrays`, необходимо упомянуть еще один полезный метод, не входящий в класс `Arrays`.

Копирование массива

В стандартной библиотеке Java имеется статический (`static`) метод с именем `System.arraycopy()`, который способен гораздо быстрее копировать массивы, чем получается в случае «ручного» копирования элементов в цикле `for`. Метод `System.arraycopy()` перегружен для всех нужных типов. Следующий пример манипулирует массивами целых чисел (`int`):

```
//: arrays/CopyingArrays.java
// Using System.arraycopy()
import java.util.*;
import static net.mindview.util.Print.*;

public class CopyingArrays {
    public static void main(String[] args) {
        int[] i = new int[7];
        int[] j = new int[10];
        Arrays.fill(i, 47);
        Arrays.fill(j, 99);
        print("i = " + Arrays.toString(i));
        print("j = " + Arrays.toString(j));
        System.arraycopy(i, 0, j, 0, i.length);
    }
}
```

```

print("j = " + Arrays.toString(j));
int[] k = new int[5];
Arrays.fill(k, 103);
System.arraycopy(i, 0, k, 0, k.length);
print("k = " + Arrays.toString(k));
Arrays.fill(k, 103);
System.arraycopy(k, 0, i, 0, k.length);
print("i = " + Arrays.toString(i));
// Objects:
Integer[] u = new Integer[10];
Integer[] v = new Integer[5];
Arrays.fill(u, new Integer(47));
Arrays.fill(v, new Integer(99));
print("u = " + Arrays.toString(u));
print("v = " + Arrays.toString(v));
System.arraycopy(v, 0, u, u.length/2, v.length);
print("u = " + Arrays.toString(u));
}
} /* Output:
i = [47, 47, 47, 47, 47, 47, 47]
j = [99, 99, 99, 99, 99, 99, 99, 99, 99, 99]
j = [47, 47, 47, 47, 47, 47, 99, 99, 99, 99]
k = [47, 47, 47, 47, 47]
i = [103, 103, 103, 103, 103, 47, 47]
u = [47, 47, 47, 47, 47, 47, 47, 47, 47, 47]
v = [99, 99, 99, 99, 99]
u = [47, 47, 47, 47, 47, 99, 99, 99, 99, 99]
*///:~

```

В аргументах `arraycopy()` передается исходный массив; смещение копируемого блока в исходном массиве; приемный массив; смещение копируемого блока в приемном массиве; и количество копируемых элементов. Естественно, попытки выхода за границу массива приводят к исключению.

Пример показывает, что копироваться могут как массивы примитивов, так и массивы объектов. Однако копирование массивов объектов ограничивается копированием ссылок — сами объекты при этом не дублируются. Такое копирование называется *поверхностным* (за дополнительной информацией обращайтесь к электронным приложениям к книге). Метод `System.arraycopy()` не выполняет ни автоматической упаковки, ни автоматической распаковки — типы двух массивов должны точно совпадать.

- 18.** (3) Создайте и заполните массив объектов `BerylliumSphere`. Скопируйте его в новый массив и продемонстрируйте, что копирование было поверхностным.

Сравнение массивов

В классе `Arrays` имеется перегруженный метод `equals()` для сравнения целых массивов, для всех примитивных типов и для `Object`. При сравнении массивов проверяются следующие условия: в массивах должно быть равное количество элементов и каждый элемент должен быть эквивалентен соответствующему элементу другого массива, для проверки этого условия используется метод `equals()`. (Для примитивов привлекается метод `equals()` соответствующего класса-«обертки»; например, метод `Integer.equals()` для целого числа `int`.) Вот пример:

```

//: arrays/ComparingArrays.java
// использование Arrays.equals()
import java.util.*;
import static net.mindview.util.Print.*;

public class ComparingArrays {
    public static void main(String[] args) {
        int[] a1 = new int[10];
        int[] a2 = new int[10];
        Arrays.fill(a1, 47);
        Arrays.fill(a2, 47);
        print(Arrays.equals(a1, a2));
        a2[3] = 11;
        print(Arrays.equals(a1, a2));
        String[] s1 = new String[4];
        Arrays.fill(s1, "Hi");
        String[] s2 = { new String("Hi"), new String("Hi"),
            new String("Hi"), new String("Hi") };
        print(Arrays.equals(s1, s2));
    }
} /* Output:
true
false
true
*///:~

```

Изначально `a1` и `a2` равны, поэтому программа выводит `true`, но затем один из элементов изменяется, поэтому выводится `false`. В последнем случае все элементы `s1` указывают на один и тот же объект, но `s2` содержит пять уникальных объектов. Однако равенство массивов определяется на основании содержимого (через `Object.equals()`), поэтому результат равен `true`.

19. (2) Создайте класс с полем `int`, инициализируемым по аргументу конструктора. Создайте два массива таких объектов, используя идентичные значения инициализации для каждого массива, и покажите, что по мнению `Arrays.equals()` они не равны. Добавьте в класс метод `equals()` для решения проблемы.
20. (4) Продемонстрируйте работу `deepEquals()` для многомерных массивов.

Сравнения элементов массивов

Сортировка должна выполнять сравнения на основании фактического типа объекта. Конечно, можно написать разные методы сортировки для всех разных типов, но такой код нельзя будет повторно использовать для новых типов.

Важнейшая цель при проектировании программы — это «отделить изменяющуюся составляющую от постоянной», в нашем случае постоянная часть — это алгоритм сортировки общего применения, а изменяющаяся — способ сравнения объектов. Поэтому вместо прямого вписывания кода сравнения в различные подпрограммы сортировки используется паттерн «Стратегия»¹. В этом паттерне та часть кода, что изменяется от случая к случаю, инкапсулируется в отдельном классе (объект стратегии). Этот

¹ «Приемы объектно-ориентированного проектирования», Э. Гамма и др. (Питер, 2013). См. «Thinking in Patterns (with Java)» на сайте www.MindView.net.

объект передается коду, который остается неизменным, и использует стратегию для реализации своего алгоритма. При таком подходе можно создать разные объекты для выражения разных способов сравнения и передать их одному коду сортировки.

В Java существует два способа предоставить описание процесса сравнения. Первый основан на использовании метода «естественного» сравнения, который включается в класс при реализации интерфейса `java.lang.Comparable`. Это очень простой интерфейс с одним методом, `compareTo()`. Метод получает в качестве аргумента другой объект того же типа и возвращает отрицательное значение, если аргумент меньше текущего объекта, ноль, если аргумент равен ему, и положительное значение, если аргумент больше, чем текущий объект.

Вот класс, который реализует интерфейс `Comparable` и демонстрирует сравнение, вызывая для этого метод стандартной библиотеки `Arrays.sort()`:

```
//: arrays/CompType.java
// Реализация Comparable в классе.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class CompType implements Comparable<CompType> {
    int i;
    int j;
    private static int count = 1;
    public CompType(int n1, int n2) {
        i = n1;
        j = n2;
    }
    public String toString() {
        String result = "[i = " + i + ", j = " + j + "]";
        if(count++ % 3 == 0)
            result += "\n";
        return result;
    }
    public int compareTo(CompType rv) {
        return (i < rv.i ? -1 : (i == rv.i ? 0 : 1));
    }
    private static Random r = new Random(47);
    public static Generator<CompType> generator() {
        return new Generator<CompType>() {
            public CompType next() {
                return new CompType(r.nextInt(100), r.nextInt(100));
            }
        };
    }
    public static void main(String[] args) {
        CompType[] a =
            Generated.array(new CompType[12], generator());
        print("before sorting:");
        print(Arrays.toString(a));
        Arrays.sort(a);
        print("after sorting:");
        print(Arrays.toString(a));
    }
} /* Output:
```

продолжение ➤

before sorting:

```
[[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
, [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]
, [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
, [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
]
```

after sorting:

```
[[i = 9, j = 78], [i = 11, j = 22], [i = 16, j = 40]
, [i = 20, j = 58], [i = 22, j = 7], [i = 51, j = 89]
, [i = 58, j = 55], [i = 61, j = 29], [i = 68, j = 0]
, [i = 88, j = 28], [i = 93, j = 61], [i = 98, j = 61]
]
```

*///:~

Когда вы определяете функцию сравнения, то берете на себя ответственность за действия, вкладываемые в сравнение одного из объектов с другим. В нашем случае в сравнении объектов участвуют только переменные *i*, а переменные *j* игнорируются.

Метод `generator()` дает объект, реализующий интерфейс `Generator`, используя для этого анонимный внутренний класс (см. главу 8). Затем с помощью описанных методов создаются объекты `CompType`, инициализированные случайными значениями. В методе `main()` генератор заполняет массив объектов `CompType`, который затем сортируется. Если интерфейс `Comparable` не был реализован, во время выполнения при попытке вызова `sort()` возбуждается исключение `ClassCastException`. Это объясняется тем, что `sort()` выполняет нисходящее преобразование своего аргумента к интерфейсу `Comparable`.

Теперь предположим, что вы получаете класс, не реализующий `Comparable`, или класс, *реализующий* этот интерфейс, но не так, как вам хочется, и вы предпочли бы написать собственный метод для сравнения. Для решения этой проблемы создается отдельный класс, реализующий интерфейс `Comparator` (кратко описанный в главе 11). Это пример паттерна проектирования «Стратегия». Он содержит два метода: `compare()` и `equals()`. Впрочем, вам не понадобится реализовывать метод `equals()`, разве что в особых случаях для повышения производительности, поскольку любой созданный вами класс наследуется от корневого класса `Object`, в котором уже есть метод `equals()`. Поэтому обычно вызывается предлагаемый по умолчанию метод `equals()` класса `Object`, следовательно, условие реализации интерфейса соблюдается.

Класс `Collections` (который мы рассмотрим в следующей главе) содержит метод `reverseOrder()`, который производит реализацию `Comparator`, меняющую порядок сортировки на обратный. Все это можно применить к `CompType`:

```
//: arrays/Reverse.java
// Использование Collections.reverseOrder()
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class Reverse {
    public static void main(String[] args) {
        CompType[] a = Generated.array(
            new CompType[12], CompType.generator());
        print("before sorting:");
        print(Arrays.toString(a));
```

```

    Arrays.sort(a, Collections.reverseOrder());
    print("after sorting:");
    print(Arrays.toString(a));
}
} /* Output:
before sorting:
[[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
, [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]
, [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
, [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
]
after sorting:
[[i = 98, j = 61], [i = 93, j = 61], [i = 88, j = 28]
, [i = 68, j = 0], [i = 61, j = 29], [i = 58, j = 55]
, [i = 51, j = 89], [i = 22, j = 7], [i = 20, j = 58]
, [i = 16, j = 40], [i = 11, j = 22], [i = 9, j = 78]
]
*///:~

```

Вы также можете написать собственную реализацию `Comparator`. Следующая реализация сравнивает объекты `CompType` по значениям `j` (вместо значений `i`):

```

//: arrays/ComparatorTest.java
// Реализация Comparator для класса.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

class CompTypeComparator implements Comparator<CompType> {
    public int compare(CompType o1, CompType o2) {
        return (o1.j < o2.j ? -1 : (o1.j == o2.j ? 0 : 1));
    }
}

public class ComparatorTest {
    public static void main(String[] args) {
        CompType[] a = Generated.array(
            new CompType[12], CompType.generator());
        print("before sorting:");
        print(Arrays.toString(a));
        Arrays.sort(a, new CompTypeComparator());
        print("after sorting:");
        print(Arrays.toString(a));
    }
} /* Output:
before sorting:
[[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
, [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]
, [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
, [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
]
after sorting:
[[i = 68, j = 0], [i = 22, j = 7], [i = 11, j = 22]
, [i = 88, j = 28], [i = 61, j = 29], [i = 16, j = 40]
, [i = 58, j = 55], [i = 20, j = 58], [i = 93, j = 61]
, [i = 98, j = 61], [i = 9, j = 78], [i = 51, j = 89]
]
*///:~

```

21. (3) Попробуйте отсортировать массив объектов из упражнения 18. Реализуйте `Comparable` для решения проблемы. Теперь создайте `Comparator` для сортировки объектов в обратном порядке.

Сортировка массива

Со встроенными методами для сортировки вы можете упорядочить любой массив простейших типов или любой массив объектов, который либо реализует `Comparable`, либо имеет связанный объект `Comparator`¹. Следующий пример генерирует случайные строки (`String`) и упорядочивает их:

```
//: arrays/StringSorting.java
// Сортировка массива строк.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class StringSorting {
    public static void main(String[] args) {
        String[] sa = Generated.array(new String[20],
            new RandomGenerator.String(5));
        print("Before sort: " + Arrays.toString(sa));
        Arrays.sort(sa);
        print("After sort: " + Arrays.toString(sa));
        Arrays.sort(sa, Collections.reverseOrder());
        print("Reverse sort: " + Arrays.toString(sa));
        Arrays.sort(sa, String.CASE_INSENSITIVE_ORDER);
        print("Case-insensitive sort: " + Arrays.toString(sa));
    }
} /* Output:
Before sort: [YNzbr, nyGcF, OWZnT, cQrGs, eGZMm, JMROE,
suEcU, OneOE, dLsmw, HLGEa, hKcxr, EqUCB, bkIna, Mesbt,
WHkjU, rUkZP, gwsqP, zDyCy, RFJQA, HxxHv]
After sort: [EqUCB, HLGEa, HxxHv, JMROE, Mesbt, OWZnT,
OneOE, RFJQA, WHkjU, YNzbr, bkIna, cQrGs, dLsmw, eGZMm,
gwsqP, hKcxr, nyGcF, rUkZP, suEcU, zDyCy]
Reverse sort: [zDyCy, suEcU, rUkZP, nyGcF, hKcxr, gwsqP,
eGZMm, dLsmw, cQrGs, bkIna, YNzbr, WHkjU, RFJQA, OneOE,
OWZnT, Mesbt, JMROE, HxxHv, HLGEa, EqUCB]
Case-insensitive sort: [bkIna, cQrGs, dLsmw, eGZMm, EqUCB,
gwsqP, hKcxr, HLGEa, HxxHv, JMROE, Mesbt, nyGcF, OneOE,
OWZnT, RFJQA, rUkZP, suEcU, WHkjU, YNzbr, zDyCy]
*///:~
```

После просмотра результата работы программы вы сразу сможете заметить, что алгоритм сортировки строк является *лексикографическим*: все слова, начинающиеся с прописных букв, помещаются в начало списка, и только после них следуют слова с первой строчной буквой. (Обычно так сортируются записи в телефонных книгах.) Чтобы сгруппировать слова без учета регистра, используйте режим `String.CASE_INSENSITIVE_ORDER`, как показано в последнем вызове `sort()` предыдущего примера.

Алгоритм сортировки из стандартной библиотеки Java разработан с учетом максимальной эффективности для сортируемого типа: для простейших типов используется

¹ Как ни странно, в Java 1.0 и 1.1 не было поддержки сортировки строк!

быстрая сортировка (quick sort), а для объектов применяется устойчивая сортировка слиянием (stable merge). Поэтому вам не следует тратить свое время, разрабатывая эффективные методы сортировки, по крайней мере до тех пор, пока ваше средство профилирования не покажет вам, что сортировка является «узким местом» программы.

Поиск в отсортированном массиве

Как только массив отсортирован, в нем можно быстро найти определенный элемент, привлекая для этого методы `Arrays.binarySearch()`. Однако очень важно понимать, что при поиске в неупорядоченных массивах результат будет непредсказуем. Следующий пример использует генератор `RandomGenerator.Integer` для заполнения массива, и тот же генератор производит значения для поиска:

```
//: arrays/ArraySearching.java
// Использование Arrays.binarySearch().
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class ArraySearching {
    public static void main(String[] args) {
        Generator<Integer> gen =
            new RandomGenerator.Integer(1000);
        int[] a = ConvertTo.primitive(
            Generated.array(new Integer[25], gen));
        Arrays.sort(a);
        print("Sorted array: " + Arrays.toString(a));
        while(true) {
            int r = gen.next();
            int location = Arrays.binarySearch(a, r);
            if(location >= 0) {
                print("Location of " + r + " is " + location +
                    ", a[" + location + "] = " + a[location]);
                break; // Out of while loop
            }
        }
    }
} /* Output:
Sorted array: [128, 140, 200, 207, 258, 258, 278, 288, 322,
429, 511, 520, 522, 551, 555, 589, 693, 704, 809, 861, 861,
868, 916, 961, 998]
Location of 322 is 8, a[8] = 322
*///:~
```

В цикле `while` производятся случайные значения для поиска, до тех пор пока в массиве не будет найдено одно из них.

Метод `Arrays.binarySearch()` возвращает большее или равное нулю значение, если при поиске был найден нужный элемент. В противном случае методом возвращается отрицательное значение, представляющее позицию, где следует вставить элемент, если бы отсортированный массив поддерживался «вручную». Получаемое значение:
-(позиция для вставки) - 1

Позиция для вставки — это индекс первого элемента, оказавшегося больше, чем ключ поиска, или значение `a.size()`, если все элементы массива меньше ключа поиска.

Если в массиве присутствуют дубликаты, процедура поиска не гарантирует, какой именно из дубликатов будет найден. Используемый алгоритм не был разработан в расчете на дубликаты, хотя их присутствие не запрещается. Впрочем, если вам понадобится сортированный список без повторяющихся элементов, используйте класс `TreeSet` (для поддержки элементов в упорядоченном состоянии) или класс `LinkedHashSet` (для поддержки элементов в порядке вставки), которые будут описаны чуть позже. Он берет заботу о деталях на себя и делает все автоматически. Только в случае проблем с производительностью стоит заменить контейнер `TreeSet` массивом, который придется поддерживать «вручную».

Если вы отсортировали массив объектов с использованием интерфейса `Comparator` (массивы примитивов нельзя сортировать таким способом), то придется включить этот же `Comparator` в вызов метода `binarySearch()` (используя перегруженную версию метода). Например, мы можем немного изменить программу `StringSorting.java` так, чтобы добавить в нее поиск:

```
//: arrays/AlphabeticSearch.java
// Поиск с использованием Comparator.
import java.util.*;
import net.mindview.util.*;

public class AlphabeticSearch {
    public static void main(String[] args) {
        String[] sa = Generated.array(new String[30],
            new RandomGenerator.String(5));
        Arrays.sort(sa, String.CASE_INSENSITIVE_ORDER);
        System.out.println(Arrays.toString(sa));
        int index = Arrays.binarySearch(sa, sa[10],
            String.CASE_INSENSITIVE_ORDER);
        System.out.println("Index: " + index + "\n" + sa[index]);
    }
} /* Output:
[bkIna, cQrGs, cXZJo, dLsmw, eGZMm, EqUCB, gwsqP, hKcxr,
HLGEa, HqXum, HxxHv, JMRoE, JmzMs, Mesbt, MNvqe, nyGcF,
ogoYW, OneOE, OWZnT, RFJQA, rUkZP, sgqia, slJrL, suEcU,
uTpnX, vpFfv, WHkjU, xxEAj, YNzbr, zDyCy]
Index: 10
HxxHv
*///:~
```

Объект `Comparator` необходимо передать в перегруженный метод `binarySearch()` в третьем аргументе. В рассмотренном примере успешный поиск гарантирован, поскольку искомое значение берется прямо из массива.

- 22.** (2) Покажите, что вызов `binarySearch()` для несортированного массива приводит к непредсказуемым результатам.
- 23.** (2) Создайте массив `Integer`, заполните его случайными значениями `int` (с использованием автоматической упаковки) и отсортируйте в обратном порядке с использованием `Comparator`.
- 24.** (3) Покажите, что в классе из упражнения 19 возможен поиск.

Резюме

В этой главе было показано, что Java предоставляет основательную поддержку низкоуровневых массивов фиксированного размера. Такие массивы отдают предпочтение производительности перед гибкостью, как и в модели массивов C и C++. В исходной версии Java низкоуровневые массивы фиксированного размера были абсолютно необходимы — не только потому, что создатели Java решили включить примитивные типы (также по соображениям производительности), но и потому, что поддержка контейнеров в этой версии была минимальной. Таким образом, в ранних версиях Java массивы всегда были предпочтительным вариантом.

В последующих версиях Java поддержка контейнеров значительно улучшилась, и теперь контейнеры превосходят массивы во всех отношениях, кроме производительности, даже при том, что производительность контейнеров была значительно улучшена. Как указывается в других местах книги, проблемы производительности все равно обычно никогда не возникают там, где вы склонны искать их причины.

С добавлением автоматической упаковки и обобщений хранение примитивов в контейнерах существенно упростилось, что становится дополнительным доводом в пользу замены низкоуровневых массивов контейнерами. Так как обобщения предоставляют контейнеры, безопасные по отношению к типам, массивы уже не обладают преимуществами и в этом отношении.

Как упоминалось в этой главе (и как вы увидите при попытке использовать их), обобщения плохо совместимы с массивами. Часто даже когда вам удастся как-то совместить обобщения с массивами (см. следующую главу), во время компиляции все равно будут выдаваться предупреждения.

При обсуждении конкретных примеров создатели языка Java несколько раз напрямую говорили мне, что следует использовать контейнеры вместо массивов (я использовал массивы для демонстрации конкретных приемов, поэтому у меня такого выбора не было).

Все эти аспекты показывают, что при программировании на Java последних версий следует отдавать предпочтение контейнерам перед массивами. И только если вы будете твердо убеждены в том, что производительность создает проблемы (а переход на массивы изменит ситуацию), переработайте код для использования массивов.

Это заявление кажется довольно смелым, но в некоторых языках вообще нет низкоуровневых массивов фиксированного размера. В них есть только контейнеры с изменяемым размером, существенно превосходящие по своей функциональности традиционные массивы C/C++/Java. Например, в Python¹ имеется тип `list`, который использует базовый синтаксис массива, но обладает существенно большей функциональностью — он даже может использоваться для наследования:

```
#: arrays/PythonLists.py
```

```
aList = [1, 2, 3, 4, 5]
print type(aList) # <type 'list'>
print aList # [1, 2, 3, 4, 5]
```

продолжение ➤

¹ См. www.Python.org.

```

print aList[4] # 5    Простейшее индексирование
aList.append(6) # Размеры list могут изменяться
aList += [7, 8] # добавление list в list
print aList # [1, 2, 3, 4, 5, 6, 7, 8]
aSlice = aList[2:4]
print aSlice # [3, 4]

class MyList(list): # Наследование от list
    # Определение метода, указатель 'this' передается явно:
    def getReversed(self):
        reversed = self[:] # Копирование list с использованием срезов
        reversed.reverse() # Встроенный метод list
        return reversed

list2 = MyList(aList) # Для создания объектов 'new' не требуется
print type(list2) # <class '__main__.MyList'>
print list2.getReversed() # [8, 7, 6, 5, 4, 3, 2, 1]
#::~

```

Базовый синтаксис Python был представлен в предыдущей главе. Здесь список создается простым заключением последовательности объектов, разделенных запятыми, в квадратные скобки. Результат представляет собой объект с типом времени выполнения `list` (вывод команд `print` приводится в комментариях в той же строке). При выводе списка выводится такой же результат, как при использовании метода `Arrays.toString()` в Java.

Создание «среза», то есть субпоследовательности списка, осуществляется включением оператора `:` в операцию индексирования. Тип `list` содержит много других встроенных операций.

`MyList` — определение класса; базовые классы перечисляются в круглых скобках. Внутри класса команды `def` определяют методы, а первый аргумент метода автоматически эквивалентен `this` в языке Java, если не считать того, что в Python он задается явно, а идентификатор `self` определяется по соглашению (то есть не является ключевым словом). Обратите внимание на автоматическое наследование конструктора.

Хотя в Python все данные представляются объектами (включая целочисленные и вещественные типы), у программиста все равно остается лазейка: для оптимизации частей кода, критичных по быстродействию, можно написать расширения на C, C++ или специальном языке Ругех, предназначенном для простого ускорения кода. Таким образом достигается «объектная чистота» без ущерба для быстродействия.

Язык PHP¹ идет еще дальше: в нем существует только один тип массива, который используется как для массивов с целочисленным индексированием, так и для ассоциативных массивов (`Map`).

После многолетней эволюции Java интересно поразмыслить над тем, стоило бы включать поддержку примитивов и низкоуровневых массивов, если бы работа над языком началась заново? Если бы их не было, можно было бы получить по-настоящему «чистый» объектно-ориентированный язык (что бы там ни говорили, Java не является «чистым» ОО-языком, притом именно из-за низкоуровневого балласта). Исходный

¹ См. www.php.net.

аргумент — эффективность — всегда выглядит заманчиво, но как мы уже видели, со временем проектировщики постепенно отказываются от этой идеи в пользу высокоуровневых компонентов (таких, как контейнеры). К этому стоит добавить, что при встраивании контейнеров в базовый синтаксис языка (как это сделано в некоторых языках) перед компилятором открывается гораздо больше возможностей для оптимизации.

Впрочем, если не отвлекаться на теоретические рассуждения, мы неразрывно связаны с массивами, и вы будете постоянно сталкиваться с ними при чтении кода. Впрочем, решения с контейнерами почти всегда оказываются предпочтительными.

25. (3) Перепишите пример `PythonLists.py` на Java.

17

Подробнее о контейнерах

В главе 11 были представлены основные концепции и функциональность библиотеки контейнеров Java; этого было достаточно, чтобы вы могли приступить к работе с контейнерами. В этой главе эта важная библиотека будет рассмотрена на более глубоком уровне.

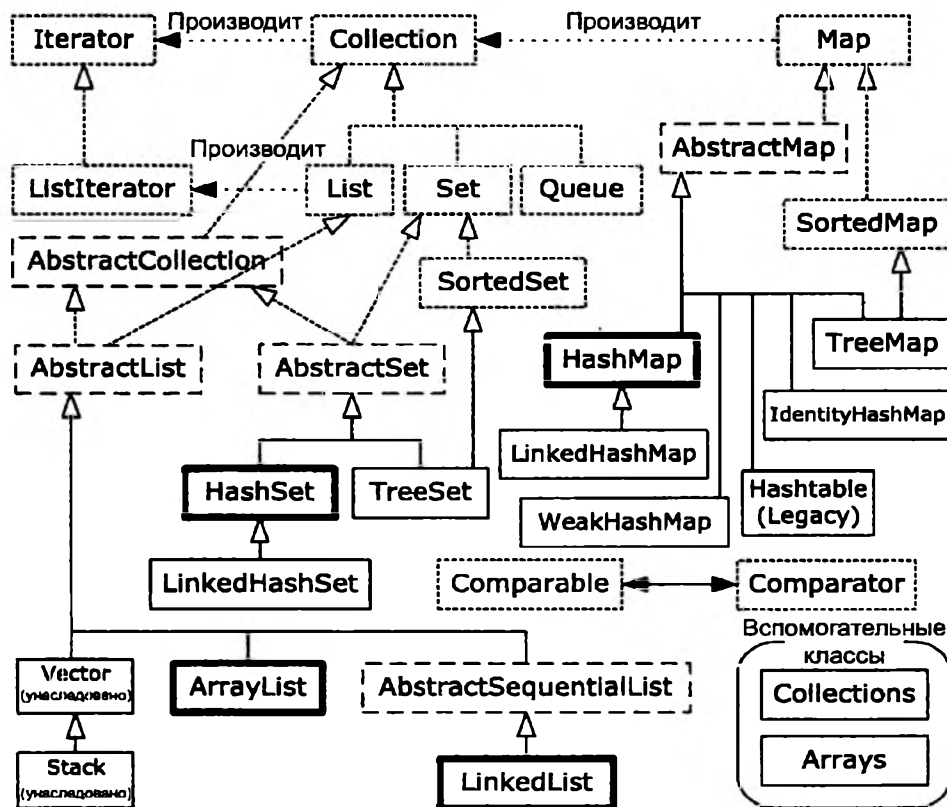
Чтобы в полной мере использовать возможности библиотеки контейнеров, необходимо знать больше того, что рассказано в главе 11, но эта глава базируется на более сложном материале (таком, как обобщения), поэтому она была отложена до более поздней стадии. После более подробного обзора контейнеров вы узнаете, как работает хеширование и как написать методы `hashCode()` и `equals()` для работы с хеширующими контейнерами. Вы узнаете, почему некоторые контейнеры существуют в нескольких версиях и как выбрать между этими версиями. Глава завершается описанием некоторых инструментов общего назначения и специализированных классов.

Полная таксономия контейнеров

В разделе «Резюме» главы 11 приведена упрощенная диаграмма библиотеки контейнеров Java. Далее представлена более полная диаграмма, включающая абстрактные классы и унаследованные компоненты (кроме реализаций `Queue`).

В Java SE5 были добавлены:

- ❑ Интерфейс `Queue` (реализуемый измененной версией `LinkedList`, как было показано в главе 11) и его реализации: `PriorityQueue` и различные разновидности `BlockingQueue`, которые будут представлены в главе 21).
- ❑ Интерфейс `ConcurrentMap` и его реализация `ConcurrentHashMap`, также предназначенная для использования в многопоточных программах и представленная в главе 21.
- ❑ `CopyOnWriteArrayList` и `CopyOnWriteArraySet` (также предназначены для многопоточных программ).



- ❑ EnumSet и EnumMap, специальные реализации Set и Map для перечислений (см. главу 19).
- ❑ Некоторые вспомогательные возможности класса Collections.

Блоки с контуром из длинных штрихов представляют абстрактные классы; также на диаграмме присутствуют классы, имена которых начинаются с «Abstract». На первый взгляд это выглядит несколько странно, но это всего лишь вспомогательные частичные реализации некоторого интерфейса. Например, создавая собственную версию Set, вы не будете начинать с интерфейса Set с реализацией всех методов; вместо этого вы определите класс, производный от AbstractSet, и выполните минимальную необходимую работу для создания нового класса. Тем не менее библиотека контейнеров содержит достаточную функциональность практически для любых ситуаций, так что обычно на классы, имена которых начинаются с Abstract, можно не обращать внимания.

Заполнение контейнеров

Хотя проблема вывода контейнеров была решена, заполнение контейнеров страдает от тех же недостатков, что и java.util.Arrays. Как и в случае с Arrays, существует вспомогательный класс Collections со статическими служебными методами, к числу которых принадлежит метод с именем fill(). Как и версия из Arrays, эта реализация fill() просто дублирует одну ссылку на объект в контейнере. Кроме того, она работает только для объектов, но полученный список можно передать конструктору или методу addAll():

```
//: containers/FillingLists.java
// Методы Collections.fill() и Collections.nCopies().
import java.util.*;
```

продолжение ➤

```

class StringAddress {
    private String s;
    public StringAddress(String s) { this.s = s; }
    public String toString() {
        return super.toString() + " " + s;
    }
}

public class FillingLists {
    public static void main(String[] args) {
        List<StringAddress> list= new ArrayList<StringAddress>(
            Collections.nCopies(4, new StringAddress("Hello")));
        System.out.println(list);
        Collections.fill(list, new StringAddress("World!"));
        System.out.println(list);
    }
} /* Output: (Sample)
[StringAddress@82ba41 Hello, StringAddress@82ba41 Hello,
StringAddress@82ba41 Hello, StringAddress@82ba41 Hello]
[StringAddress@923e30 World!, StringAddress@923e30 World!,
StringAddress@923e30 World!, StringAddress@923e30 World!]
*///:~

```

Этот пример демонстрирует два способа заполнения `Collection` ссылками на один объект. Первый, `Collections.nCopies()`, создает объект `List`, который передается конструктору и используется для заполнения `ArrayList`.

Метод `toString()` в `StringAddress` вызывает метод `Object.toString()`, который возвращает имя класса с шестнадцатеричным беззнаковым представлением хеш-кода объекта (сгенерированным методом `hashCode()`). Из результатов видно, что все ссылки указывают на один объект, причем ситуация не изменяется и после вызова второго метода `Collections.fill()`. Полезность метода `fill()` снижается еще и из-за того, что он может только заменять элементы, уже присутствующие в списке, и не может добавлять новые элементы.

Решение с Generator

Практически все подтипы `Collection` имеют конструктор, который получает другой объект `Collection`, используемый для заполнения нового контейнера. Таким образом, для простого создания тестовых данных достаточно построить класс с конструктором, в аргументах которого передается объект `Generator` (см. главы 15 и 16) и количество объектов `quantity`:

```

//: net/mindview/util/CollectionData.java
// Заполнение коллекции с использованием объекта генератора.
package net.mindview.util;
import java.util.*;

public class CollectionData<T> extends ArrayList<T> {
    public CollectionData(Generator<T> gen, int quantity) {
        for(int i = 0; i < quantity; i++)
            add(gen.next());
    }
}
// Обобщенный вспомогательный метод:
public static <T> CollectionData<T>

```



```
list(Generator<T> gen, int quantity) {
    return new CollectionData<T>(gen, quantity);
}
} ///:~
```

Объект `Generator` используется для включения в контейнер нужного количества объектов. Полученный контейнер может быть передан конструктору любого объекта `Collection`, и этот конструктор скопирует данные в самого себя. Метод `addAll()`, присутствующий в любом подтипе `Collection`, также может использоваться для заполнения существующих объектов `Collection`.

Обобщенный вспомогательный метод сокращает объем кода, необходимого для использования класса.

Класс `CollectionData` демонстрирует использование паттерна проектирования «Адаптер¹»; он адаптирует `Generator` к конструктору `Collection`.

Следующий пример инициализирует `LinkedHashSet`:

```
//: containers/CollectionDataTest.java
import java.util.*;
import net.mindview.util.*;

class Government implements Generator<String> {
    String[] foundation = ("strange women lying in ponds " +
        "distributing swords is no basis for a system of " +
        "government").split(" ");
    private int index;
    public String next() { return foundation[index++]; }
}

public class CollectionDataTest {
    public static void main(String[] args) {
        Set<String> set = new LinkedHashSet<String>(
            new CollectionData<String>(new Government(), 15));
        // Использование вспомогательного метода:
        set.addAll(CollectionData.list(new Government(), 15));
        System.out.println(set);
    }
} /* Output:
[strange, women, lying, in, ponds, distributing, swords,
is, no, basis, for, a, system, of, government]
*///:~
```

Элементы следуют в порядке вставки, потому что `LinkedHashSet` поддерживает связанный список с сохранением этого порядка.

Все генераторы, определенные в главе 16, теперь доступны через адаптер `CollectionData`. В следующем примере используются два из них:

```
//: containers/CollectionDataGeneration.java
// Использование генераторов, определенных в главе Arrays.
import java.util.*;
```

продолжение ➤

¹ Возможно, такое использование не соответствует формальному определению адаптера, приведенному в книге «Приемы объектно-ориентированного проектирования», но мне кажется, что оно близко ему по духу.

```
import net.mindview.util.*;

public class CollectionDataGeneration {
    public static void main(String[] args) {
        System.out.println(new ArrayList<String>(
            CollectionData.list( // Convenience method
                new RandomGenerator.String(9), 10)));
        System.out.println(new HashSet<Integer>(
            new CollectionData<Integer>(
                new RandomGenerator.Integer(), 10)));
    }
} /* Output:
[YNzbrnyGc, FOWZnTcQr, GseGZMmJM, RoEsuEcUO, neOEdLsmw,
HLGEahKcx, rEqUCBbkI, naMesbtWH, kjUrUkZPg, wsqPzDyCy]
[573, 4779, 871, 4367, 6090, 7882, 2017, 8037, 3455, 299]
*///:~
```

Длина объекта `String`, производимого `RandomGenerator.String`, определяется аргументом конструктора.

Генераторы Map

Аналогичный подход можно использовать и для `Map`, но для этого потребуется класс `Pair`, поскольку для заполнения `Map` при каждом вызове метода `next()` генератора должна производиться пара объектов (ключ и значение):

```
//: net/mindview/util/Pair.java
package net.mindview.util;

public class Pair<K,V> {
    public final K key;
    public final V value;
    public Pair(K k, V v) {
        key = k;
        value = v;
    }
} ///:~
```

Поля `key` и `value` объявлены открытыми (`public`) и неизменяемыми (`final`), так что `Pair` становится *объектом передачи данных*, доступным только для чтения.

Адаптер `Map` теперь может использовать различные комбинации `Generator`, `Iterable` и констант для заполнения объектов инициализации `Map`:

```
//: net/mindview/util/MapData.java
// Заполнение данными Map с использованием объекта генератора.
package net.mindview.util;
import java.util.*;

public class MapData<K,V> extends LinkedHashMap<K,V> {
    // Один генератор Pair:
    public MapData(Generator<Pair<K,V>> gen, int quantity) {
        for(int i = 0; i < quantity; i++) {
            Pair<K,V> p = gen.next();
            put(p.key, p.value);
        }
    }
}
```

```

// Два разных генератора:
public MapData(Generator<K> genK, Generator<V> genV,
    int quantity) {
    for(int i = 0; i < quantity; i++) {
        put(genK.next(), genV.next());
    }
}
// Генератор ключа и одно значение:
public MapData(Generator<K> genK, V value, int quantity){
    for(int i = 0; i < quantity; i++) {
        put(genK.next(), value);
    }
}
// Iterable и генератор значения:
public MapData(Iterable<K> genK, Generator<V> genV) {
    for(K key : genK) {
        put(key, genV.next());
    }
}
// Iterable и одно значение:
public MapData(Iterable<K> genK, V value) {
    for(K key : genK) {
        put(key, value);
    }
}
// Обобщенные вспомогательные методы:
public static <K,V> MapData<K,V>
map(Generator<Pair<K,V>> gen, int quantity) {
    return new MapData<K,V>(gen, quantity);
}
public static <K,V> MapData<K,V>
map(Generator<K> genK, Generator<V> genV, int quantity) {
    return new MapData<K,V>(genK, genV, quantity);
}
public static <K,V> MapData<K,V>
map(Generator<K> genK, V value, int quantity) {
    return new MapData<K,V>(genK, value, quantity);
}
public static <K,V> MapData<K,V>
map(Iterable<K> genK, Generator<V> genV) {
    return new MapData<K,V>(genK, genV);
}
public static <K,V> MapData<K,V>
map(Iterable<K> genK, V value) {
    return new MapData<K,V>(genK, value);
}
} ///:~

```

Таким образом, появляется выбор между использованием одного генератора `Generator<Pair<K,V>>`, двух отдельных генераторов, одного генератора и константы, реализации `Iterable` (а значит, любых реализаций `Collection`) и генератора или `Iterable` и значения. Обобщенные вспомогательные методы сокращают объем необходимого кода для создания объектов `MapData`.

Рассмотрим пример использования `MapData`. Генератор `Letters` также реализует `Iterable`, предоставляя `Iterator`; это позволяет использовать его для тестирования методов `MapData.map()`, которые работают с `Iterable`:

```

//: containers/MapDataTest.java
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

class Letters implements Generator<Pair<Integer,String>>,
    Iterable<Integer> {
    private int size = 9;
    private int number = 1;
    private char letter = 'A';
    public Pair<Integer,String> next() {
        return new Pair<Integer,String>(
            number++, "" + letter++);
    }
    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            public Integer next() { return number++; }
            public boolean hasNext() { return number < size; }
            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
}

public class MapDataTest {
    public static void main(String[] args) {
        // Генератор Pair:
        print(MapData.map(new Letters(), 11));
        // Два разных генератора:
        print(MapData.map(new CountingGenerator.Character(),
            new RandomGenerator.String(3), 8));
        // Генератор ключей и одно значение:
        print(MapData.map(new CountingGenerator.Character(),
            "Value", 6));
        // Iterable и генератор значений:
        print(MapData.map(new Letters(),
            new RandomGenerator.String(3)));
        // Iterable и одно значение:
        print(MapData.map(new Letters(), "Pop"));
    }
} /* Output:
{1=A, 2=B, 3=C, 4=D, 5=E, 6=F, 7=G, 8=H, 9=I, 10=J, 11=K}
{a=YNz, b=brn, c=yGc, d=FOw, e=ZnT, f=cQr, g=Gse, h=GZM}
{a=Value, b=Value, c=Value, d=Value, e=Value, f=Value}
{1=mJM, 2=RoE, 3=suE, 4=cUO, 5=neO, 6=EdL, 7=smw, 8=HLG}
{1=Pop, 2=Pop, 3=Pop, 4=Pop, 5=Pop, 6=Pop, 7=Pop, 8=Pop}
*///:~

```

В этом примере также используются генераторы из главы 16.

С помощью этих средств можно сгенерировать произвольный набор данных для `Map` и `Collection`, а затем инициализировать `Map` или `Collection` с использованием конструктора или методов `Map.putAll()` или `Collection.addAll()`.

Использование классов Abstract

Альтернативный подход к проблеме производства тестовых данных для контейнеров основан на создании нестандартных реализаций `Collection` и `Map`. Каждый контейнер `java.util` имеет собственный класс `Abstract`, который предоставляет частичную реализацию контейнера; разработчику остается только реализовать необходимые методы для получения нужного контейнера. Если контейнер доступен только для чтения (что типично для тестовых данных), количество методов, которые требуется предоставить, сводится к минимуму.

И хотя в данной ситуации это не особенно необходимо, следующее решение демонстрирует другой паттерн проектирования: «Легковес». Он применяется в тех случаях, когда обычное решение требует слишком большого количества объектов или создание обычных объектов требует слишком больших затрат памяти. Паттерн «Легковес» выводит наружу часть составляющих объекта, чтобы обращение к ним производилось более эффективно через внешнюю таблицу (или посредством других вычислений для экономии памяти).

В этом примере важно то, что он демонстрирует, как относительно просто создаются специализации `Map` и `Collection` наследованием от классов `java.util.Abstract`. Чтобы создать контейнер `Map`, доступный только для чтения, вы наследуете от `AbstractMap` и реализуете `entrySet()`. Чтобы создать множество `Set`, доступное только для чтения, разработчик определяет класс, производный от `AbstractSet`, и реализует `iterator()` и `size()`.

В качестве набора данных в этом примере используется набор названий стран мира и их столиц¹. Метод `capitals()` создает контейнер `Map` с этими данными, а метод `names()` создает `List` с названиями стран. В обоих случаях можно получить частичный список, передав аргумент `int` с нужным размером:

```
//: net/mindview/util/Countries.java
// Контейнеры Map и List с тестовыми данными
// для паттерна "Легковес"
package net.mindview.util;
import java.util.*;
import static net.mindview.util.Print.*;

public class Countries {
    public static final String[][] DATA = {
        // Африка
        {"ALGERIA", "Algiers"}, {"ANGOLA", "Luanda"},
        {"BENIN", "Porto-Novo"}, {"BOTSWANA", "Gaberone"},
        {"BULGARIA", "Sofia"}, {"BURKINA FASO", "Ouagadougou"},
        {"BURUNDI", "Bujumbura"},
        {"CAMEROON", "Yaounde"}, {"CAPE VERDE", "Praia"},
        {"CENTRAL AFRICAN REPUBLIC", "Bangui"},
        {"CHAD", "N'djamena"}, {"COMOROS", "Moroni"},
        {"CONGO", "Brazzaville"}, {"DJIBOUTI", "Djibouti"},
        {"EGYPT", "Cairo"}, {"EQUATORIAL GUINEA", "Malabo"},
        {"ERITREA", "Asmara"}, {"ETHIOPIA", "Addis Ababa"},
    }
```

продолжение ↗

¹ Данные были найдены в Интернете. Со временем читатели внесли ряд исправлений.

```

{"GABON","Libreville"}, {"THE GAMBIA","Banjul"},
{"GHANA","Accra"}, {"GUINEA","Conakry"},
{"BISSAU","Bissau"},
{"COTE D'IVOIR (IVORY COAST)","Yamoussoukro"},
{"KENYA","Nairobi"}, {"LESOTHO","Maseru"},
{"LIBERIA","Monrovia"}, {"LIBYA","Tripoli"},
{"MADAGASCAR","Antananarivo"}, {"MALAWI","Lilongwe"},
{"MALI","Bamako"}, {"MAURITANIA","Nouakchott"},
{"MAURITIUS","Port Louis"}, {"MOROCCO","Rabat"},
{"MOZAMBIQUE","Maputo"}, {"NAMIBIA","Windhoek"},
{"NIGER","Niamey"}, {"NIGERIA","Abuja"},
{"RWANDA","Kigali"},
{"SAO TOME E PRINCIPE","Sao Tome"},
{"SENEGAL","Dakar"}, {"SEYCHELLES","Victoria"},
{"SIERRA LEONE","Freetown"}, {"SOMALIA","Mogadishu"},
{"SOUTH AFRICA","Pretoria/Cape Town"},
{"SUDAN","Khartoum"},
{"SWAZILAND","Mbabane"}, {"TANZANIA","Dodoma"},
{"TOGO","Lome"}, {"TUNISIA","Tunis"},
{"UGANDA","Kampala"},
{"DEMOCRATIC REPUBLIC OF THE CONGO (ZAIRE)","Kinshasa"},
{"ZAMBIA","Lusaka"}, {"ZIMBABWE","Harare"},
// Азия
{"AFGHANISTAN","Kabul"}, {"BAHRAIN","Manama"},
{"BANGLADESH","Dhaka"}, {"BHUTAN","Thimphu"},
{"BRUNEI","Bandar Seri Begawan"},
{"CAMBODIA","Phnom Penh"},
{"CHINA","Beijing"}, {"CYPRUS","Nicosia"},
{"INDIA","New Delhi"}, {"INDONESIA","Jakarta"},
{"IRAN","Tehran"}, {"IRAQ","Baghdad"},
{"ISRAEL","Jerusalem"}, {"JAPAN","Tokyo"},
{"JORDAN","Amman"}, {"KUWAIT","Kuwait City"},
{"LAOS","Vientiane"}, {"LEBANON","Beirut"},
{"MALAYSIA","Kuala Lumpur"}, {"THE MALDIVES","Male"},
{"MONGOLIA","Ulan Bator"},
{"MYANMAR (BURMA)","Rangoon"},
{"NEPAL","Katmandu"}, {"NORTH KOREA","P'yongyang"},
{"OMAN","Muscat"}, {"PAKISTAN","Islamabad"},
{"PHILIPPINES","Manila"}, {"QATAR","Doha"},
{"SAUDI ARABIA","Riyadh"}, {"SINGAPORE","Singapore"},
{"SOUTH KOREA","Seoul"}, {"SRI LANKA","Colombo"},
{"SYRIA","Damascus"},
{"TAIWAN (REPUBLIC OF CHINA)","Taipei"},
{"THAILAND","Bangkok"}, {"TURKEY","Ankara"},
{"UNITED ARAB EMIRATES","Abu Dhabi"},
{"VIETNAM","Hanoi"}, {"YEMEN","Sana'a"},
// Австралия и Океания
{"AUSTRALIA","Canberra"}, {"FIJI","Suva"},
{"KIRIBATI","Bairiki"},
{"MARSHALL ISLANDS","Dalap-Uliga-Darrit"},
{"MICRONESIA","Palikir"}, {"NAURU","Yaren"},
{"NEW ZEALAND","Wellington"}, {"PALAU","Koror"},
{"PAPUA NEW GUINEA","Port Moresby"},
{"SOLOMON ISLANDS","Honaira"}, {"TONGA","Nuku'alofa"},
{"TUVALU","Fongafale"}, {"VANUATU","Port-Vila"},
{"WESTERN SAMOA","Apia"},
// Восточная Европа и страны бывшего СССР

```

```

{"ARMENIA","Yerevan"}, {"AZERBAIJAN","Baku"},
{"BELARUS (BYELORUSSIA)","Minsk"},
{"GEORGIA","Tbilisi"},
{"KAZAKSTAN","Almaty"}, {"KYRGYZSTAN","Alma-Ata"},
{"MOLDOVA","Chisinau"}, {"RUSSIA","Moscow"},
{"TAJIKISTAN","Dushanbe"}, {"TURKMENISTAN","Ashkabad"},
{"UKRAINE","Kyiv"}, {"UZBEKISTAN","Tashkent"},
// Европа
{"ALBANIA","Tirana"}, {"ANDORRA","Andorra la Vella"},
{"AUSTRIA","Vienna"}, {"BELGIUM","Brussels"},
{"BOSNIA","-"}, {"HERZEGOVINA","Sarajevo"},
{"CROATIA","Zagreb"}, {"CZECH REPUBLIC","Prague"},
{"DENMARK","Copenhagen"}, {"ESTONIA","Tallinn"},
{"FINLAND","Helsinki"}, {"FRANCE","Paris"},
{"GERMANY","Berlin"}, {"GREECE","Athens"},
{"HUNGARY","Budapest"}, {"ICELAND","Reykjavik"},
{"IRELAND","Dublin"}, {"ITALY","Rome"},
{"LATVIA","Riga"}, {"LIECHTENSTEIN","Vaduz"},
{"LITHUANIA","Vilnius"}, {"LUXEMBOURG","Luxembourg"},
{"MACEDONIA","Skopje"}, {"MALTA","Valletta"},
{"MONACO","Monaco"}, {"MONTENEGRO","Podgorica"},
{"THE NETHERLANDS","Amsterdam"}, {"NORWAY","Oslo"},
{"POLAND","Warsaw"}, {"PORTUGAL","Lisbon"},
{"ROMANIA","Bucharest"}, {"SAN MARINO","San Marino"},
{"SERBIA","Belgrade"}, {"SLOVAKIA","Bratislava"},
{"SLOVENIA","Ljuijana"}, {"SPAIN","Madrid"},
{"SWEDEN","Stockholm"}, {"SWITZERLAND","Berne"},
{"UNITED KINGDOM","London"}, {"VATICAN CITY","---"},
// Северная и Центральная Америка
{"ANTIGUA AND BARBUDA","Saint John's"},
{"BAHAMAS","Nassau"},
{"BARBADOS","Bridgetown"}, {"BELIZE","Belmopan"},
{"CANADA","Ottawa"}, {"COSTA RICA","San Jose"},
{"CUBA","Havana"}, {"DOMINICA","Roseau"},
{"DOMINICAN REPUBLIC","Santo Domingo"},
{"EL SALVADOR","San Salvador"},
{"GRENADA","Saint George's"},
{"GUATEMALA","Guatemala City"},
{"HAITI","Port-au-Prince"},
{"HONDURAS","Tegucigalpa"}, {"JAMAICA","Kingston"},
{"MEXICO","Mexico City"}, {"NICARAGUA","Managua"},
{"PANAMA","Panama City"}, {"ST. KITTS","-"},
{"NEVIS","Basseterre"}, {"ST. LUCIA","Castries"},
{"ST. VINCENT AND THE GRENADINES","Kingstown"},
{"UNITED STATES OF AMERICA","Washington, D.C."},
// Южная Америка
{"ARGENTINA","Buenos Aires"},
{"BOLIVIA","Sucre (legal)/La Paz(administrative)"},
{"BRAZIL","Brasilia"}, {"CHILE","Santiago"},
{"COLOMBIA","Bogota"}, {"ECUADOR","Quito"},
{"GUYANA","Georgetown"}, {"PARAGUAY","Asuncion"},
{"PERU","Lima"}, {"SURINAME","Paramaribo"},
{"TRINIDAD AND TOBAGO","Port of Spain"},
{"URUGUAY","Montevideo"}, {"VENEZUELA","Caracas"},
};
// Использование AbstractMap реализацией entrySet()
private static class FlyweightMap
extends AbstractMap<String,String> {

```

```

private static class Entry
implements Map.Entry<String,String> {
    int index;
    Entry(int index) { this.index = index; }
    public boolean equals(Object o) {
        return DATA[index][0].equals(o);
    }
    public String getKey() { return DATA[index][0]; }
    public String getValue() { return DATA[index][1]; }
    public String setValue(String value) {
        throw new UnsupportedOperationException();
    }
    public int hashCode() {
        return DATA[index][0].hashCode();
    }
}
// Использование AbstractSet реализацией size() и iterator()
static class EntrySet
extends AbstractSet<Map.Entry<String,String>> {
    private int size;
    EntrySet(int size) {
        if(size < 0)
            this.size = 0;
        // Не может быть больше массива:
        else if(size > DATA.length)
            this.size = DATA.length;
        else
            this.size = size;
    }
    public int size() { return size; }
    private class Iter
    implements Iterator<Map.Entry<String,String>> {
        // Только один объект Entry на Iterator:
        private Entry entry = new Entry(-1);
        public boolean hasNext() {
            return entry.index < size - 1;
        }
        public Map.Entry<String,String> next() {
            entry.index++;
            return entry;
        }
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
    public
    Iterator<Map.Entry<String,String>> iterator() {
        return new Iter();
    }
}
private static Set<Map.Entry<String,String>> entries =
    new EntrySet(DATA.length);
public Set<Map.Entry<String,String>> entrySet() {
    return entries;
}
}
// Создание частичной карты из 'size' стран:
static Map<String,String> select(final int size) {

```



```

return new FlyweightMap() {
    public Set<Map.Entry<String,String>> entrySet() {
        return new EntrySet(size);
    }
};
}
static Map<String,String> map = new FlyweightMap();
public static Map<String,String> capitals() {
    return map; // Полная карта
}
public static Map<String,String> capitals(int size) {
    return select(size); // Частичная карта
}
static List<String> names =
    new ArrayList<String>(map.keySet());
// Все имена:
public static List<String> names() { return names; }
// Частичный список:
public static List<String> names(int size) {
    return new ArrayList<String>(select(size).keySet());
}
public static void main(String[] args) {
    print(capitals(10));
    print(names(10));
    print(new HashMap<String,String>(capitals(3)));
    print(new LinkedHashMap<String,String>(capitals(3)));
    print(new TreeMap<String,String>(capitals(3)));
    print(new Hashtable<String,String>(capitals(3)));
    print(new HashSet<String>(names(6)));
    print(new LinkedHashSet<String>(names(6)));
    print(new TreeSet<String>(names(6)));
    print(new ArrayList<String>(names(6)));
    print(new LinkedList<String>(names(6)));
    print(capitals().get("BRAZIL"));
}
} /* Output:
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo,
BOTSWANA=Gaberone, BULGARIA=Sofia, BURKINA
FASO=Ouagadougou, BURUNDI=Bujumbura, CAMEROON=Yaounde, CAPE
VERDE=Praia, CENTRAL AFRICAN REPUBLIC=Bangui}
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO,
BURUNDI, CAMEROON, CAPE VERDE, CENTRAL AFRICAN REPUBLIC]
{BENIN=Porto-Novo, ANGOLA=Luanda, ALGERIA=Algiers}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
[BULGARIA, BURKINA FASO, BOTSWANA, BENIN, ANGOLA, ALGERIA]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
Brasilia
*///:~

```

Двумерный массив DATA объявлен открытым (`public`), чтобы он мог использоваться где угодно. Контейнер `FlyweightMap` должен реализовать метод `entrySet()`, для которого необходима как пользовательская реализация `Set`, так и пользовательский класс

`Map.Entry`. А вот собственно «легковесная» часть: в каждом объекте `Map.Entry` хранится индекс вместо фактического ключа и значения. При вызове `getKey()` или `getValue()` индекс используется для возвращения соответствующего элемента `DATA`. Класс `EntrySet` следит за тем, чтобы его размер не превысил размер `DATA`.

Другая часть паттерна реализуется в `EntrySet.Iterator`. Вместо создания объекта `Map.Entry` для каждой пары данных в `DATA` создается только один объект `Map.Entry` на итератор. Объект `Entry` используется как «окно» к данным; в нем содержится только индекс (`index`) в статическом массиве строк. При каждом вызове `next()` для итератора индекс в `Entry` увеличивается, чтобы он указывал на следующую пару, после чего `next()` возвращает объект `Entry` этого итератора¹.

Метод `select()` производит объект `FlyweightMap`, содержащий `EntrySet` желаемого размера; он используется в перегруженных методах `capitals()` и `names()`, что продемонстрировано в `main()`.

В некоторых тестах ограниченный размер `Countries` создает проблемы. Аналогичный подход можно использовать для создания инициализируемых контейнеров, содержащих набор данных произвольного размера. Этот класс представляет собой список `List`, который может иметь произвольный размер и (по сути) предварительно инициализироваться данными `Integer`:

```
//: net/mindview/util/CountingIntegerList.java
// Список произвольной длины, содержащий тестовые данные.
package net.mindview.util;
import java.util.*;

public class CountingIntegerList
    extends AbstractList<Integer> {
    private int size;
    public CountingIntegerList(int size) {
        this.size = size < 0 ? 0 : size;
    }
    public Integer get(int index) {
        return Integer.valueOf(index);
    }
    public int size() { return size; }
    public static void main(String[] args) {
        System.out.println(new CountingIntegerList(30));
    }
} /* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
*///:~
```

Чтобы создать на базе `AbstractList` контейнер `List`, доступный только для чтения, необходимо реализовать `get()` и `size()`. И снова используется решение на базе паттерна «Легковес»: `get()` производит значение по запросу, так что реально заполнять `List` не обязательно.

¹ Контейнеры `Map` в `java.util` выполняют групповое копирование с использованием `getKey()` и `getValue()`, и это работает. Если пользовательская реализация `Map` будет просто копировать весь объект `Map.Entry`, возникнут проблемы.

В следующем листинге представлен контейнер Map, предварительно инициализированный уникальными значениями Integer и String; он тоже может иметь произвольный размер:

```
//: net/mindview/util/CountingMapData.java
// Контейнер Map неограниченной длины с тестовыми данными.
package net.mindview.util;
import java.util.*;

public class CountingMapData
extends AbstractMap<Integer,String> {
    private int size;
    private static String[] chars =
        "A B C D E F G H I J K L M N O P Q R S T U V W X Y Z"
        .split(" ");
    public CountingMapData(int size) {
        if(size < 0) this.size = 0;
        this.size = size;
    }
    private static class Entry
implements Map.Entry<Integer,String> {
        int index;
        Entry(int index) { this.index = index; }
        public boolean equals(Object o) {
            return Integer.valueOf(index).equals(o);
        }
        public Integer getKey() { return index; }
        public String getValue() {
            return
                chars[index % chars.length] +
                Integer.toString(index / chars.length);
        }
        public String setValue(String value) {
            throw new UnsupportedOperationException();
        }
        public int hashCode() {
            return Integer.valueOf(index).hashCode();
        }
    }
    public Set<Map.Entry<Integer,String>> entrySet() {
        // LinkedHashSet сохраняет порядок инициализации:
        Set<Map.Entry<Integer,String>> entries =
            new LinkedHashSet<Map.Entry<Integer,String>>();
        for(int i = 0; i < size; i++)
            entries.add(new Entry(i));
        return entries;
    }
    public static void main(String[] args) {
        System.out.println(new CountingMapData(60));
    }
} /* Output:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0,
9=J0, 10=K0, 11=L0, 12=M0, 13=N0, 14=O0, 15=P0, 16=Q0,
17=R0, 18=S0, 19=T0, 20=U0, 21=V0, 22=W0, 23=X0, 24=Y0,
25=Z0, 26=A1, 27=B1, 28=C1, 29=D1, 30=E1, 31=F1, 32=G1,
33=H1, 34=I1, 35=J1, 36=K1, 37=L1, 38=M1, 39=N1, 40=O1,
41=P1, 42=Q1, 43=R1, 44=S1, 45=T1, 46=U1, 47=V1, 48=W1,
49=X1, 50=Y1, 51=Z1, 52=A2, 53=B2, 54=C2, 55=D2, 56=E2,
57=F2, 58=G2, 59=H2}
*///:~
```

Здесь вместо создания пользовательской реализации `Set` создается класс `LinkedHashSet`, так что паттерн «Легковес» реализован не в полной мере.

1. (1) Создайте контейнер `List` (опробуйте как `ArrayList`, так и `LinkedList`) и заполните его данными `Countries`. Отсортируйте список и выведите его, затем многократно примените к списку `Collections.shuffle()`. После каждого применения выводите содержимое контейнера, чтобы было видно, что метод `shuffle()` каждый раз переставляет элементы списка в разном порядке.
2. (2) Создайте контейнеры `Map` и `Set` со всеми странами, названия которых начинаются с буквы «А».
3. (1) Используя данные `Countries`, несколько раз заполните контейнер `Set` одними данными и убедитесь в том, что в `Set` в конечном итоге каждый экземпляр входит в множество однократно. Попробуйте сделать то же самое с `HashSet`, `LinkedHashSet` и `TreeSet`.
4. (2) Создайте инициализатор `Collection`, который открывает файл и разбивает его на слова при помощи `TextFile`, после чего использует слова как источник данных для `Collection`. Покажите, что ваше решение работает.
5. (3) Измените пример `CountingMapData.java` и обеспечьте полноценную реализацию паттерна «Легковес»; для этого добавьте класс `EntrySet` вроде использованного в `Countries.java`.

Функциональность `Collection`

В таблице 17.1 перечислены все возможные операции `Collection` (кроме методов, унаследованных от корневого класса `Object`), а значит, и операции со списками (`List`) и множествами (`Set`). (Объекты `List` также обладают несколькими дополнительными возможностями.) Карты `map` не являются производными от `Collection` и поэтому рассматриваются отдельно.

Таблица 17.1. Функциональность `Collection`

Метод	Предназначение
<code>boolean add(T)</code>	Проверяет, хранится ли в контейнере аргумент обобщенного типа <code>T</code> . Возвращает <code>false</code> , если аргумент не был добавлен. (Это «необязательный» метод, о нем мы узнаем чуть позже.)
<code>boolean addAll(Collection<? extends T>)</code>	Добавляет все элементы, содержащиеся в аргументе. Возвращает <code>true</code> , если был добавлен хотя бы один элемент (необязательный метод)
<code>void clear()</code>	Удаляет все элементы из контейнера (необязательный метод)
<code>boolean contains(T)</code>	Возвращает <code>true</code> , если в контейнере хранится аргумент обобщенного типа <code>T</code>
<code>boolean containsAll(Collection<?>)</code>	Возвращает <code>true</code> , если в контейнере содержатся все элементы, имеющиеся в аргументе
<code>boolean isEmpty()</code>	Возвращает <code>true</code> , если в контейнере нет элементов

Метод	Предназначение
<code>Iterator<T> iterator()</code>	Возвращает объект <code>Iterator<T></code> , который можно использовать для перемещения по элементам контейнера
<code>boolean remove(Object)</code>	Если аргумент содержится в контейнере, то один его экземпляр из него удаляется. Возвращает <code>true</code> , если удаление прошло успешно (необязательный метод)
<code>boolean removeAll(Collection<?>)</code>	Удаляет все элементы, содержащиеся в аргументе. Возвращает <code>true</code> , если было проведено хотя бы одно удаление (необязательный метод)
<code>boolean retainAll(Collection<?>)</code>	Оставляет в контейнере только те элементы, которые присутствуют в аргументе («пересечение» на языке теории множеств). Возвращает <code>true</code> , если произошли какие-либо изменения (необязательный метод)
<code>int size()</code>	Возвращает количество элементов в контейнере
<code>Object[] toArray()</code>	Возвращает массив, содержащий все элементы контейнера
<code><T> T[] toArray(T[] a)</code>	Возвращает массив, содержащий все элементы контейнера, тип которых совпадает с типом массива-аргумента <code>a</code> (вместо <code>Object</code>)

Заметьте, что здесь нет метода `get()` для произвольной выборки элементов. Причина в том, что в иерархию `Collection` также входит множество (`Set`), у которого имеется своя собственная внутренняя структура расположения элементов (что лишает доступ к произвольному элементу всякого смысла). Таким образом, если вам понадобится просмотреть все элементы коллекции, придется применить итератор.

Следующий пример демонстрирует все перечисленные методы. И снова они работают со всяким контейнером, производным от `Collection`, в качестве «наименьшего общего кратного» здесь используется контейнер `ArrayList`:

```

//: containers/CollectionMethods.java
// Операции, которые могут выполняться
// со всеми разновидностями Collection.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class CollectionMethods {
    public static void main(String[] args) {
        Collection<String> c = new ArrayList<String>();
        c.addAll(Countries.names(6));
        c.add("ten");
        c.add("eleven");
        print(c);
        // Создание массива на базе List:
        Object[] array = c.toArray();
        // Создание массива String на базе List:
        String[] str = c.toArray(new String[0]);
        // Поиск наибольшего и наименьшего элементов;
        // смысл может зависеть от того, как реализован
        // интерфейс Comparable:
        print("Collections.max(c) = " + Collections.max(c));
        print("Collections.min(c) = " + Collections.min(c));
        // Добавление Collection в другой объект Collection
    }
}

```

продолжение ➤

```

Collection<String> c2 = new ArrayList<String>();
c2.addAll(Countries.names(6));
c.addAll(c2);
print(c);
c.remove(Countries.DATA[0][0]);
print(c);
c.remove(Countries.DATA[1][0]);
print(c);
// Удаление всех компонентов, присутствующих
// в коллекции-аргументе:
c.removeAll(c2);
print(c);
c.addAll(c2);
print(c);
// Элемент присутствует в Collection?
String val = Countries.DATA[3][0];
print("c.contains(" + val + ") = " + c.contains(val));
// Содержимое Collection входит в другой объект Collection?
print("c.containsAll(c2) = " + c.containsAll(c2));
Collection<String> c3 =
    ((List<String>)c).subList(3, 5);
// Оставить все элементы, присутствующие
// в c2 и c3 (пересечение множеств):
c2.retainAll(c3);
print(c2);
// Удалить из c2 все элементы,
// также присутствующие в c3:
c2.removeAll(c3);
print("c2.isEmpty() = " + c2.isEmpty());
c = new ArrayList<String>();
c.addAll(Countries.names(6));
print(c);
c.clear(); // Удаление всех элементов
print("after c.clear():" + c);
}
} /* Output:
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO,
ten, eleven]
Collections.max(c) = ten
Collections.min(c) = ALGERIA
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO,
ten, eleven, ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA,
BURKINA FASO]
[ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO, ten,
eleven, ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA
FASO]
[BENIN, BOTSWANA, BULGARIA, BURKINA FASO, ten, eleven,
ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
[ten, eleven]
[ten, eleven, ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA,
BURKINA FASO]
c.contains(BOTSWANA) = true
c.containsAll(c2) = true
[ANGOLA, BENIN]
c2.isEmpty() = true
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
after c.clear():[]
*///:~

```

Контейнеры `ArrayList` создаются с различными наборами данных и преобразуются в объекты `Collection`; следовательно, ничего, кроме методов интерфейса `Collection`, в примере не используется. Метод `main()` использует простые «упражнения» для демонстрации всех методов `Collection`.

Последующие разделы описывают различные реализации списков (`List`), множеств (`Set`) и карт (`Map`), и в каждом случае отмечено (звездочкой), какой контейнер следует выбирать по умолчанию. Описания устаревших классов `Vector`, `Stack` и `Hashtable` отложены до конца главы — хотя пользоваться ими не стоит, они все еще встречаются в старом коде.

Необязательные операции

Методы, выполняющие различные операции добавления и удаления, относятся к *необязательным* операциям в интерфейсе `Collection`. Это означает, что реализация класса не обязана предоставлять работоспособные определения таких методов.

Такое определение интерфейса весьма необычно. Как вы уже видели, в объектно-ориентированном проектировании интерфейс определяет *контракт*. Он говорит: «Как бы вы ни реализовали этот интерфейс, я гарантирую, что вы сможете отправлять объекту эти сообщения¹». «Необязательная» операция нарушает этот фундаментальный принцип; речь идет о том, что некоторые методы не выполняют осмысленных действий. Вместо этого они возбуждают исключения! Казалось бы, безопасность типов во время компиляции теряется.

Впрочем, не все так плохо. Если операция является необязательной, компилятор все равно ограничивает вас вызовом методов этого интерфейса. Ситуация не похожа на динамические языки, в которых можно вызвать любой метод для любого объекта и только во время выполнения узнать, сработает ли конкретный вызов². Кроме того, многие методы, которые получают объект `Collection` в аргументе, ограничиваются чтением из `Collection`, а все методы чтения в `Collection` являются обязательными.

Зачем определять методы как «необязательные»? Это предотвращает размножение интерфейсов в архитектуре. Во многих библиотеках контейнеров появляется невразумительная куча интерфейсов, описывающих все вариации на основную тему. В интерфейсах даже невозможно отразить все особые случаи, потому что кто-нибудь всегда сможет изобрести новый интерфейс. Благодаря подходу с «неподдерживаемыми операциями» в библиотеке контейнеров Java достигается очень важная цель: контейнеры просты в изучении и использовании. Неподдерживаемые операции — особый случай, который можно отложить до возникновения необходимости. Однако для того, чтобы этот подход работал, необходимо выполнение некоторых условий.

¹ Я использую термин «интерфейс» для описания как формального ключевого слова `interface`, так и в более общем смысле «методы, поддерживаемые любым классом или субклассом».

² В таком изложении это кажется странным и не очень практичным, но как было показано ранее (особенно в главе 14), такое динамическое поведение иногда открывает очень полезные возможности.

1. Исключение `UnsupportedOperationException` должно быть редким событием. Иначе говоря, для большинства классов все операции должны работать, и только в отдельных случаях операция может оказаться неподдерживаемой. Это условие выполняется для библиотеки контейнеров Java, поскольку классы, используемые в 99% случаев — `ArrayList`, `LinkedList`, `HashSet` и `HashMap`, а также другие конкретные реализации, — поддерживают все операции. Неподдерживаемые операции предоставляют «лазейку» на тот случай, если вы захотите создать новую реализацию `Collection` без предоставления осмысленных определений всех методов интерфейса `Collection` и при этом интегрировать ее в существующую библиотеку.
2. Если операция не поддерживается, должна существовать разумная вероятность того, что исключение `UnsupportedOperationException` обнаружится во время работы над реализацией, а не после отправки продукта пользователю. В конце концов, исключение указывает на ошибку программирования: реализация используется некорректно.

Стоит заметить, что неподдерживаемые операции могут обнаруживаться только во время выполнения, а следовательно, представляют динамическую проверку типов. Программистам с опытом работы на языках со статической типизацией (таких, как C++) может показаться, что Java — просто еще один язык со статической типизацией. Безусловно, в Java имеется статическая проверка типов, но наряду с ней также присутствует и значительная доля динамической типизации, так что трудно однозначно отнести этот язык к тому или иному типу. Как только вы осознаете это, вы начнете видеть другие примеры динамической проверки типов в Java.

Неподдерживаемые операции

Самая частая причина неподдерживаемых операций — контейнеры, работающие на базе структуры данных с фиксированным размером. Такие контейнеры создаются в результате преобразования массива в `List` методом `Arrays.asList()`. Также можно *заставить* любой контейнер (включая `Map`) выдавать исключения `UnsupportedOperationException` посредством использования «немодифицирующих» методов класса `Collections`. В этом примере представлены оба случая:

```

//: containers/Unsupported.java
// Неподдерживаемые операции в контейнерах Java.
import java.util.*;

public class Unsupported {
    static void test(String msg, List<String> list) {
        System.out.println("--- " + msg + " ---");
        Collection<String> c = list;
        Collection<String> subList = list.subList(1,8);
        // Копирование подписки:
        Collection<String> c2 = new ArrayList<String>(subList);
        try { c.retainAll(c2); } catch(Exception e) {
            System.out.println("retainAll(): " + e);
        }
        try { c.removeAll(c2); } catch(Exception e) {
            System.out.println("removeAll(): " + e);
        }
    }
}

```



```

try { c.clear(); } catch(Exception e) {
    System.out.println("clear(): " + e);
}
try { c.add("X"); } catch(Exception e) {
    System.out.println("add(): " + e);
}
try { c.addAll(c2); } catch(Exception e) {
    System.out.println("addAll(): " + e);
}
try { c.remove("C"); } catch(Exception e) {
    System.out.println("remove(): " + e);
}
// Метод List.set() изменяет значение,
// но не изменяет размер структуры данных:
try {
    list.set(0, "X");
} catch(Exception e) {
    System.out.println("List.set(): " + e);
}
}
public static void main(String[] args) {
    List<String> list =
        Arrays.asList("A B C D E F G H I J K L".split(" "));
    test("Modifiable Copy", new ArrayList<String>(list));
    test("Arrays.asList()", list);
    test("unmodifiableList()",
        Collections.unmodifiableList(
            new ArrayList<String>(list)));
}
} /* Output:
--- Modifiable Copy ---
--- Arrays.asList() ---
retainAll(): java.lang.UnsupportedOperationException
removeAll(): java.lang.UnsupportedOperationException
clear(): java.lang.UnsupportedOperationException
add(): java.lang.UnsupportedOperationException
addAll(): java.lang.UnsupportedOperationException
remove(): java.lang.UnsupportedOperationException
--- unmodifiableList() ---
retainAll(): java.lang.UnsupportedOperationException
removeAll(): java.lang.UnsupportedOperationException
clear(): java.lang.UnsupportedOperationException
add(): java.lang.UnsupportedOperationException
addAll(): java.lang.UnsupportedOperationException
remove(): java.lang.UnsupportedOperationException
List.set(): java.lang.UnsupportedOperationException
*///:~

```

Так как `Arrays.asList()` создает `List` на базе массива фиксированного размера, вполне разумно, что поддерживаются только операции, не изменяющие размер массива. Любой метод, приводящий к изменению размера нижележащей структуры данных, выдает исключение `UnsupportedOperationException` для обозначения вызова неподдерживаемого метода (ошибка программирования).

Учтите, что результат `Arrays.asList()` всегда можно передать в аргументе конструктора любой реализации `Collection` (или воспользоваться методом `addAll()`, или статическим методом `Collections.addAll()`) для создания обычного контейнера, допускающего

использование всех методов, — эта возможность продемонстрирована при первом вызове `test()` в `main()`. Такой вызов создает новую структуру данных с возможностью изменения размера.

«Немодифицирующие» методы класса `Collections` упаковывают контейнер в объект-заместитель, выдающий исключение `UnsupportedOperationException` при попытке выполнения операции, каким-либо образом изменяющей контейнер. Целью этих методов является создание «константного» объекта контейнера. Полный список «немодифицирующих» методов `Collections` приводится ниже.

Последний блок `try` в `test()` проверяет метод `set()`, являющийся частью `List`. Этот момент интересен, потому что он демонстрирует эффективное использование granularity механизма «неподдерживаемых операций» — итоговые «интерфейсы» могут различаться на один метод между объектом, возвращаемым методом `Arrays.asList()`, и объектом, возвращаемым `Collections.unmodifiableList()`. Метод `Arrays.asList()` возвращает `List` фиксированного размера, а `Collections.unmodifiableList()` создает список, который не может изменяться. Как видно из результатов, изменение элементов контейнера `List`, возвращаемого `Arrays.asList()`, не создает проблем, потому что оно не нарушает характеристики фиксированного размера этого списка. Но при этом очевидно, что результат `unmodifiableList()` не должен изменяться никаким образом. Если бы использовались интерфейсы, для этого понадобились бы два дополнительных интерфейса: один с рабочим методом `set()`, а другой без него. Дополнительные интерфейсы потребовались бы и для различных немодифицирующих подтипов `Collection`.

В документации метода, получающего контейнер в аргументе, должно быть указано, какие из необязательных методов должны быть реализованы.

6. (2) Контейнер `List` содержит дополнительные «необязательные» операции, не включенные в `Collection`. Напишите версию `Unsupported.java` с проверкой этих дополнительных необязательных операций.

Функциональность `List`

На базовом уровне контейнеры `List` просты в использовании. Чаще всего используются методы `add()` для добавления объектов, `get()` для их выборки по одному и `iterator()` для получения итератора последовательности.

Вдобавок, на самом деле есть два типа списков: основной `ArrayList`, который особенно хорош при извлечении произвольных элементов из контейнера, и гораздо более мощный список `LinkedList` (который не разрабатывался для быстрого доступа к произвольным элементам, но зато он обладает более полным набором методов).

Методы следующего примера охватывают совокупность различных видов действий: то, что может каждый список (метод `basicTest()`), перемещение по списку посредством итератора (`iterMotion()`) в сравнении с изменением списка с помощью итератора (`iterManipulation()`), просмотр результатов операций со списками (`testVisual()`) и операции, доступные только в классе `LinkedList`.

```
//: containers/Lists.java
// Операции со списками List.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class Lists {
    private static boolean b;
    private static String s;
    private static int i;
    private static Iterator<String> it;
    private static ListIterator<String> lit;
    public static void basicTest(List<String> a) {
        a.add(1, "x"); // Добавление в позиции 1
        a.add("x");    // Добавление в конце
        // Добавление коллекции:
        a.addAll(Countries.names(25));
        // Добавление коллекции с позиции 3:
        a.addAll(3, Countries.names(25));
        b = a.contains("1"); // Значение присутствует?
        // Вся коллекция?
        b = a.containsAll(Countries.names(25));
        // Списки обеспечивают произвольный доступ -
        // быстрый для ArrayList, медленный для LinkedList:
        s = a.get(1)        ; // Получение (типизованного) объекта в позиции 1
        i = a.indexOf("1"); // Получение индекса объекта
        b = a.isEmpty();    // Список содержит элементы?
        it = a.iterator();  // Обычный Iterator
        lit = a.listIterator(); // ListIterator
        lit = a.listIterator(3); // Начать с позиции 3
        i = a.lastIndexOf("1"); // Последнее вхождение
        a.remove(1);        // Удаление в позиции 1
        a.remove("3");      // Удаление заданного объекта
        a.set(1, "y");      // Записать "y" в позицию 1
        // Оставить все элементы, присутствующие в аргументе
        // (пересечение двух множеств):
        a.retainAll(Countries.names(25));
        // Удаление всех элементов, присутствующих в аргументе:
        a.removeAll(Countries.names(25));
        i = a.size();       // Определение размера
        a.clear();          // Удаление всех элементов
    }
    public static void iterMotion(List<String> a) {
        ListIterator<String> it = a.listIterator();
        b = it.hasNext();
        b = it.hasPrevious();
        s = it.next();
        i = it.nextIndex();
        s = it.previous();
        i = it.previousIndex();
    }
    public static void iterManipulation(List<String> a) {
        ListIterator<String> it = a.listIterator();
        it.add("47");
        // Переход к элементу после add():
```

продолжение ➤

```

    it.next();
    // Удаление элемента за вновь созданным:
    it.remove();
    // Переход к элементу после remove():
    it.next();
    // Изменение элемента после удаленного:
    it.set("47");
}
public static void testVisual(List<String> a) {
    print(a);
    List<String> b = Countries.names(25);
    print("b = " + b);
    a.addAll(b);
    a.addAll(b);
    print(a);
    // Вставка, удаление и замена элементов
    // с использованием ListIterator:
    ListIterator<String> x = a.listIterator(a.size()/2);
    x.add("one");
    print(a);
    print(x.next());
    x.remove();
    print(x.next());
    x.set("47");
    print(a);
    // Перебор списка в обратном направлении:
    x = a.listIterator(a.size());
    while(x.hasPrevious())
        printnb(x.previous() + " ");
    print();
    print("testVisual finished");
}
// Некоторые операции, поддерживаемые только для LinkedList:
public static void testLinkedList() {
    LinkedList<String> ll = new LinkedList<String>();
    ll.addAll(Countries.names(25));
    print(ll);
    // Интерпретировать как стек - занесение:
    ll.addFirst("one");
    ll.addFirst("two");
    print(ll);
    // Аналог чтения с вершины стека (без извлечения):
    print(ll.getFirst());
    // Аналог извлечения из стека:
    print(ll.removeFirst());
    print(ll.removeFirst());
    // Интерпретировать как очередь с извлечением элементов
    // с конца:
    print(ll.removeLast());
    print(ll);
}
public static void main(String[] args) {
    // Каждый раз создается и заполняется новый список:
    basicTest(
        new LinkedList<String>(Countries.names(25)));
}

```

```

    basicTest(
        new ArrayList<String>(Countries.names(25)));
    iterMotion(
        new LinkedList<String>(Countries.names(25)));
    iterMotion(
        new ArrayList<String>(Countries.names(25)));
    iterManipulation(
        new LinkedList<String>(Countries.names(25)));
    iterManipulation(
        new ArrayList<String>(Countries.names(25)));
    testVisual(
        new LinkedList<String>(Countries.names(25)));
    testLinkedList();
}
} /* (Execute to see output) *///:~

```

В методах `basicTest()` и `iterMotion()` вызовы методов списка делаются просто для демонстрации правильной формы записи, и хотя их возвращаемое значение сохраняется, оно не используется. В некоторых случаях возвращаемое значение не сохраняется в принципе. Вам следует внимательно изучить описание этих методов в документации JDK на сайте java.sun.com, перед тем как работать с ними.

7. (4) Создайте объекты `ArrayList` и `LinkedList`, заполните каждый из них с использованием генератора `Countries.names()`. Выведите списки с использованием обычного итератора `Iterator`, затем вставьте один список в другой с использованием `ListIterator`, выполняя вставку через позицию. Теперь выполните вставку, начиная от конца первого списка и перемещаясь в обратном направлении.
8. (7) Создайте обобщенный класс односвязного списка с именем `SList`, который для простоты не реализует интерфейс `List`. Каждый объект узла `Link` должен содержать ссылку на следующий элемент списка, но не на предыдущий (в отличие от двусвязного списка `LinkedList`, который поддерживает ссылки в обоих направлениях). Создайте собственную реализацию `SListIterator`, которая (опять же для простоты) не реализует `ListIterator`. Кроме `toString()`, класс `SList` должен содержать только метод `iterator()`, который создает `SListIterator`. Вставка и удаление элементов из `SList` должны выполняться только через `SListIterator`. Напишите код для демонстрации `SList`.

Set и порядок хранения

Примеры `Set` из главы 11 неплохо демонстрируют основные операции, выполняемые с множествами (`Set`). Однако в этих примерах для удобства использовались определенные типы Java (такие, как `Integer` и `String`), предназначенные для использования в контейнерах. При создании собственных типов учтите, что множество должно каким-то образом поддерживать порядок хранения. Способ поддержания этого порядка зависит от реализации `Set`. Таким образом, разные реализации `Set` могут различаться не только поведением, но и требованиями к типу объекта, который можно поместить в конкретную реализацию `Set`:

Таблица 17.2. Множества Set

Название класса/ интерфейса	Описание
Set (интерфейс)	Каждый элемент, добавляемый в множество Set, должен быть уникальным; в противном случае дубликат не добавляется. Все объекты, помещаемые в Set, должны определять метод equals() для выполнения сравнения. Интерфейс Set полностью идентичен интерфейсу Collection. Множество Set не гарантирует того, что хранимые в нем элементы будут располагаться в определенном порядке
HashSet*	Для реализаций Set, у которых первостепенное значение имеет быстрый поиск. Хранимые объекты должны определять метод hashCode()
TreeSet	Упорядоченное множество, реализованное на основе дерева. Из него можно извлекать упорядоченную последовательность элементов. Элементы также должны реализовать интерфейс Comparable
LinkedHashSet	Обладает аналогичной HashSet скоростью поиска, а также своими силами (используя связанный список) запоминает порядок добавления элементов (порядок вставки). Таким образом, при переборе элементов этого множества они следуют в порядке вставки. Элементы также должны определять hashCode()

Звездочка рядом с HashSet указывает, что при отсутствии других ограничений следует выбирать именно этот вариант, потому что он оптимизирован по скорости.

Способ определения метода hashCode() будет описан в этой главе позднее. Вы всегда должны создавать метод equals() как для хешированного множества, так и для множества на базе дерева, а метод hashCode() необходим только в том случае, если вы намереваетесь хранить объекты нового типа во множестве HashSet (что весьма вероятно, и всегда следует рассматривать этот выбор как основу для первой реализации множества) или LinkedHashSet. Впрочем, хороший стиль программирования подразумевает, что при переопределении метода equals() необходимо переопределять и метод hashCode().

Следующий пример демонстрирует методы, которые должны определяться для успешного использования типа с конкретной реализацией Set:

```
//: containers/TypesForSets.java
// Методы, необходимые для хранения типа в Set.
import java.util.*;

class SetType {
    int i;
    public SetType(int n) { i = n; }
    public boolean equals(Object o) {
        return o instanceof SetType && (i == ((SetType)o).i);
    }
    public String toString() { return Integer.toString(i); }
}

class HashType extends SetType {
    public HashType(int n) { super(n); }
    public int hashCode() { return i; }
}
```

```

class TreeType extends SetType
implements Comparable<TreeType> {
    public TreeType(int n) { super(n); }
    public int compareTo(TreeType arg) {
        return (arg.i < i ? -1 : (arg.i == i ? 0 : 1));
    }
}

public class TypesForSets {
    static <T> Set<T> fill(Set<T> set, Class<T> type) {
        try {
            for(int i = 0; i < 10; i++)
                set.add(
                    type.getConstructor(int.class).newInstance(i));
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
        return set;
    }
    static <T> void test(Set<T> set, Class<T> type) {
        fill(set, type);
        fill(set, type); // Попытка добавления дубликатов
        fill(set, type);
        System.out.println(set);
    }
    public static void main(String[] args) {
        test(new HashSet<HashType>(), HashType.class);
        test(new LinkedHashSet<HashType>(), HashType.class);
        test(new TreeSet<TreeType>(), TreeType.class);
        // Следующие операции не работают:
        test(new HashSet<SetType>(), SetType.class);
        test(new HashSet<TreeType>(), TreeType.class);
        test(new LinkedHashSet<SetType>(), SetType.class);
        test(new LinkedHashSet<TreeType>(), TreeType.class);
        try {
            test(new TreeSet<SetType>(), SetType.class);
        } catch(Exception e) {
            System.out.println(e.getMessage());
        }
        try {
            test(new TreeSet<HashType>(), HashType.class);
        } catch(Exception e) {
            System.out.println(e.getMessage());
        }
    }
} /* Output: (Sample)
[2, 4, 9, 8, 6, 1, 3, 7, 5, 0]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[9, 9, 7, 5, 1, 2, 6, 3, 0, 7, 2, 4, 4, 7, 9, 1, 3, 6, 2,
4, 3, 0, 5, 0, 8, 8, 8, 6, 5, 1]
[0, 5, 5, 6, 5, 0, 3, 1, 9, 8, 4, 2, 3, 9, 7, 3, 4, 4, 0,
7, 1, 9, 6, 2, 1, 8, 2, 8, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8,
9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8,
9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
java.lang.ClassCastException: SetType cannot be cast to

```

продолжение ➤

```
java.lang.Comparable
java.lang.ClassCastException: HashType cannot be cast to
java.lang.Comparable
*///:~
```

Чтобы показать, какие методы необходимы для конкретной реализации Set и в то же время избежать дублирования кода, мы создаем три класса. Базовый класс SetType просто сохраняет значение int и выдает его представление методом toString(). Так как все классы, хранимые в Set, должны содержать метод equals(), этот метод также помещается в базовый класс. Равенство базируется на значении int i.

Класс HashType является производным от SetType и добавляет метод hashCode(), необходимый для помещения объекта в хешированную реализацию Set.

Интерфейс Comparable, реализованный TreeType, необходим в тех случаях, когда объект должен использоваться в сортированном контейнере — таком, как SortedSet (единственной реализацией которого является TreeSet). Заметьте, что в методе compareTo() я *не употребил* «простое и очевидное» выражение return i-i2. Это обычная ошибка программиста, и правильно работает такое выражение только для случая «беззнаковых» целых чисел (в Java нет ключевого слова unsigned). Для целых чисел со знаком, используемых в Java, такое выражение неверно, поскольку этот тип недостаточно велик для вмещения разности двух целых со знаком. Если i — большое положительное целое число, а j — большое отрицательное целое, то при вычислении выражения i-j произойдет переполнение и возврат отрицательного числа, которое в данном контексте вызовет ошибку выполнения.

Обычно метод compareTo() должен обеспечивать естественное упорядочение, не противоречащее методу equals(). Если equals() выдает true для конкретного сравнения, то метод compareTo() должен возвращать нуль для этого сравнения, а если equals() выдает false, то результат compareTo() должен быть отличен от нуля.

В TypesForSets методы fill() и test() определяются с использованием обобщений; тем самым предотвращается дублирование кода. Для проверки поведения Set метод test() вызывает fill() для тестового множества три раза, пытаясь создать объекты-дубликаты. Метод fill() получает Set произвольного типа и объект Class того же типа. Объект Class используется для поиска конструктора с аргументом int, который затем вызывается для добавления элементов в Set.

Из результатов видно, что в HashSet элементы хранятся в совершенно загадочном порядке (который будет объяснен позднее в этой главе), в LinkedHashSet элементы хранятся в порядке вставки, а TreeSet хранит элементы в порядке сортировки (из-за особенностей реализации compareTo() он соответствует сортировке по убыванию).

Если вы попытаетесь использовать типы, не поддерживающие необходимые операции для Set, возможны серьезные проблемы. Включение объекта SetType или TreeType, не включающего переопределенный метод hashCode(), в любую реализацию с хешированием приводит к появлению дубликатов и нарушению первичного контракта Set. Это весьма неприятно, потому что нарушение не приводит к ошибке времени выполнения. Однако у hashCode() имеется реализация по умолчанию, поэтому такое поведение законно, даже несмотря на его некорректность. Единственным надежным способом обеспечения правильности такой программы является встраивание модульных тестов

в систему сборки (за дополнительной информацией обращайтесь к приложению <http://MindView.net/Books/BetterJava>).

При включении в `TreeSet` типа, не реализующего `Comparable`, будет получен более определенный результат: при попытке использования объекта как `Comparable` возбуждается исключение.

SortedSet

Элементы отсортированного множества `SortedSet` гарантированно хранятся в порядке сортировки, что позволяет задействовать дополнительную функциональность, обеспечиваемую интерфейсом `SortedSet`.

- ❑ `Comparator comparator()`: возвращает объект `Comparator`, использующийся для данного множества, или `null` для естественного упорядочивания.
- ❑ `Object first()`: производит наименьший элемент множества.
- ❑ `Object last()`: производит наибольший элемент множества.
- ❑ `SortedSet subSet(fromElement, toElement)`: производит подмножество данного множества, в которое включены элементы от `fromElement` (включительно) до `toElement` (не включая).
- ❑ `SortedSet headSet(toElement)`: производит подмножество, содержащее все элементы, меньшие `toElement`.
- ❑ `SortedSet tailSet(fromElement)`: производит подмножество, содержащее все элементы, большие либо равные `fromElement`.

Вот простая иллюстрация сказанного:

```
//: containers/SortedSetDemo.java
// Операции с TreeSet.
import java.util.*;
import static net.mindview.util.Print.*;

public class SortedSetDemo {
    public static void main(String[] args) {
        SortedSet<String> sortedSet = new TreeSet<String>();
        Collections.addAll(sortedSet,
            "one two three four five six seven eight"
                .split(" "));
        print(sortedSet);
        String low = sortedSet.first();
        String high = sortedSet.last();
        print(low);
        print(high);
        Iterator<String> it = sortedSet.iterator();
        for(int i = 0; i <= 6; i++) {
            if(i == 3) low = it.next();
            if(i == 6) high = it.next();
            else it.next();
        }
        print(low);
        print(high);
        print(sortedSet.subSet(low, high));
    }
}
```

продолжение ➤

```

        print(sortedSet.headSet(high));
        print(sortedSet.tailSet(low));
    }
} /* Output:
[eight, five, four, one, seven, six, three, two]
eight
two
one
two
[one, seven, six, three]
[eight, five, four, one, seven, six, three]
[one, seven, six, three, two]
*///:~

```

Заметьте, что элементы в множестве `SortedSet` отсортированы согласно их функциям сравнения, а не в соответствии с очередностью добавления. Для сохранения порядка вставки следует использовать `LinkedHashSet`.

9. (2) Используйте генератор `RandomGenerator.String` для заполнения множества `TreeSet` с алфавитной сортировкой. Выведите содержимое `TreeSet`, чтобы подтвердить правильность сортировки.
10. (7) Используя `LinkedList` в качестве базовой реализации, определите собственную реализацию `SortedSet`.

Очереди

Если не считать многопоточных версий, в Java SE5 используются всего две реализации `Queue`: `LinkedList` и `PriorityQueue`. Они различаются прежде всего по поведению упорядочения, а не по производительности. В следующем примере задействованы многие реализации `Queue` (не все из них работают в этом примере), включая многопоточные реализации. Элементы помещаются с одного конца очереди, а извлекаются с другого:

```

//: containers/QueueBehavior.java
// Сравнение поведения некоторых очередей
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

public class QueueBehavior {
    private static int count = 10;
    static <T> void test(Queue<T> queue, Generator<T> gen) {
        for(int i = 0; i < count; i++)
            queue.offer(gen.next());
        while(queue.peek() != null)
            System.out.print(queue.remove() + " ");
        System.out.println();
    }
    static class Gen implements Generator<String> {
        String[] s = ("one two three four five six seven " +
            "eight nine ten").split(" ");
        int i;
        public String next() { return s[i++]; }
    }
}

```

```

public static void main(String[] args) {
    test(new LinkedList<String>(), new Gen());
    test(new PriorityQueue<String>(), new Gen());
    test(new ArrayBlockingQueue<String>(count), new Gen());
    test(new ConcurrentLinkedQueue<String>(), new Gen());
    test(new LinkedBlockingQueue<String>(), new Gen());
    test(new PriorityBlockingQueue<String>(), new Gen());
}
} /* Output:
one two three four five six seven eight nine ten
eight five four nine one seven six ten three two
one two three four five six seven eight nine ten
one two three four five six seven eight nine ten
one two three four five six seven eight nine ten
eight five four nine one seven six ten three two
*///:~

```

Вы видите, что за исключением приоритетных очередей Queue производит элементы в порядке их помещения в Queue.

Приоритетные очереди

Приоритетные очереди были кратко представлены в главе 11. Давайте рассмотрим более интересную задачу: список дел, в котором каждый объект содержит строку и два приоритета, первичный и вторичный. Порядок следования элементов списка снова определяется реализацией Comparable:

```

//: containers/ToDoList.java
// Нетривиальное использование PriorityQueue.
import java.util.*;

class ToDoList extends PriorityQueue<ToDoList.ToDoItem> {
    static class ToDoItem implements Comparable<ToDoItem> {
        private char primary;
        private int secondary;
        private String item;
        public ToDoItem(String td, char pri, int sec) {
            primary = pri;
            secondary = sec;
            item = td;
        }
        public int compareTo(ToDoItem arg) {
            if(primary > arg.primary)
                return +1;
            if(primary == arg.primary)
                if(secondary > arg.secondary)
                    return +1;
            else if(secondary == arg.secondary)
                return 0;
            return -1;
        }
        public String toString() {
            return Character.toString(primary) +
                secondary + ": " + item;
        }
    }
}

```

продолжение ➤

```

public void add(String td, char pri, int sec) {
    super.add(new ToDoItem(td, pri, sec));
}
public static void main(String[] args) {
    ToDoList toDoList = new ToDoList();
    toDoList.add("Empty trash", 'C', 4);
    toDoList.add("Feed dog", 'A', 2);
    toDoList.add("Feed bird", 'B', 7);
    toDoList.add("Mow lawn", 'C', 3);
    toDoList.add("Water lawn", 'A', 1);
    toDoList.add("Feed cat", 'B', 1);
    while(!toDoList.isEmpty())
        System.out.println(toDoList.remove());
}
} /* Output:
A1: Water lawn
A2: Feed dog
B1: Feed cat
B7: Feed bird
C3: Mow lawn
C4: Empty trash
*///:~

```

Как видите, в приоритетных очередях упорядочение элементов происходит автоматически.

11. (2) Создайте класс с полем `Integer`, которое инициализируется значением от 0 до 100 средствами `java.util.Random`. Реализуйте интерфейс `Comparable` с использованием этого поля `Integer`. Заполните `PriorityQueue` объектами своего класса и извлеките значения методом `poll()`, чтобы показать, что они извлекаются в нужном порядке.

Деки

Дек (двусторонняя очередь) ведет себя как очередь, но с возможностью добавления и удаления элементов с обоих концов. В `LinkedList` имеются методы поддержки операций дека, но явно определенного интерфейса для деков в стандартных библиотеках Java не существует. Таким образом, `LinkedList` не может реализовать этот интерфейс, и вы не можете выполнить восходящее преобразование к интерфейсу `Deque` (в отличие от интерфейса `Queue` в предыдущем примере). Впрочем, класс `Deque` можно создать посредством композиции и просто предоставить доступ к соответствующим методам из `LinkedList`:

```

//: net/mindview/util/Deque.java
// Создание Deque из LinkedList.
package net.mindview.util;
import java.util.*;

public class Deque<T> {
    private LinkedList<T> deque = new LinkedList<T>();
    public void addFirst(T e) { deque.addFirst(e); }
    public void addLast(T e) { deque.addLast(e); }
    public T getFirst() { return deque.getFirst(); }
    public T getLast() { return deque.getLast(); }
}

```

```

public T removeFirst() { return deque.removeFirst(); }
public T removeLast() { return deque.removeLast(); }
public int size() { return deque.size(); }
public String toString() { return deque.toString(); }
// И другие методы по мере необходимости...
} ///:~

```

Вероятно, при использовании этого класса Deque в своих программах вы обнаружите, что для удобства работы в него стоит добавить и другие методы.

Простой пример использования класса Deque:

```

//: containers/DequeTest.java
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DequeTest {
    static void fillTest(Deque<Integer> deque) {
        for(int i = 20; i < 27; i++)
            deque.addFirst(i);
        for(int i = 50; i < 55; i++)
            deque.addLast(i);
    }
    public static void main(String[] args) {
        Deque<Integer> di = new Deque<Integer>();
        fillTest(di);
        print(di);
        while(di.size() != 0)
            printnb(di.removeFirst() + " ");
        print();
        fillTest(di);
        while(di.size() != 0)
            printnb(di.removeLast() + " ");
    }
} /* Output:
[26, 25, 24, 23, 22, 21, 20, 50, 51, 52, 53, 54]
26 25 24 23 22 21 20 50 51 52 53 54
54 53 52 51 50 20 21 22 23 24 25 26
*///:~

```

Необходимость включения элементов и их извлечения с обоих концов встречается реже, так что деки используются не так часто, как очереди.

Карты (Map)

Как вы узнали в главе 11, отличительная особенность карты (также называемой *ассоциативным массивом*) состоит в том, что в ней хранятся связи «ключ-значение» (пары), чтобы к значениям можно было обращаться по ключу. Стандартная библиотека Java содержит различные типы карт: классы `HashMap`, `TreeMap`, `LinkedHashMap`, `WeakHashMap`, `ConcurrentHashMap` и `IdentityHashMap`. Они имеют одинаковый интерфейс (поскольку все реализуют интерфейс `Map`), но отличаются в производительности, порядке добавления и представления пар объектов, сроках хранения объектов и способах сравнения ключей. Количество реализаций интерфейса `Map` убедительно свидетельствует о важности этой разновидности контейнеров.

Для более глубокого понимания Map полезно посмотреть, как конструируются ассоциативные массивы. Возьмем очень простую реализацию:

```
//: containers/AssociativeArray.java
// Связывание ключей со значениями.
import static net.mindview.util.Print.*;

public class AssociativeArray<K,V> {
    private Object[][] pairs;
    private int index;
    public AssociativeArray(int length) {
        pairs = new Object[length][2];
    }
    public void put(K key, V value) {
        if(index >= pairs.length)
            throw new ArrayIndexOutOfBoundsException();
        pairs[index++] = new Object[]{ key, value };
    }
    @SuppressWarnings("unchecked")
    public V get(K key) {
        for(int i = 0; i < index; i++)
            if(key.equals(pairs[i][0]))
                return (V)pairs[i][1];
        return null; // Ключ не найден
    }
    public String toString() {
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < index; i++) {
            result.append(pairs[i][0].toString());
            result.append(" : ");
            result.append(pairs[i][1].toString());
            if(i < index - 1)
                result.append("\n");
        }
        return result.toString();
    }
    public static void main(String[] args) {
        AssociativeArray<String,String> map =
            new AssociativeArray<String,String>(6);
        map.put("sky", "blue");
        map.put("grass", "green");
        map.put("ocean", "dancing");
        map.put("tree", "tall");
        map.put("earth", "brown");
        map.put("sun", "warm");
        try {
            map.put("extra", "object"); // Выход за границу контейнера
        } catch(ArrayIndexOutOfBoundsException e) {
            print("Too many objects!");
        }
        print(map);
        print(map.get("ocean"));
    }
} /* Output:
Too many objects!
sky : blue
grass : green
```

```
ocean : dancing
tree : tall
earth : brown
sun : warm
dancing
*///:~
```

Важнейшие методы ассоциативных массивов — `put()` и `get()`, но для наглядности метод `toString()` был переопределен для вывода пар «ключ-значение». Метод `main()` заполняет `AssociativeArray` парами строк и выводит полученную карту и результат выборки одного из значений методом `get()`. При вызове `get()` передается ключ, а метод возвращает ассоциированное значение или `null`, если поиск оказался неудачным. Пожалуй, метод `get()` использует наименее эффективный способ поиска: он проверяет элементы от начала массива, используя `equals()` для сравнения ключей. Впрочем, эта реализация ориентирована на простоту, а не на эффективность.

Итак, приведенная версия полезна в учебных целях, но не очень эффективна и обладает фиксированным размером, ухудшающим ее гибкость. К счастью, у карт из `java.util` таких проблем нет, и их можно использовать в этом примере.

12. (1) Используйте карты `HashMap`, `TreeMap` и `LinkedHashMap` в методе `main()` примера `AssociativeArray.java`.
13. (4) Используйте пример `AssociativeArray.java` для создания счетчика слов, связывающего ключ `String` со значением `Integer`. Используя средства `net.mindview.util.TextFile`, откройте текстовый файл, разбейте его на слова по пропускam и знакам препинания и подсчитайте вхождения каждого слова в тексте.

Производительность

Для карт очень важен вопрос производительности. Например, использование линейного поиска в `get()` при поиске ключа — достаточно медленный процесс. Карта `HashMap` значительно увеличивает скорость такого поиска. Вместо неторопливого поиска ключа она работает со специальным значением, называемым *хеш-кодом* (`hash code`). Хеш-код — это способ преобразования некоторой информации об объекте в «относительно уникальное» целое число, связанное с этим объектом. Метод `hashCode()` встроен в корневой класс `Object`, поэтому хеш-код может генерироваться для любых объектов `Java`. Карта `HashMap` задействует метод `hashCode()` для быстрого поиска ключа. Это приводит к впечатляющему всплеску производительности¹.

¹ Если это ускорение в работе все же не удовлетворяет вашим запросам к эффективности, вы можете дополнительно ускорить поиск в таблице, написав свою собственную карту `Map` для хранения определенных типов, чтобы избавиться от постоянного преобразования к типу `Object`. Энтузиастам оптимизации могу посоветовать второе издание книги Дональда Кнута «Искусство программирования» (*The Art of Computer Programming*), третий том «Сортировка и поиск» (*Sorting and Searching*), где списки с переполнением рекомендуется заменить на массивы, которые имеют два преимущества: их можно оптимизировать для хранения на диске, а также они помогут сэкономить время при создании и уничтожении отдельных элементов уборщиком мусора.

В табл. 3 перечислены основные реализации `Map`. Звездочка рядом с `HashMap` указывает, что при отсутствии других ограничений следует выбирать именно этот вариант, потому что он оптимизирован по скорости. Другие реализации ориентированы на другие характеристики, а следовательно, уступают `HashMap` по скорости.

Таблица 17.3. Карты `Map`

Название класса/ интерфейса	Описание
<code>HashMap*</code>	Реализация карты на основе хеш-таблицы (используйте вместо устаревшего класса <code>Hashtable</code>). Поиск и вставка пар занимает небольшое постоянное время. Производительность можно настроить, указав в конструкторах особые значения для <i>емкости</i> и <i>коэффициента загрузки</i>
<code>LinkedHashMap</code>	Похожа на <code>HashMap</code> , однако при переборе выдает пары в порядке их добавления, или согласно принципу LRU («наименее используемые идут первыми», <i>least-recently-used</i>). Лишь немногим медленнее <code>HashMap</code> , за исключением процесса перечисления элементов, где она быстрее за счет внутреннего связанного списка, отвечающего за хранение порядка добавления элементов
<code>TreeMap</code>	Реализация карты на базе красно-черного дерева. При просмотре ключей или пар видно, что они соблюдают определенный порядок (который определяется интерфейсами <code>Comparable</code> или <code>Comparator</code>). Область применения карты <code>TreeMap</code> — выдача результатов в отсортированном виде. Карта <code>TreeMap</code> — единственная карта с методом <code>subMap()</code> , который позволяет выделять из карты некоторую ее часть
<code>WeakHashMap</code>	Карта, состоящая из «слабых» ключей, которые не препятствуют освобождению объектов, на которые ссылается карта; разработана для решения определенного класса задач. Если за пределами карты ссылок на ключ нет, он может быть удален уборщиком мусора
<code>ConcurrentHashMap</code>	Потоково-безопасная версия <code>Map</code> , не использующая синхронизационную блокировку (см. главу 21)
<code>IdentityHashMap</code>	Хеш-таблица, использующая для сравнения ключей оператор <code>==</code> вместо метода <code>equals()</code> . Применяется в особых случаях, не для рядовых целей

Хеширование — самый распространенный способ хранения элементов в карте. Позднее мы рассмотрим его более подробно.

Требования к ключам, используемым в `Map`, не отличаются от требований к элементам `Set`. Работа с ними уже была показана в примере `TypesForSets.java`. Для каждого ключа должен быть определен метод `equals()`. Если ключ используется в хешированной версии `Map`, для него также должен быть определен метод `hashCode()`. Ключи, используемые в `TreeMap`, должны реализовать `Comparable`.

Следующий пример демонстрирует операции, доступные через интерфейс `Map`, используя определенный ранее набор данных `CountingMapData`:

```
//: containers/Maps.java
// Операции с картами.
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;
```



```

public class Maps {
    public static void printKeys(Map<Integer,String> map) {
        println("Size = " + map.size() + ", ");
        println("Keys: ");
        print(map.keySet()); // Создание множества ключей
    }
    public static void test(Map<Integer,String> map) {
        print(map.getClass().getSimpleName());
        map.putAll(new CountingMapData(25));
        // Map реализует поведение 'Set' для ключей:
        map.putAll(new CountingMapData(25));
        printKeys(map);
        // Создание коллекции значений:
        println("Values: ");
        print(map.values());
        print(map);
        print("map.containsKey(11): " + map.containsKey(11));
        print("map.get(11): " + map.get(11));
        print("map.containsValue(\"F0\"):" +
            + map.containsValue("F0"));
        Integer key = map.keySet().iterator().next();
        print("First key in map: " + key);
        map.remove(key);
        printKeys(map);
        map.clear();
        print("map.isEmpty(): " + map.isEmpty());
        map.putAll(new CountingMapData(25));
        // Операции с Set изменяют Map:
        map.keySet().removeAll(map.keySet());
        print("map.isEmpty(): " + map.isEmpty());
    }
    public static void main(String[] args) {
        test(new HashMap<Integer,String>());
        test(new TreeMap<Integer,String>());
        test(new LinkedHashMap<Integer,String>());
        test(new IdentityHashMap<Integer,String>());
        test(new ConcurrentHashMap<Integer,String>());
        test(new WeakHashMap<Integer,String>());
    }
} /* Output:
HashMap
Size = 25, Keys: [15, 8, 23, 16, 7, 22, 9, 21, 6, 1, 14,
24, 4, 19, 11, 18, 3, 12, 17, 2, 13, 20, 10, 5, 0]
Values: [P0, I0, X0, Q0, H0, W0, J0, V0, G0, B0, O0, Y0,
E0, T0, L0, S0, D0, M0, R0, C0, N0, U0, K0, F0, A0]
{15=P0, 8=I0, 23=X0, 16=Q0, 7=H0, 22=W0, 9=J0, 21=V0, 6=G0,
1=B0, 14=O0, 24=Y0, 4=E0, 19=T0, 11=L0, 18=S0, 3=D0, 12=M0,
17=R0, 2=C0, 13=N0, 20=U0, 10=K0, 5=F0, 0=A0}
map.containsKey(11): true
map.get(11): L0
map.containsValue("F0"): true
First key in map: 15
Size = 24, Keys: [8, 23, 16, 7, 22, 9, 21, 6, 1, 14, 24, 4,
19, 11, 18, 3, 12, 17, 2, 13, 20, 10, 5, 0]
map.isEmpty(): true
map.isEmpty(): true
...
*///:~

```

Метод `printKeys()` показывает, как получить коллекцию (`Collection`) из карты (`Map`). Метод `keySet()` производит множество (`Set`), элементами которого являются все ключи таблицы. Благодаря усовершенствованной поддержке вывода в Java SE5 можно просто вывести результат вызова метода `values()`, который производит коллекцию `Collection` со всеми значениями карты (`Map`). (Заметьте, что ключи карты должны быть уникальны, в то время как значения могут повторяться.) Так как полученные коллекции значений и ключей неразрывно связаны с картой, любое проведенное в них изменение немедленно отразится на ней.

Остальная часть программы демонстрирует простые операции с картами (`Map`), а затем проверяет существующие типы карт.

14. (3) Покажите, что `java.util.Properties` работает в приведенной программе.

SortedMap

Если вы используете отсортированную карту типа `SortedMap` (к ней относится один класс `TreeMap`), все ключи гарантированно следуют в определенном порядке, что позволяет определить в интерфейсе `SortedMap` несколько дополнительных методов.

- ❑ `Comparator comparator()`: возвращает интерфейс `Comparator` для сравнения объектов данной карты или `null` для естественного упорядочивания.
- ❑ `T firstKey()`: производит наименьшее значение ключа.
- ❑ `T lastKey()`: производит наибольшее значение ключа.
- ❑ `SortedMap subMap(fromKey, toKey)`: возвращает подкарту данной карты (`Map`), начиная с ключа `fromKey` (включительно) и заканчивая ключом `toKey` (не включая).
- ❑ `SortedMap headMap(toKey)`: возвращает подкарту данной карты `Map`, содержащую ключи, меньшие `toKey`.
- ❑ `SortedMap tailMap(fromKey)`: возвращает подкарту данной карты `Map`, содержащую ключи, большие или равные `fromKey`.

Вот пример, весьма похожий на `SortedSetDemo.java`, он демонстрирует дополнительные возможности класса `TreeMap`:

```
//: containers/SortedMapDemo.java
// Операции с TreeMap.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;
public class SortedMapDemo {
    public static void main(String[] args) {
        TreeMap<Integer,String> sortedMap =
            new TreeMap<Integer,String>(new CountingMapData(10));
        print(sortedMap);
        Integer low = sortedMap.firstKey();
        Integer high = sortedMap.lastKey();
        print(low);
        print(high);
        Iterator<Integer> it = sortedMap.keySet().iterator();
        for(int i = 0; i <= 6; i++) {
```

```

        if(i == 3) low = it.next();
        if(i == 6) high = it.next();
        else it.next();
    }
    print(low);
    print(high);
    print(sortedMap.subMap(low, high));
    print(sortedMap.headMap(high));
    print(sortedMap.tailMap(low));
}
} /* Output:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0,
9=J0}
0
9
3
7
{3=D0, 4=E0, 5=F0, 6=G0}
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0}
{3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0}
*///:~

```

Здесь пары хранятся в порядке сортировки по ключам. Так как в `TreeMap` существует некоторое подобие порядка, концепция «позиции» становится более логичной, и появляется возможность получения первых и последних элементов, а также подкарт.

LinkedHashMap

Карта `LinkedHashMap` производит хеширование для ускорения работы, но при этом также поддерживает порядок вставки пар объектов при переборе (например, `System.out.println()` выполняет перебор, и вы видите его результаты). Вдобавок конструктор позволяет настроить `LinkedHashMap` в соответствии с принципом выборки «наименее используемые идут первыми» (LRU, least-recently-used). Алгоритм учитывает частоту доступа к элементам, и те из них, что востребовались реже других, помещаются в начало списка. Это помогает создавать программы, которые в целях экономии периодически высвобождают ресурсы. Вот простой пример, иллюстрирующий обе эти характеристики:

```

//: containers/LinkedHashMapDemo.java
// Операции с LinkedHashMap.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class LinkedHashMapDemo {
    public static void main(String[] args) {
        LinkedHashMap<Integer,String> linkedMap =
            new LinkedHashMap<Integer,String>(
                new CountingMapData(9));
        print(linkedMap);
        // Порядок LRU:
        linkedMap =
            new LinkedHashMap<Integer,String>(16, 0.75f, true);
        linkedMap.putAll(new CountingMapData(9));
        print(linkedMap);
    }
}

```

продолжение ↗

```

    for(int i = 0; i < 6; i++) // Обращения к элементам:
        linkedMap.get(i);
    print(linkedMap);
    linkedMap.get(0);
    print(linkedMap);
}
} /* Output:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0}
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0}
{6=G0, 7=H0, 8=I0, 0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0}
{6=G0, 7=H0, 8=I0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 0=A0}
*///:~

```

Результат работы программы показывает, что пары действительно перебираются в порядке их вставки, даже при использовании LRU. Однако после обращения только к первым шести элементам последние три сместились в начало списка. Когда затем был еще раз запрошен элемент «0», он оказался в самом конце списка.

Хеширование и хеш-коды

В примерах главы 11 в качестве ключей `HashMap` использовались уже существующие классы. Эти примеры работали, так как готовые классы содержат все необходимое для правильной работы класса в качестве ключа хеш-таблицы.

Часто ошибки происходят, когда разработчик сам пишет класс, который должен стать ключом `HashMap`, и забывает реализовать весь необходимый служебный код. Например, рассмотрим систему прогноза погоды, которая ставит в соответствие объекты `Groundhog` (сурок) объектам `Prediction` (прогноз погоды). Все выглядит довольно просто — вы создаете два класса: `Groundhog` для ключей таблицы и `Prediction` для сопоставленного ему значения:

```

//: containers/Groundhog.java
// Выглядит весьма правдоподобно, но не работает.

```

```

public class Groundhog {
    protected int number;
    public Groundhog(int n) { number = n; }
    public String toString() {
        return "Groundhog #" + number;
    }
} ///:~

```

```

//: containers/Prediction.java
// Прогнозирование погоды по поведению сурка.
import java.util.*;

```

```

public class Prediction {
    private static Random rand = new Random(47);
    private boolean shadow = rand.nextDouble() > 0.5;
    public String toString() {
        if(shadow)
            return "Six more weeks of Winter!";
        else
            return "Early Spring!";
    }
} ///:~

```

```

//: containers/SpringDetector.java
// Какая же будет погода?
import java.lang.reflect.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class SpringDetector {
    // Используем Groundhog или класс, производный от него:
    public static <T extends Groundhog>
    void detectSpring(Class<T> type) throws Exception {
        Constructor<T> ghog = type.getConstructor(int.class);
        Map<Groundhog, Prediction> map =
            new HashMap<Groundhog, Prediction>();
        for(int i = 0; i < 10; i++)
            map.put(ghog.newInstance(i), new Prediction());
        print("map = " + map);
        Groundhog gh = ghog.newInstance(3);
        print("Looking up prediction for " + gh);
        if(map.containsKey(gh))
            print(map.get(gh));
        else
            print("Key not found: " + gh);
    }
    public static void main(String[] args) throws Exception {
        detectSpring(Groundhog.class);
    }
} /* Output:
map = {Groundhog #3=Early Spring!, Groundhog #7=Early
Spring!, Groundhog #5=Early Spring!, Groundhog #9=Six more
weeks of Winter!, Groundhog #8=Six more weeks of Winter!,
Groundhog #0=Six more weeks of Winter!, Groundhog #6=Early
Spring!, Groundhog #4=Six more weeks of Winter!, Groundhog
#1=Six more weeks of Winter!, Groundhog #2=Early Spring!}
Looking up prediction for Groundhog #3
Key not found: Groundhog #3
*///:~

```

Каждому объекту `Groundhog` присваивается порядковый номер, поэтому поиск объекта `Prediction` в `HashMap` может выполняться по принципу «найти прогноз `Prediction`, ассоциированный с местоположением `Groundhog` под номером 3». Класс `Prediction` содержит логическую переменную (`boolean`), которая инициализируется с помощью метода `java.util.random()`, а также метод `toString()`, который выводит полученный результат. В методе `detectSpring()` используется отражение для создания объектов класса `Groundhog` или производных от него. Так нам будет удобнее перейти к усовершенствованному объекту `Groundhog`, решающему возникшую проблему.

Карта `HashMap` заполняется объектами `Groundhog` и ассоциированными с ними объектами `Prediction`. Затем она выводится, чтобы вы смогли проверить ее содержимое. После этого объект `Groundhog` под номером 3 становится ключом при поиске прогноза `Prediction` для местоположения `Groundhog` №3 (мы видим, что оно присутствует в `Map`).

Все выглядит просто, но тем не менее не работает. Проблема состоит в том, что класс `Groundhog` унаследован от общего корневого класса `Object`, а для генерирования хеш-кода объекта используется реализация `hashCode()` этого класса. По умолчанию этот метод возвращает адрес объекта. Таким образом, хеш-код первого экземпляра `Groundhog(3)`

не совпадает с хеш-кодом второго экземпляра `Groundhog(3)`, который применялся в качестве ключа при поиске.

Можно предположить, что необходимо просто переопределить метод `hashCode()`, написав в классе свою собственную версию. Но и это не поможет до тех пор, пока вы не сделаете еще одну вещь: не переопределите метод `equals()`, который также является частью класса `Object`. Этот метод используется картой `HashMap` для проверки того, равен ли ваш ключ какому-либо ключу из содержащихся в ней.

«Правильный» метод `equals()` должен удовлетворять следующим пяти условиям:

- ❑ рефлексивность: для любого `x` вызов `x.equals(x)` должен возвращать `true`;
- ❑ симметричность: для любых `x` и `y` вызов `x.equals(y)` должен возвращать `true`, если и только если `y.equals(x)` возвращает `true`;
- ❑ транзитивность: для любых `x`, `y` и `z`, если `x.equals(y)` возвращает `true` и `y.equals(z)` возвращает `true`, тогда `x.equals(z)` должно вернуть `true`;
- ❑ стабильность: для любых `x` и `y` многократные вызовы `x.equals(y)` согласованно возвращают `true` или согласованно возвращают `false` (при условии, что информация, используемая при сравнениях объектов, не изменяется);
- ❑ для любых `x`, отличных от `null`, `x.equals(null)` должно возвращать `false`.

Метод `equals()` в классе `Object` по умолчанию сводится к сравнению адресов, поэтому один объект `Groundhog(3)` никогда не будет равен другому объекту `Groundhog(3)`. Таким образом, чтобы задействовать ваши собственные классы в качестве ключей `HashMap`, необходимо переопределить и метод `hashCode()`, и метод `equals()`, как показано в следующем примере, который исправляет ошибку предыдущей программы:

```
//: containers/Groundhog2.java
// Класс, используемый в качестве ключа HashMap,
// должен переопределять hashCode() и equals().
```

```
public class Groundhog2 extends Groundhog {
    public Groundhog2(int n) { super(n); }
    public int hashCode() { return number; }
    public boolean equals(Object o) {
        return o instanceof Groundhog2 &&
            (number == ((Groundhog2)o).number);
    }
} ///:~
```

```
//: containers/SpringDetector2.java
// Работоспособный класс ключа
```

```
public class SpringDetector2 {
    public static void main(String[] args) throws Exception {
        SpringDetector.detectSpring(Groundhog2.class);
    }
} /* Output:
```

```
map = {Groundhog #2=Early Spring!, Groundhog #4=Six more
weeks of Winter!, Groundhog #9=Six more weeks of Winter!,
Groundhog #8=Six more weeks of Winter!, Groundhog #6=Early
Spring!, Groundhog #1=Six more weeks of Winter!, Groundhog
#3=Early Spring!, Groundhog #7=Early Spring!, Groundhog
```

```
#5=Early Spring!, Groundhog #0=Six more weeks of Winter!}
Looking up prediction for Groundhog #3
Early Spring!
*///:~
```

Метод `Groundhog2.hashCode()` использует в качестве хеш-кода порядковый номер сурка. Здесь программист обязан следить за тем, чтобы два различных места не имели одинакового порядкового номера. От метода `hashCode()` не требуется возвращения строго уникального хеш-кода (об этом еще будет сказано в этой главе), но вот метод `equals()` должен однозначно определять, эквивалентны ли два объекта. Здесь метод `equals()` основывается на номере сурка; таким образом, если два объекта `Groundhog2` существуют как ключи в `HashMap` под одним и тем же номером, объекты будут считаться неравными.

Хотя кажется, что метод `equals()` просто проверяет, является ли его аргумент объектом типа `Groundhog2` (с помощью ключевого слова `instanceof`, которое более подробно описывалось в главе 14), на самом деле выражение с `instanceof` незаметно выполняет и вторую миссию: проверяет, не является ли ссылка на объект нулевой (`null`), поскольку ключевое слово `instanceof` при сравнении объекта со ссылкой `null` возвращает значение «ложь» (`false`). Предполагая, что переданный аргумент имеет нужный тип и не является `Null`, метод `equals()` проводит сравнение, основанное на фактических числах `ghNumber`. На этот раз при запуске программы вы убедитесь, что результат получается верный.

При написании собственных классов, использующих `HashSet`, необходимо уделить внимание тем же вопросам, что возникают при создании ключей для `HashMap`.

Понимание `hashCode()`

Рассмотренный пример — только начало. Он показывает, что если не переопределить методы `equals()` и `hashCode()` для ключей, применяемых в хешированных структурах данных (`HashSet`, `HashMap`, `LinkedHashSet` и `LinkedHashMap`), они просто не смогут нормально функционировать. Но чтобы *качественно* решать такие задачи, необходимо понимать, что происходит внутри хешированных структур данных.

Во-первых, для чего вообще нужно хеширование? Для поиска объекта по другому объекту. Но это можно сделать с помощью `TreeMap` или реализовать собственную карту (`Map`). В отличие от реализации с хешированием следующий пример реализует `Map` на базе двух контейнеров `ArrayList`. В отличие от `AssociativeArray.java`, он включает полную реализацию интерфейса `Map`, чем и объясняется присутствие `entrySet()`:

```
//: containers/SlowMap.java
// Реализация Map на базе списков ArrayList.
import java.util.*;
import net.mindview.util.*;

public class SlowMap<K,V> extends AbstractMap<K,V> {
    private List<K> keys = new ArrayList<K>();
    private List<V> values = new ArrayList<V>();
    public V put(K key, V value) {
        V oldValue = get(key); // Старое значение или null
        if(!keys.contains(key)) {
            keys.add(key);
            values.add(value);
        } else
```

продолжение ➤

```

        values.set(keys.indexOf(key), value);
        return oldValue;
    }
    public V get(Object key) { // key относится к типу Object, не K
        if(!keys.contains(key))
            return null;
        return values.get(keys.indexOf(key));
    }
    public Set<Map.Entry<K,V>> entrySet() {
        Set<Map.Entry<K,V>> set= new HashSet<Map.Entry<K,V>>();
        Iterator<K> ki = keys.iterator();
        Iterator<V> vi = values.iterator();
        while(ki.hasNext())
            set.add(new MapEntry<K,V>(ki.next(), vi.next()));
        return set;
    }
    public static void main(String[] args) {
        SlowMap<String,String> m= new SlowMap<String,String>();
        m.putAll(Countries.capitals(15));
        System.out.println(m);
        System.out.println(m.get("BULGARIA"));
        System.out.println(m.entrySet());
    }
} /* Output:
{CAMEROON=Yaounde, CHAD=N'djamena, CONGO=Brazzaville, CAPE
VERDE=Praia, ALGERIA=Algiers, COMOROS=Moroni, CENTRAL
AFRICAN REPUBLIC=Bangui, BOTSWANA=Gaberone,
BURUNDI=Bujumbura, BENIN=Porto-Novo, BULGARIA=Sofia,
EGYPT=Cairo, ANGOLA=Luanda, BURKINA FASO=Ouagadougou,
DJIBOUTI=Djibouti}
Sofia
[CAMEROON=Yaounde, CHAD=N'djamena, CONGO=Brazzaville, CAPE
VERDE=Praia, ALGERIA=Algiers, COMOROS=Moroni, CENTRAL
AFRICAN REPUBLIC=Bangui, BOTSWANA=Gaberone,
BURUNDI=Bujumbura, BENIN=Porto-Novo, BULGARIA=Sofia,
EGYPT=Cairo, ANGOLA=Luanda, BURKINA FASO=Ouagadougou,
DJIBOUTI=Djibouti]
*///:~

```

Метод `put()` просто помещает ключи и значения в соответствующий объект `ArrayList`. В соответствии с правилами интерфейса `Map` он должен возвращать старый ключ или `null`, если старый ключ отсутствует.

Также в соответствии со спецификацией `Map` метод `get()` выдает `null`, если ключ отсутствует в `SlowMap`. Если ключ существует, то он используется для получения числового индекса, обозначающего его позицию в списке `keys`, а число — для получения ассоциированного значения из списка `values` индексированием. Обратите внимание: в `get()` ключ `key` относится к типу `Object`, а не к параметризованному типу `K`, как можно было бы ожидать (и как было в `AssociativeArray.java`). Это следствие того, что обобщения были включены в язык `Java` на поздней стадии, — если бы обобщения присутствовали в языке с самого начала, метод `get()` мог бы использовать тип своего параметра.

Метод `Map.entrySet()` должен производить множество объектов `Map.Entry`. Однако `Map.Entry` — интерфейс, описывающий структуру, зависимую от реализации, поэтому если вы захотите создать собственный тип `Map`, то вам также придется определить реализацию `Map.Entry`:


```

//: containers/MapEntry.java
// Простая реализация Map.Entry
// для пробной реализации Map.
import java.util.*;

public class MapEntry<K,V> implements Map.Entry<K,V> {
    private K key;
    private V value;
    public MapEntry(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey() { return key; }
    public V getValue() { return value; }
    public V setValue(V v) {
        V result = value;
        value = v;
        return result;
    }
    public int hashCode() {
        return (key==null ? 0 : key.hashCode()) ^
            (value==null ? 0 : value.hashCode());
    }
    public boolean equals(Object o) {
        if(!(o instanceof MapEntry)) return false;
        MapEntry me = (MapEntry)o;
        return
            (key == null ?
                me.getKey() == null : key.equals(me.getKey())) &&
            (value == null ?
                me.getValue()== null : value.equals(me.getValue()));
    }
    public String toString() { return key + "=" + value; }
} ///:~

```

Здесь очень простой класс `MapEntry` хранит и получает ключи и значения. Он используется в `entrySet()` для создания множества (`Set`) пар «ключ-значение». Обратите внимание: `entrySet()` использует `HashSet` для хранения пар, а `MapEntry` использует упрощенный подход, просто используя вызов `hashCode()` для ключа. И хотя это решение выглядит очень просто и вроде бы работает в тривиальном тесте из `SlowMap.main()`, оно не может считаться правильной реализацией из-за копирования ключей и значений. Правильная реализация `entrySet()` должна создавать представление для `Map` (вместо копирования), через которое может производиться модификация исходной карты (чего копия сделать не позволяет). Одно из возможных решений представлено в упражнении 16.

Обратите внимание: метод `equals()` в `MapEntry` должен проверять как ключи, так и значения. Смысл метода `hashCode()` будет описан ниже.

Представление содержимого `SlowMap` в формате `String` автоматически создается методом `toString()`, определенным в `AbstractMap`.

В `SlowMap.main()` происходит загрузка карты `SlowMap` с последующим выводом ее содержимого. Вызов `get()` показывает, что оно работает.

15. (1) Повторите упражнение 13 с использованием `SlowMap`.

16. (7) Примените тесты из `Maps.java` к `SlowMap` и проверьте, работает ли класс. Исправьте в `SlowMap` все, что работает некорректно.
17. (2) Реализуйте оставшуюся часть интерфейса `Map` для `SlowMap`.
18. (3) Взяв за образец `SlowMap.java`, создайте аналогичную реализацию множества `SlowSet`.

Хеширование ради скорости

Пример `SlowMap.java` показывает, что создать новую карту не так уж сложно. Но, как и предполагает ее имя, карта `SlowMap` («медленная карта») не отличается быстротой, поэтому при наличии более эффективных альтернатив вы вряд ли станете ее использовать. Проблема кроется в процессе поиска ключа: ключи находятся в неупорядоченном состоянии, с ними приходится использовать простой линейный поиск, а неповоротливее процесса на свете просто не существует.

«Изюминка» хеширования — именно скорость: хеширование позволяет провести поиск весьма оперативно. Так как «узким местом» является поиск ключа, одним из решений этой задачи могло бы стать поддержание всех ключей в упорядоченном виде, чтобы при поиске заставить работать метод `Collections.binarySearch()` (одно из упражнений в конце этой главы основано на таком действии).

По сути, при хешировании необходимо всего лишь хранить ключи *где-то*, где их можно потом быстро найти. Как вы не раз могли убедиться, самая быстрая структура для хранения группы элементов — массив, поэтому он и используется для хранения информации о ключах (обязательно отметьте, что я сказал «информации о ключах», а не о «ключях» самих по себе). В этой же главе мы убедились, что массивы, после их создания и выделения хранилища, больше не меняют свой размер, поэтому возникает новая проблема: в карте (`map`) мы хотели бы содержать неограниченное количество значений, однако число ключей ограничено размером массива, так что же делать?

Ответ: в массиве не будут храниться ключи. По объекту-ключу будет строиться число, которое будет использоваться для индексирования массива. Это число — *хеш-код* (`hash code`) — возвращается методом `hashCode()` (в компьютерных терминах такой метод называется *хеш-функцией*), который определен в классе `Object` и, предположительно, переопределен вашим классом. Для решения проблемы с потенциально недостаточным размером массива разрешается соотносить один индекс массива с несколькими ключами, то есть не исключены *коллизии*. Из-за этого неважно, насколько вместителен массив — каждый ключ сможет найти в нем для себя подходящее место.

Таким образом, процесс поиска значения начинается с вычисления хеш-кода и поиска его в массиве. Если бы вы могли гарантировать, что «накладок» в массиве нет (что могло бы случиться, будь у вас ограниченное количество хранимых элементов), то получилась бы *идеальная хеш-функция*, но это уже особый случай¹. Во всех других случаях с коллизиями справляется *внешнее связывание*: элемент массива содержит не конкретное значение, но указывает на цепочку значений. В этой цепочке обнаружение

¹ Идеальная хеш-функция реализована в классах `Java SE5 EnumMap` и `EnumSet`, потому что перечисления (`enum`) определяют фиксированное количество экземпляров (см. главу 19).

производится с помощью обычного линейного алгоритма, с помощью метода `equals()`. Конечно, эта часть поиска гораздо медленнее, но если хеш-функция работает хорошо, на каждую ячейку массива придется максимум два-три значения. Поэтому вместо поиска в списке всех ключей вы быстро переходите к нужной позиции, где остается сравнить несколько вхождений для выявления искомого значения. Это гораздо эффективнее, и именно потому таблица `HashMap` такая быстрая.

Зная основы хеширования, можно реализовать простую хешированную карту (`Map`):

```

//: containers/SimpleHashMap.java
// Демонстрация карты с хешированием.
import java.util.*;
import net.mindview.util.*;

public class SimpleHashMap<K,V> extends AbstractMap<K,V> {
    // В качестве размера хеш-таблицы следует выбирать
    // простое число, чтобы обеспечить равномерность распределения:
    static final int SIZE = 997;
    // Физический массив обобщений создать нельзя, но можно
    // прийти к нему через восходящее преобразование:
    @SuppressWarnings("unchecked")
    LinkedList<MapEntry<K,V>>[] buckets =
        new LinkedList[SIZE];
    public V put(K key, V value) {
        V oldValue = null;
        int index = Math.abs(key.hashCode()) % SIZE;
        if(buckets[index] == null)
            buckets[index] = new LinkedList<MapEntry<K,V>>();
        LinkedList<MapEntry<K,V>> bucket = buckets[index];
        MapEntry<K,V> pair = new MapEntry<K,V>(key, value);
        boolean found = false;
        ListIterator<MapEntry<K,V>> it = bucket.listIterator();
        while(it.hasNext()) {
            MapEntry<K,V> iPair = it.next();
            if(iPair.getKey().equals(key)) {
                oldValue = iPair.getValue();
                it.set(pair); // Заменяем старое значение новым
                found = true;
                break;
            }
        }
        if(!found)
            buckets[index].add(pair);
        return oldValue;
    }
    public V get(Object key) {
        int index = Math.abs(key.hashCode()) % SIZE;
        if(buckets[index] == null) return null;
        for(MapEntry<K,V> iPair : buckets[index])
            if(iPair.getKey().equals(key))
                return iPair.getValue();
        return null;
    }
    public Set<Map.Entry<K,V>> entrySet() {
        Set<Map.Entry<K,V>> set= new HashSet<Map.Entry<K,V>>();
        for(LinkedList<MapEntry<K,V>> bucket : buckets) {
            if(bucket == null) continue;

```

продолжение ➤

```

        for(MapEntry<K,V> mpair : bucket)
            set.add(mpair);
    }
    return set;
}
public static void main(String[] args) {
    SimpleHashMap<String,String> m =
        new SimpleHashMap<String,String>();
    m.putAll(Countries.capitals(25));
    System.out.println(m);
    System.out.println(m.get("ERITREA"));
    System.out.println(m.entrySet());
}
} /* Output:
{CAMEROON=Yaounde, CONGO=Brazzaville, CHAD=N'djamena, COTE
D'IVOIR (IVORY COAST)=Yamoussoukro, CENTRAL AFRICAN
REPUBLIC=Bangui, GUINEA=Conakry, BOTSWANA=Gaberone,
BISSAU=Bissau, EGYPT=Cairo, ANGOLA=Luanda, BURKINA
FASO=Ouagadougou, ERITREA=Asmara, THE GAMBIA=Banjul,
KENYA=Nairobi, GABON=Libreville, CAPE VERDE=Praia,
ALGERIA=Algiers, COMOROS=Moroni, EQUATORIAL GUINEA=Malabo,
BURUNDI=Bujumbura, BENIN=Porto-Novo, BULGARIA=Sofia,
GHANA=Accra, DJIBOUTI=Djibouti, ETHIOPIA=Addis Ababa}
Asmara
[CAMEROON=Yaounde, CONGO=Brazzaville, CHAD=N'djamena, COTE
D'IVOIR (IVORY COAST)=Yamoussoukro, CENTRAL AFRICAN
REPUBLIC=Bangui, GUINEA=Conakry, BOTSWANA=Gaberone,
BISSAU=Bissau, EGYPT=Cairo, ANGOLA=Luanda, BURKINA
FASO=Ouagadougou, ERITREA=Asmara, THE GAMBIA=Banjul,
KENYA=Nairobi, GABON=Libreville, CAPE VERDE=Praia,
ALGERIA=Algiers, COMOROS=Moroni, EQUATORIAL GUINEA=Malabo,
BURUNDI=Bujumbura, BENIN=Porto-Novo, BULGARIA=Sofia,
GHANA=Accra, DJIBOUTI=Djibouti, ETHIOPIA=Addis Ababa]
*///:~

```

Так как «ячейки» хеш-таблицы часто называют *узловыми группами* (buckets), массив, представляющий фактическую таблицу, называют buckets. Чтобы распределение было более равномерным, количество узлов обычно выбирают из простых чисел¹. Заметьте, что таблица — это массив списков LinkedList, поэтому она автоматически приспособлена к возникновению коллизий — каждый новый элемент группы просто добавляется в конец списка соответствующей группы. И хотя Java не позволяет создавать массивы обобщений, вы можете создать *ссылку* на такой массив. В данном случае удобно выполнить восходящее преобразование к такому массиву для предотвращения лишних приведений типа в коде.

В методе put() для ключа вызывается метод hashCode(), а полученное значение преобразуется в положительное число. Чтобы полученный результат ограничивался размером

¹ Как оказывается, простые числа — это не самый лучший размер для хеш-таблиц, и последние реализации хеш-таблиц в Java используют размер, равный степени числа 2 (после тщательного тестирования). Операции деления или получения остатка на современных процессорах выполняются относительно медленно, а для метода возведения ключа в квадрат вместо деления можно использовать маскирование крайних разрядов. Метод get() вызывается чаще других, поэтому операция получения остатка (%) отнимает большую часть времени, а метод «степеней двойки» устраняет эти затраты (правда, это может повлиять на работу некоторых методов hashCode()).

массива `buckets`, для хеш-кода применяется оператор остатка от деления. Если в полученном узле таблицы находится ссылка `null`, значит, список элементов еще пуст, и для этого узла создается новый список `LinkedList` для хранения элементов. Впрочем, чаще процесс протекает по-другому: в списке производится поиск дубликатов, и если они есть, старое значение помещается в `oldValue`, а на его место помещается новое значение. Флаг `found` следит за тем, была ли найдена старая пара «ключ-значение», если она не найдена, то новая пара добавляется к концу списка.

Метод `get()` вычисляет индекс в массиве `buckets` так же, как это делается в `put()` (это важно, чтобы программа пришла к той же ячейке). Если в ней имеется список элементов `LinkedList`, в нем проводится окончательный поиск.

Учтите, что реализация не оптимизирована по быстродействию; она всего лишь демонстрирует операции, выполняемые хеширующей картой. Если вас интересует оптимизированная реализация, обратитесь к исходному коду `java.util.HashMap`. Кроме того, для простоты `SimpleHashMap` использует в `entrySet()` тот же подход, что и в `SlowMap`, — чрезмерно упрощенный и не подходящий для карты общего назначения.

19. (1) Повторите упражнение 13 для `SimpleHashMap`.
20. (3) Измените реализацию `SimpleHashMap` так, чтобы она сообщала о коллизиях. Чтобы протестировать результат, добавьте один набор данных дважды, чтобы спровоцировать коллизию.
21. (2) Измените реализацию `SimpleHashMap` так, чтобы она сообщала количество необходимых «проб» при возникновении коллизий. Иначе говоря, сколько вызовов `next()` должно выполняться итераторами, обходящими списки `LinkedList` в поисках совпадений?
22. (4) Реализуйте методы `clear()` и `remove()` для `SimpleHashMap`.
23. (3) Реализуйте остаток интерфейса `Map` для `SimpleHashMap`.
24. (5) По образцу `SimpleHashMap.java` создайте и протестируйте `SimpleHashSet`.
25. (6) Вместо того чтобы использовать `ListIterator` для каждой узловой группы, измените класс `MapEntry` так, чтобы он представлял собой автономный односвязный список (каждый объект `MapEntry` должен содержать ссылку на следующий объект `MapEntry`). Измените остальной код `SimpleHashMap.java`, чтобы новое решение работало правильно.

Переопределение `hashCode()`

Теперь, когда вы понимаете, как работает хеширование, вам будет проще разобраться в реализации `hashCode()`.

Прежде всего вы не управляете созданием конкретного значения, используемого для индексирования массива узловых групп. Оно зависит от емкости конкретного объекта `HashMap`, и эта емкость может варьироваться в зависимости от текущего заполнения контейнера и *коэффициента загрузки* таблицы (этот термин будет объяснен позднее). Значение, возвращаемое вашим методом `hashCode()`, будет затем обработано для

получения индекса узла (в классе `SimpleHashMap` обработка сводится к ограничению хеш-кода размерами массива).

Наиболее важная характеристика метода `hashCode()` — возвращение одного и того же хеш-кода для определенного объекта при каждом вызове метода. Если ваш объект при помещении в `HashMap` методом `put()` возвращает один хеш-код, а при получении значения из таблицы методом `get()` — другой, выборка объектов из таблицы станет невозможной. Поэтому, если метод ключа `hashCode()` зависит от изменяемых данных объекта, пользователь должен знать, что изменение этих данных повлечет за собой обновление ключа, поскольку будет сгенерирован другой хеш-код.

Вдобавок вы, скорее всего, *не будете* производить хеш-код своего объекта на основе его уникальных данных — в особенности неудобно использовать в методе `hashCode()` ссылку `this`, дающую адрес объекта, так как вы не сможете получить ключ, идентичный тому, который использовался при вызове `put()` для исходной пары «ключ-значение». Именно такая проблема возникла в программе `SpringDetector.java`, где по умолчанию метод `hashCode()` возвращает *именно* адрес объекта. Следовательно, для получения хеш-кода объекта стоит брать такие данные, которые идентифицируют его неким содержательным образом.

Проиллюстрировать вышесказанное может класс `String`. У строк `String` есть одна особенность: если в программе существует несколько объектов `String`, содержащих одинаковые последовательности символов, то все ссылки на эти строки указывают на один и тот же участок памяти. Поэтому важно, чтобы метод `hashCode()` у двух разных экземпляров строк, созданных выражением `new String("hello")`, возвращал одинаковые хеш-коды. Это можно увидеть, запустив следующую программу:

```
//: containers/StringHashCode.java

public class StringHashCode {
    public static void main(String[] args) {
        String[] hellos = "Hello Hello".split(" ");
        System.out.println(hellos[0].hashCode());
        System.out.println(hellos[1].hashCode());
    }
} /* Output: (Sample)
69609650
69609650
*///:~
```

Метод `hashCode()` для класса `String` базируется на содержимом строки.

Для достижения своей цели метод `hashCode()` должен быть быстрым и содержательным, то есть он обязан возвращать хеш-код, вычисленный на основании внутреннего состояния объекта. Помните, что от этого значения не следует ждать обязательной уникальности — лучше склоняться к скорости, чем к уникальности, — но объект должен однозначно идентифицироваться комбинацией `hashCode()` и `equals()`.

Так как число, возвращаемое методом `hashCode()`, в процессе вычисления индекса узла несколько раз изменяется, диапазон возвращаемых чисел не имеет значения; они просто должны быть целыми (`int`).

Есть и еще один аспект: хорошо продуманный метод `hashCode()` возвращает равномерно распределенные значения. Если результаты метода имеют склонность к «сгучиванию», `HashMap` или `HashSet` в нескольких местах будут излишне загружены, что немедленно приведет к снижению эффективности поиска по сравнению со скоростью поиска в равномерно загруженной таблице.

В своей книге *Effective Java* (издательство Addison-Wesley, 2001) Джошуа Блош предлагает простой рецепт создания подходящего метода `hashCode()`.

1. Сохраните некоторое ненулевое значение (скажем, 17) в целочисленной (`int`) переменной с названием `result`.
2. Для каждого значимого поля `f` в вашем классе (значимое поле — это то, которое учитывает ваш метод `equals()`) вычислите целое (`int`) значение хеш-кода `c`.

Тип поля	Вычисления
<code>Boolean</code>	<code>c = (f ? 0 : 1)</code>
<code>byte</code> , <code>char</code> , <code>short</code> или <code>int</code>	<code>c = (int)f</code>
<code>long</code>	<code>c = (int)(f ^ (f >>> 32))</code>
<code>float</code>	<code>c = Float.floatToIntBits(f);</code>
<code>double</code>	<code>long l = Double.doubleToLongBits(f);</code> <code>c = (int)(l ^ (l >>> 32))</code>
<code>Object</code> , где метод <code>equals()</code> вызывает <code>equals()</code> для каждого поля	<code>c = f.hashCode()</code>
Массив	К каждому элементу применяются описанные выше правила

3. Скомбинируйте хеш-коды, полученные по указанным формулам:

```
result = 37*result + c;
```

4. Возвратите `result`.
5. Взгляните на полученный результат, возвращаемый методом `hashCode()`, и удостоверьтесь в том, что одинаковые экземпляры имеют совпадающий хеш-код.

Пример реализации этой схемы:

```
//: containers/CountedString.java
// Написание хорошей реализации hashCode().
import java.util.*;
import static net.mindview.util.Print.*;

public class CountedString {
    private static List<String> created =
        new ArrayList<String>();
    private String s;
    private int id = 0;
    public CountedString(String str) {
```

продолжение ➤

```

s = str;
created.add(s);
// id - общее количество экземпляров данной строки,
// используемых классом CountedString:
for(String s2 : created)
    if(s2.equals(s))
        id++;
}
public String toString() {
    return "String: " + s + " id: " + id +
        " hashCode(): " + hashCode();
}
public int hashCode() {
    // Очень простая схема:
    // вернуть s.hashCode() * id
    // по рецепту Джошуа Блоша:
    int result = 17;
    result = 37 * result + s.hashCode();
    result = 37 * result + id;
    return result;
}
public boolean equals(Object o) {
    return o instanceof CountedString &&
        s.equals(((CountedString)o).s) &&
        id == ((CountedString)o).id;
}
public static void main(String[] args) {
    Map<CountedString,Integer> map =
        new HashMap<CountedString,Integer>();
    CountedString[] cs = new CountedString[5];
    for(int i = 0; i < cs.length; i++) {
        cs[i] = new CountedString("hi");
        map.put(cs[i], i); // Упаковка int -> Integer
    }
    print(map);
    for(CountedString cstring : cs) {
        print("Looking up " + cstring);
        print(map.get(cstring));
    }
}
} /* Output: (Sample)
{String: hi id: 4 hashCode(): 146450=3, String: hi id: 1
hashCode(): 146447=0, String: hi id: 3 hashCode():
146449=2, String: hi id: 5 hashCode(): 146451=4, String: hi
id: 2 hashCode(): 146448=1}
Looking up String: hi id: 1 hashCode(): 146447
0
Looking up String: hi id: 2 hashCode(): 146448
1
Looking up String: hi id: 3 hashCode(): 146449
2
Looking up String: hi id: 4 hashCode(): 146450
3
Looking up String: hi id: 5 hashCode(): 146451
4
*///:~

```

Класс `CountedString` содержит строку (`String`) и переменную `id`, показывающую, сколько объектов `CountedString` содержат одинаковые строки. Подсчет строк происходит

в конструкторе, с использованием перебора статического списка `ArrayList`, в котором хранятся все созданные строки.

Оба важных метода класса — `hashCode()`, и `equals()` — дают результаты, основанные на значимых полях; если бы они опирались только на данные строки или только на переменную `id`, даже для различных объектов могли бы получиться одни и те же значения.

В методе `main()` создается несколько объектов `CountedString` с одинаковыми строками, чтобы показать, что на самом деле возникают уникальные объекты (для этого и создана переменная `id`). Также показана таблица `HashMap`, чтобы вы увидели, как хранятся внутри нее данные (без видимого порядка), а поиск каждого ключа проводится индивидуально, чтобы вы убедились в работоспособности механизма поиска.

В качестве второго примера рассмотрим класс `Individual` — базовый класс для библиотеки `typeinfo.pet` из главы 14. Класс `Individual` использовался в этой главе, но определение было отложено на будущее, чтобы вы могли понять реализацию:

```
//: typeinfo/pets/Individual.java
package typeinfo.pets;

public class Individual implements Comparable<Individual> {
    private static long counter = 0;
    private final long id = counter++;
    private String name;
    public Individual(String name) { this.name = name; }
    // Параметр 'name' не обязателен:
    public Individual() {}
    public String toString() {
        return getClass().getSimpleName() +
            (name == null ? "" : " " + name);
    }
    public long id() { return id; }
    public boolean equals(Object o) {
        return o instanceof Individual &&
            id == ((Individual)o).id;
    }
    public int hashCode() {
        int result = 17;
        if(name != null)
            result = 37 * result + name.hashCode();
        result = 37 * result + (int)id;
        return result;
    }
    public int compareTo(Individual arg) {
        // Сначала сравнить по имени класса:
        String first = getClass().getSimpleName();
        String argFirst = arg.getClass().getSimpleName();
        int firstCompare = first.compareTo(argFirst);
        if(firstCompare != 0)
            return firstCompare;
        if(name != null && arg.name != null) {
            int secondCompare = name.compareTo(arg.name);
            if(secondCompare != 0)
                return secondCompare;
        }
        return (arg.id < id ? -1 : (arg.id == id ? 0 : 1));
    }
} ///:~
```

Метод `compareTo()` использует иерархию сравнений, поэтому он создает последовательность, которая сначала сортируется по типу, затем по имени `name` (если оно есть) и в последнюю очередь возвращается к порядку создания. Следующий пример показывает, как это происходит:

```
//: containers/IndividualTest.java
import holding.MapOfList;
import typeinfo.pets.*;
import java.util.*;

public class IndividualTest {
    public static void main(String[] args) {
        Set<Individual> pets = new TreeSet<Individual>();
        for(List<? extends Pet> lp :
            MapOfList.petPeople.values())
            for(Pet p : lp)
                pets.add(p);
        System.out.println(pets);
    }
} /* Output:
[Cat Elsie May, Cat Pinkola, Cat Shackleton, Cat Stanford
aka Stinky el Negro, Cymric Molly, Dog Margrett, Mutt Spot,
Pug Louie aka Louis Snorkelstein Dupree, Rat Fizzy, Rat
Freckly, Rat Fuzzy]
*///:~
```

Так как у всех питомцев в этом примере есть имена, они сортируются сначала по типу, затем по имени внутри типа.

Написать правильные реализации `hashCode()` и `equals()` для нового класса достаточно непросто. В проекте Apache Jakarta Commons (jakarta.apache.org/commons), в пакете `lang` можно найти несколько инструментов, способных облегчить эту задачу (проект включает в себя несколько других потенциально полезных библиотек; похоже, он становится ответом сообщества программистов Java сообществу программистов C++ www.boost.org).

26. (2) Добавьте в `CountedString` поле `char`, которое также инициализируется в конструкторе. Измените методы `hashCode()` и `equals()`, чтобы в них включалось значение этого поля.
27. (3) Измените метод `hashCode()` из `CountedString.java` — удалите использование комбинации с `id`. Продемонстрируйте, что `CountedString` по-прежнему работает как ключ. В чем проблема такого решения?
28. (4) Измените код `net/mindview/util/Tuple.java` и сделайте его классом общего назначения — добавьте методы `hashCode()`, `equals()` и реализуйте `Comparable` для всех типов `Tuple`.

Выбор реализации

Вероятно, вы уже поняли, что на самом деле существуют только четыре фундаментальные разновидности контейнеров: карта (`Map`), список (`List`), множество (`Set`) и очередь

(Queue), но для каждой имеется несколько различных реализаций. Если вам необходима функциональность определенного интерфейса, на основании чего вы примете решение о подходящей для него реализации?

Чтобы ответить на этот вопрос, надо осознать, что каждый компонент имеет свои характерные свойства, преимущества и недостатки. Например, на диаграмме классов в начале этой главы видно, что контейнеры `Hashtable`, `Vector` и `Stack` не вредят старому коду. С другой стороны, в новом коде эти классы лучше вообще не применять.

Разные типы `Queue` в библиотеке Java различаются только по тому, как они получают и передают значения (о том, почему это важно, рассказано в главе 21).

В определении различий между другими контейнерами часто исходят из того, на основе каких структур данных, «физически» реализующих необходимый интерфейс, они строятся. Это значит, что, например, при использовании списков `ArrayList` и `LinkedList`, реализующих интерфейс `List`, базовые операции `List` будут работать одинаково независимо от выбранного типа. Однако класс `ArrayList` основан на массиве, в то время как класс `LinkedList` реализован как двунаправленный список: в нем хранятся отдельные объекты, связанные ссылками на предыдущий и следующий элементы списка. По этой причине список `LinkedList` хорошо подходит тогда, когда планируется выполнять много вставок и удалений из середины списка. (В нем также имеются дополнительные возможности, описанные в абстрактном классе `AbstractSequentialList`.) В противном случае список `ArrayList` обычно работает быстрее.

В качестве другого примера возьмем множества: интерфейс `Set` реализован классами `TreeSet`, `HashSet` и `LinkedHashSet`¹. Все они ведут себя по-разному: класс `HashSet` предназначен для повседневного использования и обеспечивает хорошую скорость поиска. `LinkedHashSet` хранит пары в порядке добавления, а множество `TreeSet` работает на базе карты `TreeMap` и спроектировано специально как множество, постоянно находящееся в упорядоченном состоянии. Вы выбираете реализацию в зависимости от того, что вам необходимо.

Иногда разные реализации некоторого контейнера имеют общие операции, но отличаются по производительности. В этом случае реализация выбирается на основании частоты использования той или иной операции и того, насколько быстро она должна выполняться. В таких случаях решение должно приниматься по результатам тестирования производительности.

Среда тестирования

Чтобы предотвратить дублирование кода и обеспечить логическую целостность тестирования, я выделил базовую функциональность тестирования в отдельную программную среду. Следующий код определяет базовый класс, на основе которого вы создаете список анонимных внутренних классов — по одному для каждого текста. Каждый из внутренних классов вызывается как часть процесса тестирования. Такое решение позволяет легко добавлять и удалять тесты.

¹ А также классами `EnumSet` и `CopyOnWriteArraySet`, которые являются особыми случаями. Впрочем, в этом разделе мы стараемся рассматривать более общие случаи.

Перед нами еще один пример паттерна проектирования «Шаблонный метод». Хотя мы следуем типичному для «Шаблонного метода» приему переопределения метода `Test.test()` в каждом конкретном тесте, в данном случае базовый код (который остается неизменным) выделяется в отдельный класс `Tester`¹. Тип тестируемого контейнера задается обобщенным параметром `C`:

```
//: containers/Test.java
// Среда для проведения хронометражного тестирования контейнеров.

public abstract class Test<C> {
    String name;
    public Test(String name) { this.name = name; }
    // Метод переопределяется для разных тестов.
    // Возвращает фактическое количество повторений теста.
    abstract int test(C container, TestParam tp);
} ///:~
```

В каждом объекте `Test` хранится имя теста. При вызове метода `test()` должен передаваться тестируемый контейнер с «объектом передачи данных», содержащим различные параметры конкретного теста: `size` (количество элементов в контейнере), `loops` (количество итераций в тесте) и т. д. Эти параметры могут использоваться (или не использоваться) в каждом конкретном тесте.

Для каждого контейнера выполняется серия вызовов `test()` с разными `TestParam`, поэтому `TestParam` также содержит статические методы `array()`, упрощающие создание массивов объектов `TestParam`. Первая версия `array()` получает переменный список аргументов с чередующимися значениями `size` и `loops`, а вторая получает также список, внутренними значениями которого являются строки `String`, что позволяет использовать ее для разбора аргументов командной строки:

```
//: containers/TestParam.java
// "Объект передачи данных".

public class TestParam {
    public final int size;
    public final int loops;
    public TestParam(int size, int loops) {
        this.size = size;
        this.loops = loops;
    }
    // Создание массива TestParam по списку аргументов переменной длины:
    public static TestParam[] array(int... values) {
        int size = values.length/2;
        TestParam[] result = new TestParam[size];
        int n = 0;
        for(int i = 0; i < size; i++)
            result[i] = new TestParam(values[n++], values[n++]);
        return result;
    }
    // Convert a String array to a TestParam array:
    public static TestParam[] array(String[] values) {
        int[] vals = new int[values.length];
        for(int i = 0; i < vals.length; i++)
```

¹ Кишиштоф Соболевский помог мне разобраться с обобщениями в этом примере.

```

        vals[i] = Integer.decode(values[i]);
    return array(vals);
}
} ///:~

```

Тестируемый контейнер передается вместе со списком объектов `Test` методу `Tester.run()` (перегруженные обобщенные вспомогательные методы, сокращающие объем кода, необходимого для их использования). Метод `Tester.run()` вызывает подходящий перегруженный конструктор, после чего вызывает метод `timedTest()`, который выполняет каждый тест в списке для данного контейнера. `timedTest()` повторяет каждый тест для каждого из объектов `TestParam` в `paramList`. Так как `paramList` инициализируется на основании статического массива `defaultParams`, вы можете изменить `paramList` для всех тестов переназначением `defaultParams` или же изменить `paramList` для одного теста, передав для него специальный список `paramList`:

```

//: containers/Tester.java
// Применяет объекты Test к спискам разных контейнеров.
import java.util.*;

public class Tester<C> {
    public static int fieldWidth = 8;
    public static TestParam[] defaultParams= TestParam.array(
        10, 5000, 100, 5000, 1000, 5000, 10000, 500);
    // Переопределите этот метод, чтобы изменить
    // предварительную инициализацию до выполнения теста:
    protected C initialize(int size) { return container; }
    protected C container;
    private String headline = "";
    private List<Test<C>> tests;
    private static String stringField() {
        return "%" + fieldWidth + "s";
    }
    private static String numberField() {
        return "%" + fieldWidth + "d";
    }
    private static int sizeWidth = 5;
    private static String sizeField = "%" + sizeWidth + "s";
    private TestParam[] paramList = defaultParams;
    public Tester(C container, List<Test<C>> tests) {
        this.container = container;
        this.tests = tests;
        if(container != null)
            headline = container.getClass().getSimpleName();
    }
    public Tester(C container, List<Test<C>> tests,
        TestParam[] paramList) {
        this(container, tests);
        this.paramList = paramList;
    }
    public void setHeadline(String newHeadline) {
        headline = newHeadline;
    }
    // Вспомогательные обобщенные методы:
    public static <C> void run(C cntnr, List<Test<C>> tests){
        new Tester<C>(cntnr, tests).timedTest();
    }
}

```

```

public static <C> void run(C cntnr,
    List<Test<C>> tests, TestParam[] paramList) {
    new Tester<C>(cntnr, tests, paramList).timedTest();
}
private void displayHeader() {
    // Вычислить ширину и дополнить '-':
    int width = fieldWidth * tests.size() + sizeWidth;
    int dashLength = width - headline.length() - 1;
    StringBuilder head = new StringBuilder(width);
    for(int i = 0; i < dashLength/2; i++)
        head.append('-');
    head.append(' ');
    head.append(headline);
    head.append(' ');
    for(int i = 0; i < dashLength/2; i++)
        head.append('-');
    System.out.println(head);
    // Вывести заголовки столбцов:
    System.out.format(sizeField, "size");
    for(Test test : tests)
        System.out.format(stringField(), test.name);
    System.out.println();
}
// Выполнить тесты для текущего контейнера:
public void timedTest() {
    displayHeader();
    for(TestParam param : paramList) {
        System.out.format(sizeField, param.size);
        for(Test<C> test : tests) {
            C kontainer = initialize(param.size);
            long start = System.nanoTime();
            // Вызов переопределенного метода:
            int reps = test.test(kontainer, param);
            long duration = System.nanoTime() - start;
            long timePerRep = duration / reps; // Наносекунды
            System.out.format(numberField(), timePerRep);
        }
        System.out.println();
    }
}
} ///:~

```

Методы `stringField()` и `numberField()` создают строки форматирования для вывода результатов. Стандартную ширину форматирования можно изменить, присваивая значение статической переменной `fieldWidth`. Метод `displayHeader()` форматирует и выводит информацию заголовка для каждого текста.

Если вам потребуется выполнить специальную инициализацию, переопределите метод `initialize()`. Он создает инициализированный объект-контейнер правильного размера — вы можете либо изменить существующий контейнер, либо создать новый. Из `test()` видно, что результат сохраняется в локальной ссылке с именем `kontainer`, это позволяет заменить хранимый контейнер совершенно другим инициализированным контейнером.

Метод `Test.test()` должен возвращать количество операций, выполненных тестом; оно используется для вычисления количества наносекунд, необходимых для каждой

операции. Учтите, что `System.nanoTime()` обычно выдает значения с точностью больше 1 (которая зависит от конкретной машины или операционной системы), что вносит некоторую погрешность в результаты.

Результаты могут изменяться от компьютера к компьютеру; тесты предназначены только для сравнения относительной производительности разных контейнеров.

Выбор List

Ниже приводятся тесты производительности для основных операций `List`. Для сравнения также приведены важнейшие операции `Queue`. Для тестирования двух видов контейнеров создаются два разных списка тестов. В данном случае операции `Queue` относятся только к `LinkedList`.

```
//: containers/ListPerformance.java
// тестирование производительности операций
// при работе с List.
// {Args: 100 500} Небольшие значения, чтобы тесты
// не занимали много времени.
import java.util.*;
import net.mindview.util.*;

public class ListPerformance {
    static Random rand = new Random();
    static int reps = 1000;
    static List<Test<List<Integer>>> tests =
        new ArrayList<Test<List<Integer>>>();
    static List<Test<LinkedList<Integer>>> qTests =
        new ArrayList<Test<LinkedList<Integer>>>();
    static {
        tests.add(new Test<List<Integer>>>("add") {
            int test(List<Integer> list, TestParam tp) {
                int loops = tp.loops;
                int listSize = tp.size;
                for(int i = 0; i < loops; i++) {
                    list.clear();
                    for(int j = 0; j < listSize; j++)
                        list.add(j);
                }
                return loops * listSize;
            }
        });
        tests.add(new Test<List<Integer>>>("get") {
            int test(List<Integer> list, TestParam tp) {
                int loops = tp.loops * reps;
                int listSize = list.size();
                for(int i = 0; i < loops; i++)
                    list.get(rand.nextInt(listSize));
                return loops;
            }
        });
        tests.add(new Test<List<Integer>>>("set") {
            int test(List<Integer> list, TestParam tp) {
                int loops = tp.loops * reps;
                int listSize = list.size();
```

продолжение ➤

```

        for(int i = 0; i < loops; i++)
            list.set(rand.nextInt(listSize), 47);
        return loops;
    }
});
tests.add(new Test<List<Integer>>("iteradd") {
    int test(List<Integer> list, TestParam tp) {
        final int LOOPS = 1000000;
        int half = list.size() / 2;
        ListIterator<Integer> it = list.listIterator(half);
        for(int i = 0; i < LOOPS; i++)
            it.add(47);
        return LOOPS;
    }
});
tests.add(new Test<List<Integer>>("insert") {
    int test(List<Integer> list, TestParam tp) {
        int loops = tp.loops;
        for(int i = 0; i < loops; i++)
            list.add(5, 47); // Минимизация затрат на произвольный доступ
        return loops;
    }
});
tests.add(new Test<List<Integer>>("remove") {
    int test(List<Integer> list, TestParam tp) {
        int loops = tp.loops;
        int size = tp.size;
        for(int i = 0; i < loops; i++) {
            list.clear();
            list.addAll(new CountingIntegerList(size));
            while(list.size() > 5)
                list.remove(5); // Минимизация затрат на произвольный доступ
        }
        return loops * size;
    }
});
// Тестирование поведения очереди:
qTests.add(new Test<LinkedList<Integer>>("addFirst") {
    int test(LinkedList<Integer> list, TestParam tp) {
        int loops = tp.loops;
        int size = tp.size;
        for(int i = 0; i < loops; i++) {
            list.clear();
            for(int j = 0; j < size; j++)
                list.addFirst(47);
        }
        return loops * size;
    }
});
qTests.add(new Test<LinkedList<Integer>>("addLast") {
    int test(LinkedList<Integer> list, TestParam tp) {
        int loops = tp.loops;
        int size = tp.size;
        for(int i = 0; i < loops; i++) {
            list.clear();
            for(int j = 0; j < size; j++)
                list.addLast(47);
        }
    }
});

```



```

        return loops * size;
    }
});
qTests.add(
    new Test<LinkedList<Integer>>("rmFirst") {
        int test(LinkedList<Integer> list, TestParam tp) {
            int loops = tp.loops;
            int size = tp.size;
            for(int i = 0; i < loops; i++) {
                list.clear();
                list.addAll(new CountingIntegerList(size));
                while(list.size() > 0)
                    list.removeFirst();
            }
            return loops * size;
        }
    });
qTests.add(new Test<LinkedList<Integer>>("rmLast") {
    int test(LinkedList<Integer> list, TestParam tp) {
        int loops = tp.loops;
        int size = tp.size;
        for(int i = 0; i < loops; i++) {
            list.clear();
            list.addAll(new CountingIntegerList(size));
            while(list.size() > 0)
                list.removeLast();
        }
        return loops * size;
    }
});
}
static class ListTester extends Tester<List<Integer>> {
    public ListTester(List<Integer> container,
        List<Test<List<Integer>>> tests) {
        super(container, tests);
    }
    // Заполняется до нужного размера перед каждым тестом:
    @Override protected List<Integer> initialize(int size){
        container.clear();
        container.addAll(new CountingIntegerList(size));
        return container;
    }
    // Вспомогательный метод:
    public static void run(List<Integer> list,
        List<Test<List<Integer>>> tests) {
        new ListTester(list, tests).timedTest();
    }
}
public static void main(String[] args) {
    if(args.length > 0)
        Tester.defaultParams = TestParam.array(args);
    // Выполняет с массивом только эти два теста:
    Tester<List<Integer>> arrayTest =
        new Tester<List<Integer>>(null, tests.subList(1, 3)){
            // Будет вызываться перед каждым тестом. Создает список
            // на базе массива без возможности изменения размера:
            @Override protected
            List<Integer> initialize(int size) {

```

продолжение ➤

```

        Integer[] ia = Generated.array(Integer.class,
            new CountingGenerator.Integer(), size);
        return Arrays.asList(ia);
    }
};
arrayTest.setHeadline("Array as List");
arrayTest.timedTest();
Tester.defaultParams= TestParam.array(
    10, 5000, 100, 5000, 1000, 1000, 10000, 200);
if(args.length > 0)
    Tester.defaultParams = TestParam.array(args);
ListTester.run(new ArrayList<Integer>(), tests);
ListTester.run(new LinkedList<Integer>(), tests);
ListTester.run(new Vector<Integer>(), tests);
Tester.fieldWidth = 12;
Tester<LinkedList<Integer>> qTest =
    new Tester<LinkedList<Integer>>(
        new LinkedList<Integer>(), qTests);
qTest.setHeadline("Queue tests");
qTest.timedTest();
}
} /* Output: (Sample)
--- Array as List ---
size      get      set
  10      130      183
 100      130      164
1000      129      165
10000     129      165
----- ArrayList -----
size      add      get      set iteradd  insert  remove
  10      121      139      191      435      3952      446
 100       72      141      191      247      3934      296
1000       98      141      194      839      2202      923
10000     122      144      190     6880     14042     7333
----- LinkedList -----
size      add      get      set iteradd  insert  remove
  10      182      164      198      658      366      262
 100     106      202      230      457      108      201
1000     133     1289     1353      430      136      239
10000    172    13648    13187      435      255      239
----- Vector -----
size      add      get      set iteradd  insert  remove
  10      129      145      187      290      3635      253
 100       72      144      190      263      3691      292
1000       99      145      193      846      2162      927
10000     108      145      186     6871     14730     7135
----- Queue tests -----
size      addFirst  addLast    rmFirst    rmLast
  10           199          163         251         253
 100           98           92         180         179
1000           99           93         216         212
10000          111          109         262         384
*///:~

```

Каждый тест необходимо тщательно продумать, чтобы получить осмысленные результаты. Например, тест `add` очищает список, а затем заполняет его до заданного размера. Вызов `clear()` является частью теста и может повлиять на время выполнения (особенно

для небольших тестов). Хотя результаты выглядят вполне разумно, можно переписать тестовую среду так, чтобы вызов подготовительного метода (который в данном случае будет включать вызов `clear()`) осуществлялся вне хронометражного цикла.

Чтобы получить правильные временные данные, для каждого теста необходимо точно вычислить количество выполняемых операций и вернуть это значение из `test()`.

Тесты `get` и `set` используют генератор случайных чисел для произвольных обращений к `List`. Из выходных данных видно, что для списков на базе массивов и `ArrayList` обращения выполняются очень быстро независимо от размера списка, тогда как для `LinkedList` время обращения существенно увеличивается с увеличением размера списка. Очевидно, связанные списки не очень хорошо подходят для частых произвольных обращений.

Тест `iteradd` использует итератор для вставки новых элементов в середине списка. Для `ArrayList` эти операции занимают больше времени с увеличением списка, но для `LinkedList` они обходятся относительно дешево и выполняются за постоянное время независимо от размера. Это логично: при вставке контейнер `ArrayList` должен выделять место и сдвигать все ссылки вперед. Перемещение обходится дороже с ростом `ArrayList`. Контейнеры `LinkedList` достаточно создать ссылку на новый элемент без модификации остальных элементов списка, поэтому затраты остаются примерно одинаковыми и не зависят от размера списка.

Тесты `insert` и `remove` используют ячейку 5 для вставки и удаления (вместо одного из концов списка). У `LinkedList` предусмотрена особая обработка концов списка — это повышает скорость при использовании `LinkedList` как очереди. Однако при добавлении или удалении элементов в середине списка вносятся затраты на произвольный доступ, которые, как мы уже видели, изменяются в зависимости от реализации `List`. При выполнении вставки и удаления в позиции 5 затраты на произвольный доступ должны быть пренебрежимо малыми, и в результатах будет отражено время вставки и удаления без учета специализированных оптимизаций для концов списка. Из результатов видно, что затраты на вставку и удаление в `LinkedList` относительно невелики и не изменяются с размером списка, но для `ArrayList` вставка обходится *очень* дорого, а затраты возрастают с размером списка.

Из тестов `Queue` видно, как быстро `LinkedList` выполняет вставку и удаление на концах списка; такое поведение оптимально для поведения очередей.

В общем случае достаточно вызвать `Tester.run()` с передачей контейнера и списка `tests`. Однако в данном случае необходимо переопределить метод `initialize()`, чтобы список очищался и заполнялся заново перед каждым тестом — в противном случае контроль за размером списка будет теряться между тестами. Класс `ListTester` наследует от `Tester` и выполняет эту инициализацию с использованием `CountingIntegerList`. Вспомогательный метод `run()` также переопределяется.

Также полезно сравнить эффективность обращения к массиву и контейнеру (прежде всего `ArrayList`). В первом тесте `main()` создается специальный объект `Test` с использованием анонимного внутреннего класса. Метод `initialize()` переопределяется для создания нового объекта при каждом вызове (хранимый объект `container` игнорируется, так что при вызове этого конструктора `Tester` аргумент `container` равен `null`).

Новый объект создается методом `Generated.array()` (который определяется в главе 16) и `Arrays.asList()`. В этом случае можно выполнить только два теста, потому что при использовании `List` на базе массива вставка и удаление элементов невозможны, поэтому нужные тесты выбираются из списка `tests` при помощи метода `List.subList()`.

Для операций `get()` и `set()` с произвольным доступом реализация `List` на базе массива работает чуть быстрее `ArrayList`, но для `LinkedList` те же операции обходятся намного дороже, потому что этот контейнер не рассчитан на операции произвольного доступа.

От контейнера `Vector` следует держаться подальше; он включен в библиотеку только для поддержки старого кода (и работает в программах только потому, что он был адаптирован в `List` для обеспечения совместимости).

Вероятно, лучше всего использовать `ArrayList` по умолчанию и переходить на `LinkedList`, если вам потребуется дополнительная функциональность или же при возникновении проблем производительности из-за слишком большого числа вставок и удалений в середине массива. Если вы работаете с группой элементов фиксированного размера, либо используйте реализацию `List` на базе массива (создаваемую вызовом `Arrays.asList()`), либо при необходимости — фактический массив.

`CopyOnWriteArrayList` — специализированная реализация `List`, предназначенная для многопоточного программирования; она рассматривается в главе 21.

29. (2) Измените код `ListPerformance.java` так, чтобы в `List` вместо `Integer` хранились объекты `String`. Для создания тестовых значений используйте генератор из главы 16.
30. (3) Сравните производительность `Collections.sort()` для `ArrayList` и `LinkedList`.
31. (5) Создайте контейнер, инкапсулирующий массив `String`. Контейнер должен поддерживать только добавление и удаление `String`, так что при использовании проблем с преобразованием типа не будет. Если размер внутреннего массива недостаточен для следующего добавления, контейнер должен автоматически изменять его размер. В `main()` сравните производительность контейнера с `ArrayList<String>`.
32. (2) Повторите предыдущее упражнение для контейнера `int` и сравните производительность с `ArrayList<Integer>`. Включите в сравнение процесс инкрементирования каждого объекта в контейнере.
33. (5) Создайте реализацию `FastTraverseLinkedList`, которая во внутреннем представлении использует `LinkedList` для быстрых вставок/удалений и `ArrayList` для быстрого перебора и операций `get()`. Протестируйте, внося соответствующие изменения в `ListPerformance.java`.

Опасности микротестов

При написании так называемых *микротестов* необходимо действовать внимательно, не делать слишком много предположений и по возможности сузить тесты, чтобы в них действительно измерялись только интересующие вас операции. Также следует позаботиться о том, чтобы тесты выполнялись достаточно долго для получения интересных данных, и принять во внимание, что некоторые технологии Java HotSpot активизируются только при выполнении программы в течение определенного времени (это обстоятельство также важно учитывать для программ с малым временем выполнения).

Результаты зависят от компьютера и JVM, поэтому вы должны провести эти тесты самостоятельно и убедиться в том, что ваши результаты сходны с приведенными в книге. Прежде всего вас должны интересовать не абсолютные числа, а относительные результаты между разными типами контейнеров.

Возможно, профилировщик справится с анализом производительности лучше, чем вы. У Java существует свой профилировщик (см. приложение <http://MindView.net/Books/BetterJava>); также доступны профилировщики сторонних фирм, как бесплатные/свободно распространяемые, так и коммерческие.

Аналогичный пример относится к методу `Math.random()`. Он выдает значения от нуля до единицы — но включаются ли границы в набор допустимых значений? Иначе говоря, как должен обозначаться диапазон в системе математических обозначений — $(0,1)$, $[0,1]$, $(0,1]$ или $[0,1)$? Возможно, тестовая программа поможет получить ответ на этот вопрос:

```
//: containers/RandomBounds.java
// Выдает ли Math.random() значения 0.0 и 1.0?
// {RunByHand}
import static net.mindview.util.Print.*;

public class RandomBounds {
    static void usage() {
        print("Usage:");
        print("\tRandomBounds lower");
        print("\tRandomBounds upper");
        System.exit(1);
    }
    public static void main(String[] args) {
        if(args.length != 1) usage();
        if(args[0].equals("lower")) {
            while(Math.random() != 0.0)
                ; // Продолжать попытки
            print("Produced 0.0!");
        }
        else if(args[0].equals("upper")) {
            while(Math.random() != 1.0)
                ; // Продолжать попытки
            print("Produced 1.0!");
        }
        else
            usage();
    }
} ///:~
```

Программа запускается командой

```
java RandomBounds lower
```

или

```
java RandomBounds upper
```

В обоих случаях выполнение программы приходится прерывать вручную, так что на первый взгляд `Math.random()` не генерирует значения 0.0 или 1.0. Однако такой эксперимент обманчив. Если учесть, что от 0 до 1 находится около 2^{62} разных значений типа `double`, время экспериментального ожидания одного конкретного значения может

превысить срок жизни компьютера (и даже экспериментатора). На самом деле значение 0.0 включается в результаты `Math.random()`, так что на математическом языке диапазон должен обозначаться $[0,1)$. Будьте осмотрительны при анализе экспериментов и старайтесь разобраться в их ограничениях.

Выбор между множествами

В зависимости от нужного поведения можно выбрать одну из нескольких реализаций множества — `TreeSet`, `HashSet` и `LinkedHashSet`. Следующая программа проводит тестирование этих реализаций, демонстрируя их достоинства и недостатки:

```
//: containers/SetPerformance.java
// Демонстрация различий производительности
// в разных реализациях Set.
// {Args: 100 5000} Небольшие значения, чтобы тесты
// не занимали много времени.
import java.util.*;

public class SetPerformance {
    static List<Test<Set<Integer>>> tests =
        new ArrayList<Test<Set<Integer>>>();
    static {
        tests.add(new Test<Set<Integer>>>("add") {
            int test(Set<Integer> set, TestParam tp) {
                int loops = tp.loops;
                int size = tp.size;
                for(int i = 0; i < loops; i++) {
                    set.clear();
                    for(int j = 0; j < size; j++)
                        set.add(j);
                }
                return loops * size;
            }
        });
        tests.add(new Test<Set<Integer>>>("contains") {
            int test(Set<Integer> set, TestParam tp) {
                int loops = tp.loops;
                int span = tp.size * 2;
                for(int i = 0; i < loops; i++)
                    for(int j = 0; j < span; j++)
                        set.contains(j);
                return loops * span;
            }
        });
        tests.add(new Test<Set<Integer>>>("iterate") {
            int test(Set<Integer> set, TestParam tp) {
                int loops = tp.loops * 10;
                for(int i = 0; i < loops; i++) {
                    Iterator<Integer> it = set.iterator();
                    while(it.hasNext())
                        it.next();
                }
                return loops * set.size();
            }
        });
    }
}
```

```

public static void main(String[] args) {
    if(args.length > 0)
        Tester.defaultParams = TestParam.array(args);
    Tester.fieldWidth = 10;
    Tester.run(new TreeSet<Integer>(), tests);
    Tester.run(new HashSet<Integer>(), tests);
    Tester.run(new LinkedHashSet<Integer>(), tests);
}
} /* Output: (Sample)
----- TreeSet -----
size      add  contains  iterate
  10       746      173      89
 100       501      264      68
1000        714      410      69
10000       1975      552      69
----- HashSet -----
size      add  contains  iterate
  10       308       91      94
 100       178       75      73
1000        216      110      72
10000        711      215     100
----- LinkedHashSet -----
size      add  contains  iterate
  10       350       65      83
 100       270       74      55
1000        303      111      54
10000       1615      256      58
*///:~

```

Множество `HashSet` превосходит по скоростным показателям `TreeSet` при проведении практически всех операций (в особенности добавления и поиска элементов, которые являются важнейшими операциями). Множество `TreeSet` создано только потому, что поддерживает хранимые элементы в упорядоченном виде, и имеет смысл к нему обращаться, только если вам понадобится отсортированное множество. Из-за внутренней структуры, необходимой для поддержки сортировки, а также потому, что перебор в таких реализациях является более вероятной операцией, `TreeSet` обычно превосходит `HashSet` по скорости перебора.

Заметьте, что класс `LinkedHashSet` вставляет элементы немного медленнее, чем `HashSet`; так происходит из-за необходимости поддерживать не только хешированный контейнер, но и связанный список.

- 34.** (1) Измените код `SetPerformance.java` так, чтобы в `Set` вместо `Integer` хранились объекты `String`. Для создания тестовых значений используйте генератор из главы 16.

Выбор между картами

Следующая программа демонстрирует сильные и слабые стороны разных реализаций `Map`:

```

//: containers/MapPerformance.java
// Демонстрация различий производительности
// в разных реализациях Map.
// {Args: 100 5000} Небольшие значения, чтобы тесты
// не занимали много времени.
import java.util.*;

```

продолжение ➤

```

public class MapPerformance {
    static List<Test<Map<Integer,Integer>>> tests =
        new ArrayList<Test<Map<Integer,Integer>>>();
    static {
        tests.add(new Test<Map<Integer,Integer>>("put") {
            int test(Map<Integer,Integer> map, TestParam tp) {
                int loops = tp.loops;
                int size = tp.size;
                for(int i = 0; i < loops; i++) {
                    map.clear();
                    for(int j = 0; j < size; j++)
                        map.put(j, j);
                }
                return loops * size;
            }
        });
        tests.add(new Test<Map<Integer,Integer>>("get") {
            int test(Map<Integer,Integer> map, TestParam tp) {
                int loops = tp.loops;
                int span = tp.size * 2;
                for(int i = 0; i < loops; i++)
                    for(int j = 0; j < span; j++)
                        map.get(j);
                return loops * span;
            }
        });
        tests.add(new Test<Map<Integer,Integer>>("iterate") {
            int test(Map<Integer,Integer> map, TestParam tp) {
                int loops = tp.loops * 10;
                for(int i = 0; i < loops; i++) {
                    Iterator it = map.entrySet().iterator();
                    while(it.hasNext())
                        it.next();
                }
                return loops * map.size();
            }
        });
    }
    public static void main(String[] args) {
        if(args.length > 0)
            Tester.defaultParams = TestParam.array(args);
        Tester.run(new TreeMap<Integer,Integer>(), tests);
        Tester.run(new HashMap<Integer,Integer>(), tests);
        Tester.run(new LinkedHashMap<Integer,Integer>(), tests);
        Tester.run(
            new IdentityHashMap<Integer,Integer>(), tests);
        Tester.run(new WeakHashMap<Integer,Integer>(), tests);
        Tester.run(new Hashtable<Integer,Integer>(), tests);
    }
} /* Output: (Sample)
----- TreeMap -----
size    put    get iterate
  10     748    168    100
  100    506    264     76
 1000    771    450     78
10000   2962    561     83
----- HashMap -----
size    put    get iterate

```


10	281	76	93
100	179	70	73
1000	267	102	72
10000	1305	265	97

```

----- LinkedHashMap -----
size      put      get  iterate
  10      354      100    72
  100     273      89    50
 1000     385     222    56
10000    2787     341    56
----- IdentityHashMap -----
size      put      get  iterate
  10      290     144   101
  100     204     287   132
 1000     508     336    77
10000     767     266    56
----- WeakHashMap -----
size      put      get  iterate
  10      484     146   151
  100     292     126   117
 1000     411     136   152
10000    2165     138   555
----- Hashtable -----
size      put      get  iterate
  10      264     113   113
  100     181     105    76
 1000     260     201    80
10000    1245     134    77
*///:~

```

Вставка для всех реализаций Map (кроме IdentityHashMap) существенно замедляется с увеличением размера Map. Однако в общем случае поиск требует значительно меньших затрат, чем вставка, — и это хорошо, потому что находить элементы в контейнере обычно требуется гораздо чаще, чем вставлять их.

Производительность Hashtable примерно равна производительности HashMap. Впрочем, это и не удивительно — реализация HashMap должна заменить устаревший контейнер Hashtable, поэтому она использует те же механизмы хранения и поиска данных (о которых будет рассказано позднее).

TreeMap показывает куда более скромные результаты. Как и TreeSet, она предназначена для создания упорядоченных списков. В древовидной структуре, положенной в основу TreeMap, элементы всегда хранятся в упорядоченном виде, а значит, их не нужно специально сортировать. После заполнения TreeMap можно использовать методы keySet() для получения множества (Set) ключей карты и toArray() для создания массива ключей. После этого статический метод Arrays.binarySearch() позволит быстро найти нужные объекты в отсортированном массиве. Конечно, все это делается только тогда, когда по каким-либо причинам реализация HashMap вам не подходит, так как именно HashMap обеспечивает быстрое нахождение объектов. К тому же создать карту HashMap из существующей TreeMap очень просто: для этого нужно просто определить один объект или вызвать putAll(). Вывод: когда вы используете таблицу, вашим основным выбором должен стать класс HashMap, и только при необходимости для часто сортируемой карты (Map) следует выбирать TreeMap.

Реализация `LinkedHashMap` работает медленнее, чем `HashMap`, так как ей приходится поддерживать не только хешированный контейнер, но и связанный список (для сохранения порядка вставки). По этой причине перебор выполняется быстрее.

Производительность `IdentityHashMap` отличается от других карт, поскольку в ней сравнение выполняется не методом `equals()`, а оператором `==`. Реализация `WeakHashMap` будет описана позднее в этой главе.

35. (1) Измените код `MapPerformance.java` и включите в него тесты для реализации `SlowMap`.
36. (5) Измените карту `SlowMap`, чтобы вместо двух списков `ArrayList` в ней хранился один список объектов `ArrayList` или `MapEntry`. Покажите, что эта версия класса работает правильно. С помощью программы `MapPerformance.java` проверьте быстродействие нового класса. После этого измените метод `put()` так, чтобы после помещения очередной пары он проводил сортировку карты, а в методе `get()` для поиска ключа примените метод `Collections.binarySearch()`. Сравните быстродействие этой версии таблицы с предыдущими реализациями.
37. (2) Модифицируйте класс `SimpleHashMap` так, чтобы в нем использовались списки `ArrayList` вместо `LinkedList`. Измените программу `MapPerformance.java` и сравните быстродействие двух реализаций.

Факторы, влияющие на производительность `HashMap`

Реализацию `HashMap` можно настроить «вручную», чтобы добиться оптимального быстродействия в вашем конкретном приложении. Но чтобы понять, какие проблемы решаются настройкой `HashMap`, необходимо изучить несколько терминов.

- ❑ **Емкость** (`capacity`): количество узлов хеш-таблицы.
- ❑ **Начальная емкость** (`initial capacity`): количество узлов при первичном создании хеш-таблицы. `HashMap` и `HashSet` позволяют указать начальную емкость при вызове конструктора.
- ❑ **Размер** (`size`): количество заполненных узлов, находящихся в таблице.
- ❑ **Коэффициент загрузки** (`load factor`): отношение размер/емкость. Коэффициент загрузки, равный нулю, — это пустая таблица, 0,5 — это таблица, заполненная наполовину и т. д. Незначительно загруженная таблица практически не имеет коллизий и поэтому оптимальна для вставок и произвольного поиска (но замедляет перебор с использованием итератора). В конструкторах `HashSet` и `HashMap` может передаваться коэффициент загрузки; при достижении таблицей этого уровня заполнения контейнер автоматически увеличит емкость (количество узлов) примерно в два раза и перераспределит набор существующих элементов по новым позициям (это называется *перехешированием* (`rehashing`)).

По умолчанию в `HashMap` устанавливается коэффициент загрузки, равный 0,75 (перехеширование не происходит, пока не заполнится три четверти таблицы). В принципе, это разумный компромисс между временем поиска и занимаемой памятью. Более высокий коэффициент загрузки уменьшает потребность таблицы в памяти, но увеличивает время поиска, а это очень важно, поскольку большую часть времени занимает именно поиск (к нему относится и вызов методов `put()` и `get()`).

Если вы знаете, что в таблице `HashMap` будет храниться много элементов, имеет смысл сделать ее емкость достаточно большой, чтобы предотвратить слишком частое проведение перехеширования¹.

- 38.** (3) В интерактивной документации JDK найдите описание класса `HashMap`. Создайте карту `HashMap`, заполните ее элементами и определите ее коэффициент загрузки. Оцените скорость поиска, а затем попытайтесь увеличить быстродействие, создав новую карту `HashMap` с большей начальной емкостью, скопируйте старую версию в новую и проведите проверку скорости поиска для новой таблицы.
- 39.** (6) Добавьте в `SimpleHashMap` закрытый (`private`) метод `rehash()`, который проводит перехеширование, когда коэффициент загрузки превышает 0,75. В процессе перехеширования удвойте число ячеек, затем проведите поиск первого простого числа после полученного значения, чтобы узнать, сколько надо сделать новых ячеек.

Вспомогательные средства работы с коллекциями

Также существуют некоторые автономные вспомогательные средства для работы с контейнерами, представленные статическими методами класса `java.util.Collections`. Некоторые из них нам уже встречались — как, например, `addAll()`, `reverseOrder()` и `binarySearch()`. В следующей таблице обобщения используются там, где они актуальны.

<code>checkedCollection(Collection<T>, Class<T> type) checkedList(List<T>, Class<T> type) checkedMap(Map<K,V>, Class<K> keyType, Class<V> valueType) checkedSet(Set<T>, Class<T> type) checkedSortedMap(SortedMap<K,V>, Class<K> keyType, Class<V> valueType)</code>	Представление <code>Collection</code> (или конкретного подтипа <code>Collection</code>), обеспечивающее <i>динамическую</i> безопасность типов. Используется в ситуациях, в которых невозможно использование версии со статической проверкой. См. раздел «Динамическая безопасность типов» главы 14
---	---

продолжение ➤

¹ Джошуа Блош замечает в своем неофициальном сообщении: «...Я полагаю, что мы совершили ошибку, позволив деталям реализации (таким, как размер таблицы и коэффициент ее загрузки) „просочиться“ в программные интерфейсы API. Клиент скорее должен был бы сообщать ожидаемый максимальный размер таблицы, и эти параметры следовало выводить из него. Выбирая параметры самостоятельно, клиент легко может навредить работе таблицы. Как крайность можно вспомнить параметр `capacityIncrement` класса `Vector`. Никому не следовало пользоваться им, а нам не нужно было давать такую возможность. Если приравнять ему ненулевое значение, трудоемкость операций возрастет с линейной до квадратичной. Другими словами, исчезает производительность. Со временем мы поняли эти вещи. Если вы посмотрите на класс `IdentityHashMap`, то увидите, что низкоуровневых параметров для настройки в нем нет».

checkedSortedSet(SortedSet<T>, Class<T> type)	
max(Collection) min(Collection)	Находит наибольший или наименьший элемент аргумента с использованием «естественного» сравнения объектов Collection
max(Collection, Comparator) min(Collection, Comparator)	Находит наибольший или наименьший элемент Collection с использованием сравнения по критерию Comparator
indexOfSubList(List source, List target)	Находит начальный индекс <i>первого</i> вхождения target в source – или –1, если вхождения не обнаружены
lastIndexOfSubList(List source, List target)	Находит начальный индекс <i>последнего</i> вхождения target в source – или –1, если вхождения не обнаружены
replaceAll(List<T>, T oldVal, T newVal)	Заменяет все вхождения oldVal на newVal
reverse(List)	Переставляет элементы в обратном порядке
reverseOrder() reverseOrder(Comparator<T>)	Возвращает Comparator для обратного порядка перебора объектов коллекции, реализующей Comparable<T>. Вторая версия изменяет порядок для переданного объекта Comparator
rotate(List, int distance)	Перемещает все элементы вперед на расстояние distance; при этом элементы в конце коллекции перемещаются в ее начало
shuffle(List) shuffle(List, Random)	Генерирует случайную перестановку для заданного списка. Первая форма использует внутренний генератор случайных чисел, во второй форме вы можете передать собственную версию
sort(List<T>) sort(List<T>, Comparator<? super T> c)	Сортирует List<T> с использованием «естественного» упорядочения. Вторая форма позволяет задать свой объект Comparator для сортировки
copy(List<? super T> dest, List<? extends T> src)	Копирует элементы из src в dest
swap(List, int i, int j)	Меняет местами элементы i и j в List. Скорее всего, работает быстрее, чем реализация, которую вы напишете «вручную»
fill(List<? super T>, T x)	Заменяет все элементы списка значением x
nCopies(int n, T x)	Возвращает неизменяемый экземпляр List<T> размера n, все ссылки в котором указывают на x
disjoint(Collection, Collection)	Возвращает true, если две коллекции не имеют общих элементов
frequency(Collection, Object x)	Возвращает количество элементов в Collection, равных x
emptyList() emptyMap() emptySet()	Возвращает неизменяемый пустой экземпляр List, Map или Set. Возвращается обобщение, поэтому полученная реализация Collection параметризуется по нужному типу

singleton(T x) singletonList(T x) singletonMap(K key, V value)	Возвращает неизменяемый экземпляр Set<T>, List<T> или Map<K,V> с одним элементом, созданным на базе заданных аргументов
list(Enumeration<T> e)	Создает экземпляр ArrayList<T>, содержащий элементы в том порядке, в котором они возвращаются Enumeration (предшественник Iterator). Используется для преобразования старого кода
enumeration(Collection<T>)	Создает Enumeration<T> «в старом стиле» для аргумента

Обратите внимание: методы min() и max() работают с объектами Collection, а не List, так что вам не нужно беспокоиться о том, отсортирован объект Collection или нет. (Как упоминалось ранее, список List или массив необходимо отсортировать перед вызовом binarySearch().)

Следующий пример демонстрирует использование большинства вспомогательных методов из приведенной выше таблицы:

```
//: containers/Utilities.java
// Простая демонстрация вспомогательных методов Collections.
import java.util.*;
import static net.mindview.util.Print.*;

public class Utilities {
    static List<String> list = Arrays.asList(
        "one Two three Four five six one".split(" "));
    public static void main(String[] args) {
        print(list);
        print("'list' disjoint (Four)? : " +
            Collections.disjoint(list,
                Collections.singletonList("Four")));
        print("max: " + Collections.max(list));
        print("min: " + Collections.min(list));
        print("max w/ comparator: " + Collections.max(list,
            String.CASE_INSENSITIVE_ORDER));
        print("min w/ comparator: " + Collections.min(list,
            String.CASE_INSENSITIVE_ORDER));
        List<String> sublist =
            Arrays.asList("Four five six".split(" "));
        print("indexOfSubList: " +
            Collections.indexOfSubList(list, sublist));
        print("lastIndexOfSubList: " +
            Collections.lastIndexOfSubList(list, sublist));
        Collections.replaceAll(list, "one", "Yo");
        print("replaceAll: " + list);
        Collections.reverse(list);
        print("reverse: " + list);
        Collections.rotate(list, 3);
        print("rotate: " + list);
        List<String> source =
            Arrays.asList("in the matrix".split(" "));
        Collections.copy(list, source);
        print("copy: " + list);
        Collections.swap(list, 0, list.size() - 1);
        print("swap: " + list);
    }
}
```

продолжение ➤

```

Collections.shuffle(list, new Random(47));
print("shuffled: " + list);
Collections.fill(list, "pop");
print("fill: " + list);
print("frequency of 'pop': " +
    Collections.frequency(list, "pop"));
List<String> dups = Collections.nCopies(3, "snap");
print("dups: " + dups);
print("'list' disjoint 'dups'?": " +
    Collections.disjoint(list, dups));
// Получение Enumeration "в старом стиле":
Enumeration<String> e = Collections.enumeration(dups);
Vector<String> v = new Vector<String>();
while(e.hasMoreElements())
    v.addElement(e.nextElement());
// Преобразование Vector "в старом стиле"
// в List с использованием Enumeration:
ArrayList<String> arrayList =
    Collections.list(v.elements());
print("arrayList: " + arrayList);
}
} /* Output:
[one, Two, three, Four, five, six, one]
'list' disjoint (Four)?: false
max: three
min: Four
max w/ comparator: Two
min w/ comparator: five
indexOfSubList: 3
lastIndexOfSubList: 3
replaceAll: [Yo, Two, three, Four, five, six, Yo]
reverse: [Yo, six, five, Four, three, Two, Yo]
rotate: [three, Two, Yo, Yo, six, five, Four]
copy: [in, the, matrix, Yo, six, five, Four]
swap: [Four, the, matrix, Yo, six, five, in]
shuffled: [six, matrix, the, Four, Yo, five, in]
fill: [pop, pop, pop, pop, pop, pop, pop]
frequency of 'pop': 7
dups: [snap, snap, snap]
'list' disjoint 'dups'?": true
arrayList: [snap, snap, snap]
*///:~

```

В выходных результатах объясняется поведение каждого вспомогательного метода. Обратите внимание на различия в результатах `min()` и `max()` при использовании компаратора `String.CASE_INSENSITIVE_ORDER` из-за различий в регистре.

Сортировка и поиск в списках

Инструменты, предназначенные для сортировки списков и поиска в них, имеют точно такие же имена и описания, как те, что использовались нами для проведения аналогичных операций над массивами, но это статические методы класса `Collections`, в то время как для массивов применялся класс `Arrays`. Вот пример с использованием данных `list` из `Utilities.java`:

```

//: containers/ListSortSearch.java
// Поиск и сортировка списков с использованием
// вспомогательного класса Collections.
import java.util.*;
import static net.mindview.util.Print.*;

public class ListSortSearch {
    public static void main(String[] args) {
        List<String> list =
            new ArrayList<String>(Utilities.list);
        list.addAll(Utilities.list);
        print(list);
        Collections.shuffle(list, new Random(47));
        print("Shuffled: " + list);
        // Использование ListIterator для отсечения последних элементов:
        ListIterator<String> it = list.listIterator(10);
        while(it.hasNext()) {
            it.next();
            it.remove();
        }
        print("Trimmed: " + list);
        Collections.sort(list);
        print("Sorted: " + list);
        String key = list.get(7);
        int index = Collections.binarySearch(list, key);
        print("Location of " + key + " is " + index +
            ", list.get(" + index + ") = " + list.get(index));
        Collections.sort(list, String.CASE_INSENSITIVE_ORDER);
        print("Case-insensitive sorted: " + list);
        key = list.get(7);
        index = Collections.binarySearch(list, key,
            String.CASE_INSENSITIVE_ORDER);
        print("Location of " + key + " is " + index +
            ", list.get(" + index + ") = " + list.get(index));
    }
} /* Output:
[one, Two, three, Four, five, six, one, one, Two, three,
Four, five, six, one]
Shuffled: [Four, five, one, one, Two, six, six, three,
three, five, Four, Two, one, one]
Trimmed: [Four, five, one, one, Two, six, six, three,
three, five]
Sorted: [Four, Two, five, five, one, one, six, six, three,
three]
Location of six is 7, list.get(7) = six
Case-insensitive sorted: [five, five, Four, one, one, six,
six, three, three, Two]
Location of three is 7, list.get(7) = three
*///:~

```

Как и в случае с поиском и сортировкой в массивах, при сортировке с использованием `Comparator` этот же объект `Comparator` должен использоваться и при вызове `binarySearch()`.

- 40.** (5) Создайте класс, содержащий два объекта `String`, и реализуйте для этого класса интерфейс `Comparable` так, чтобы его метод проводил сравнение с учетом только первой строки. Заполните массив и список `ArrayList` объектами этого класса, применив

генератор `RandomGenerator`. Покажите, что сортировка работает правильно. Теперь создайте объект `Comparator`, который проводит сравнение с учетом только второй строки, после чего покажите, что и в этом случае сравнение проходит верно; выполните также двоичный поиск методом `binarySearch()` с использованием вашего объекта `Comparator`.

41. (3) Измените класс из предыдущего упражнения так, чтобы он работал с `HashSet` и использовался в качестве ключа в `HashMap`.
42. (2) Измените упражнение 40 так, чтобы в нем использовалась сортировка по алфавиту.

Получение неизменяемых коллекций и карт

Довольно часто бывает удобно иметь версию коллекции (`Collection`) или карты (`Map`), доступную только для чтения. Это поможет сделать тот же класс `Collections`; один из его методов получает исходный контейнер и возвращает контейнер, пригодный только для чтения. Существуют несколько разновидностей таких методов: для коллекции (`Collection`) (если вы не захотите обозначить ее тип более точно), списка (`List`), множества (`Set`) и карты (`Map`). Следующий пример показывает, как создавать и использовать версии контейнеров с фиксированным состоянием.

```
//: containers/ReadOnly.java
// Использование методов Collections.unmodifiable().
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class ReadOnly {
    static Collection<String> data =
        new ArrayList<String>(Countries.names(6));
    public static void main(String[] args) {
        Collection<String> c =
            Collections.unmodifiableCollection(
                new ArrayList<String>(data));
        print(c);           // Чтение разрешено
        //! c.add("one"); // Но изменение невозможно.

        List<String> a = Collections.unmodifiableList(
            new ArrayList<String>(data));
        ListIterator<String> lit = a.listIterator();
        print(lit.next()); // Чтение разрешено
        //! lit.add("one"); // Но изменение невозможно.

        Set<String> s = Collections.unmodifiableSet(
            new HashSet<String>(data));
        print(s);           // Чтение разрешено
        //! s.add("one"); // Но изменение невозможно.

        // Для SortedSet:
        Set<String> ss = Collections.unmodifiableSortedSet(
            new TreeSet<String>(data));

        Map<String,String> m = Collections.unmodifiableMap(
            new HashMap<String,String>(Countries.capitals(6)));
```



```

print(m); // Чтение разрешено
///! m.put("Ralph", "Howdy!");

// Для SortedMap:
Map<String,String> sm =
    Collections.unmodifiableSortedMap(
        new TreeMap<String,String>(Countries.capitals(6)));
}
} /* Output:
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
ALGERIA
[BULGARIA, BURKINA FASO, BOTSWANA, BENIN, ANGOLA, ALGERIA]
{BULGARIA=Sofia, BURKINA FASO=Ouagadougou,
BOTSWANA=Gaberone, BENIN=Porto-Novo, ANGOLA=Luanda,
ALGERIA=Algiers}
*///:~

```

Вызов «unmodifiable-метода» для конкретного типа не запрещается во время компиляции, но после выполнения преобразования вызовы таких методов, изменяющие содержимое конкретного контейнера, приведут к возникновению исключения `UnsupportedOperationException`.

В любом случае, *перед тем* как сделать контейнер неизменяемым, его следует заполнить какими-либо значимыми данными. После заполнения контейнера лучше всего заменить существующую ссылку новой, полученной методом, делающим контейнер «неизменяемым». Наложив однажды запрет на изменения, вы больше не сможете что-либо модифицировать в этом контейнере. Вместе с тем, этот инструмент позволяет хранить изменяемый контейнер в классе как его закрытый (`private`) член, а по запросу возвращать ссылку на контейнер, допускающий только чтение. Таким образом, у вас остается возможность менять содержимое контейнера в классе, но все внешние пользователи смогут лишь читать его.

Синхронизация коллекции или карты

Синхронизация — это важное понятие из области *многозадачности*, более сложной темы, которой мы не будем касаться вплоть до главы 21. Сейчас я просто отмечу, что класс `Collections` предоставляет способ автоматически синхронизировать контейнер целиком. Форма записи такой операции сходна с вызовом методов, производящих «неизменяемые» контейнеры.

```

//: containers/Synchronization.java
// Использование методов Collections.synchronized.
import java.util.*;

public class Synchronization {
    public static void main(String[] args) {
        Collection<String> c =
            Collections.synchronizedCollection(
                new ArrayList<String>());
        List<String> list = Collections.synchronizedList(
            new ArrayList<String>());
        Set<String> s = Collections.synchronizedSet(
            new HashSet<String>());
        Set<String> ss = Collections.synchronizedSortedSet(

```

продолжение ➤

```

    new TreeSet<String>());
    Map<String,String> m = Collections.synchronizedMap(
        new HashMap<String,String>());
    Map<String,String> sm =
        Collections.synchronizedSortedMap(
            new TreeMap<String,String>());
    }
} ///:~

```

Новый контейнер следует немедленно передать соответствующему «synchronized-методу», как показано выше. Тем самым вы сводите к минимуму опасность внешнего доступа к несинхронизированному контейнеру.

Срочный отказ

В контейнерах Java работает механизм, не позволяющий изменять их содержимое сразу нескольким процессам. Проблема возникает, если вы проводите перебор элементов контейнера, и в это время некоторый другой процесс пытается вставить, удалить или изменить один из объектов контейнера. Возможно, этот объект вами уже просмотрен, или вам еще предстоит провести с ним некоторые действия, или размер контейнера уменьшился после того, как вы узнали его с помощью метода `size()`, — опасных ситуаций, подобных описанным, может возникнуть великое множество. Библиотека контейнеров Java поддерживает механизм «срочного отказа», который следит за тем, не было ли проведено в контейнере изменений, не относящихся к текущему процессу. Если вдруг окажется, что кто-то еще изменяет контейнер, немедленно возбуждается исключение `ConcurrentModificationException`. В этом и заключается принцип «срочного отказа» — механизм не пытается обнаружить проблему позднее или использовать более сложный алгоритм.

Увидеть действие механизма срочного отказа своими глазами довольно просто — нужно создать итератор, а затем добавить к коллекции новый элемент, поместив его в позицию, на которую указывает итератор, как в следующем примере:

```

//: containers/FailFast.java
// Демонстрация механизма "срочного отказа".
import java.util.*;

public class FailFast {
    public static void main(String[] args) {
        Collection<String> c = new ArrayList<String>();
        Iterator<String> it = c.iterator();
        c.add("An object");
        try {
            String s = it.next();
        } catch (ConcurrentModificationException e) {
            System.out.println(e);
        }
    }
} /* Output:
java.util.ConcurrentModificationException
*///:~

```

Здесь исключение возникает потому, что объект помещается в контейнер *после того*, как для этого контейнера был запрошен итератор. Возможность изменения двумя независимыми частями программы одного и того же контейнера ведет к неопределенности,

поэтому исключение предупреждает вас о том, что код нужно поменять — в нашем случае надо запрашивать итератор *после того*, как все элементы будут добавлены в контейнер.

Реализации `ConcurrentHashMap`, `CopyOnWriteArrayList` и `CopyOnWriteArraySet` используют механизмы, предотвращающие выдачу исключения `ConcurrentModificationExceptions`.

Удержание ссылок

Библиотека `java.lang.ref` содержит набор классов, которые позволяют проводить уборку мусора с большей гибкостью и особенно полезны при наличии в программе больших объектов, потенциально опасных в смысле нехватки памяти. Существуют три класса, унаследованных от абстрактного класса `Reference`: `SoftReference`, `WeakReference` и `PhantomReference`. Каждый из них предоставляет свой способ контроля над уборщиком мусора, если *только* объект достижим с помощью одного из объектов `Reference`.

Если объект *достижим*, это значит, что где-то в вашей программе его можно обнаружить. В обычном варианте это значит, что у вас есть ссылка на этот объект, хранящийся в стеке, или ссылка на объект, в котором есть ссылка на рассматриваемый объект; промежуточных звеньев может быть много. Если объект достижим, уборщик мусора не имеет права удалить его, так как объект все еще используется программой. Если объект недостижим, программа уже не может никаким образом обратиться к нему, потому его удаление уборщиком мусора не вызовет осложнений.

Класс `Reference` используется тогда, когда необходимо продолжить хранить ссылку на объект (вы хотите иметь возможность обратиться к этому объекту), но в то же время вы хотите разрешить уборщику мусора удаление объекта. Таким образом, у вас остается возможность обращаться к объекту, но если свободной памяти не осталось, вы разрешаете освобождение этого объекта.

Объект `Reference` служит посредником между вами и обычной ссылкой, и других обычных ссылок на объект (тех, что не «упакованы» в объектах `Reference`) существовать не должно. Если уборщик мусора обнаружит, что объект достижим напрямую по обычной ссылке, он не станет его освобождать.

Классы `SoftReference`, `WeakReference` и `PhantomReference` «располагаются» по порядку: каждый следующий «слабее», чем предыдущий, и отвечает за свой уровень достижимости. Класс `SoftReference` предназначен для реализации буферов памяти. Класс `WeakReference` необходим при реализации «канонического отображения», где экземпляры объектов могут одновременно использоваться в различных местах программы, для экономии памяти, что не предотвращает освобождения ключей (или значений). Класс `PhantomReference` предназначен для планирования «предсмертных» действий объекта и позволяет осуществить их более гибкими способами в сравнении со стандартным механизмом финализации Java.

Для `SoftReference` и `WeakReference` у вас есть выбор — помещать или не помещать их во вспомогательный класс `ReferenceQueue` (механизм «предсмертной» очистки объекта), однако создание `PhantomReference` возможно только в объекте `ReferenceQueue`. Вот простой пример:

```

//: containers/References.java
// Демонстрация объектов Reference.
import java.lang.ref.*;
import java.util.*;

class VeryBig {
    private static final int SIZE = 10000;
    private long[] la = new long[SIZE];
    private String ident;
    public VeryBig(String id) { ident = id; }
    public String toString() { return ident; }
    protected void finalize() {
        System.out.println("Finalizing " + ident);
    }
}

public class References {
    private static ReferenceQueue<VeryBig> rq =
        new ReferenceQueue<VeryBig>();
    public static void checkQueue() {
        Reference<? extends VeryBig> inq = rq.poll();
        if(inq != null)
            System.out.println("In queue: " + inq.get());
    }
    public static void main(String[] args) {
        int size = 10;
        // Или использовать размер, заданный в командной строке:
        if(args.length > 0)
            size = new Integer(args[0]);
        LinkedList<SoftReference<VeryBig>> sa =
            new LinkedList<SoftReference<VeryBig>>();
        for(int i = 0; i < size; i++) {
            sa.add(new SoftReference<VeryBig>(
                new VeryBig("Soft " + i), rq));
            System.out.println("Just created: " + sa.getLast());
            checkQueue();
        }
        LinkedList<WeakReference<VeryBig>> wa =
            new LinkedList<WeakReference<VeryBig>>();
        for(int i = 0; i < size; i++) {
            wa.add(new WeakReference<VeryBig>(
                new VeryBig("Weak " + i), rq));
            System.out.println("Just created: " + wa.getLast());
            checkQueue();
        }
        SoftReference<VeryBig> s =
            new SoftReference<VeryBig>(new VeryBig("Soft"));
        WeakReference<VeryBig> w =
            new WeakReference<VeryBig>(new VeryBig("Weak"));
        System.gc();
        LinkedList<PhantomReference<VeryBig>> pa =
            new LinkedList<PhantomReference<VeryBig>>();
        for(int i = 0; i < size; i++) {
            pa.add(new PhantomReference<VeryBig>(
                new VeryBig("Phantom " + i), rq));
            System.out.println("Just created: " + pa.getLast());
            checkQueue();
        }
    }
}
/* (Execute to see output) *///:~

```

Запустив программу (вывод желательно направить в текстовый файл, чтобы просмотреть результат с разбивкой на страницы), вы увидите, что объекты удаляются уборщиком мусора, несмотря на то, что у вас все еще есть доступ к ним через объекты `Reference` (для получения реального объекта используется метод `get()`). Вы также заметите, что `ReferenceQueue` всегда выдает объект `Reference`, содержащий ссылку `null`. Чтобы использовать это обстоятельство, объявите класс, производный от `Reference`, и добавьте в него нужные дополнительные методы.

WeakHashMap

Библиотека контейнеров содержит специальную карту (`Map`) для хранения «слабых» ссылок: `WeakHashMap`. Этот класс разработан, чтобы упростить реализацию «канонического отображения»: с целью экономии памяти для конкретного значения создается только один экземпляр. Когда программе понадобится этот объект, она выбирает его из карты и использует (вместо того, чтобы создавать «с нуля»). Карту можно заполнить значениями во время инициализации, но обычно значения создаются по мере необходимости.

Так как этот механизм предназначен для экономии памяти, очень удобно то, что `WeakHashMap` позволяет уборщику мусора автоматически удалять ключи и значения. Вам не нужно делать ничего особенного, при помещении в `WeakHashMap` они будут автоматически «заворачиваться» в объекты `WeakReference`. Присоединенная процедура позволяет проводить очистку, когда ключи становятся больше не нужны, как показано в следующем примере:

```
//: containers/CanonicalMapping.java
// Демонстрация WeakHashMap.
import java.util.*;

class Element {
    private String ident;
    public Element(String id) { ident = id; }
    public String toString() { return ident; }
    public int hashCode() { return ident.hashCode(); }
    public boolean equals(Object r) {
        return r instanceof Element &&
            ident.equals(((Element)r).ident);
    }
    protected void finalize() {
        System.out.println("Finalizing " +
            getClass().getSimpleName() + " " + ident);
    }
}

class Key extends Element {
    public Key(String id) { super(id); }
}

class Value extends Element {
    public Value(String id) { super(id); }
}

public class CanonicalMapping {
    public static void main(String[] args) {
        int size = 1000;
```

продолжение ➤

```
// Или использовать размер, заданный в командной строке:
if(args.length > 0)
    size = new Integer(args[0]);
Key[] keys = new Key[size];
WeakHashMap<Key,Value> map =
    new WeakHashMap<Key,Value>();
for(int i = 0; i < size; i++) {
    Key k = new Key(Integer.toString(i));
    Value v = new Value(Integer.toString(i));
    if(i % 3 == 0)
        keys[i] = k; // Сохраняем как "настоящие" ссылки
    map.put(k, v);
}
System.gc();
}
} /* (Execute to see output) *///:~
```

Класс `Key` должен иметь методы `hashCode()` и `equals()`, поскольку он используется в качестве ключа хешированной структуры данных (см. ранее в этой главе).

При запуске программы вы увидите, что уборщик мусора пропускает каждый третий ключ, так как для них были созданы обычные ссылки, помещенные в массив `keys`, а это значит, что для их объектов не разрешена уборка мусора.

Контейнеры Java версий 1.0/1.1

К сожалению, огромный объем кода на Java был написан с использованием старых контейнеров, и даже новый код зачастую «грешит» устаревшими вызовами. Поэтому, хотя вам настоятельно не рекомендуется писать новые программы, привлекая эти старые классы, знать об их существовании все же стоит. Впрочем, прежние контейнеры были довольно ограниченными и много о них не напишешь. (Они уже остались в прошлом, поэтому я постараюсь удержаться от нападок на некоторые ужасные решения, принятые их проектировщиками.)

Vector и Enumeration

Единственной последовательностью, обладающей способностью саморасширения, в Java версий 1.0/1.1 был контейнер `Vector`, поэтому он использовался весьма интенсивно. У него так много пороков, что я не стану даже упоминать о них (любопытным рекомендую посмотреть первое издание книги, которое можно загрузить с сайта *www.MindView.net*). В общих чертах это аналог списка `ArrayList`, но только с длинными и неуклюжими именами методов. В библиотеке контейнеров Java 2 класс `Vector` был переработан, теперь с ним можно обращаться как с коллекцией (`Collection`) или как со списком (`List`). Но не стоит воспринимать эти изменения чересчур оптимистично и думать, что `Vector` как контейнер стал лучше — он включен в библиотеку исключительно для поддержки старого кода.

Итератор, вошедший в Java версий 1.0/1.1, был назван новым именем «перечисление» (`enumeration`), вместо устоявшегося термина «итератор», к которому привыкли все разработчики. Интерфейс `Enumeration` компактнее, чем интерфейс `Iterator`, он содержит только два метода, которые имеют более длинные имена: метод `boolean`

`hasMoreElements()` возвращает `true`, если в последовательности имеются еще элементы, а метод `Object nextElement()` возвращает следующий элемент последовательности, если таковой присутствует (иначе возбуждается исключение).

`Enumeration` — это просто интерфейс, у него нет реализации, и бывает, что новые библиотеки используют этот устаревший интерфейс — это неприятно, но не смертельно. И хотя по возможности в собственном коде следует применять итератор `Iterator`, вам стоит быть начеку и быть готовыми к обработке `Enumeration`.

Вдобавок вы можете получить итератор `Enumeration` для любой коллекции (`Collection`), при помощи метода `Collections.enumeration()`, как это сделано в следующем примере:

```
//: containers/Enumerations.java
// Контейнер Java 1.0/1.1 Vector и итератор Enumeration.
import java.util.*;
import net.mindview.util.*;

public class Enumerations {
    public static void main(String[] args) {
        Vector<String> v =
            new Vector<String>(Countries.names(10));
        Enumeration<String> e = v.elements();
        while(e.hasMoreElements())
            System.out.print(e.nextElement() + ", ");
        // Получение Enumeration для Collection:
        e = Collections.enumeration(new ArrayList<String>());
    }
} /* Output:
ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO,
BURUNDI, CAMEROON, CAPE VERDE, CENTRAL AFRICAN REPUBLIC,
*///:~
```

Для получения `Enumeration` следует вызвать метод `elements()`, после чего можно проводить перебор элементов (только в прямом порядке).

Последняя строка программы создает список `ArrayList` и применяет к нему метод `enumeration()`, чтобы адаптировать итератор `Iterator` к старому `Enumeration`. Таким образом, даже если вы имеете старый код, которому необходим `Enumeration`, он может использоваться с новыми контейнерами.

Hashtable

Как вы могли видеть из сравнительного анализа быстродействия контейнеров, в основном контейнер `Hashtable` весьма схож с картой `HashMap`, даже имена методов у них похожи. А это означает, что в новом коде нет никаких причин использовать класс `Hashtable` вместо `HashMap`.

Stack

Понятие стека мы рассмотрели чуть раньше, когда изучали список `LinkedList`. Что действительно странно в реализации стека, имеющегося в Java 1.0/1.1, — вместо использования класса `Vector` посредством композиции класс `Stack` был *унаследован* от

него. Таким образом, этот класс обладает всеми характеристиками класса `Vector`, плюс несколько дополнительных возможностей. Сложно понять, действительно ли создатели класса думали, что так нужно поступать всегда и везде, или так наивно подошли к проектированию; в любом случае код класса явно не проходил рецензирования перед выпуском, поэтому недостатки его мы можем наблюдать *до сих пор* (никогда не используйте его).

Ниже приведена простая демонстрация, в которой в стек помещаются строковые представления элементов перечисления (`enum`). Она также показывает, что в качестве стека можно использовать `LinkedList` или класс `Stack`, созданный нами в главе 11.

```
//: containers/Stacks.java
// Демонстрация класса Stack.
import java.util.*;
import static net.mindview.util.Print.*;
enum Month { JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
    JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER }

public class Stacks {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<String>();
        for(Month m : Month.values())
            stack.push(m.toString());
        print("stack = " + stack);
        // Работа со стеком как с Vector:
        stack.addElement("The last line");
        print("element 5 = " + stack.elementAt(5));
        print("popping elements:");
        while(!stack.empty())
            printnb(stack.pop() + " ");

        // Использование LinkedList как стека:
        LinkedList<String> lstack = new LinkedList<String>();
        for(Month m : Month.values())
            lstack.addFirst(m.toString());
        print("lstack = " + lstack);
        while(!lstack.isEmpty())
            printnb(lstack.removeFirst() + " ");

        // Использование класса Stack из главы 11:
        net.mindview.util.Stack<String> stack2 =
            new net.mindview.util.Stack<String>();
        for(Month m : Month.values())
            stack2.push(m.toString());
        print("stack2 = " + stack2);
        while(!stack2.empty())
            printnb(stack2.pop() + " ");
    }
} /* Output:
stack = [JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY,
AUGUST, SEPTEMBER, OCTOBER, NOVEMBER]
element 5 = JUNE
popping elements:
The last line NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE
MAY APRIL MARCH FEBRUARY JANUARY lstack = [NOVEMBER,
OCTOBER, SEPTEMBER, AUGUST, JULY, JUNE, MAY, APRIL, MARCH,
```



```

FEBRUARY, JANUARY]
NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL MARCH
FEBRUARY JANUARY stack2 = [NOVEMBER, OCTOBER, SEPTEMBER,
AUGUST, JULY, JUNE, MAY, APRIL, MARCH, FEBRUARY, JANUARY]
NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL MARCH
FEBRUARY JANUARY
*///:~

```

Строковое представление строится по константам перечисления `Month`, которые добавляются в стек методом `push()`, а позже извлекаются оттуда методом `pop()`. Чтобы специально подчеркнуть эту возможность, показано, как можно вызвать методы класса `Vector` для объекта `Stack`. Это делается очень просто, так как, по определению наследования, класс `Stack` *«является частным случаем»* класса `Vector`. Таким образом, все операции, описанные в классе `Vector`, применимы и для класса `Stack`: например, можно получить произвольный элемент стека методом `elementAt()`.

Как уже было сказано, если вам понадобятся свойства стека, используйте для этого класс `LinkedList` или класс `net.mindview.util.Stack`, созданный на базе класса `LinkedList`.

BitSet

Класс `BitSet` служит для эффективного хранения большого количества «переключателей» с двумя состояниями (включено-выключено). Он эффективен только с точки зрения размеров; по эффективности доступа он обычно уступает низкоуровневым массивам.

К тому же минимальный размер данных, хранимых классом `BitSet`, эквивалентен типу длинного целого (`long`): 64 бита. Поэтому при хранении данных меньшего размера — например, 8-битовых значений — `BitSet` станет крайне неэффективным; в таком случае лучше создать свой класс, или просто массив, чтобы хранить в нем флаги. (Впрочем, это утверждение истинно только при создании большого количества объектов, содержащих списки данных с двумя состояниями, и принимать решение следует только на основании профилирования и других метрик. Если вы будете руководствоваться только тем, что данные, по вашему мнению, занимают слишком много памяти, это приведет лишь к усложнению и напрасным тратам времени.)

Стандартный контейнер автоматически меняет свой размер по мере добавления в него новых элементов, и контейнер `BitSet` не является исключением. Следующий пример показывает, как работает этот класс:

```

//: containers/Bits.java
// Демонстрация BitSet.
import java.util.*;
import static net.mindview.util.Print.*;

public class Bits {
    public static void printBitSet(BitSet b) {
        print("bits: " + b);
        StringBuilder bbits = new StringBuilder();
        for(int j = 0; j < b.size(); j++)
            bbits.append(b.get(j) ? "1" : "0");
        print("bit pattern: " + bbits);
    }
}

```

продолжение ➤

[illegible]

```
26, 31}
bit pattern:
11010101010100000011011111000010000000000000000000000000000000
00000
set bit 127: {127}
set bit 255: {255}
set bit 1023: {1023, 1024}
*///:~
```

Для создания случайных чисел `byte`, `short` и `int` используется генератор случайных чисел, после чего полученные числа превращаются в соответствующую маску битов (с помощью класса `BitSet`). Это прекрасно работает, поскольку изначальный размер данных контейнера `BitSet` — 64 бита и расширять его нет необходимости. Затем создаются контейнеры `BitSet` большего размера.

`EnumSet` (см. главу 19) обычно предпочтительнее `BitSet` при работе с фиксированным набором флагов, которому можно присвоить имя (вместо числовых позиций битов), — это снижает вероятность ошибок, потому что `EnumSet` позволяет работать с именами вместо позиций конкретных битов. `EnumSet` также предотвращает случайное добавление новых флагов, что приводит к появлению серьезных, трудно обнаружимых ошибок. Существует совсем немного ситуаций, в которых есть смысл использовать `BitSet` вместо `EnumSet`, — если необходимое количество флагов неизвестно заранее, или если присваивание имен флагам не имеет смысла, или если вам нужны специальные операции `BitSet` (см. документацию `BitSet` и `EnumSet` в JDK).

Резюме

Пожалуй, библиотека контейнеров является самой важной библиотекой в объектно-ориентированном языке. В большинстве программ контейнеры используются чаще любых других библиотечных компонентов. Некоторые языки (например, Python) даже содержат встроенные компоненты основных контейнеров (списки, карты и множества).

Как было показано в главе 11, контейнеры позволяют выполнить без особых усилий много интересных операций. Однако на какой-то стадии для правильного использования контейнеров вам потребуется больше знать о них — в частности, нужно разбираться в хешировании для написания собственного метода `hashCode()` (и знать, когда это необходимо), а также уметь выбрать из нескольких реализаций ту, которая лучше всего подходит для ваших целей. В этой главе рассматриваются эти концепции, а также приводится дополнительная полезная информация о библиотеке контейнеров. К завершению этой главы вы должны быть хорошо подготовлены к использованию контейнеров Java в повседневной работе.

Хорошо спроектировать библиотеку контейнеров непросто (впрочем, это относится к большинству библиотек). В C++ классы контейнеров охватывали весь спектр задач с использованием множества разных классов. Такой подход был лучше того, что было до появления контейнеров C++ (ничего не было), но он плохо адаптировался для Java. Я видел и другую крайность: библиотеку контейнеров из одного класса, который одновременно работал как линейная последовательность и как ассоциативный массив. Библиотека контейнеров Java выдерживает баланс между двумя крайностями:

полноценная функциональность, которую можно ожидать от проверенной временем библиотеки контейнеров, которую проще изучить и использовать, чем классы контейнеров C++ и другие похожие библиотеки контейнеров. Результат порой выглядит немного странно. Однако в отличие от некоторых решений, принятых на ранней стадии разработки библиотек Java, это не случайности, а тщательно продуманные решения, которые базируются на компромиссах, связанных со сложностью реализации.

18

Система ввода-вывода Java

Создание хорошей системы ввода-вывода — одна из труднейших задач проектировщика языка. На это указывает разнообразие подходов, используемых при разработке систем ввода-вывода.

Основная сложность состоит в том, что необходимо учесть все возможные ситуации. Это не только наличие множества источников и приемников данных, с которыми необходимо поддерживать связь (файлы, консоль, сетевые соединения), но и необходимость реализации различных форм этой связи (последовательный доступ, произвольный, буферизованный, двоичный, символьный, построчный, пословный и т. д.).

Разработчики библиотеки Java решили начать с создания огромного количества классов. Вообще говоря, в библиотеке ввода-вывода Java так много классов, что она выглядит довольно устрашающе (парадоксально, но сама система ввода-вывода Java в действительности не нуждается в таком количестве классов). Потом, после выхода первой версии языка Java, в библиотеке ввода-вывода последовали значительные изменения: к ориентированным на передачу и прием *байтов* классам добавились классы на базе Юникода, работающие с *символами*. В JDK 1.4 классы `nio` (от сочетания «new I/O», «новый ввод-вывод», название, которое мы будем использовать и через несколько лет, когда их уже будет трудно назвать «новыми») призваны улучшить производительность и функциональность. В результате, прежде чем использовать операции ввода-вывода, вам придется изучить порядочный ворох классов. Вдобавок не менее важно понять и изучить эволюцию библиотеки ввода-вывода, несмотря на очевидную первую реакцию: «Оставьте историю себе! Покажите просто, как работать с библиотекой!» На самом деле, если мы не уясним причин изменений, проведенных в библиотеке ввода-вывода, то вскоре запутаемся в ней и не сможем твердо аргументировать сделанный нами выбор в пользу того или иного класса.

Эта глава поможет понять основы классов, отвечающих за ввод-вывод в библиотеке Java, а также покажет, как следует их использовать.

Класс File

Перед тем как перейти к классам, которые осуществляют реальные запись и чтение данных, мы рассмотрим вспомогательные инструменты библиотеки, чтобы облегчить обращение с файлами и каталогами.

Имя класса `File` весьма обманчиво: легко подумать, что оно всегда относится к файлу, но это не так. Класс `File` может представлять *имя* определенного файла или набор файлов, находящихся в каталоге. Если класс представляет каталог, можно вызвать его метод `list()`, чтобы получить массив строк, содержащих имена всех файлов. Использовать в данной ситуации массив (а не более гибкий контейнер) очень удобно: количество файлов в каталоге выражается конечным числом, как и размер массива, а если понадобится узнать имена файлов в другом каталоге, надо просто создать еще один объект `File`. Вообще говоря, для этого класса куда лучше подошло бы имя `FilePath` (путь к файлу). Следующий раздел покажет, как использовать этот класс в совокупности с тесно связанным с ним интерфейсом `FilenameFilter`.

Получение содержимого каталогов

Предположим, что вам необходимо увидеть список содержимого каталога. Объект `File` позволяет получить этот список двумя способами. Если вызвать метод `list()` без аргументов, то результатом будет полный список файлов и каталогов (точнее, их названий), содержащихся в данном каталоге. Но если вам нужен ограниченный список — например, список, включающий в себя только файлы, имеющие расширение `.java`, — используйте «фильтр для каталогов», представляющий собой класс, который настраивает последующий выбор объектов `File`.

Следующий пример реализует эту идею. Заметьте, что полученный список был без всяких дополнительных усилий отсортирован (по алфавиту) с помощью метода `java.util.Array.sort()` и компаратора `String.CASE_INSENSITIVE_ORDER`:

```
//: io/DirList.java
// Вывод содержимого каталога с использованием регулярных выражений.
// {Args: "D.*\.java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList {
    public static void main(String[] args) {
        File path = new File(".");
        String[] list;
        if(args.length == 0)
            list = path.list();
        else
            list = path.list(new DirFilter(args[0]));
        Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
        for(String dirItem : list)
            System.out.println(dirItem);
    }
}
```

```
class DirFilter implements FilenameFilter {
    private Pattern pattern;
    public DirFilter(String regex) {
        pattern = Pattern.compile(regex);
    }
    public boolean accept(File dir, String name) {
        return pattern.matcher(name).matches();
    }
} /* Output:
DirectoryDemo.java
DirList.java
DirList2.java
DirList3.java
*///:~
```

Класс `DirFilter` реализует интерфейс `FilenameFilter`. Обратите внимание, как просто выглядит интерфейс `FilenameFilter`:

```
public interface FilenameFilter {
    boolean accept(File dir, String name);
}
```

Интерфейс `DirFilter` существует по единственной причине: он предоставляет метод `accept()` для метода `list()`, чтобы метод `list()` мог вызывать `accept()` для определения имен файлов, включаемых в список. Такая структура часто называется *обратным вызовом* (callback). А точнее, перед нами пример паттерна проектирования «Стратегия»: `list()` реализует базовую функциональность, а вы предоставляете «стратегию» в форме `FilenameFilter`; она завершает алгоритм, необходимый `list()` для предоставления его сервиса. Так как `list()` в своем аргументе получает объект `FilenameFilter`, ему можно передать любой объект любого класса, реализующего интерфейс `FilenameFilter`, чтобы выбрать поведение метода `list()` (даже во время выполнения). Таким образом легко изменять результат работы метода `list()`. В этом состоит цель обратного вызова — обеспечить подходящую гибкость кода.

Метод `accept()` принимает в качестве аргументов объект `File`, представляющий каталог, в котором был найден данный файл, и строку, содержащую имя файла. Вы вправе игнорировать эти аргументы или нет, но обычно используется имя файла. Помните, что метод `list()` вызывает метод `accept()` для всех файлов, обнаруженных в каталоге, чтобы определить, какие из них следует включить в список; это зависит от логического результата, возвращаемого методом `accept()`.

Метод `accept()` использует объект поиска по регулярному выражению `matcher` для проверки того, соответствует ли имя файла регулярному выражению `regex`. Метод `list()` возвращает массив.

Анонимные внутренние классы

Описанный пример идеально подходит для демонстрации преимуществ внутренних классов (описанных в главе 10). Для начала создадим метод `filter()`, который возвращает ссылку на объект `FilenameFilter`:

```
//: io/DirList2.java
// Использование анонимных внутренних классов.
// {Args: "D.*\java"}
```

продолжение ➤

```

import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList2 {
    public static FilenameFilter filter(final String regex) {
        // Создание анонимного внутреннего класса:
        return new FilenameFilter() {
            private Pattern pattern = Pattern.compile(regex);
            public boolean accept(File dir, String name) {
                return pattern.matcher(name).matches();
            }
        }; // Конец анонимного внутреннего класса:
    }
    public static void main(String[] args) {
        File path = new File(".");
        String[] list;
        if(args.length == 0)
            list = path.list();
        else
            list = path.list(filter(args[0]));
        Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
        for(String dirItem : list)
            System.out.println(dirItem);
    }
} /* Output:
DirectoryDemo.java
DirList.java
DirList2.java
DirList3.java
*///:~

```

Заметьте, что аргумент метода `filter()` должен быть неизменным (`final`). Это необходимо для того, чтобы анонимный внутренний класс смог получить к нему доступ даже за пределами своей области действия.

Несомненно, структура программы улучшилась — хотя бы потому, что объект `FilenameFilter` теперь неразрывно связан с внешним классом `DirList2`. Впрочем, можно продолжить совершенствование и определить безымянный внутренний класс как аргумент метода `list()`; в этом случае код станет еще компактнее:

```

//: io/DirList3.java
// Построение анонимного внутреннего класса "на месте".
// {Args: "D.*\*.java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList3 {
    public static void main(final String[] args) {
        File path = new File(".");
        String[] list;
        if(args.length == 0)
            list = path.list();
        else
            list = path.list(new FilenameFilter() {
                private Pattern pattern = Pattern.compile(args[0]);

```



```
        public boolean accept(File dir, String name) {
            return pattern.matcher(name).matches();
        }
    });
    Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
    for(String dirItem : list)
        System.out.println(dirItem);
}
} /* Output:
DirectoryDemo.java
DirList.java
DirList2.java
DirList3.java
*///:~
```

На этот раз неизменным (`final`) оказывается аргумент метода `main()`, так как безымянный внутренний класс использует параметр командной строки (`args[0]`) напрямую.

Именно так безымянные внутренние классы позволяют быстро создать специализированный «одноразовый» класс для решения конкретной задачи. Одно из преимуществ такого решения заключается в том, что составные части кода, решающего определенную задачу, находятся в одном месте. С другой стороны, полученный код читается сложнее, поэтому применять его следует осмотнительно.

1. (3) Измените программу `DirList.java` (или одну из ее разновидностей) так, чтобы объект `FilenameFilter` открывал и читал каждый файл (с использованием класса `net.mindview.util.TextFile`) и принимал файл на основании того, присутствуют ли в этом файле аргументы командной строки.
2. (2) Создайте класс с именем `SortedDirList`, конструктор которого принимает информацию о пути к файлу и на основе этой информации составляет отсортированный список файлов в указанном каталоге. Создайте два перегруженных метода `list()`, один должен возвращать весь список файлов, другой — подмножество списка, соответствующее аргументу (регулярное выражение).
3. (3) Измените программу `DirList.java` (или одну из ее разновидностей) так, чтобы она суммировала размеры выбранных файлов.

Вспомогательные средства для работы с каталогами

В программировании часто встречается задача выполнения операций с наборами файлов — либо находящихся в локальном каталоге, либо найденных при переборе всего дерева каталогов. Полезно иметь инструмент, который будет создавать наборы файлов за вас. Следующий вспомогательный класс создает либо массив объектов `File` для локального каталога (метод `local()`), либо контейнер `List<File>` для всего дерева каталогов, начиная с заданного каталога (метод `walk()`). Объекты `File` полезнее файлов, потому что они содержат больше информации. Файлы выбираются на основании переданного вами регулярного выражения:

```
//: net/mindview/util/Directory.java
// Создание последовательности объектов File, соответствующих
// регулярному выражению - либо в локальном каталоге, либо directory,
```

продолжение ➤

```

// посредством обхода дерева каталогов.
package net.mindview.util;
import java.util.regex.*;
import java.io.*;
import java.util.*;

public final class Directory {
    public static File[]
    local(File dir, final String regex) {
        return dir.listFiles(new FilenameFilter() {
            private Pattern pattern = Pattern.compile(regex);
            public boolean accept(File dir, String name) {
                return pattern.matcher(
                    new File(name).getName()).matches();
            }
        });
    }
    public static File[]
    local(String path, final String regex) { // Перегружен
        return local(new File(path), regex);
    }
    // Кортеж для возвращения пары объектов:
    public static class TreeInfo implements Iterable<File> {
        public List<File> files = new ArrayList<File>();
        public List<File> dirs = new ArrayList<File>();
        // The default iterable element is the file list:
        public Iterator<File> iterator() {
            return files.iterator();
        }
        void addAll(TreeInfo other) {
            files.addAll(other.files);
            dirs.addAll(other.dirs);
        }
        public String toString() {
            return "dirs: " + PPrint.pformat(dirs) +
                "\n\nfiles: " + PPrint.pformat(files);
        }
    }
    public static TreeInfo
    walk(String start, String regex) { // Начало рекурсии
        return recurseDirs(new File(start), regex);
    }
    public static TreeInfo
    walk(File start, String regex) { // Перегружен
        return recurseDirs(start, regex);
    }
    public static TreeInfo walk(File start) { // Bce
        return recurseDirs(start, ".*");
    }
    public static TreeInfo walk(String start) {
        return recurseDirs(new File(start), ".*");
    }
    static TreeInfo recurseDirs(File startDir, String regex){
        TreeInfo result = new TreeInfo();
        for(File item : startDir.listFiles()) {
            if(item.isDirectory()) {
                result.dirs.add(item);
                result.addAll(recurseDirs(item, regex));
            }
        }
    }
}

```

```

    } else // Обычный файл
    {
        if(item.getName().matches(regex))
            result.files.add(item);
    }
    return result;
}
// Простая проверка:
public static void main(String[] args) {
    if(args.length == 0)
        System.out.println(walk("."));
    else
        for(String arg : args)
            System.out.println(walk(arg));
}
} ///:~

```

Метод `local()` использует разновидность `File.list()` — метод `listFiles()`, возвращающий массив объектов `File`. Также из листинга видно, что он использует `FilenameFilter`. Если вам понадобится список `List` вместо массива, преобразуйте результат самостоятельно вызовом `Arrays.asList()`.

Метод `walk()` преобразует имя начального каталога в объект `File` и вызывает метод `recurseDirs()`, который выполняет рекурсивный обход каталога, получая новую информацию при каждой рекурсии. Чтобы обычные файлы не путались с каталогами, возвращаемое значение представляет собой «кортеж» объектов — список `List` с обычными файлами, и другой — с каталогами. Поля намеренно объявлены открытыми, потому что целью `TreeInfo` является простое объединение объектов — если бы вы просто возвращали `List`, вы бы не стали делать его закрытым, так что возвращение пары объектов не означает, что их нужно объявлять закрытыми. Обратите внимание: `TreeInfo` реализует интерфейс `Iterable<File>`, который производит файлы, так что у вас имеется «перебор по умолчанию» для списка файлов; чтобы указать каталоги, используйте запись «`.dirs`».

Метод `TreeInfo.toString()` использует класс «аккуратного вывода», чтобы вывод было удобнее просматривать. По умолчанию методы `toString()` для контейнеров выводят все элементы контейнера в одну строку. Для больших коллекций такие результаты трудно читать, поэтому мы воспользуемся альтернативным форматированием. Следующий класс добавляет новые строки и отступы для каждого элемента:

```

//: net/mindview/util/PPrint.java
// Аккуратный вывод коллекций
package net.mindview.util;
import java.util.*;
public class PPrint {
    public static String pformat(Collection<?> c) {
        if(c.size() == 0) return "[]";
        StringBuilder result = new StringBuilder("[");
        for(Object elem : c) {
            if(c.size() != 1)
                result.append("\n ");
            result.append(elem);
        }
        if(c.size() != 1)
            result.append("\n");
    }
}

```

продолжение ➤

```

    result.append("]");
    return result.toString();
}
public static void pprint(Collection<?> c) {
    System.out.println(pformat(c));
}
public static void pprint(Object[] c) {
    System.out.println(pformat(Arrays.asList(c)));
}
} ///:~

```

Метод `pformat()` создает отформатированную строку на базе `Collection`, а метод `pprint()` использует `pformat()` для выполнения своей работы. Обратите внимание на отдельную обработку особых случаев: отсутствие элементов и один элемент. Также имеется версия `pprint()` для массивов.

Вспомогательный класс `Directory` включен в пакет `net.mindview.util`, чтобы с ним было удобно работать. Пример использования:

```

//: io/DirectoryDemo.java
// Пример использования вспомогательного класса Directory.
import java.io.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DirectoryDemo {
    public static void main(String[] args) {
        // Все каталоги
        PPrint.pprint(Directory.walk(".").dirs);
        // Все файлы, имена которых начинаются с 'T'
        for(File file : Directory.local(".", "T.*"))
            print(file);
        print("-----");
        // Все файлы Java, имена которых начинаются с 'T'
        for(File file : Directory.walk(".", "T.*\\.java"))
            print(file);
        print("=====");
        // Файлы классов, имена которых содержат "Z" или "z":
        for(File file : Directory.walk(".", ".*[Zz].*\\.class"))
            print(file);
    }
} /* Output: (Sample)
[.\xfiles]
.\TestEOF.class
.\TestEOF.java
.\TransferTo.class
.\TransferTo.java
-----
.\TestEOF.java
.\TransferTo.java
.\xfiles\ThawAlien.java
=====
.\FreezeAlien.class
.\GZIPcompress.class
.\ZipCompress.class
*///:~

```

Если вам непонятен второй аргумент в вызовах `local()` и `walk()`, вернитесь к главе 13 и освежите в памяти синтаксис регулярных выражений.

Можно пойти еще дальше и создать класс, который будет обходить каталоги и обрабатывать содержащиеся в них файлы в соответствии с объектом Strategy (еще один пример паттерна проектирования «Стратегия»):

```
//: net/mindview/util/ProcessFiles.java
package net.mindview.util;
import java.io.*;

public class ProcessFiles {
    public interface Strategy {
        void process(File file);
    }
    private Strategy strategy;
    private String ext;
    public ProcessFiles(Strategy strategy, String ext) {
        this.strategy = strategy;
        this.ext = ext;
    }
    public void start(String[] args) {
        try {
            if(args.length == 0)
                processDirectoryTree(new File("."));
            else
                for(String arg : args) {
                    File fileArg = new File(arg);
                    if(fileArg.isDirectory())
                        processDirectoryTree(fileArg);
                    else {
                        // Разрешить пользователю не указывать расширение:
                        if(!arg.endsWith("." + ext))
                            arg += "." + ext;
                        strategy.process(
                            new File(arg).getCanonicalFile());
                    }
                }
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
    }
    public void
    processDirectoryTree(File root) throws IOException {
        for(File file : Directory.walk(
            root.getAbsolutePath(), ".*\\" + ext))
            strategy.process(file.getCanonicalFile());
    }
    // Демонстрация использования:
    public static void main(String[] args) {
        new ProcessFiles(new ProcessFiles.Strategy() {
            public void process(File file) {
                System.out.println(file);
            }
        }, "java").start(args);
    }
} /* (Execute to see output) *///:~
```

Интерфейс «Стратегия» вложен в ProcessFiles, поэтому реализовать его приходится в записи ProcessFiles.Strategy, предоставляющей больше контекстной информации для читателя кода. ProcessFiles выполняет всю работу по поиску файлов с конкретным

расширением (аргумент ext конструктора), а когда подходящий файл будет найден, он просто передается объекту Strategy (который также является аргументом конструктора).

Если аргументы не заданы, ProcessFiles считает, что вы хотите перебрать все подкаталоги текущего каталога. Также можно задать конкретный файл с расширением или без (расширение будет добавлено при необходимости) или один или несколько каталогов.

В методе main() представлен простейший вариант использования вспомогательного класса; он выводит имена всех файлов с исходным кодом Java в соответствии с заданной командной строкой.

4. (2) Используйте вызов Directory.walk() для суммирования размеров всех файлов в дереве каталогов, имена которых соответствуют заданному регулярному выражению.
5. (1) Измените код ProcessFiles.java так, чтобы вместо фиксированного расширения поиск осуществлялся по регулярному выражению.

Проверка и создание каталогов

Возможности класса File не ограничиваются представлением существующих файлов или каталогов. Объект File также можно использовать для создания нового каталога или даже дерева каталогов, если последние не существуют. Можно также проверить характеристики файлов (размер, дату последнего изменения, режим чтения/записи), проверить, представляет ли объект File файл или каталог, удалить файл. Следующая программа демонстрирует некоторые методы класса File (чтобы узнать обо всех его возможностях, обращайтесь к документации JDK на сайте <http://java.sun.com>):

```
//: io/MakeDirectories.java
// Использование класса File для создания
// каталогов и выполнения операций с файлами.
// {Args: MakeDirectoriesTest}
import java.io.*;

public class MakeDirectories {
    private static void usage() {
        System.err.println(
            "Usage:MakeDirectories path1 ... \n" +
            "Creates each path\n" +
            "Usage:MakeDirectories -d path1 ... \n" +
            "Deletes each path\n" +
            "Usage:MakeDirectories -r path1 path2\n" +
            "Renames from path1 to path2");
        System.exit(1);
    }
    private static void fileData(File f) {
        System.out.println(
            "Absolute path: " + f.getAbsolutePath() +
            "\n Can read: " + f.canRead() +
            "\n Can write: " + f.canWrite() +
            "\n getName: " + f.getName() +
            "\n getParent: " + f.getParent() +
            "\n getPath: " + f.getPath() +
            "\n length: " + f.length() +
```

```

        "\n lastModified: " + f.lastModified());
    if(f.isFile())
        System.out.println("It's a file");
    else if(f.isDirectory())
        System.out.println("It's a directory");
}
public static void main(String[] args) {
    if(args.length < 1) usage();
    if(args[0].equals("-r")) {
        if(args.length != 3) usage();
        File
            old = new File(args[1]),
            rname = new File(args[2]);
        old.renameTo(rname);
        fileData(old);
        fileData(rname);
        return; // Выход из main
    }
    int count = 0;
    boolean del = false;
    if(args[0].equals("-d")) {
        count++;
        del = true;
    }
    count--;
    while(++count < args.length) {
        File f = new File(args[count]);
        if(f.exists()) {
            System.out.println(f + " exists");
            if(del) {
                System.out.println("deleting..." + f);
                f.delete();
            }
        }
        else { // Не существует
            if(!del) {
                f.mkdirs();
                System.out.println("created " + f);
            }
        }
        fileData(f);
    }
}
} /* Output: (80% match)
created MakeDirectoriesTest
Absolute path: d:\aaa-TIJ4\code\io\MakeDirectoriesTest
Can read: true
Can write: true
getName: MakeDirectoriesTest
getParent: null
getPath: MakeDirectoriesTest
length: 0
lastModified: 1101690308831
It's a directory
*///:~

```

В методе `fileData()` представлены различные методы, предназначенные для получения информации о файлах и каталогах.

Первым делом в методе `main()` вызывается метод `renameTo()`, который позволяет переименовывать (или перемещать) файлы, используя для этого второй аргумент — еще один объект `File`, который указывает на новое местоположение или имя.

Немного поэкспериментировав с этой программой, вы увидите, что создать пути произвольной сложности очень просто, поскольку всю работу за вас фактически делает метод `mkdirs()`.

6. (5) Используйте `ProcessFiles` для поиска в заданном поддереве каталогов всех файлов с исходным кодом Java, измененных после конкретной даты.

Ввод и вывод

В библиотеках ввода-вывода часто используется абстракция *потока* (`stream`) — произвольного источника или приемника данных, который способен производить или получать некоторую информацию. Поток скрывает подробности низкоуровневых операций, выполняемых с данными непосредственно в устройствах ввода-вывода.

Классы библиотеки ввода-вывода Java разделены на две части — одни осуществляют ввод, другие вывод. В этом можно убедиться, просмотрев HTML-документацию JDK в вашем браузере. Вследствие применения наследования все классы, производные от базовых классов `InputStream` или `Reader`, имеют методы с именами `read()` для чтения одиночных байтов или массива байтов. Аналогично, все классы, унаследованные от базовых классов `OutputStream` или `Writer`, имеют методы с именами `write()` для записи одиночных байтов или массива байтов. Впрочем, вы вряд ли станете использовать эти методы напрямую — они в основном предназначены для других классов, а другие классы на их основе предоставляют более удобный интерфейс. Объекты потоков редко создаются на основе одного класса; чаще несколько объектов «наслаиваются» друг на друга для получения необходимой функциональности (пример паттерна проектирования «Декоратор», о котором будет рассказано в этом разделе). Необходимость создания потока на основе нескольких объектов — главная причина затруднений с библиотекой ввода-вывода Java у новичков.

Для изучения полезно разделить классы по категориям в зависимости от их функций. В первой версии Java (1.0) разработчики решили, что все, связанное с вводом данных, должно быть унаследовано от базового класса `InputStream`, а все, имеющее отношение к выводу данных, — от класса `OutputStream`.

В этой главе используется тот же подход, что и в предыдущих главах: я приведу обзор классов, но за полной информацией (например, полным списком методов конкретного класса) следует обращаться к документации JDK.

Типы `InputStream`

Назначение базового класса `InputStream` — представлять классы, которые получают данные из различных источников. Такими источниками могут быть:

- ☐ массив байтов;
- ☐ строка (объект `String`);

- ❑ файл;
- ❑ «канал» (pipe), который работает как настоящий физический трубопровод: вы помещаете данные в один его «конец», а извлекаете их из другого конца;
- ❑ последовательность различных потоков, которые можно объединить в одном потоке;
- ❑ другие источники, подобные соединениям с сетью Интернет (такие возможности описаны в книге *Thinking In Enterprise Java*, доступной на сайте www.MindView.net).

Каждый из указанных источников имеет ассоциированный с ним подкласс базового класса `InputStream` (табл. 18.1). Дополнительно, существует еще класс `FilterInputStream`, который также является производным классом `InputStream` и представляет собой основу для классов-«декораторов», добавляющих к входным потокам полезные свойства и интерфейсы. Его мы обсудим чуть позже.

Таблица 18.1. Разновидности входных потоков `InputStream`

Класс	Назначение	Аргументы конструктора/ порядок применения
<code>ByteArrayInputStream</code>	Позволяет использовать буфер в памяти (массив байтов) в качестве источника данных для входного потока	Буфер, из которого берутся данные. Как источник данных. Присоедините этот поток к классу <code>FilterInputStream</code> , чтобы получить удобный интерфейс
<code>StringBufferInputStream</code>	Превращает строку (<code>String</code>) во входной поток данных <code>InputStream</code>	Строка. Лежащая в основе класса реализация на самом деле использует класс <code>StringBuffer</code> . Как источник данных. Присоедините этот поток к классу <code>FilterInputStream</code> , чтобы получить удобный интерфейс
<code>FileInputStream</code>	Для чтения информации из файла	Строка (<code>String</code>) с именем файла или объекты <code>File</code> и <code>FileDescriptor</code> . Как источник данных. Присоедините этот поток к классу <code>FilterInputStream</code> , чтобы получить расширенные возможности
<code>PipedInputStream</code>	Производит данные, которые были записаны в соответствующий выходной поток <code>PipedOutputStream</code> . Реализует концепцию канала	Объект <code>PipedOutputStream</code> . Как источник данных в многозадачном окружении. Присоедините этот поток к классу <code>FilterInputStream</code> , чтобы получить удобный интерфейс
<code>SequenceInputStream</code>	Сливает два или более потока <code>InputStream</code> в единый поток	Два объекта-потока <code>InputStream</code> или перечисление <code>Enumeration</code> для контейнера, в котором содержатся все потоки. Как источник данных. Присоедините этот поток к классу <code>FilterInputStream</code> , чтобы получить расширенные возможности

продолжение ➤

Класс	Назначение	Аргументы конструктора/ порядок применения
FilterInputStream	Абстрактный класс, предоставляющий интерфейс для классов-«декораторов», которые добавляют к существующим потокам полезные свойства. См. табл. 18.3	См. табл. 18.3

Типы OutputStream

В данную категорию (табл. 18.2) попадают классы, определяющие, куда направляются ваши данные: в массив байтов (здесь нельзя использовать строки (String) напрямую; предполагается, что вы сможете воссоздать их из массива байтов), в файл или канал.

Вдобавок, класс `FilterOutputStream` предоставляет базовый интерфейс для классов-«декораторов», которые способны добавлять к существующим потокам полезные свойства и интерфейсы. Подробности мы отложим на потом.

Таблица 18.2. Разновидности выходных потоков `OutputStream`

Класс	Назначение	Аргументы конструктора/ порядок применения
<code>ByteArrayOutputStream</code>	Создает буфер в памяти. Все данные, посылаемые в этот поток, размещаются в созданном буфере	Начальный размер буфера (аргумент не является обязательным). Для указания местоположения ваших данных. Присоедините этот поток к объекту <code>FilterOutputStream</code> , чтобы получить удобный интерфейс
<code>FileOutputStream</code>	Отправка данных в файл на диске	Строка с именем файла, или объекты <code>File</code> или <code>FileDescriptor</code> . Используется для указания местоположения данных. Присоедините этот поток к объекту <code>FilterOutputStream</code> , чтобы получить удобный интерфейс
<code>PipedOutputStream</code>	Все данные, записываемые в поток, автоматически появляются в ассоциированном с ним входном потоке <code>PipedInputStream</code> . Реализует понятие канала	Объект <code>PipedInputStream</code> . Используется для указания местонахождения данных в многозадачном окружении. Присоедините этот поток к объекту <code>FilterOutputStream</code> , чтобы получить удобный интерфейс
<code>FilterOutputStream</code>	Абстрактный класс, предоставляющий интерфейс для классов-декораторов, которые добавляют к существующим потокам полезные свойства. См. табл. 18.4	См. табл. 18.4

Добавление атрибутов и интерфейсов

Декораторы (decorators), или *надстройки*, были представлены в главе 15. Библиотека ввода-вывода Java требует разнообразных сочетаний функциональных возможностей; это и стало основной причиной для применения паттерна «Декоратор»¹. Это причина для наличия в библиотеке ввода-вывода классов-«фильтров»: абстрактный класс фильтра является основой для всех существующих декораторов. (Декоратор должен обладать таким же интерфейсом, как «декорируемый» объект, но он также может расширять этот интерфейс; соответствующие примеры встречаются в классах-фильтрах.)

Впрочем, у паттерна «Декоратор» есть свои недостатки. Декораторы предоставляют дополнительную гибкость при написании программы (можно легко совмещать различные атрибуты), но при этом код получается излишне сложным. Причина запутанности библиотеки ввода-вывода Java состоит в том, что для получения желаемого объекта приходится создавать много дополнительных объектов — «ядро» ввода-вывода и набор декораторов.

Классы, предоставляющие интерфейс для управления конкретным объектом `InputStream` или `OutputStream`, называются `FilterInputStream` и `FilterOutputStream` — честно говоря, выбор имен выглядит странно. Эти классы являются производными от базовых классов библиотеки ввода-вывода, `InputStream` и `OutputStream`. Наследование от этих классов — основное требование к классу-декоратору (поскольку таким образом обеспечивается некоторый общий интерфейс для декорируемых объектов).

Чтение из `InputStream` с использованием `FilterInputStream`

Классы, производные от `FilterInputStream` (см. табл. 18.3), выполняют две различные миссии. `DataInputStream` позволяет читать из потока различные типы простейших данных и строки. (Все методы этого класса начинаются словом `read`, с последующим именем примитива, например `readByte()`, `readFloat()` и т. п.) Этот класс, вместе с «симметричным» ему классом `DataOutputStream`, обеспечивает передачу примитивных данных в потоках из одного места в другое. Эти «места» определяются классами, описанными в табл. 18.1.

Другие классы изменяют внутренние механизмы входного потока: делают его буферизованным, запоминают количество прочитанных строк (позволяя вам запросить их по номеру или номерам), позволяют поместить в поток только что считанный символ. Судя по всему, последние два класса сделаны исключительно для удобства построения компилятора (вероятно, были добавлены в ходе эксперимента по «написанию компилятора Java на Java»), и прока в повседневном программировании от них немного.

Ввод данных почти всегда нуждается в буферизации, вне зависимости от присоединенного устройства ввода-вывода, поэтому имеет смысл сделать отдельный класс (или просто специальный метод) не для буферизованного ввода данных, а наоборот, для прямого.

¹ Трудно сказать, насколько удачным было такое решение, особенно если сравнить с простотой библиотек ввода-вывода в других языках. Тем не менее причина была именно такой.

Таблица 18.3. Разновидности `FilterInputStream`

Класс	Назначение	Аргументы конструктора/ порядок применения
<code>DataInputStream</code>	Используется в сочетании с классом <code>DataOutputStream</code> и позволяет читать примитивы (целые (<code>int</code> , <code>long</code>), символы (<code>char</code>) и т. д.) из потока без привязки к платформе	Входной поток <code>InputStream</code> . Содержит все необходимое для чтения всех примитивных типов
<code>BufferedInputStream</code>	Используется для предотвращения физического чтения с устройства при каждом новом запросе данных. Вы просто говорите: «Используем буфер»	Входной поток <code>InputStream</code> , размер буфера (необязательно). По существу, никакого интерфейса этот класс не предоставляет, он просто присоединяет к потоку буфер. Добавьте дополнительно объект, предоставляющий интерфейс
<code>LineNumberInputStream</code>	Следит за количеством считанных из потока строк; вы можете узнать их число, вызвав метод <code>getLineNumber()</code> , а также перейти к определенной строке с помощью метода <code>setLineNumber(int)</code>	Входной поток <code>InputStream</code> . Этот класс просто добавляет подсчет строк, поэтому к нему стоит добавить еще одну, более полезную, интерфейсный объект
<code>PushbackInputStream</code>	Имеет односимвольный буфер, который позволяет помещать в поток только что считанный символ	Входной поток <code>InputStream</code> . Обычно используется при сканировании файлов для компилятора, и, скорее всего, включен в библиотеку именно поэтому. Вероятно, вам он не понадобится

Запись в `OutputStream` с использованием `FilterOutputStream`

«Двоичным дополнением» класса `DataInputStream` является класс `DataOutputStream` (табл. 18.4), который позволяет форматировать и записывать в поток примитивы и строки таким образом, что `DataInputStream` их сможет прочесть и правильно обработать на любой машине и на любой платформе. Все имена методов начинаются со слова `write` (запись), за которым следует имя примитива, например `writeByte()`, `writeFloat()` и т. п.

Класс `PrintStream` изначально был предназначен для вывода значений в таком формате, который позволял бы понять их человеку. В данном аспекте он отличается от класса `DataOutputStream`, потому что цель последнего — разместить данные в потоке так, чтобы `DataInputStream` смог их правильно распознать.

Основные методы класса `PrintStream` — `print()` и `println()`, они перегружены для наиболее часто используемых типов. Разница между ними состоит в том, что метод `println()` после распечатки значения осуществляет перевод на новую строку, в то время как метод `print()` этого не делает.

На практике класс `PrintStream` довольно-таки неудобен, поскольку он перехватывает и скрывает все исключения `IOException`. (Приходится явно вызывать метод `checkError()`, который возвращает `true`, если во время исполнения какого-либо из методов класса произошла ошибка.) К тому же интернационализация этого класса выполнена недостаточно качественно, и он не обрабатывает перевод строки платформенно-независимым способом (все эти проблемы решены в классе `PrintWriter`, описанном ниже).

Класс `BufferedOutputStream` модифицирует поток так, чтобы использовалась буферизация при записи данных, которая предотвращает слишком частые обращения к физическому устройству. Вероятно, при выводе данных в поток всегда стоит применять буферизацию.

Таблица 18.4. Разновидности `FilterOutputStream`

Класс	Назначение	Аргументы конструктора/ порядок применения
<code>DataOutputStream</code>	Используется в сочетании с классом <code>DataInputStream</code> , позволяет записывать в поток примитивы (целые, символы и т. д.) независимо от платформы	Выходной поток <code>OutputStream</code> . Содержит все необходимое для записи простейших типов данных
<code>PrintStream</code>	Используется для форматирования данных при выводе. Если класс <code>DataOutputStream</code> отвечает за хранение данных, то этот класс отвечает за <i>отображение</i> данных	Выходной поток <code>OutputStream</code> , а также необязательный логический (<code>boolean</code>) аргумент, показывающий, нужно ли очищать буфер записи при переходе на новую строку. Должен быть последним в «наслоении» объектов для выходного потока <code>OutputStream</code> . Вероятно, вы будете часто его использовать
<code>BufferedOutputStream</code>	Используется для буферизации вывода, то есть для предотвращения прямой записи в устройство каждый раз при записи данных. Для записи содержимого буфера в устройство используется метод <code>flush()</code>	Выходной поток <code>OutputStream</code> , а также размер буфера записи (необязательно). Не предоставляет интерфейса как такового, просто добавляет буферизацию. К классу следует присоединить более содержательный интерфейсный объект

Классы Reader и Writer

В Java версии 1.1 в библиотеку ввода-вывода были внесены значительные изменения. Когда в первый раз видишь множество новых классов, основанных на базовых классах `Reader` и `Writer`, возникает мысль, что они сконцентрируют внимание на себе, изжив классы `InputStream` и `OutputStream`. Тем не менее это совершенно не так. Хотя некоторые

аспекты изначальной библиотеки ввода-вывода, основанной на потоках, объявлены устаревшими (а при попытке использовать их вы получите предупреждение от компилятора), классы `InputStream` и `OutputStream` все еще предоставляют достаточно полезную функциональность байт-ориентированного ввода-вывода. Одновременно с этим классы `Reader` и `Writer` позволяют проводить операции символьно-ориентированного ввода-вывода в кодировке Юникод. Дополнительно:

1. Выпуск Java версии 1.1 добавил новые классы и к иерархии потоков, основанной на классах `InputStream` и `OutputStream`. Очевидно, что эти классы и не хотели заменять.
2. Бывают ситуации, когда «байтовые» классы должны использоваться *в сочетании* с «символьными». Для этого в библиотеке появились классы-«преобразователи»: `InputStreamReader` конвертирует `InputStream` в `Reader`, а `OutputStreamWriter` трансформирует `OutputStream` в `Writer`.

Основной причиной появления иерархий классов `Reader` и `Writer` стала интернационализация. Старая библиотека ввода-вывода поддерживала только 8-битовые байтовые потоки и зачастую неверно обращалась с 16-битовыми символами Юникода. Именно благодаря символам Юникода возможна интернационализация программ (простейший тип `Java char` (символ) также основан на Юникоде), поэтому новые классы отвечают за их правильное использование в операциях ввода-вывода. Вдобавок, новые средства разработаны так, что работают быстрее старых классов.

Источники и приемники данных

Практически у всех изначальных потоковых классов имеются соответствующие классы `Reader` и `Writer` со схожими функциями, однако работающие с символами Юникода. Впрочем, существует немало ситуаций, где правильным, а зачастую и единственным выбором становятся классы, ориентированные на прием и посылку байтов; в особенности это относится к библиотекам сжатия данных `java.util.zip`. Поэтому лучше всего *попытаться* использовать классы `Reader` и `Writer` где только возможно — вы обнаружите ситуации, в которых эти классы неприменимы, по сообщениям об ошибках от компилятора.

В табл. 18.5 показано соответствие между источниками и получателями информации двух иерархий библиотеки ввода-вывода Java.

Таблица 18.5. Соответствие между источниками и получателями информации двух иерархий библиотеки ввода-вывода Java

Источники и приемники: классы Java 1.0	Соответствующие классы Java 1.1
<code>InputStream</code>	<code>Reader</code> адаптер: <code>InputStreamReader</code>
<code>OutputStream</code>	<code>Writer</code> адаптер: <code>OutputStreamWriter</code>
<code>FileInputStream</code>	<code>FileReader</code>
<code>FileOutputStream</code>	<code>FileWriter</code>
<code>StringBufferInputStream</code> (объявлен устаревшим)	<code>StringReader</code>

Источники и приемники: классы Java 1.0	Соответствующие классы Java 1.1
(отсутствует)	StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

В основном интерфейсы соответствующих друг другу классов из двух разных иерархий очень сходны, если не совпадают.

Изменение поведения потока

Для потоков `InputStream` и `OutputStream` существуют классы-«декораторы» на основе классов `FilterInputStream` и `FilterOutputStream`. Они позволяют модифицировать изначальный поток ввода-вывода так, как это необходимо в данной ситуации. Иерархия на основе классов `Reader` и `Writer` также использует данный подход, но по-другому.

В табл. 18.6 соответствие классов менее точное, чем в предыдущей таблице. Это связано со структурой классов: в то время как `BufferedOutputStream` является подклассом `FilterOutputStream`, класс `BufferedWriter` не унаследован от базового класса `FilterWriter` (от него вообще не происходит ни одного класса, хотя он и является абстрактным — видимо, его поместили в библиотеку просто для полноты картины). Впрочем, у интерфейсов этих классов много общего.

Таблица 18.6. Соответствие между фильтрами двух иерархий библиотеки ввода-вывода Java

Фильтры: классы Java 1.0	Соответствующие классы Java 1.1
<code>FilterInputStream</code>	<code>FilterReader</code>
<code>FilterOutputStream</code>	<code>FilterWriter</code> (абстрактный класс без субклассов)
<code>BufferedInputStream</code>	<code>BufferedReader</code> (также есть метод для чтения строк <code>readLine()</code>)
<code>BufferedOutputStream</code>	<code>BufferedWriter</code>
<code>DataInputStream</code>	Используйте класс <code>DataInputStream</code> (за исключением чтения строк методом <code>readLine()</code> — для их чтения предпочтителен класс <code>BufferedReader</code>)
<code>PrintStream</code>	<code>PrintWriter</code>
<code>LineNumberInputStream</code> (объявлен устаревшим)	<code>LineNumberReader</code>
<code>StreamTokenizer</code>	<code>StreamTokenizer</code> (используйте конструктор с аргументом <code>Reader</code>)
<code>PushBackInputStream</code>	<code>PushBackReader</code>

Одно правило очевидно: для чтения строк больше не следует употреблять класс `DataInputStream` (при такой попытке компилятор сообщит вам, что этот метод для чтения строк устарел), вместо него используйте класс `BufferedReader`. Во всех других ситуациях класс `DataInputStream` остается основным вариантом во всем многообразии библиотеки ввода-вывода.

Чтобы облегчить переход к классу `PrintWriter`, в него добавили конструктор, который принимает в качестве аргумента выходной поток `OutputStream` (обычный конструктор принимает класс `Writer`). Однако возможности вывода на экран и форматирования в классе `PrintWriter` практически идентичны тем, что имелись в классе `PrintStream`.

В Java SE5 были добавлены конструкторы `PrintWriter`, упрощающие создание файлов при выводе (см. далее).

К тому же в конструкторе класса `PrintWriter` можно указать дополнительный флаг, чтобы содержимое буфера сбрасывалось каждый раз при записи новой строки (методом `println()`).

Классы, оставленные без изменений

Некоторые классы остались неизменными между версиями Java 1.0 и 1.1:

- ☐ `DataOutputStream`
- ☐ `File`
- ☐ `RandomAccessFile`
- ☐ `SequenceInputStream`

Следует особо отметить, что без изменений остался класс `DataOutputStream`, так что для передачи данных независимым от платформы и машины способом по-прежнему используются иерархии `InputStream` и `OutputStream`.

RandomAccessFile: сам по себе

Класс `RandomAccessFile` предназначен для работы с файлами, содержащими записи известного размера, по которым можно перемещаться (методом `seek()`) с последующим чтением или записью данных. Записи не обязаны иметь одинаковый размер; вы просто должны уметь определить их размер и то, где они располагаются в файле.

Поначалу с трудом верится, что класс `RandomAccessFile` не является частью иерархии `InputStream` и `OutputStream`. Но тем не менее никаких связей с этими классами и их иерархиями у него нет, разве что он реализует интерфейсы `DataInput` и `DataOutput` (эти интерфейсы также реализуются классами `DataInputStream` и `DataOutputStream`). Он не использует функциональность существующих классов из иерархии `InputStream` и `OutputStream` — это полностью независимый класс, написанный «с нуля», с собственными методами. Причина кроется, скорее всего, в том, что класс `RandomAccessFile` позволяет свободно перемещаться по файлу в прямом и обратном направлении, что для других типов ввода-вывода невозможно. Так или иначе, он стоит особняком и напрямую унаследован от корневого класса `Object`.

По сути, класс `RandomAccessFile` похож на пару совмещенных в одном классе потоков `DataInputStream` и `DataOutputStream`, дополненных методом `getFilePointer()`, показывающим текущую позицию в файле, и методом `length()`, определяющим максимальный размер файла. Вдобавок конструкторам этого класса передается второй аргумент (по аналогии с методом `fopen()` в С), устанавливающий порядок использования файла: только для чтения (строка «r») или и для чтения, и для записи (строка «rw»). Поддержки файлов только для записи нет, поэтому разумно предположить, что класс `RandomAccessFile` можно было бы унаследовать от `DataInputStream` без потери функциональности.

Прямое позиционирование допустимо только для класса `RandomAccessFile`, и работает оно только в случае файлов. Класс `BufferedInputStream` позволяет пометить некоторую позицию потока методом `mark()`, а затем вернуться к ней вызовом `reset()`. Однако эта возможность ограничена (позиция запоминается в единственной внутренней переменной) и потому нечасто востребована.

Большая часть (если не вся) функциональности класса `RandomAccessFile` в JDK 1.4 также реализуется *отображаемыми в память файлами* (memory-mapped files) из нового пакета `nio`. Мы обсудим их чуть позже.

Типичное использование потоков ввода-вывода

Хотя из классов библиотеки ввода-вывода, реализующих потоки, можно составить множество разнообразных конфигураций, обычно используется несколько наиболее употребимых. Ниже рассматриваются некоторые примеры типичного использования ввода-вывода.

Упрощенная обработка исключений — вывод информации на консоль — подходит только для небольших примеров и вспомогательных классов. В вашем коде следует использовать более совершенные механизмы обработки.

Буферизованное чтение из файла

Чтобы открыть файл для посимвольного чтения, используется класс `FileInputStream`, имя файла извлекается из строки (`String`) или объекта `File`. Ускорить процесс чтения помогает буферизация ввода, для чего полученная ссылка передается конструктору класса `BufferedReader`. Так как `BufferedReader` также предоставляет метод `readLine()`, ничего более вам не потребуется. Если метод `readLine()` возвращает ссылку `null`, значит, вы находитесь в конце файла:

```
//: io/BufferedInputFile.java
import java.io.*;

public class BufferedInputFile {
    // Вывод исключений на консоль:
    public static String
    read(String filename) throws IOException {
        // Чтение по строкам:
        BufferedReader in = new BufferedReader(
```

продолжение ➤

```

    new FileReader(filename));
    String s;
    StringBuilder sb = new StringBuilder();
    while((s = in.readLine()) != null)
        sb.append(s + "\n");
    in.close();
    return sb.toString();
}
public static void main(String[] args)
throws IOException {
    System.out.print(read("BufferedInputFile.java"));
}
} /* (Execute to see output) *///:~

```

Объект `StringBuilder sb` служит для того, чтобы собрать воедино весь прочитанный текст (включая переводы строк, поскольку метод `readLine()` их отбрасывает). Эта строка затем еще не раз встретится в программе. В завершение файл закрывается вызовом `close()`¹.

Формально метод `close()` должен вызываться в завершающем методе `finalize()`, а последний должен вызываться при выходе из программы (вне зависимости от того, сработал ли уборщик мусора). Однако это было реализовано недостаточно согласованно, и потому единственный способ завершить ввод-вывод безопасно — явно вызвать метод `close()`.

7. (2) Откройте текстовый файл для построчного чтения. Считывайте из него строки в объект `String`, который затем помещается в `LinkedList`. Выведите все строки в обратном порядке средствами класса `LinkedList`.
8. (1) Измените упражнение 7 так, чтобы имя файла с обрабатываемым текстом передавалось в командной строке.
9. (1) Измените упражнение 8 так, чтобы буквы во всех прочитанных строках преобразовывались к верхнему регистру. Направьте результат в «устройство» стандартного вывода `System.out`.
10. (2) Измените упражнение 8, чтобы программа получала в дополнительных аргументах командной строки слова, которые нужно найти в файле. Выведите все строки, где были найдены указанные слова.
11. (2) В примере `innerclasses/GreenhouseController.java` класс `GreenhouseController` содержит жестко запрограммированный набор событий. Измените программу так, чтобы она читала события и их относительное время из текстового файла. (Уровень сложности 8: используйте для построения событий паттерн «Фабричный метод» — см. *Thinking in Patterns (with Java)* на сайте www.MindView.net).

¹ В исходном варианте архитектуры предполагалось, что метод `close()` будет вызываться при выполнении `finalize()`, и подобное определение `finalize()` встречается для классов ввода-вывода. Однако, как было указано ранее, функциональность `finalize()` работает не так, как ее изначально планировали проектировщики Java (а если говорить точнее — она была безвозвратно «запорота»), так что единственным надежным решением можно считать явный вызов `close()` для файлов.

Чтение из памяти

В этом примере результат `String` из `BufferedInputFile.read()` используется для создания класса `StringReader`. После этого символы читаются методом `read()`, и каждый следующий символ посылается на консоль.

```
//: io/MemoryInput.java
import java.io.*;

public class MemoryInput {
    public static void main(String[] args)
        throws IOException {
        StringReader in = new StringReader(
            BufferedInputFile.read("MemoryInput.java"));
        int c;
        while((c = in.read()) != -1)
            System.out.print((char)c);
    }
} /* (Execute to see output) *///:~
```

Заметьте, что метод `read()` возвращает следующий символ в формате `int`, поэтому для правильного вывода на консоль приходится приводить его результат к типу `char`.

Форматированное чтение из памяти

Для чтения «форматированных» данных применяется класс `DataInputStream`, ориентированный на ввод-вывод байтов (а не символов). Это вынуждает нас работать с классами иерархии `InputStream`, а не их аналогов на основе класса. Можно прочитать все что угодно (например, файл) в виде байтов через `InputStream`, но здесь используется тип `String`:

```
//: io/FormattedMemoryInput.java
import java.io.*;

public class FormattedMemoryInput {
    public static void main(String[] args)
        throws IOException {
        try {
            DataInputStream in = new DataInputStream(
                new ByteArrayInputStream(
                    BufferedInputFile.read(
                        "FormattedMemoryInput.java").getBytes()));
            while(true)
                System.out.print((char)in.readByte());
        } catch EOFException e {
            System.err.println("End of stream");
        }
    }
} /* (Execute to see output) *///:~
```

Поток `ByteArrayInputStream` должен получать массив байтов. Для его создания в `String` имеется метод `getBytes()`. После этой операции у вас оказывается необходимый поток `InputStream`, который можно передать `DataInputStream`.

При чтении байтов из форматированного потока `DataInputStream` методом `readByte()` любое полученное значение будет законным результатом, поэтому он неприменим для идентификации конца потока. Вместо этого можно использовать метод `available()`,

который сообщает, сколько еще осталось символов. Следующий пример показывает, как читать файл побайтно:

```
//: io/TestEOF.java
// Проверка конца файла при побайтовом чтении.
import java.io.*;

public class TestEOF {
    public static void main(String[] args)
        throws IOException {
        DataInputStream in = new DataInputStream(
            new BufferedInputStream(
                new FileInputStream("TestEOF.java")));
        while(in.available() != 0)
            System.out.print((char)in.readByte());
    }
} /* (Execute to see output) *///:~
```

Заметьте, что метод `available()` всегда работает по-разному, в зависимости от источника данных; дословно его функция описывается следующим образом: «количество байтов, которые можно прочитать *без блокировки*». Когда читается файл, доступны все его байты, но для другого рода потоков это не обязательно верно, потому используйте этот метод разумно.

Определить конец входного потока можно и с помощью перехвата исключения. Однако использование механизма исключений в таких целях считается злоупотреблением.

Вывод в файл

Объект `FileWriter` записывает данные в файл. Буферизация применяется при вводе/выводе практически всегда (попробуйте прочитать файл без нее, и вы увидите, насколько ее отсутствие влияет на производительность — скорость чтения уменьшится в несколько раз), поэтому мы декорируем его в `BufferedWriter`. После этого подключается `PrintWriter`, чтобы выполнять форматированный вывод. Файл с данными, созданный такой конфигурацией ввода-вывода, можно прочитать как обычный текстовый файл.

```
//: io/BasicFileOutput.java
import java.io.*;

public class BasicFileOutput {
    static String file = "BasicFileOutput.out";
    public static void main(String[] args)
        throws IOException {
        BufferedReader in = new BufferedReader(
            new StringReader(
                BufferedInputFile.read("BasicFileOutput.java")));
        PrintWriter out = new PrintWriter(
            new BufferedWriter(new FileWriter(file)));
        int lineCount = 1;
        String s;
        while((s = in.readLine()) != null )
            out.println(lineCount++ + ": " + s);
        out.close();
        // Вывод сохраненного файла:
        System.out.println(BufferedInputFile.read(file));
    }
}
```

```
}  
} /* (Execute to see output) *///:~
```

При записи строк в файл к ним добавляются их номера. Заметьте, что `LineNumberInputStream` для этого *не применяется*, поскольку этот класс никому не нужен. Как и показано в рассматриваемом примере, своя собственная нумерация реализуется тривиально.

Когда данные входного потока исчерпываются, метод `readLine()` возвращает `null`. Для потока `out` явно вызывается метод `close()`; если не вызвать его для всех выходных файловых потоков, в буферах могут остаться данные, и файл получится неполным.

Сокращенная запись для вывода в текстовые файлы

В Java SE5 у `PrintWriter` появился вспомогательный конструктор, чтобы вам не приходилось явно проделывать все необходимое каждый раз, когда вы захотите создать текстовый файл и записать в него информацию. Вот как выглядит пример `BasicFileOutput.java`, переписанный с использованием этого синтаксиса:

```
//: io/FileOutputShortcut.java  
import java.io.*;  
  
public class FileOutputShortcut {  
    static String file = "FileOutputShortcut.out";  
    public static void main(String[] args)  
        throws IOException {  
        BufferedReader in = new BufferedReader(  
            new StringReader(  
                BufferedInputStream.read("FileOutputShortcut.java")));  
        // Сокращенная запись:  
        PrintWriter out = new PrintWriter(file);  
        int lineCount = 1;  
        String s;  
        while((s = in.readLine()) != null )  
            out.println(lineCount++ + ": " + s);  
        out.close();  
        // Вывод сохраненного файла:  
        System.out.println(BufferedInputStream.read(file));  
    }  
} /* (Execute to see output) *///:~
```

Вы по-прежнему пользуетесь преимуществами буферизации, но вам не нужно реализовывать ее самостоятельно. К сожалению, для других частых задач записи сокращенная запись не предусмотрена, так что типичный ввод-вывод по-прежнему требует большого количества избыточного текста. Впрочем, вспомогательный класс `TextFile`, который используется в книге и будет определен позднее в этой главе, упрощает эти стандартные задачи.

12. (3) Измените упражнение 8, чтобы оно также открывало текстовый файл для записи. Запишите в файл строки из `LinkedList` вместе с номерами строк (не пытайтесь использовать классы `LineNumber`).
13. (3) Измените пример `BasicFileOutput.java`, чтобы в нем использовался объект `LineNumberReader` для подсчета строк. Обратите внимание, насколько проще подсчитывать строки самостоятельно.

14. (2) На базе кода `BasicFileOutput.java` напишите программу для сравнения производительности записи в файл при использовании буферизованного и небуферизованного ввода-вывода.

Сохранение и восстановление данных

Объект `PrintWriter` форматирует данные, так чтобы их мог прочитать человек. Однако для вывода информации, предназначенной для другого потока, следует использовать классы `DataOutputStream` (для записи данных) и `DataInputStream` (для чтения данных). Конечно, природа этих потоков может быть любой, но в нашем случае открывается файл, буферизованный как для чтения, так и для записи. Надстройки `DataOutputStream` и `DataInputStream` ориентированы на отправку байтов, поэтому для них потребуются потоки `OutputStream` и `InputStream`.

```
//: io/StoringAndRecoveringData.java
import java.io.*;

public class StoringAndRecoveringData {
    public static void main(String[] args)
        throws IOException {
        DataOutputStream out = new DataOutputStream(
            new BufferedOutputStream(
                new FileOutputStream("Data.txt")));
        out.writeDouble(3.14159);
        out.writeUTF("That was pi");
        out.writeDouble(1.41413);
        out.writeUTF("Square root of 2");
        out.close();
        DataInputStream in = new DataInputStream(
            new BufferedInputStream(
                new FileInputStream("Data.txt")));
        System.out.println(in.readDouble());
        // Только readUTF() правильно восстановит
        // объект String в кодировке UTF для Java:
        System.out.println(in.readUTF());
        System.out.println(in.readDouble());
        System.out.println(in.readUTF());
    }
} /* Output:
3.14159
That was pi
1.41413
Square root of 2
*///:~
```

Если вы используете поток `DataOutputStream` для записи данных, язык Java гарантирует, что эти же данные в точно таком же виде будут восстановлены входным потоком `DataInputStream` — невзирая на платформу, на которой производится запись или чтение. Это чрезвычайно ценно, и это знает любой, так или иначе соприкасавшийся с вопросами переносимости программ. Возможные проблемы пропадают сами по себе — достаточно иметь интерпретатор Java на обеих платформах¹.

¹ Язык разметки XML — еще один способ переноса данных между различными платформами, к тому же он не зависит от того, есть ли среда выполнения Java на машине или нет. О работе с данными XML будет рассказано позднее в этой главе.

Единственным надежным способом записать в поток `DataOutputStream` строку (`String`) так, чтобы ее можно было потом правильно считать потоком `DataInputStream`, является кодирование UTF-8, реализуемое методами `readUTF()` и `writeUTF()`. UTF-8 относится к многобайтовым форматам, а длина кодирования зависит от используемого набора символов. Если вы работаете только с кодировкой ASCII или основным подмножеством ASCII (в котором значимы только семь битов), «удвоение» данных приводит к неоправданным затратам пространства и/или нагрузке на сеть. Поэтому UTF-8 кодирует символы ASCII одним байтом, а символы из других кодировок записывает двумя или тремя байтами. Вдобавок в первых двух байтах строки хранится ее длина. Впрочем, методы `readUTF()` и `writeUTF()` используют специальную модификацию UTF-8 для Java¹ (она описана в документации JDK), и для правильного считывания из другой программы (не на Java) строки, записанной методом `writeUTF()`, вам придется добавить в нее специальный код, обеспечивающий ее правильное считывание.

Методы `readUTF()` и `writeUTF()` позволяют смешивать строки и другие типы данных, записываемые потоком `DataOutputStream`, так как вы знаете, что строки будут правильно сохранены в Юникоде и их будет легко восстановить потоком `DataInputStream`.

Метод `writeDouble()` записывает число `double` в поток, а соответствующий ему метод `readDouble()` затем восстанавливает его (для других типов также существуют подобные методы). Но чтобы правильно интерпретировать любые данные, вы должны точно знать их расположение в потоке, поскольку с таким же успехом можно прочесть число `double` как какую-то последовательность байтов или символов. Поэтому данные в файле должны храниться в определенном формате, или придется использовать дополнительную информацию, которая будет показывать, какие именно данные находятся в тех или иных местах. Заметьте, что сериализация объектов (описанная в этой главе чуть позже) предоставляет, наверное, простейший способ записи и восстановления сложных структур данных.

- 15.** (4) Найдите описания `DataOutputStream` и `DataInputStream` в документации JDK. На базе кода `StoringAndRecoveringData.java` напишите программу, которая сохраняет, а затем читает все возможные типы, поддерживаемые классами `DataOutputStream` и `DataInputStream`. Убедитесь в том, что запись и чтение данных были выполнены правильно.

Чтение/запись файлов с произвольным доступом

Работа с классом `RandomAccessFile` напоминает использование комбинации `DataInputStream` и `DataOutputStream` (потому что он реализует те же интерфейсы: `DataInput` и `DataOutput`). Кроме того, вы можете использовать метод `seek()` для перемещения по файлу и изменения значений.

¹ В стандарте Unicode 3.0 (раздел 3.8, Transformations) говорится, что UTF-8 сериализует значения в последовательность от одного до четырех байтов. Описание UTF-8 во второй редакции ISO/IEC 10646 допускает также пятый и шестой байты, но это не является корректным для стандарта Unicode. Сказанное в двух предыдущих предложениях является особенностью именно Java-реализации UTF-8. — *Примеч. ред.*

При использовании `RandomAccessFile` необходимо знать структуру файла, чтобы правильно работать с ним. У класса `RandomAccessFile` имеются специализированные методы для записи и чтения примитивов и строк UTF-8. Пример:

```
//: io/UsingRandomAccessFile.java
import java.io.*;

public class UsingRandomAccessFile {
    static String file = "rtest.dat";
    static void display() throws IOException {
        RandomAccessFile rf = new RandomAccessFile(file, "r");
        for(int i = 0; i < 7; i++)
            System.out.println(
                "Value " + i + ": " + rf.readDouble());
        System.out.println(rf.readUTF());
        rf.close();
    }
    public static void main(String[] args)
        throws IOException {
        RandomAccessFile rf = new RandomAccessFile(file, "rw");
        for(int i = 0; i < 7; i++)
            rf.writeDouble(i*1.414);
        rf.writeUTF("The end of the file");
        rf.close();
        display();
        rf = new RandomAccessFile(file, "rw");
        rf.seek(5*8);
        rf.writeDouble(47.00001);
        rf.close();
        display();
    }
} /* Output:
Value 0: 0.0
Value 1: 1.414
Value 2: 2.828
Value 3: 4.242
Value 4: 5.656
Value 5: 7.069999999999999
Value 6: 8.484
The end of the file
Value 0: 0.0
Value 1: 1.414
Value 2: 2.828
Value 3: 4.242
Value 4: 5.656
Value 5: 47.00001
Value 6: 8.484
The end of the file
*///:~
```

Метод `display()` открывает файл и выводит семь элементов как значения `double`. В методе `main()` файл сначала создается, затем открывается и изменяется. Так как значение `double` всегда состоит из 8 байтов, при вызове `seek()` для позиционирования к значению `double` с номером 5 нужное смещение определяется простым умножением `5*8`.

Как уже было замечено, класс `RandomAccessFile` практически ничем не связан с иерархией библиотеки ввода-вывода Java, если не считать реализации интерфейсов `DataInput` и `DataOutput`. Таким образом, комбинировать его с подклассами `InputStream` и `OutputStream` не получится. Можно предположить, что класс доступа к буферу памяти `ByteArrayInputStream` стал бы хорошим потоком произвольного доступа, но класс `RandomAccessFile` работает только с файлами. Буферизация уже встроена в него и присоединить ее, как это делается с потоками, нельзя.

Все возможности настройки сводятся ко второму аргументу конструктора: можно открыть `RandomAccessFile` только для чтения (указав строку «r») или для чтения/записи (строка «rw»).

Также стоит рассмотреть возможность употребления вместо класса `RandomAccessFile` механизма отображаемых в память файлов.

- 16.** (2) Найдите описание `RandomAccessFile` в документации JDK. На базе кода `UsingRandomAccessFile.java` напишите программу, которая сохраняет, а затем читает все возможные типы, поддерживаемые классом `RandomAccessFile`. Убедитесь в том, что запись и чтение данных были выполнены правильно.

Каналы

В этой главе были коротко упомянуты классы каналов `PipedInputStream`, `PipedOutputStream`, `PipedReader` и `PipedWriter`. Это не значит, что они редко используются или не слишком полезны, просто их смысл и действие нельзя донести до понимания до тех пор, пока не объяснена многозадачность: каналы предназначены для связи между отдельными *программными потоками*. Поэтому они будут подробно описаны в главе 21, там же для них имеется несколько примеров.

Средства чтения и записи файлов

Очень часто в программировании используется стандартная последовательность операций: файл считывается в память, там он изменяется, а потом снова записывается на диск. Одна из неприятностей с библиотекой ввода-вывода Java состоит в том, что для выполнения таких достаточно частых операций вам придется написать некоторое количество кода — не существует вспомогательных функций, на которые можно переложить такую деятельность. Что еще хуже, с декораторами вообще трудно запомнить, как открываются файлы. Потому имеет смысл добавить в вашу библиотеку вспомогательные классы, которые легко сделают нужное за вас. В Java SE5 добавился вспомогательный конструктор `PrintWriter`, который позволяет легко открыть текстовый файл для записи. Однако существует много стандартных операций, которые выполняются в программах снова и снова, и избыточный код таких операций было бы лучше исключить.

Ниже показан класс `TextFile`, использовавшийся в предшествующих примерах для упрощения чтения и записи файлов. Он содержит статические методы для чтения и записи текстовых файлов как одной большой строки. Также можно создать экземпляр

класса `TextFile`, который будет хранить содержимое файла в списке `ArrayList` (чтобы при работе с содержимым файла были доступны все возможности `ArrayList`):

```

//: net/mindview/util/TextFile.java
// Статические функции для чтения и записи
// текстовых файлов, а также для работы с файлом
// как со списком ArrayList.
package net.mindview.util;
import java.io.*;
import java.util.*;

public class TextFile extends ArrayList<String> {
    // Чтение всего файла как одной строки:
    public static String read(String fileName) {
        StringBuilder sb = new StringBuilder();
        try {
            BufferedReader in= new BufferedReader(new FileReader(
                new File(fileName).getAbsolutePath()));
            try {
                String s;
                while((s = in.readLine()) != null) {
                    sb.append(s);
                    sb.append("\n");
                }
            } finally {
                in.close();
            }
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
        return sb.toString();
    }
    // Запись файла одним вызовом метода:
    public static void write(String fileName, String text) {
        try {
            PrintWriter out = new PrintWriter(
                new File(fileName).getAbsolutePath());
            try {
                out.print(text);
            } finally {
                out.close();
            }
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
    }
    // Чтение файла с разбивкой по регулярному выражению:
    public TextFile(String fileName, String splitter) {
        super(Arrays.asList(read(fileName).split(splitter)));
        // Регулярное выражение split() часто оставляет
        // пустую строку в первой позиции:
        if(get(0).equals("")) remove(0);
    }
    // Обычное чтение по строкам:
    public TextFile(String fileName) {
        this(fileName, "\n");
    }
}

```

```
public void write(String fileName) {
    try {
        PrintWriter out = new PrintWriter(
            new File(fileName).getAbsoluteFile());
        try {
            for(String item : this)
                out.println(item);
        } finally {
            out.close();
        }
    } catch(IOException e) {
        throw new RuntimeException(e);
    }
}

// Простой тест:
public static void main(String[] args) {
    String file = read("TextFile.java");
    write("test.txt", file);
    TextFile text = new TextFile("test.txt");
    text.write("test2.txt");
    // Разбивка на сортированные списки уникальных слов:
    TreeSet<String> words = new TreeSet<String>(
        new TextFile("TextFile.java", "\\W+"));
    // Вывод слов в верхнем регистре:
    System.out.println(words.headSet("a"));
}

} /* Output:
[0, ArrayList, Arrays, Break, BufferedReader,
BufferedWriter, Clean, Display, File, FileReader,
FileWriter, IOException, Normally, Output, PrintWriter,
Read, Regular, RuntimeException, Simple, Static, String,
StringBuilder, System, TextFile, Tools, TreeSet, W, Write]
*///::~~
```

Метод `read()` присоединяет к объекту `StringBuilder` каждую прочитанную строку и символ новой строки, усеченный во время чтения. Затем он возвращает объект `String` с полным содержимым всего файла. Метод `write()` открывает файл и записывает в него текстовую строку.

Обратите внимание: в любом коде, открывающем файл, вызов `close()` заключается в секцию `finally`, чтобы гарантировать корректное закрытие файла.

Конструктор использует метод `read()` для преобразования файла в `String`, после чего вызывает метод `String.split()`, чтобы разбить результат по символам новой строки (если вы будете часто использовать этот класс, то, возможно, захотите переписать этот конструктор для повышения эффективности). Увы, соответствующего метода для слияния строк нет, так что придется для записи строк обойтись нестатическим методом `write()`.

Так как этот класс предназначен для упрощения процесса чтения и записи файлов, все исключения `IOException` преобразуются в `RuntimeException`, чтобы пользователю не приходилось использовать блоки `try-catch`. Возможно, вам придется создать другую версию, которая будет передавать `IOException` на сторону вызова.

В методе `main()` выполняется простой тест, позволяющий удостовериться в правильной работе методов.

Несмотря на то что кода в этом классе немного, его применение позволит сэкономить вам уйму времени и сделать вашу жизнь проще, в чем вы еще будете иметь возможность убедиться чуть позже.

Также для чтения текстовых файлов можно воспользоваться классом `java.util.Scanner`, появившимся в Java SE5. Однако этот класс предназначен только для чтения файлов, но не для записи, и этот инструмент (не включенный в `java.io`) предназначен в основном для создания обработчиков программного кода или «мини-языков».

17. (4) Используя `TextFile` и `Map<Character, Integer>`, создайте программу для подсчета вхождений символов в файл. (Например, если в файле буква «а» встречается 12 раз, с этим символом связывается значение типа `Integer`, равное 12.)
18. (1) Измените пример `TextFile.java` так, чтобы он передавал исключения `IOException` вызывающей стороне.

Чтение двоичных файлов

Следующий вспомогательный класс похож на `TextFile.java` — он также упрощает процесс чтения, но используется для двоичных файлов:

```
//: net/mindview/util/BinaryFile.java
// Чтение файлов в двоичной форме.
package net.mindview.util;
import java.io.*;

public class BinaryFile {
    public static byte[] read(File bFile) throws IOException{
        BufferedInputStream bf = new BufferedInputStream(
            new FileInputStream(bFile));
        try {
            byte[] data = new byte[bf.available()];
            bf.read(data);
            return data;
        } finally {
            bf.close();
        }
    }
    public static byte[]
    read(String bFile) throws IOException {
        return read(new File(bFile).getAbsoluteFile());
    }
} ///:~
```

Один перегруженный метод получает аргумент `File`; во втором передается имя файла (аргумент `String`). Оба метода возвращают полученный массив байтов.

Метод `available()` возвращает размер массива, а эта перегруженная версия метода `read()` заполняет массив.

19. (2) Используя классы `BinaryFile` и `Map<Byte, Integer>`, создайте программу для подсчета вхождений разных байтов в файл.
20. (4) Используя метод `Directory.walk()` и `BinaryFile`, проверьте, что все файлы `.class` в дереве каталогов начинаются с шестнадцатеричной последовательности «CAFE-BABE».

Стандартный ввод-вывод

Термином *стандартный ввод-вывод* обозначается принятая в UNIX концепция единого потока информации, используемого программой (в некоторой форме эта концепция присутствует и в Windows, и во многих других операционных системах). Вся информация программы приходит из *стандартного ввода* (standard input), все данные записываются в *стандартный вывод* (standard output), а все ошибки программы пересылаются в *стандартный поток для ошибок* (standard error). Значение стандартного ввода-вывода состоит в том, что программы легко соединять в цепочку, где стандартный вывод одной программы становится стандартным входом другой программы. Этот инструмент открывает очень много полезных возможностей.

Чтение из стандартного потока ввода

Следуя модели стандартного ввода-вывода, Java определяет необходимые потоки для стандартного ввода, вывода и ошибок: `System.in`, `System.out` и `System.err`. На многих страницах книги вы не раз могли наблюдать процесс записи в стандартный вывод `System.out`, для которого уже надстроен класс форматирования данных `PrintStream`. Поток для ошибок `System.err` схож со стандартным выводом, а стандартный ввод `System.in` представляет собой «низкоуровневый» поток `InputStream` без «обертки». Это значит, что потоки `System.out` и `System.err` можно использовать напрямую, в то время как стандартный ввод `System.in` должен иметь «обертку».

Обычно чтение осуществляется построчно, методом `readLine()`, поэтому имеет смысл буферизовать стандартный ввод `System.in` посредством `BufferedReader`. Чтобы сделать это, предварительно следует конвертировать поток `System.in` в объект `Reader` при помощи класса-преобразователя `InputStreamReader`. Следующий пример просто повторяет на экране вводимую пользователем строку:

```
//: io/Echo.java
// Чтение из стандартного ввода.
// {RunByHand}
import java.io.*;

public class Echo {
    public static void main(String[] args)
        throws IOException {
        BufferedReader stdin = new BufferedReader(
            new InputStreamReader(System.in));
        String s;
        while((s = stdin.readLine()) != null && s.length() != 0)
            System.out.println(s);
        // Пустая строка или Ctrl-Z завершает работу программы.
    }
} ///:~
```

Спецификация исключения присутствует здесь из-за того, что метод `readLine()` может возбуждать исключение `IOException`. Стоит напомнить, что поток `System.in` обычно буферизуется (впрочем, как и большинство потоков).

- 21.** (1) Напишите программу, которая получает данные из стандартного ввода и преобразует все символы к верхнему регистру, после чего направляет результаты

в стандартный вывод. Передайте этой программе содержимое файла (конкретный способ перенаправления зависит от вашей операционной системы).

Замена System.out на PrintWriter

Стандартный вывод `System.out` является объектом `PrintStream`, который, в свою очередь, является производным от базового класса `OutputStream`. В классе `PrintWriter` имеется конструктор, который принимает в качестве аргумента выходной поток `OutputStream`. Таким образом, при помощи этого конструктора можно преобразовать поток стандартного вывода `System.out` в символьно-ориентированный поток `PrintWriter`:

```
//: io/ChangeSystemOut.java
// Преобразование System.out в символьный поток PrintWriter.
import java.io.*;

public class ChangeSystemOut {
    public static void main(String[] args) {
        PrintWriter out = new PrintWriter(System.out, true);
        out.println("Hello, world");
    }
} /* Output:
Hello, world
*///:~
```

Важно использовать конструктор класса `PrintWriter` с двумя аргументами и передать во втором аргументе значение «истина» (`true`), чтобы обеспечить автоматический сброс буфера на печать, иначе можно вовсе не получить никакого вывода.

Перенаправление стандартного ввода-вывода

Класс `System` позволяет вам перенаправить стандартный ввод, вывод и поток ошибок. Для этого используются простые статические методы:

- ☐ `setIn(InputStream)`
- ☐ `setOut(PrintStream)`
- ☐ `setErr(PrintStream)`

Перенаправление стандартного вывода особенно полезно тогда, когда ваша программа выдает слишком много сообщений сразу, и вы попросту не успеваете читать их, поскольку они заменяются новыми сообщениями¹. Перенаправление ввода удобно для программ, работающих с командной строкой, в которых необходимо поддерживать некоторую последовательность введенных пользователем данных. Следующий пример показывает, как использовать эти методы:

```
//: io/Redirecting.java
// Перенаправление стандартного ввода-вывода.
import java.io.*;
```

¹ В главе 22 показано еще более удобное решение — это программа с графическим интерфейсом, которая перенаправляет стандартный вывод в текстовое поле с автоматической прокруткой текста.

```
public class Redirecting {
    public static void main(String[] args)
        throws IOException {
        PrintStream console = System.out;
        BufferedInputStream in = new BufferedInputStream(
            new FileInputStream("Redirecting.java"));
        PrintStream out = new PrintStream(
            new BufferedOutputStream(
                new FileOutputStream("test.out")));
        System.setIn(in);
        System.setOut(out);
        System.setErr(out);
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
        String s;
        while((s = br.readLine()) != null)
            System.out.println(s);
        out.close(); // Не забудьте!
        System.setOut(console);
    }
} ///:~
```

Эта программа присоединяет стандартный ввод к файлу и перенаправляет стандартный ввод и поток для ошибок в другие файлы. Обратите внимание на то, как исходный объект `System.out` сохраняется в начале программы и восстанавливается перед ее завершением.

Перенаправление стандартного ввода-вывода основано на байтовом вводе/выводе, а не на символах, поэтому в примере вы видите подклассы `InputStream` и `OutputStream`, а не их символично-ориентированные эквиваленты `Reader` и `Writer`.

Управление процессами

Часто в программах Java возникает необходимость выполнения других программ операционной системы, с управлением вводом и выводом таких программ. Библиотека Java предоставляет классы для выполнения таких операций.

Одна из стандартных задач — запуск программы и направление полученных результатов на консоль. В этом разделе представлен вспомогательный класс, упрощающий выполнение этой задачи.

При его использовании могут возникать два вида ошибок: обычные ошибки, приводящие к выдаче исключений (для которых мы просто перезапускаем исключение `RuntimeException`), и ошибки, обусловленные выполнением самого процесса. Для оповещения о таких ошибках будет использоваться отдельное исключение:

```
//: net/mindview/util/OSExecuteException.java
package net.mindview.util;

public class OSExecuteException extends RuntimeException {
    public OSExecuteException(String why) { super(why); }
} ///:~
```

Для запуска программы методу `OSExecute.command()` передается строка `command`; в ней указывается команда, которая бы использовалась для выполнения программы из консоли. Команда передается конструктору `java.lang.ProcessBuilder`, а созданный объект `ProcessBuilder` используется для запуска:

```
//: net/mindview/util/OSExecute.java
// Выполнение команды операционной системы
// и передача вывода на консоль.
package net.mindview.util;
import java.io.*;

public class OSExecute {
    public static void command(String command) {
        boolean err = false;
        try {
            Process process =
                new ProcessBuilder(command.split(" ")).start();
            BufferedReader results = new BufferedReader(
                new InputStreamReader(process.getInputStream()));
            String s;
            while((s = results.readLine()) != null)
                System.out.println(s);
            BufferedReader errors = new BufferedReader(
                new InputStreamReader(process.getErrorStream()));
            // Оповещение об ошибках и возврат ненулевого значения
            // вызывающему процессу при возникновении проблем:
            while((s = errors.readLine()) != null) {
                System.err.println(s);
                err = true;
            }
        } catch (Exception e) {
            // Для системы Windows 2000, которая выдает
            // исключение для командной строки по умолчанию:
            if(!command.startsWith("CMD /C"))
                command("CMD /C " + command);
            else
                throw new RuntimeException(e);
        }
        if(err)
            throw new OSExecuteException("Errors executing " +
                command);
    }
} ///:~
```

Для получения стандартного вывода от выполняемой программы вызывается метод `getInputStream()`. Он возвращает объект `InputStream`, из которого можно читать данные.

Результаты программы передаются по строкам, поэтому для их чтения используется метод `readLine()`. В нашем примере строки просто выводятся на консоль, но их также можно сохранить и вернуть из метода `command()`.

Ошибки программы направляются в стандартный поток ошибок и сохраняются вызовом `getErrorStream()`. Если во время выполнения происходили ошибки, то информация о них выводится и выдается исключение `OSExecuteException`, чтобы вызывающая программа могла обработать проблему.

Следующий пример демонстрирует использование `OSExecute`:

```
//: io/OSExecuteDemo.java
// Демонстрация стандартного перенаправления ввода-вывода.
import net.mindview.util.*;

public class OSExecuteDemo {
    public static void main(String[] args) {
        OSExecute.command("javap OSExecuteDemo");
    }
} /* Output:
Compiled from "OSExecuteDemo.java"
public class OSExecuteDemo extends java.lang.Object{
    public OSExecuteDemo();
    public static void main(java.lang.String[]);
}
*///:~
```

В этом примере декомпилятор `javap` (включенный в поставку JDK) используется для декомпиляции программы.

- 22.** (5) Измените пример `OSExecute.java` так, чтобы вместо вывода стандартного потока ошибок результат выполнения возвращался в виде списка `List` с элементами `String`. Продемонстрируйте использование новой версии класса.

НОВЫЙ ВВОД-ВЫВОД (nio)

Библиотека «нового ввода-вывода» Java, появившаяся в JDK 1.4 в пакетах `java.nio.*`, ориентирована на единственную цель: скорость. Даже «старые» пакеты ввода-вывода были переписаны с учетом достижений `nio`, с намерением использовать преимущества повышенного быстродействия, потому улучшения вы получите, даже если не будете писать явный `nio`-код. Подъем производительности просматривается и в файловом вводе/выводе, который мы здесь рассматриваем, и в сетевом вводе/выводе, описанном в книге *Thinking In Enterprise Java*.

Увеличения скорости удалось достичь с помощью структур, близких к средствам самой операционной системы: это *каналы*¹ (`channels`) и *буферы* (`buffers`). Можно представить их в виде угольной шахты: канал представляет собой шахту, вырытую над угольным пластом (данные), а буфер — это вагонетка, которую вы посылаете в шахту. Тележка возвращается доверху наполненная углем, который вы из нее выгружаете. Таким образом прямого взаимодействия с каналом у вас нет, вы работаете с буфером и «посылаете» буфер в канал. Канал либо извлекает данные из буфера, либо помещает данные в буфер.

Единственный тип буфера, который напрямую взаимодействует с каналом, — буфер `ByteBuffer`, то есть буфер, хранящий простые байты. Если вы просмотрите документацию JDK для класса `java.nio.ByteBuffer`, то увидите, что он устроен достаточно просто: вы создаете его, указывая, сколько места надо выделить под данные; имеется набор

¹ Описываемые здесь каналы (`channels`) следует отличать от каналов (`pipe`), создаваемых классами `PipedInputStream` и `PipedOutputStream`. Первые представляют собой еще один источник данных, а вторые налаживают обмен данными между различными процессами. — *Примеч. перев.*

методов для получения и помещения данных в виде последовательности байтов или в виде примитивов. Однако возможности записать в него объект или даже простую строку нет. Этот буфер работает на низком уровне, поскольку так можно достичь более эффективной совместимости с большинством операционных систем.

Три класса из «старой» библиотеки ввода-вывода были изменены так, чтобы они позволяли получить канал `FileChannel`: это `FileInputStream`, `FileOutputStream` и позволяющий одновременное чтение и запись `RandomAccessFile`. Заметьте, что эти классы манипулируют байтами, что согласуется с низкоуровневой направленностью `nio`. Классы для символьных данных `Reader` и `Writer` не образуют каналов, однако вспомогательный класс `java.nio.channels.Channels` имеет набор методов, позволяющих произвести объекты `Reader` и `Writer` из каналов.

Ниже приведен простой пример для всех трех типов потоков; создаваемые каналы поддерживают запись, чтение/запись и только чтение:

```
//: io/GetChannel.java
// Получение каналов из потоков.
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class GetChannel {
    private static final int BSIZE = 1024;
    public static void main(String[] args) throws Exception {
        // Запись файла:
        FileChannel fc =
            new FileOutputStream("data.txt").getChannel();
        fc.write(ByteBuffer.wrap("Some text ".getBytes()));
        fc.close();
        // Добавление в конец файла:
        fc =
            new RandomAccessFile("data.txt", "rw").getChannel();
        fc.position(fc.size()); // Перемещение в конец
        fc.write(ByteBuffer.wrap("Some more".getBytes()));
        fc.close();
        // Чтение файла:
        fc = new FileInputStream("data.txt").getChannel();
        ByteBuffer buff = ByteBuffer.allocate(BSIZE);
        fc.read(buff);
        buff.flip();
        while(buff.hasRemaining())
            System.out.print((char)buff.get());
    }
} /* Output:
Some text Some more
*///:~
```

Для любого из рассмотренных выше классов потоков метод `getChannel()` произведет канал `FileChannel`. Канал довольно прост: вы можете передать ему байтовый буфер `ByteBuffer` для чтения и записи, и вы можете заблокировать некоторые участки файла для монопольного доступа (об этом чуть позже).

Для помещения байтов в буфер `ByteBuffer` можно воспользоваться одним из нескольких методов для записи данных (`put`) в виде одного или нескольких байтов или значений примитивных типов. Впрочем, как было показано в примере, можно «завернуть» уже

существующий байтовый массив в буфер `ByteBuffer` методом `wrap()`. Когда вы так делаете, байтовый массив не копируется, а используется как хранилище для полученного буфера `ByteBuffer`.

Файл `data.txt` заново открывается с помощью класса `RandomAccessFile`. Заметьте, что вы вправе перемещать канал `FileChannel` по записям, здесь он был передвинут в конец файла так, чтобы дополнительные записи присоединялись к концу файла.

Доступ «только для чтения» можно осуществить, явно выделяя память для байтового буфера `ByteBuffer` статическим методом `allocate()`. Предназначение `nio` — быстрое перемещение большого количества данных, поэтому размер буфера имеет значение: на самом деле установленный в примере размер в один килобайт меньше, чем обычно требуется (вам надо будет поэкспериментировать с вашим работающим приложением, чтобы найти оптимальное решение).

Возможно получить еще большее быстродействие, используя вместо метода `allocate()` метод `allocateDirect()`. Он производит буфер «прямого доступа», который может быть еще теснее связан с операционной системой. Однако такой буфер требует больше ресурсов, а реализация его отличается в различных операционных системах. Таким образом, опять имеет смысл поэкспериментировать с вашим работающим приложением, чтобы выяснить, дадут ли буферы прямого доступа лучшую производительность.

После того как вы вызовете метод `read()` буфера `FileChannel`, чтобы сохранить байты в буфере `ByteBuffer`, вы также должны вызвать для буфера метод `flip()`, позволяющий впоследствии извлечь из буфера его данные (да, все это выглядит немного непродуманно, но помните, что расчет делался на высокое быстродействие и поэтому все сделано на низком уровне). И если затем нам снова понадобится буфер для чтения, придется вызывать перед каждым методом `read()` метод `clear()`. Увидеть это можно в простой программе для копирования файлов:

```
//: io/ChannelCopy.java
// Копирование файлов с использованием каналов и буферов.
// {Args: ChannelCopy.java test.txt}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class ChannelCopy {
    private static final int BSIZE = 1024;
    public static void main(String[] args) throws Exception {
        if(args.length != 2) {
            System.out.println("arguments: sourcefile destfile");
            System.exit(1);
        }
        FileChannel
            in = new FileInputStream(args[0]).getChannel(),
            out = new FileOutputStream(args[1]).getChannel();
        ByteBuffer buffer = ByteBuffer.allocate(BSIZE);
        while(in.read(buffer) != -1) {
            buffer.flip(); // Подготовка к записи
            out.write(buffer);
            buffer.clear(); // Подготовка к чтению
        }
    }
}
} ///:~
```

Как видно из листинга, создаются два канала `FileChannel`: для чтения и для записи. Выделяется буфер `ByteBuffer`, а когда метод `FileChannel.read()` возвращает `-1`, это значит, что мы достигли конца входных данных (без сомнения, наследие UNIX и C). После каждого вызова метода `read()`, помещающего данные в буфер, метод `flip()` подготавливает буфер так, чтобы информация из него могла быть извлечена методом `write()`. После вызова `write()` информация все еще хранится в буфере, поэтому метод `clear()` сбрасывает все внутренние указатели, чтобы буфер снова был способен принимать данные в методе `read()`.

Впрочем, рассмотренная программа не лучшим образом выполняет копирование файлов. Специальные методы `transferTo()` и `transferFrom()` позволяют напрямую присоединить один канал к другому:

```
//: io/TransferTo.java
// Using transferTo() between channels
// {Args: TransferTo.java TransferTo.txt}
import java.nio.channels.*;
import java.io.*;

public class TransferTo {
    public static void main(String[] args) throws Exception {
        if(args.length != 2) {
            System.out.println("arguments: sourcefile destfile");
            System.exit(1);
        }
        FileChannel
            in = new FileInputStream(args[0]).getChannel(),
            out = new FileOutputStream(args[1]).getChannel();
        in.transferTo(0, in.size(), out);
        // Или:
        // out.transferFrom(in, 0, in.size());
    }
} ///:~
```

Часто такую операцию выполнять вам не придется, но знать о ней полезно.

Преобразование данных

Если вы вспомните программу `GetChannel.java`, то увидите, что для вывода информации из файла нам приходилось считывать из буфера по одному байту и преобразовывать его из типа `byte` к типу `char`. Такой подход выглядит весьма примитивно — если вы посмотрите на класс `java.nio.CharBuffer`, то увидите, что в нем есть метод `toString()`, который, как сказано в описании, «возвращает строку из символов, находящихся в данном буфере». Байтовый буфер `ByteBuffer` можно представить в виде символьного буфера `CharBuffer`, это делает метод `asCharBuffer()`, почему бы так и не поступить? Как вы увидите уже из первой команды вывода, такое решение не сработает:

```
//: io/BufferToText.java
// Преобразование текста в ByteBuffer и обратно
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.io.*;
```

```

public class BufferToText {
    private static final int BSIZE = 1024;
    public static void main(String[] args) throws Exception {
        FileChannel fc =
            new FileOutputStream("data2.txt").getChannel();
        fc.write(ByteBuffer.wrap("Some text".getBytes()));
        fc.close();
        fc = new FileInputStream("data2.txt").getChannel();
        ByteBuffer buff = ByteBuffer.allocate(BSIZE);
        fc.read(buff);
        buff.flip();
        // Не работает:
        System.out.println(buff.asCharBuffer());
        // Декодировать с использованием кодировки по умолчанию:
        buff.rewind();
        String encoding = System.getProperty("file.encoding");
        System.out.println("Decoded using " + encoding + ": "
            + Charset.forName(encoding).decode(buff));
        // А можно использовать кодировку, пригодную для печати:
        fc = new FileOutputStream("data2.txt").getChannel();
        fc.write(ByteBuffer.wrap(
            "Some text".getBytes("UTF-16BE")));
        fc.close();
        // Снова пытаемся прочесть:
        fc = new FileInputStream("data2.txt").getChannel();
        buff.clear();
        fc.read(buff);
        buff.flip();
        System.out.println(buff.asCharBuffer());
        // Используем CharBuffer для записи:
        fc = new FileOutputStream("data2.txt").getChannel();
        buff = ByteBuffer.allocate(24); // More than needed
        buff.asCharBuffer().put("Some text");
        fc.write(buff);
        fc.close();
        // Чтение и отображение:
        fc = new FileInputStream("data2.txt").getChannel();
        buff.clear();
        fc.read(buff);
        buff.flip();
        System.out.println(buff.asCharBuffer());
    }
} /* Output:
???
Decoded using Cp1252: Some text
Some text
Some text
*///:~

```

Буфер содержит обычные байты, следовательно, для превращения их в символы мы должны либо *кодировать* их по мере помещения в буфер (так, чтобы они имели смысл при их извлечении), либо *декодировать* их при извлечении из буфера. Это можно сделать с помощью класса `java.nio.charset.Charset`, который предоставляет инструменты для преобразования многих различных типов в наборы символов:

```

//: io/AvailableCharsets.java
// Вывод кодировок и их псевдонимов
import java.nio.charset.*;

```

продолжение ➤

```

import java.util.*;
import static net.mindview.util.Print.*;

public class AvailableCharsets {
    public static void main(String[] args) {
        SortedMap<String,Charset> charSets =
            Charset.availableCharsets();
        Iterator<String> it = charSets.keySet().iterator();
        while(it.hasNext()) {
            String csName = it.next();
            printnb(csName);
            Iterator aliases =
                charSets.get(csName).aliases().iterator();
            if(aliases.hasNext())
                printnb(": ");
            while(aliases.hasNext()) {
                printnb(aliases.next());
                if(aliases.hasNext())
                    printnb(", ");
            }
            print();
        }
    }
} /* Output:
Big5: csBig5
Big5-HKSCS: big5-hkscs, big5hk, big5-hkscs:unicode3.0,
big5hkscs, Big5_HKSCS
EUC-JP: eucjis, x-eucjp, csEUCPkdFmtjapanese, eucjp,
Extended_UNIX_Code_Packed_Format_for_Japanese, x-euc-jp,
euc_jp
EUC-KR: ksc5601, 5601, ksc5601_1987, ksc_5601, ksc5601-
1987, euc_kr, ks_c_5601-1987, euckr, csEUCKR
GB18030: gb18030-2000
GB2312: gb2312-1980, gb2312, EUC_CN, gb2312-80, euc-cn,
euccn, x-EUC-CN
GBK: windows-936, CP936
...
*///:~

```

Таким образом, возвращаясь к программе `BufferToText.java`, если вы вызовете для буфера метод `rewind()` (чтобы вернуться к его началу), а затем используете кодировку по умолчанию в методе `decode()`, данные буфера `CharBuffer` будут правильно выведены на консоль. Для определения кодировки по умолчанию вызовите метод `System.getProperty("file.encoding")`, который возвращает строку с названием кодировки. Передавая эту строку в метод `Charset.forName()`, вы получите объект `Charset`, с помощью которого и декодируете строку.

Другой подход — кодировать данные методом `encode()` так, чтобы при чтении файла выводилось что-нибудь понятное (см. третью часть `BufferToText.java`). Здесь для записи текста в файл используется кодировка UTF-16BE, и при последующем чтении остается только преобразовать данные в буфер `CharBuffer`, и вы получите ожидаемый текст.

Наконец, вы видите, что происходит при *записи* в буфер `ByteBuffer` через `CharBuffer` (мы узнаем об этом чуть позже). Заметьте, что для байтового буфера выделяется 24 байта. На каждый символ (`char`) отводится два байта, соответственно, буфер вместит 12 символов, а у нас в строке «Some text» их только девять. Оставшиеся нулевые байты все

равно отображаются в строке, образуемой методом `toString()` класса `CharBuffer`, как видно из результатов.

- 23. (6)** Создайте и протестируйте вспомогательный метод для вывода содержимого `CharBuffer` до позиции, за которой следуют символы, непригодные для печати.

Извлечение примитивов

Несмотря на то что в буфере `ByteBuffer` хранятся только байты, он поддерживает методы, образующие любые значения примитивов из этих байтов. Следующий пример демонстрирует вставку и чтение из буфера разнообразных значений примитивных типов с использованием этих методов:

```
/// io/GetData.java
// Получение различных представлений данных из ByteBuffer
import java.nio.*;
import static net.mindview.util.Print.*;

public class GetData {
    private static final int BSIZE = 1024;
    public static void main(String[] args) {
        ByteBuffer bb = ByteBuffer.allocate(BSIZE);
        // При выделении буфер автоматически заполняется нулями:
        int i = 0;
        while(i++ < bb.limit())
            if(bb.get() != 0)
                print("nonzero");
        print("i = " + i);
        bb.rewind();
        // Сохраняем и читаем символьный массив:
        bb.asCharBuffer().put("Howdy!");
        char c;
        while((c = bb.getChar()) != 0)
            printnb(c + " ");
        print();
        bb.rewind();
        // Сохраняем и читаем число типа short:
        bb.asShortBuffer().put((short)471142);
        print(bb.getShort());
        bb.rewind();
        // Сохраняем и читаем число типа int:
        bb.asIntBuffer().put(99471142);
        print(bb.getInt());
        bb.rewind();
        // Сохраняем и читаем число типа long:
        bb.asLongBuffer().put(99471142);
        print(bb.getLong());
        bb.rewind();
        // Сохраняем и читаем число типа float:
        bb.asFloatBuffer().put(99471142);
        print(bb.getFloat());
        bb.rewind();
        // Сохраняем и читаем число типа double:
        bb.asDoubleBuffer().put(99471142);
        print(bb.getDouble());
        bb.rewind();
    }
}
```

```

    }
} /* Output:
i = 1025
H o w d y !
12390
99471142
99471142
9.9471144E7
9.9471142E7
*///:~

```

После выделения байтового буфера мы проверяем все его значения, чтобы убедиться в том, что при выделении содержимое буфера автоматически обнуляется. Сравниваются все 1024 значения, хранимые в буфере (вплоть до последнего, индекс которого — размер буфера — возвращается методом `limit()`), и все они оказываются нулями.

Простейший способ вставить примитив в `ByteBuffer` — получить подходящее «представление» этого буфера методами `asCharBuffer()`, `asShortBuffer()` и т. п., а затем поместить в это представление значение методом `put()`. В примере мы так поступаем для каждого из примитивных типов. Единственное отклонение из этого ряда — использование буфера `ShortBuffer`, требующего приведения типов (которое усекает и изменяет результирующее значение). Все остальные представления не нуждаются в преобразовании типов.

Представления буферов

«Представления буферов» дают вам возможность интерпретировать байтовый буфер через «окно» конкретного примитивного типа. Байтовый буфер все так же хранит действительные данные и одновременно поддерживает представление, поэтому все изменения, которые вы сделаете в представлении, отразятся на содержимом байтового буфера. Как мы увидели из предыдущего примера, это удобно для вставки значений примитивов в байтовый буфер. Представления также позволяют читать значения примитивов из буфера, по одному (раз он «байтовый» буфер) или пакетами (в массивы). Следующий пример манипулирует целыми числами (`int`) в буфере `ByteBuffer` с помощью класса `IntBuffer`:

```

//: io/IntBufferDemo.java
// Работа с целыми числами в буфере ByteBuffer
// посредством буфера IntBuffer
import java.nio.*;

public class IntBufferDemo {
    private static final int BSIZE = 1024;
    public static void main(String[] args) {
        ByteBuffer bb = ByteBuffer.allocate(BSIZE);
        IntBuffer ib = bb.asIntBuffer();
        // Сохраняем массив целых чисел:
        ib.put(new int[]{ 11, 42, 47, 99, 143, 811, 1016 });
        // Чтение и запись по абсолютным позициям:
        System.out.println(ib.get(3));
        ib.put(3, 1811);
        // Установление нового предела перед возвратом к началу буфера.
        ib.flip();
    }
}

```



```

        while(ib.hasRemaining()) {
            int i = ib.get();
            System.out.println(i);
        }
    }
} /* Output:
99
11
42
47
1811
143
811
1016
*///:~

```

Перегруженный метод `put()` первый раз вызывается для помещения в буфер массива целых чисел `int`. Последующие вызовы `put()` и `get()` обращаются к конкретному числу `int` из байтового буфера `ByteBuffer`. Заметьте, что такие обращения к примитивным типам по абсолютной позиции также можно осуществить напрямую через буфер `ByteBuffer`.

Как только байтовый буфер `ByteBuffer` будет заполнен целыми числами или другими примитивами через представление, его можно передать для непосредственной записи в канал. Так же просто можно прочитать данные из канала и использовать представление для преобразования данных к конкретному примитивному типу. Следующий пример интерпретирует одну и ту же последовательность байтов как числа `short`, `int`, `float`, `long` и `double`, создавая для одного байтового буфера `ByteBuffer` различные представления:

```

//: io/ViewBuffers.java
import java.nio.*;
import static net.mindview.util.Print.*;

public class ViewBuffers {
    public static void main(String[] args) {
        ByteBuffer bb = ByteBuffer.wrap(
            new byte[]{ 0, 0, 0, 0, 0, 0, 0, 0, 'a' });
        bb.rewind();
        printnb("Byte Buffer ");
        while(bb.hasRemaining())
            printnb(bb.position() + " -> " + bb.get() + ", ");
        print();
        CharBuffer cb =
            ((ByteBuffer)bb.rewind()).asCharBuffer();
        printnb("Char Buffer ");
        while(cb.hasRemaining())
            printnb(cb.position() + " -> " + cb.get() + ", ");
        print();
        FloatBuffer fb =
            ((ByteBuffer)bb.rewind()).asFloatBuffer();
        printnb("Float Buffer ");
        while(fb.hasRemaining())
            printnb(fb.position() + " -> " + fb.get() + ", ");
        print();
        IntBuffer ib =

```

продолжение ➤

```

    ((ByteBuffer)bb.rewind()).asIntBuffer();
    printnb("Int Buffer ");
    while(ib.hasRemaining())
        printnb(ib.position()+ " -> " + ib.get() + ", ");
    print();
    LongBuffer lb =
        ((ByteBuffer)bb.rewind()).asLongBuffer();
    printnb("Long Buffer ");
    while(lb.hasRemaining())
        printnb(lb.position()+ " -> " + lb.get() + ", ");
    print();
    ShortBuffer sb =
        ((ByteBuffer)bb.rewind()).asShortBuffer();
    printnb("Short Buffer ");
    while(sb.hasRemaining())
        printnb(sb.position()+ " -> " + sb.get() + ", ");
    print();
    DoubleBuffer db =
        ((ByteBuffer)bb.rewind()).asDoubleBuffer();
    printnb("Double Buffer ");
    while(db.hasRemaining())
        printnb(db.position()+ " -> " + db.get() + ", ");
}
} /* Output:
Byte Buffer 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 0, 4 -> 0, 5 -> 0,
6 -> 0, 7 -> 97,
Char Buffer 0 -> , 1 -> , 2 -> , 3 -> a,
Float Buffer 0 -> 0.0, 1 -> 1.36E-43,
Int Buffer 0 -> 0, 1 -> 97,
Long Buffer 0 -> 97,
Short Buffer 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 97,
Double Buffer 0 -> 4.8E-322,
*///:~

```

Байтовый буфер `ByteBuffer` создается «упаковкой» массива из восьми байтов, который затем просматривается через представления для всех возможных примитивных типов. Из следующей схемы видно, как данные по-разному выглядят при просмотре через различные представления.

0	0	0	0	0	0	0	97	bytes
						a		chars
0		0		0		97		shorts
0				97				ints
0.0				1.36E-43				floats
97								longs
4.8E-322								doubles

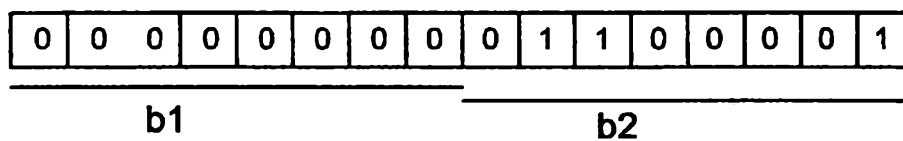
Это соответствует результату работы программы.

24. (1) Измените код `IntBufferDemo.java`, чтобы в нем использовался тип `double`.

Данные о двух концах¹

На разных компьютерах данные могут храниться с различным порядком следования байтов. Прямой порядок `big_endian` располагает старший байт по младшему адресу памяти, а для обратного порядка `little_endian` старший байт помещается по высшему адресу памяти. При хранении значения, занимающего более одного байта, такого как число `int`, `float` и т. п., вам, возможно, придется учитывать различные варианты следования байтов в памяти. Буфер `ByteBuffer` укладывает данные в порядке `big_endian`, такой же способ всегда используется для данных, пересылаемых по сети. Порядок следования байтов в буфере можно изменить методом `order()`, передав ему аргумент `ByteOrder.BIG_ENDIAN` или `ByteOrder.LITTLE_ENDIAN`.

Рассмотрим двоичное представление байтового буфера, содержащего следующие два байта:



Если вы прочитаете эти данные как тип `short` (`ByteBuffer.asShortBuffer()`), то получите число 97 (00000000 01100001), однако при изменении порядка байтов вы увидите число 24 832 (01100001 00000000).

Следующий пример показывает, как порядок следования байтов отражается на символах в зависимости от настроек буфера:

```
//: io/Endians.java
// Порядок следования байтов и хранение данных.
import java.nio.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Endians {
    public static void main(String[] args) {
        ByteBuffer bb = ByteBuffer.wrap(new byte[12]);
        bb.asCharBuffer().put("abcdef");
        print(Arrays.toString(bb.array()));
        bb.rewind();
```

продолжение ➤

¹ Упомянутые в разделе (в оригинале имеющем название «Endians») понятия «big endian» и «little endian» восходят к «Путешествиям Гулливера». Там радетели двух традиций («тупоконечники» и «остроконечники») безуспешно пытались доказать друг другу свою правоту, решая вопрос, с какого конца разбивать яйцо. Прямой и обратный порядок следования байтов особо значимы при сетевых коммуникациях, и Sun всегда здесь брала на себя роль «кормчего». Так, в 1987 году компания навела должный порядок, разработав стандарт *XDR: External Data Representation Standard* (RFC 1014), определяющий машинно-независимое представление данных, передаваемых по сети. Тем не менее до сих пор процессоры Intel (PC) и Motorola (Macintosh) хранят данные с точностью до наоборот. Давая возможность представлять буферы, Java сводит концы с концами — берет на себя ответственность за поддержание нужного порядка следования байтов, показывая тем самым, что означенная проблема того самого яйца, о котором говорил Свифт, вовсе не стоит. В конце концов, зачем бесконечно спорить по пустякам? — *Примеч. ред.*

```

    bb.order(ByteOrder.BIG_ENDIAN);
    bb.asCharBuffer().put("abcdef");
    print(Arrays.toString(bb.array()));
    bb.rewind();
    bb.order(ByteOrder.LITTLE_ENDIAN);
    bb.asCharBuffer().put("abcdef");
    print(Arrays.toString(bb.array()));
}
} /* Output:
[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
[97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102, 0]
*///:~

```

В буфере `ByteBuffer` достаточно места для хранения всех байтов символьного массива, поэтому для вывода байтов годится метод `array()`. Метод `array()` является необязательным и вызывать его следует только для буфера на базе массива; в противном случае возбуждается исключение `UnsupportedOperationException`.

Символьный массив помещается в буфер `ByteBuffer` посредством представления `CharBuffer`. При выводе содержащихся в буфере байтов вы можете видеть, что настройка по умолчанию совпадает с режимом `big_endian`, в то время как атрибут `little_endian` переставляет байты в обратном порядке.

Буферы и манипуляция данными

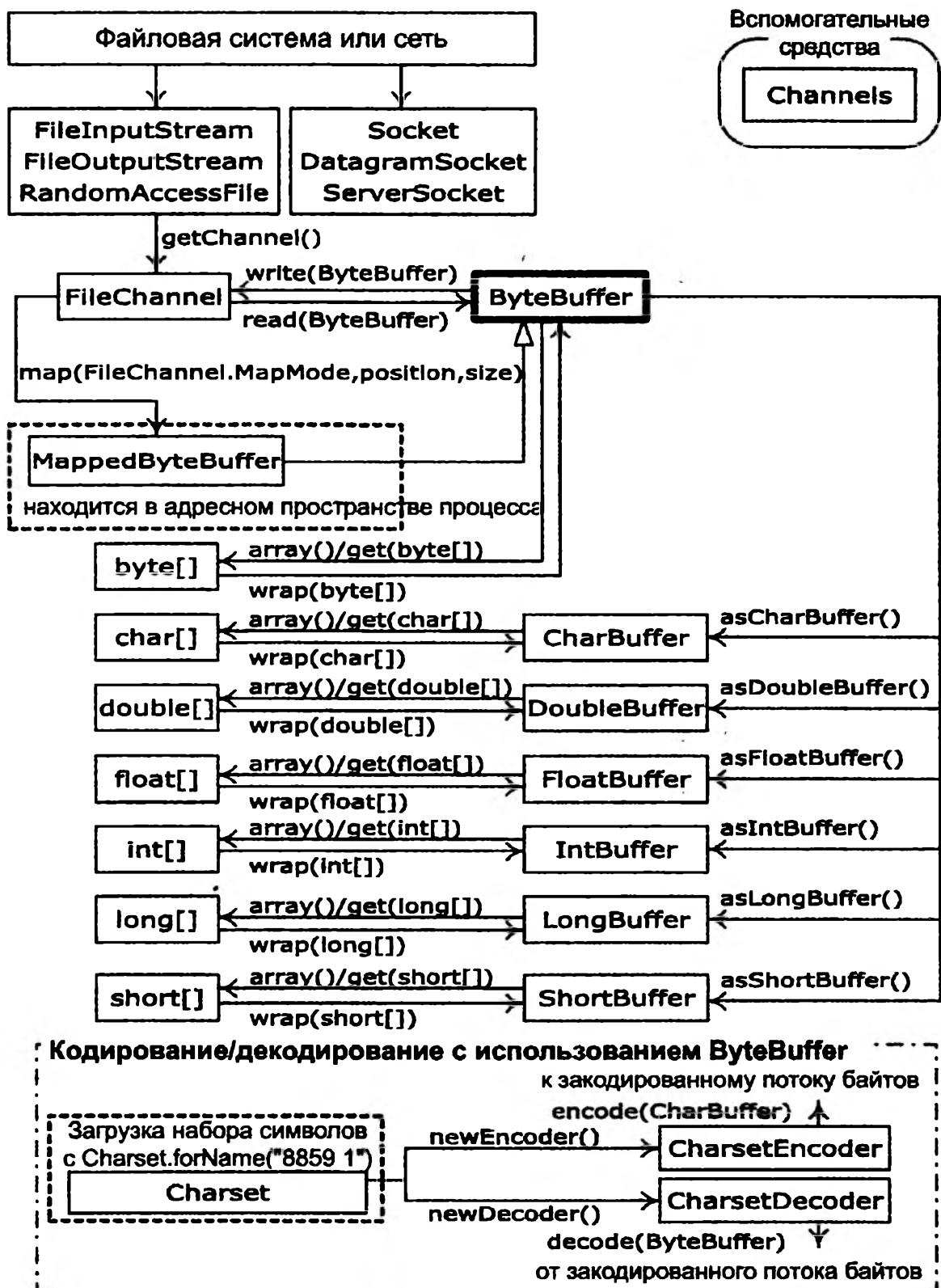
Диаграмма на с. 771 иллюстрирует отношения между классами пакета `nio`; она поможет разобраться в перемещениях и преобразованиях данных. Например, если вы захотите записать в файл байтовый массив, то сначала вложите его в буфер методом `ByteBuffer.wrap()`, затем получите из потока `FileOutputStream` канал методом `getChannel()`, а потом запишете данные буфера `ByteBuffer` в полученный канал `FileChannel`.

Заметим, что перемещать данные каналов («из» и «в») допустимо только с помощью байтовых буферов `ByteBuffer`, а для остальных примитивных типов вы можете либо создать отдельный буфер этого типа, либо получить его из байтового буфера посредством метода с префиксом `as`. Таким образом, у вас *нет* возможности преобразовать буфер с примитивными данными к байтовому буферу. Впрочем, поскольку вы в силах помещать примитивы в байтовый буфер и извлекать их оттуда с помощью представлений, это не такое уж строгое ограничение.

Подробнее о буферах

Буфер состоит из данных и четырех индексов, используемых для доступа к данным и для эффективного манипулирования ими. К этим индексам относятся *метка* (`mark`), *позиция* (`position`), *предел* (`limit`) и *вместимость* (`capacity`). Класс `Buffer` содержит методы, предназначенные для установки и сброса значений этих индексов, а также чтения их текущих значений (табл. 18.7).

Методы, вставляющие данные в буфер и считывающие их оттуда, обновляют эти индексы для учета изменений.



Следующий пример использует очень простой алгоритм (перестановка смежных символов) для шифрования и восстановления текста в буфере CharBuffer:

```
//: io/UsingBuffers.java
import java.nio.*;
import static net.mindview.util.Print.*;
```

```
public class UsingBuffers {
```

продолжение ➤

```

private static void symmetricScramble(CharBuffer buffer){
    while(buffer.hasRemaining()) {
        buffer.mark();
        char c1 = buffer.get();
        char c2 = buffer.get();
        buffer.reset();
        buffer.put(c2).put(c1);
    }
}

public static void main(String[] args) {
    char[] data = "UsingBuffers".toCharArray();
    ByteBuffer bb = ByteBuffer.allocate(data.length * 2);
    CharBuffer cb = bb.asCharBuffer();
    cb.put(data);
    print(cb.rewind());
    symmetricScramble(cb);
    print(cb.rewind());
    symmetricScramble(cb);
    print(cb.rewind());
}
} /* Output:
UsingBuffers
sUniBgfuefsr
UsingBuffers
*///:~

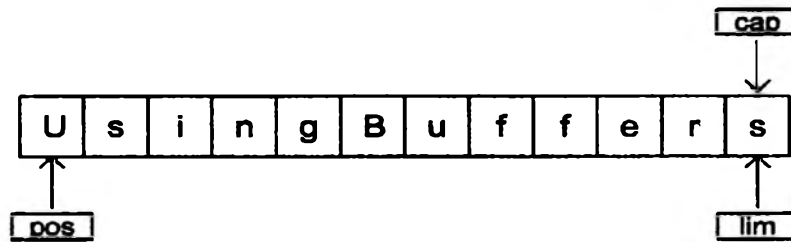
```

Таблица 18.7. Методы Buffer

Метод	Описание
capacity()	Возвращает емкость буфера
clear()	Очищает буфер, устанавливает позицию в ноль, а предел делает равным емкости. Этот метод можно вызывать для перезаписи существующего буфера
flip()	Устанавливает предельное значение равным позиции, а позицию приравнивает к нулю. Метод используется для подготовки буфера к чтению, после того как в него были записаны данные
limit()	Возвращает предел
limit(int lim)	Устанавливает предел
mark()	Приравнивает метке значение позиции
position()	Возвращает значение позиции
position(int pos)	Устанавливает значение позиции
remaining()	Возвращает разность между предельным значением и позицией
hasRemaining()	Возвращает true, если между позицией и предельным значением еще остались элементы

Хотя получить буфер `CharBuffer` можно и напрямую, вызвав для символьного массива метод `wrap()`, здесь сначала выделяется служащий основой байтовый буфер `ByteBuffer`, а символьный буфер `CharBuffer` создается как представление байтового. Это подчеркивает, что в конечном счете все манипуляции производятся с байтовым буфером, поскольку именно он взаимодействует с каналом.

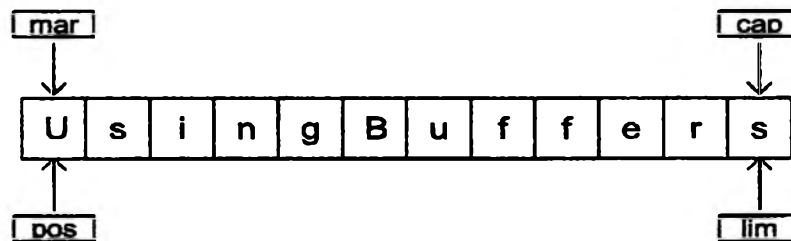
После вызова метода `symmetricScramble()` буфер выглядит следующим образом:



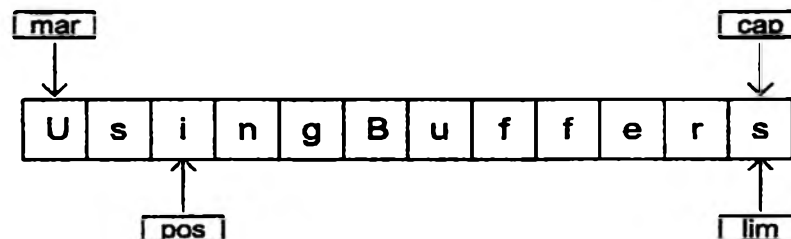
Позиция (`pos`) указывает на первый элемент буфера, *емкительность* (`cap`) и *предел* (`lim`) — на последний.

В методе `symmetricScramble()` цикл `while` выполняется до тех пор, пока *позиция* не станет равной *пределу*. *Позиция* буфера изменяется при вызове для него «относительных» методов `put()` или `get()`. Можно также использовать «абсолютные» версии методов `put()` и `get()`, которым необходимо передать аргумент-индекс, указывающий, с какого места начнет работу метод `put()` или метод `get()`. Эти методы не изменяют *позиции* буфера.

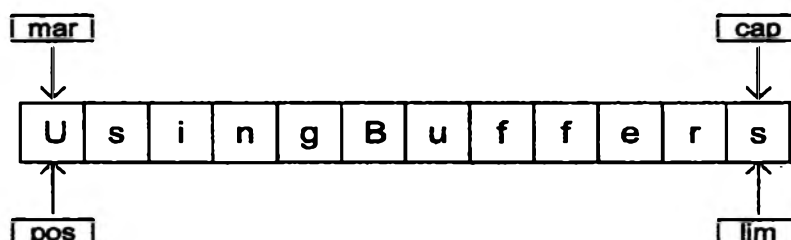
Когда управление переходит в цикл `while`, вызывается метод `mark()` для установки значения *метки* (`mar`). Состояние буфера в этот момент таково:



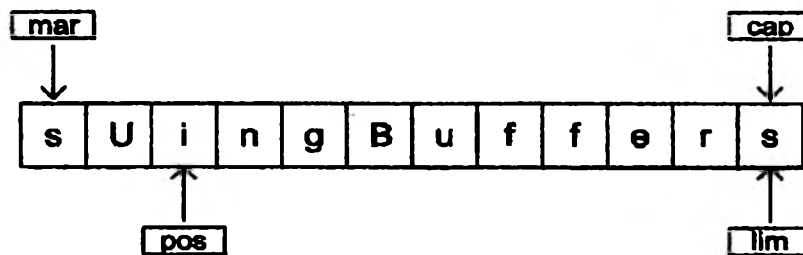
Два вызова «относительных» методов `get()` сохраняют значение первых двух символов в переменных `s1` и `s2`. После этих вызовов буфер выглядит так:



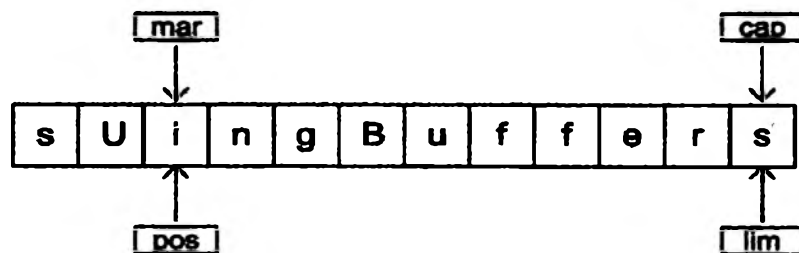
Для смешивания символов нам нужно записать символ `s2` в позицию 0, а `s1` в позицию 1. Для этого можно воспользоваться «абсолютной» версией метода `put()`, но мы приравняем позицию метке, что и делает метод `reset()`:



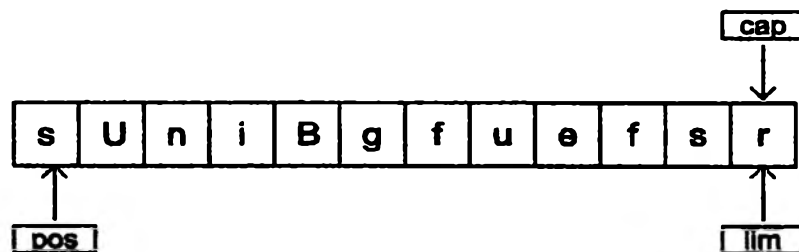
Два вызова метода `put()` записывают `c2`, а затем `c1`:



На следующей итерации значение *метки* приравнивается *позиции*:



Этот процесс продолжается до тех пор, пока не будет просмотрен весь буфер. В конце цикла `while` *позиция* находится в конце буфера. При выводе буфера на печать распечатываются только символы, находящиеся между *позицией* и *пределом*. Поэтому, если вы хотите вывести буфер целиком, придется установить *позицию* на начало буфера, используя для этого метод `rewind()`. Вот в каком состоянии находится буфер после вызова метода `rewind()` (значение *метки* стало неопределенным):



Когда снова вызывается функция `symmetricScramble()`, процесс повторяется, и буфер `CharBuffer` возвращается к своему изначальному состоянию.

Отображаемые в память файлы

Механизм отображения файлов в память позволяет вам создавать и изменять файлы, размер которых слишком велик для прямого размещения в памяти. В таком случае вы считаете, что файл целиком находится в памяти, и работаете с ним как с очень большим массивом. Такой подход значительно упрощает код, который вы пишете для изменения файла. Вот небольшой пример:

```
//: io/LargeMappedFiles.java
// Создание очень большого файла с отображением в память.
// {RunByHand}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;
import static net.mindview.util.Print.*;
```



```

public class LargeMappedFiles {
    static int length = 0x8FFFFFFF; // 128 Мбайт
    public static void main(String[] args) throws Exception {
        MappedByteBuffer out =
            new RandomAccessFile("test.dat", "rw").getChannel()
                .map(FileChannel.MapMode.READ_WRITE, 0, length);
        for(int i = 0; i < length; i++)
            out.put((byte)'x');
        print("Finished writing");
        for(int i = length/2; i < length/2 + 6; i++)
            printlnb((char)out.get(i));
    }
} ///:~

```

Для того чтобы производить одновременное чтение и запись, мы начинаем с создания объекта `RandomAccessFile`, получаем для этого файла канал, а затем вызываем метод `map()`, чтобы произвести буфер `MappedByteBuffer`, представляющий собой разновидность буфера прямого доступа. Заметьте, что необходимо указать начальную точку и длину участка, который будет проецироваться, то есть у вас есть возможность отображать маленькие участки больших файлов.

Класс `MappedByteBuffer` унаследован от буфера `ByteBuffer`, поэтому у него есть все методы последнего. Здесь показаны только самые простые вызовы методов `put()` и `get()`, но вы также можете использовать такие возможности, как метод `asCharBuffer()` и т. п.

Программа напрямую создает файл размером в 128 Мбайт, что может оказаться не «по плечу» операционной системе. Кажется, что весь файл доступен сразу, поскольку только часть его подгружается в память, в то время как остальные части выгружены. Таким образом можно работать с очень большими файлами (размером до 2 Гбайт). Заметьте, что для достижения максимальной производительности работает механизм отображения файлов используемой операционной системы.

Производительность

Хотя быстроедействие «старого» ввода-вывода было улучшено за счет переписывания его с учетом новых библиотек `nio`, техника отображения файлов качественно эффективнее. Следующая программа выполняет простое сравнение производительности:

```

//: io/MappedIO.java
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class MappedIO {
    private static int numOfInts = 4000000;
    private static int numOfUbuffInts = 200000;
    private abstract static class Tester {
        private String name;
        public Tester(String name) { this.name = name; }
        public void runTest() {
            System.out.print(name + ": ");
            try {
                long start = System.nanoTime();
                test();
                double duration = System.nanoTime() - start;
            } catch (Exception e) {}
        }
    }
}

```

продолжение ➤

```

        System.out.format("%.2f\n", duration/1.0e9);
    } catch(IOException e) {
        throw new RuntimeException(e);
    }
}
public abstract void test() throws IOException;
}
private static Tester[] tests = {
    new Tester("Stream Write") {
        public void test() throws IOException {
            DataOutputStream dos = new DataOutputStream(
                new BufferedOutputStream(
                    new FileOutputStream(new File("temp.tmp"))));
            for(int i = 0; i < numOfInts; i++)
                dos.writeInt(i);
            dos.close();
        }
    },
    new Tester("Mapped Write") {
        public void test() throws IOException {
            FileChannel fc =
                new RandomAccessFile("temp.tmp", "rw")
                    .getChannel();
            IntBuffer ib = fc.map(
                FileChannel.MapMode.READ_WRITE, 0, fc.size())
                .asIntBuffer();
            for(int i = 0; i < numOfInts; i++)
                ib.put(i);
            fc.close();
        }
    },
    new Tester("Stream Read") {
        public void test() throws IOException {
            DataInputStream dis = new DataInputStream(
                new BufferedInputStream(
                    new FileInputStream("temp.tmp")));
            for(int i = 0; i < numOfInts; i++)
                dis.readInt();
            dis.close();
        }
    },
    new Tester("Mapped Read") {
        public void test() throws IOException {
            FileChannel fc = new FileInputStream(
                new File("temp.tmp")).getChannel();
            IntBuffer ib = fc.map(
                FileChannel.MapMode.READ_ONLY, 0, fc.size())
                .asIntBuffer();
            while(ib.hasRemaining())
                ib.get();
            fc.close();
        }
    },
    new Tester("Stream Read/Write") {
        public void test() throws IOException {
            RandomAccessFile raf = new RandomAccessFile(
                new File("temp.tmp"), "rw");
            raf.writeInt(1);
        }
    }
};

```

```

        for(int i = 0; i < numOfUbuffInts; i++) {
            raf.seek(raf.length() - 4);
            raf.writeInt(raf.readInt());
        }
        raf.close();
    }
},
new Tester("Mapped Read/Write") {
    public void test() throws IOException {
        FileChannel fc = new RandomAccessFile(
            new File("temp.tmp"), "rw").getChannel();
        IntBuffer ib = fc.map(
            FileChannel.MapMode.READ_WRITE, 0, fc.size())
            .asIntBuffer();
        ib.put(0);
        for(int i = 1; i < numOfUbuffInts; i++)
            ib.put(ib.get(i - 1));
        fc.close();
    }
}
};
public static void main(String[] args) {
    for(Tester test : tests)
        test.runTest();
}
} /* Output: (90% match)
Stream Write: 0.56
Mapped Write: 0.12
Stream Read: 0.80
Mapped Read: 0.07
Stream Read/Write: 5.32
Mapped Read/Write: 0.02
*///:~

```

В предыдущих примерах книги метод `runTest()` использовался в соответствии с паттерном «Шаблонный метод» для создания среды тестирования для различных реализаций метода `test()`, определенного в анонимных внутренних subclasses. Каждый из этих subclasses выполняет свой вид теста; таким образом, методы `test()` также являются прототипами для выполнения различных действий, связанных с вводом/выводом.

Хотя кажется, что для отображаемой записи следует использовать поток `FileOutputStream`, на самом деле любые операции отображаемого вывода должны проходить через класс `RandomAccessFile`, как операции чтения/записи в рассмотренном примере.

Отметим, что в методы `test()` также включена инициализация различных объектов для работы с вводом/выводом. И несмотря на то что настройка отображаемых файлов может быть сопряжена с определенными затратами, общее преимущество по сравнению с потоковым вводом/выводом все равно получается значительным.

25. (6) Поэкспериментируйте с заменой команды `ByteBuffer.allocate()` в примерах этой главы на `ByteBuffer.allocateDirect()`. Обратите внимание на разницу в производительности; также попробуйте определить, изменилось ли время запуска инициализации программ.
26. (3) Модифицируйте `strings/JGrep.java` для использования механизма отображения файлов пакета `Java nio`.

Блокировка файлов

Блокировка файлов, появившаяся в пакете JDK 1.4, позволяет синхронизировать доступ к файлу как к совместно используемому ресурсу. Впрочем, программные потоки, претендующие на один и тот же файл, могут принадлежать различным виртуальным машинам JVM, или один поток может быть Java-поток, а другой представлять собой обычный поток операционной системы. Блокированные файлы видны другим процессам операционной системы, поскольку механизм блокировки Java напрямую связан со средствами операционной системы.

Простой пример блокировки файла:

```
//: io/FileLocking.java
import java.nio.channels.*;
import java.util.concurrent.*;
import java.io.*;

public class FileLocking {
    public static void main(String[] args) throws Exception {
        FileOutputStream fos= new FileOutputStream("file.txt");
        FileLock fl = fos.getChannel().tryLock();
        if(fl != null) {
            System.out.println("Locked File");
            TimeUnit.MILLISECONDS.sleep(100);
            fl.release();
            System.out.println("Released Lock");
        }
        fos.close();
    }
} /* Output:
Locked File
Released Lock
*///:~
```

Для блокировки всего файла целиком используется объект `FileLock`, для получения которого следует вызвать метод `tryLock()` класса `FileChannel`. (Сетевые каналы `SocketChannel`, `DatagramChannel` и `ServerSocketChannel` не нуждаются в блокировании, так как они доступны в пределах одного процесса. Вряд ли вы будете совместно использовать сокет между двумя процессами.) Метод `tryLock()` не приостанавливает программу. Он пытается овладеть объектом блокировки, но если ему это не удастся (если другой процесс уже владеет этим объектом или файл не является разделяемым), то он просто возвращает управление. Метод `lock()` ждет до тех пор, пока не удастся получить объект блокировки, или поток, в котором этот метод был вызван, не будет прерван, или же пока не будет закрыт канал, для которого был вызван метод `lock()`. Блокировка снимается методом `FileChannel.release()`.

Также возможно заблокировать часть файла, вызвав метод:

```
tryLock(long position, long size, boolean shared)
```

или

```
lock(long position, long size, boolean shared)
```

блокирующий участок файла размером `size` от позиции `position`. Третий аргумент позволяет указать, будет ли блокировка совместной.

Методы без аргументов приспособляются к изменению размеров файла, в то время как методы для блокировки участков не адаптируются к новому размеру файла. Если блокировка была наложена на область от позиции `position` до `position + size`, а затем файл увеличился и стал больше размера `position + size`, то часть файла за пределами `position + size` не блокируется. Методы без аргументов блокируют файл целиком, даже если он растет.

Поддержка блокировок с монопольным или совместным доступом должна быть встроена в операционную систему. Если операционная система не поддерживает совместные блокировки и был сделан запрос на получение такой блокировки, то используется монопольный доступ. Для получения типа блокировки (разделяемая или эксклюзивная) используется метод `FileLock.isShared()`.

Блокирование части отображаемого файла

Как упоминалось ранее, отображение файлов обычно применяется для очень больших файлов. При работе с таким большим файлом может понадобиться заблокировать некоторые его части, так чтобы другие процессы могли изменять его доступные участки. Такой подход характерен для баз данных, где он позволяет сделать базу доступной нескольким пользователям одновременно.

В следующем примере каждый из двух потоков блокирует свою собственную часть файла:

```
//: io/LockingMappedFiles.java
// Блокирование части отображаемого файла.
// {RunByHand}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class LockingMappedFiles {
    static final int LENGTH = 0x8FFFFFFF; // 128 Мбайт
    static FileChannel fc;
    public static void main(String[] args) throws Exception {
        fc =
            new RandomAccessFile("test.dat", "rw").getChannel();
        MappedByteBuffer out =
            fc.map(FileChannel.MapMode.READ_WRITE, 0, LENGTH);
        for(int i = 0; i < LENGTH; i++)
            out.put((byte)'x');
        new LockAndModify(out, 0, 0 + LENGTH/3);
        new LockAndModify(out, LENGTH/2, LENGTH/2 + LENGTH/4);
    }
    private static class LockAndModify extends Thread {
        private ByteBuffer buff;
        private int start, end;
        LockAndModify(ByteBuffer mbb, int start, int end) {
            this.start = start;
            this.end = end;
            mbb.limit(end);
            mbb.position(start);
            buff = mbb.slice();
            start();
        }
    }
}
```

продолжение ➤

```

public void run() {
    try {
        // Монопольная блокировка без перекрытия:
        FileLock fl = fc.lock(start, end, false);
        System.out.println("Locked: " + start + " to " + end);
        // Модификация:
        while(buff.position() < buff.limit() - 1)
            buff.put((byte)(buff.get() + 1));
        fl.release();
        System.out.println("Released: " + start + " to " + end);
    } catch(IOException e) {
        throw new RuntimeException(e);
    }
}
}
} ///:~

```

Класс потока `LockAndModify` устанавливает область буфера и получает его для модификации методом `slice()`. В методе `run()` запрашивается объект блокировки для файлового канала (запросить блокировку для буфера нельзя — это возможно только для канала). Вызов метода `lock()` напоминает механизм синхронизации доступа потоков к объектам, у вас появляется некая «критическая секция» с монопольным доступом к данной части файла¹.

Блокировки автоматически снимаются при завершении работы JVM, закрытии канала, для которого они были получены, но можно также явно вызвать метод `release()` объекта `FileLock`, как показано в нашем примере.

Сжатие данных

Библиотека ввода-вывода Java содержит классы, поддерживающие чтение и запись в потоки со сжатыми данными (табл. 18.8). Они базируются на уже существующих потоках ввода-вывода и предоставляют функциональность сжатия.

Эти классы не являются производными от классов `Reader` и `Writer`, а входят в иерархию `InputStream` и `OutputStream`. Это связано с тем, что библиотека сжатия работает не с символами, а с байтами. Впрочем, иногда приходится смешивать потоки двух типов. (Не забудьте, что байтовые потоки легко преобразуются в символьные — просто используйте классы `InputStreamReader` и `OutputStreamWriter`.)

Таблица 18.8. Классы для сжатия данных

Название класса	Назначение
<code>CheckedInputStream</code>	Метод <code>getChecksum()</code> возвращает контрольную сумму для любого входного потока <code>InputStream</code> (не только для потока распаковки)
<code>CheckedOutputStream</code>	Метод <code>getChecksum()</code> возвращает контрольную сумму для любого выходного потока <code>OutputStream</code> (не только для потока сжатия)
<code>DeflaterOutputStream</code>	Базовый класс для классов сжатия данных

Название класса	Назначение
ZipOutputStream	Субкласс DeflaterOutputStream, который производит сжатие данных в формате ZIP
GZIPOutputStream	Субкласс DeflaterOutputStream, который производит сжатие данных в формате GZIP
InflaterInputStream	Базовый класс для классов распаковки сжатых данных
ZipInputStream	Субкласс InflaterInputStream, который распаковывает сжатые данные, хранящиеся в формате ZIP
GZIPInputStream	Субкласс InflaterInputStream, распаковывающий сжатые данные, хранящиеся в формате GZIP

Хотя существует великое количество различных программ сжатия данных, форматы ZIP и GZIP используются, пожалуй, чаще всего. Таким образом, вы можете легко манипулировать своими сжатыми данными с помощью многочисленных программ, предназначенных для чтения и записи этих форматов.

Простое сжатие в формате GZIP

Интерфейс сжатия в формате GZIP является наиболее простым и идеально подходит для ситуаций с одним потоком данных, который необходимо уплотнить (в отличие от набора фрагментов данных). В следующем примере сжимается файл:

```

//: io/GZIPcompress.java
// {Args: GZIPcompress.java}
import java.util.zip.*;
import java.io.*;

public class GZIPcompress {
    public static void main(String[] args)
        throws IOException {
        if(args.length == 0) {
            System.out.println(
                "Usage: \nGZIPcompress file\n" +
                "\tUses GZIP compression to compress " +
                "the file to test.gz");
            System.exit(1);
        }
        BufferedReader in = new BufferedReader(
            new FileReader(args[0]));
        BufferedOutputStream out = new BufferedOutputStream(
            new GZIPOutputStream(
                new FileOutputStream("test.gz")));
        System.out.println("Writing file");
        int c;
        while((c = in.read()) != -1)
            out.write(c);
        in.close();
        out.close();
        System.out.println("Reading file");
        BufferedReader in2 = new BufferedReader(
            new InputStreamReader(new GZIPInputStream(
                new FileInputStream("test.gz"))));
        String s;
    }
}

```

продолжение →

```

        while((s = in2.readLine()) != null)
            System.out.println(s);
    }
} /* (Execute to see output) *///:~

```

Работать с классами сжатия данных очень легко: вы просто надстраиваете их для вашего выходного потока данных (`GZIPOutputStream` или `ZipOutputStream` для сжатия данных и `GZIPInputStream` или `ZipInputStream` для распаковки данных). Все остальное относится к элементарным операциям ввода-вывода. В примере продемонстрированы смешанные байтовые и символьные потоки: поток `in` основан на `Reader`, тогда как конструктор класса `GZIPOutputStream` может получать только объект `OutputStream`, но не `Writer`. Поэтому при открытии файла поток `GZIPInputStream` преобразуется в символьный поток `Reader`.

Многофайловые архивы ZIP

Библиотека, поддерживающая формат сжатия данных ZIP, обладает более широкими возможностями. С ее помощью можно легко упаковывать произвольное количество файлов, и даже существует отдельный класс для чтения файлов в формате ZIP. В библиотеке используется стандартный ZIP-формат, поэтому сжатые ею данные будут восприниматься практически любой программой, доступной в Интернете. В следующем примере используется та же структура, что и в предыдущем, но количество файлов, указываемых в командной строке, не ограничено. Вдобавок она демонстрирует использование класса `Checksum` для вычисления и проверки контрольной суммы. Таких типов контрольных сумм в Java два: один представлен классом `Adler32` (этот алгоритм быстрее), а другой классом `CRC32` (медленнее, однако гораздо точнее).

```

//: io/ZipCompress.java
// Использование формата ZIP для сжатия любого
// количества файлов, указанных в командной строке.
// {Args: ZipCompress.java}
import java.util.zip.*;

import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ZipCompress {
    public static void main(String[] args)
        throws IOException {
        FileOutputStream f = new FileOutputStream("test.zip");
        CheckedOutputStream csum =
            new CheckedOutputStream(f, new Adler32());
        ZipOutputStream zos = new ZipOutputStream(csum);
        BufferedOutputStream out =
            new BufferedOutputStream(zos);
        zos.setComment("A test of Java Zipping");
        // Впрочем, соответствующего метода getComment() нет.
        for(String arg : args) {
            print("Writing file " + arg);
            BufferedReader in =
                new BufferedReader(new FileReader(arg));
            zos.putNextEntry(new ZipEntry(arg));

```



```

    int c;
    while((c = in.read()) != -1)
        out.write(c);
    in.close();
    out.flush();
}
out.close();
// Контрольная сумма действительна только после закрытия файла!
print("Checksum: " + csum.getChecksum().getValue());
// Теперь извлечь файлы:
print("Reading file");
FileInputStream fi = new FileInputStream("test.zip");
CheckedInputStream csumi =
    new CheckedInputStream(fi, new Adler32());
ZipInputStream in2 = new ZipInputStream(csumi);
BufferedInputStream bis = new BufferedInputStream(in2);
ZipEntry ze;
while((ze = in2.getNextEntry()) != null) {
    print("Reading file " + ze);
    int x;
    while((x = bis.read()) != -1)
        System.out.write(x);
}
if(args.length == 1)
    print("Checksum: " + csumi.getChecksum().getValue());
bis.close();
// Альтернативный способ открытия и чтения ZIP-файлов:
ZipFile zf = new ZipFile("test.zip");
Enumeration e = zf.entries();
while(e.hasMoreElements()) {
    ZipEntry ze2 = (ZipEntry)e.nextElement();
    print("File: " + ze2);
    // ...после чего данные извлекаются так же, как прежде
}
/* if(args.length == 1) */
}
} /* (Execute to see output) *///:~

```

Для каждого файла, добавляемого в архив, необходимо вызвать метод `putNextEntry()` с соответствующим объектом `ZipEntry`. Класс `ZipEntry` содержит все необходимое для того, чтобы добавить к отдельной позиции ZIP-файла дополнительную информацию: имя файла, размер в сжатом и обычном виде, контрольную сумму CRC, дополнительные данные, комментарии, метод сжатия, признак каталога. Хотя в оригинальном формате ZIP можно установить пароли, библиотека Java такую возможность не поддерживает. Аналогичное ограничение встречается и при использовании контрольных сумм: потоки `CheckedInputStream` и `CheckedOutputStream` поддерживают оба вида контрольных сумм — и `Adler32`, и `CRC32`, однако в классе `ZipEntry` поддерживается только `CRC`. Это ограничение продиктовано требованиями формата ZIP, но при этом вы лишаетесь возможности использования более быстрой контрольной суммы `Adler32`.

Для извлечения файлов в классе `ZipInputStream` предусмотрен метод `getNextEntry()`, который возвращает очередное вхождение архива `ZipEntry`, если оно существует. В качестве более компактной альтернативы можно использовать для архива объект `ZipFile`, чей метод `entries()` возвращает итератор `Enumeration` для перемещения по доступным элементам архивного файла.

Чтобы иметь доступ к контрольной сумме, необходимо каким-либо образом обратиться к соответствующему объекту `Checksum`. В нашем случае сохраняются ссылки на потоки `CheckedInputStream` и `CheckedOutputStream`, хотя можно было бы просто сохранить ссылки на объекты `Checksum`.

Совершенно непонятно, зачем в библиотеку сжатия ZIP был добавлен метод `setComment()`, вставляющий в архивный файл комментарий. Как указано в примере, добавить комментарий при получении архивного файла можно, но восстановить его при чтении архива потоком `ZipInputStream` нельзя¹. Полноценная поддержка комментариев реализуется только при работе с отдельными элементами ZIP-архива, объектами `ZipEntry`.

Конечно, возможности сжатия данных при использовании библиотек GZIP и ZIP не ограничены файлами; сжимать можно все, что угодно, даже данные, передаваемые через сетевые подключения.

Архивы Java ARchives (файлы JAR)

Формат ZIP также применяется в файлах JAR (архивы Java ARchive), которые используются для упаковки группы файлов в один сжатый файл. Однако, как и всё в языке Java, файлы JAR являются кросс-платформенными. Наравне с файлами `.class`, в них могут содержаться также любые файлы, например с графикой и мультимедиа.

Файлы JAR особенно полезны при работе с Интернетом. До их появления веб-браузерам приходилось выдавать отдельный запрос к серверу для каждого файла, необходимого для запуска апплета. Вдобавок все эти файлы не сжимались. За счет объединения всех нужных для запуска апплета файлов в одном сжатом файле сокращается время запроса к серверу, при этом уменьшается и загрузка сервера. К тому же каждый элемент архива можно снабдить цифровой подписью.

Файл JAR содержит набор сжатых файлов вместе с *манифестом* (`manifest`), который их описывает. (Создать манифест можно самому; иначе эту работу выполнит программа `jar`.) Подробнее узнать о манифесте формата JAR можно в интерактивной документации пакета JDK.

Утилита `jar`, которая поставляется вместе с пакетом разработки программ JDK, автоматически сжимает файлы по вашему выбору. Запускается эта программа из командной строки:

```
jar [параметры] место_назначения [манифест] список_файлов
```

Параметры запуска — просто набор букв (дефисы или другие признаки не нужны). Пользователи систем UNIX/Linux сразу заметят сходство с программой `tar`. Допустимы следующие параметры:

¹ Возможно, дело в том, что комментарий ZIP-файла может быть многострочным объемом до 64 Кбайт (в зависимости от версии архиватора, при этом он не сжимается), в отличие от комментариев для отдельных вхождений. В таком случае текст комментария добавляется (замещается) из отдельного файла TXT. Автор готов с этим согласиться. — *Примеч. ред.*

c	Создает новый или пустой архив
t	Распечатывает содержимое архива
x	Извлекает все файлы
x файл	Извлекает все указанные файлы
f	Признак задания имени файла. Если не использовать этот параметр, jar решит, что входные данные поступают из стандартного ввода, или при создании файла — что выходные данные направляются в стандартный поток вывода
m	Указывает, что в первом аргументе передается имя файла с пользовательским манифестом
v	Выводит подробную информацию о функциональности jar
0	Просто сохраняет файлы без сжатия (для создания файлов JAR, которые можно указать в переменной окружения CLASSPATH)
M	Не включать манифест

Если в списке файлов имеется каталог, то его содержимое вместе с подкаталогами и всеми файлами автоматически помещается в файл JAR. Информация о пути файлов также сохраняется.

Далее перечислены наиболее типичные способы запуска программы jar:

```
jar cf myJarFile.jar *.class
```

Команда создает файл JAR с именем myJarFile.jar, в котором содержатся все файлы классов .class из текущего каталога, вместе с автоматически созданным манифестом.

```
jar cmf myJarFile.jar myManifestFile.mf *.class
```

Почти идентична предыдущей команде, за одним исключением — в полученный файл JAR добавляется пользовательский манифест из файла myManifestFile.mf.

```
jar tf myJarFile.jar
```

Выводит содержимое (список файлов) архива myJarFile.jar.

```
jar tvf myJarFile.jar
```

К предыдущей команде добавляется флаг v для получения более подробной информации о файлах, содержащихся в архиве myJarFile.jar.

```
jar cvf myApp.jar audio classes image
```

Предполагается, что audio, classes и image — это каталоги, их содержимое помещается в файл myApp.jar. Благодаря параметру v в процессе сжатия выводится дополнительная информация об упаковываемых файлах.

Если вы создаете файл JAR в режиме 0 (ноль), такой файл может быть включен в переменную окружения CLASSPATH:

```
CLASSPATH="lib1.jar;lib2.jar;"
```

После этого среда исполнения Java сможет искать файлы с классами в архивах lib1.jar и lib2.jar.

Утилита `jar` не настолько универсальна, как архиватор `zip`. Например, она не позволяет добавлять или обновлять файлы в уже существующем архиве JAR. Также нельзя перемещать файлы и удалять их после перемещения. Но при этом созданный файл JAR всегда читается инструментом `jar` на другой платформе (архиваторы `zip` о такой совместимости могут только мечтать)¹.

Как будет показано в главе 22, файлы JAR также используются для упаковки визуальных компонентов `JavaBean`.

Сериализация объектов

Созданный вами объект существует до тех пор, пока он вам нужен, но ни при каких условиях он не будет существовать после завершения программы. На первый взгляд это выглядит логично, но в некоторых ситуациях было бы очень полезно, если бы информация объекта продолжала сохраняться даже в то время, когда программа не работает. Затем при следующем запуске объект уже находится на своем месте и содержит ту же информацию, что при последнем запуске. Конечно, нужного эффекта можно добиться, сохраняя информацию в файл или базу данных, но в духе объектно-ориентированного программирования было бы удобно иметь возможность объявления «долговременных» объектов, чтобы система позаботилась обо всех подробностях за вас.

Сериализация (serialization) объектов Java позволяет преобразовать любой объект, реализующий интерфейс `Serializable`, в последовательность байтов, по которой затем можно полностью восстановить исходный объект. Сказанное справедливо и для сетевых соединений, а это значит, что механизм сериализации автоматически компенсирует различия между операционными системами. То есть можно создать объект на машине с ОС Windows, превратить его в последовательность байтов и послать их по сети на машину с ОС UNIX, где объект будет корректно воссоздан. Вам не надо думать о различных форматах данных, порядке следования байтов или других подробностях.

Сама по себе сериализация объектов интересна потому, что с ее помощью можно осуществить *легковесное долговременное хранение* (lightweight persistence). Это означает, что время жизни объекта определяется не только временем выполнения программы — объект существует и между запусками программы. Можно взять объект и записать его на диск, а после, при другом запуске программы, восстановить его в первоначальном виде и таким образом получить эффект «живучести». Почему такой способ хранения называется «легковесным»? Дело в том, что программа не может определить объект как «постоянный» при помощи некоторого ключевого слова, то есть долговременное хранение напрямую не поддерживается языком (хотя со временем следует этого ожидать). Система выполнения не заботится о деталях сериализации — вам приходится собственноручно сериализовывать и восстанавливать объекты вашей программы. Если вам необходим более серьезный механизм сериализации, попробуйте библиотеку `Hibernate` (<http://hibernate.sourceforge.net>). За подробностями обращайтесь к книге *Thinking In Enterprise Java*, которую можно загрузить с сайта www.MindView.net.

¹ Для платформы Windows архив JAR является не чем иным, как файлом ZIP (сменив расширение, вы сможете работать с ним как с обычным ZIP-файлом). — *Примеч. ред.*

Механизм сериализации объектов был добавлен в язык для поддержки двух расширенных возможностей. *Удаленный вызов методов* Java (RMI) позволяет работать с объектами, «обитающими» на других компьютерах, так, словно они существуют на вашей машине. При отправке сообщений удаленным объектам необходимо транспортировать аргументы и возвращаемые значения. Для этого используется сериализация объектов. Удаленный вызов методов обстоятельно обсуждается в книге *Thinking In Enterprise Java*.

Сериализация объектов также требуется визуальным компонентам JavaBean, которые описаны в главе 22. Информация состояния визуальных компонентов обычно задается во время разработки. Эту информацию о состоянии необходимо сохранить, а затем при запуске программы восстановить; данную задачу решает сериализация объектов.

Сериализовать объект достаточно просто, если он реализует интерфейс `Serializable` (это идентификационный интерфейс, в нем нет ни одного метода). Когда в язык добавили механизм сериализации, во многие классы стандартной библиотеки внесли изменения так, чтобы они были готовы к сериализации. К таким классам относятся все классы-«обертки» для простейших типов, все классы контейнеров и многие другие. Даже объекты `Class`, представляющие классы, можно сериализовать.

Для того чтобы сериализовать объект, требуется создать выходной поток `OutputStream`, который нужно упаковать в объект `ObjectOutputStream`. На этой стадии остается вызывать метод `writeObject()`, чтобы объект был сериализован и передан в выходной поток `OutputStream` (процесс сериализации объекта осуществляется на уровне байтов, поэтому используются иерархии `InputStream` и `OutputStream`). Для восстановления объекта необходимо упаковать объект `InputStream` в `ObjectInputStream`, а затем вызвать метод `readObject()`. Как обычно, такой метод возвращает ссылку на `Object`, поэтому после вызова метода следует провести нисходящее преобразование для получения объекта нужного типа.

В процессе сериализации объектов учитываются ссылки, содержащиеся в объекте. Сохраняется не только сам образ объекта, но и все связанные с ним объекты, и все объекты в связанных объектах и т. д. Это часто называют «паутиной объектов», к которой можно присоединить одиночный объект, а также массив ссылок на объекты и объекты-члены. Если бы вы создавали свой собственный механизм сериализации, отслеживание всех имеющихся в объектах ссылок стало бы весьма нелегкой задачей. Однако в сериализации объектов Java, похоже, никаких трудностей со ссылками нет — судя по всему, в этот язык встроен достаточно эффективный алгоритм создания графов объектов. Следующий пример проверяет механизм сериализации, создавая цепочку связанных объектов, каждый из которых связан со следующим сегментом цепочки, а также имеет массив ссылок на объекты другого класса с именем `Data`:

```
//: io/Worm.java
// Сериализация объектов.
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Data implements Serializable {
    private int n;
    public Data(int n) { this.n = n; }
    public String toString() { return Integer.toString(n); }
}
```

продолжение ➤

```

public class Worm implements Serializable {
    private static Random rand = new Random(47);
    private Data[] d = {
        new Data(rand.nextInt(10)),
        new Data(rand.nextInt(10)),
        new Data(rand.nextInt(10))
    };
    private Worm next;
    private char c;
    // Значение i равно количеству сегментов
    public Worm(int i, char x) {
        print("Worm constructor: " + i);
        c = x;
        if(--i > 0)
            next = new Worm(i, (char)(x + 1));
    }
    public Worm() {
        print("Default constructor");
    }
    public String toString() {
        StringBuilder result = new StringBuilder(":");
        result.append(c);
        result.append("(");
        for(Data dat : d)
            result.append(dat);
        result.append(")");
        if(next != null)
            result.append(next);
        return result.toString();
    }
    public static void main(String[] args)
        throws ClassNotFoundException, IOException {
        Worm w = new Worm(6, 'a');
        print("w = " + w);
        ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream("worm.out"));
        out.writeObject("Worm storage\n");
        out.writeObject(w);
        out.close(); // Также сбрасывает буфер
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream("worm.out"));
        String s = (String)in.readObject();
        Worm w2 = (Worm)in.readObject();
        print(s + "w2 = " + w2);
        ByteArrayOutputStream bout =
            new ByteArrayOutputStream();
        ObjectOutputStream out2 = new ObjectOutputStream(bout);
        out2.writeObject("Worm storage\n");
        out2.writeObject(w);
        out2.flush();
        ObjectInputStream in2 = new ObjectInputStream(
            new ByteArrayInputStream(bout.toByteArray()));
    }
}

```

```
s = (String)in2.readObject();
Worm w3 = (Worm)in2.readObject();
print(s + "w3 = " + w3);
}
} /* Output:
Worm constructor: 6
Worm constructor: 5
Worm constructor: 4
Worm constructor: 3
Worm constructor: 2
Worm constructor: 1
w = :a(853):b(119):c(802):d(788):e(199):f(881)
Worm storage
w2 = :a(853):b(119):c(802):d(788):e(199):f(881)
Worm storage
w3 = :a(853):b(119):c(802):d(788):e(199):f(881)
*///:~
```

Чтобы сделать все немного интереснее, массив объектов `Data` в классе `Worm` инициализируется случайными числами. (Таким образом, нельзя заподозрить компилятор в том, что он использует дополнительную информацию для хранения объектов.) Каждый сегмент `Worm` помечается порядковым номером-символом (`char`), который автоматически генерируется в процессе рекурсивного формирования связанной цепочки объектов `Worm`. При создании цепочки необходимо указать ее размер в конструкторе класса `Worm`. Для инициализации ссылки `next` рекурсивно вызывается конструктор класса `Worm`, однако с каждым разом размер цепочки уменьшается на единицу. В последнем сегменте цепочки ссылка `next` остается со значением `null`, что показывает на конец цепочки.

Все это было сделано с единственной целью — создать достаточно сложную структуру для проверки сериализации «на прочность». Впрочем, сам акт сериализации проходит проще простого. После того как создается поток `ObjectOutputStream` (на основе другого выходного потока), метод `writeObject()` записывает в него объект. Заметьте, что в поток также записывается строка (`String`). В этот же поток можно поместить все простейшие типы, используя те же методы, что и в классе `DataOutputStream` (оба потока реализуют одинаковый интерфейс).

В программе есть два похожих фрагмента кода. В первом запись и чтение производятся в файл, а во втором для разнообразия хранилищем служит массив байтов `ByteArray`. Чтение и запись объектов посредством сериализации возможны в любые классы, производные от `DataInputStream` или `DataOutputStream`, в том числе и в сетевые подключения (эта возможность описана в книге *Thinking In Enterprise Java*).

Из вывода видно, что десериализованный объект действительно содержит все ссылки, присутствовавшие в исходном объекте.

Заметьте, что в процессе восстановления объекта, реализующего интерфейс `Serializable`, никакие конструкторы (даже конструктор по умолчанию) не вызываются. Объект

восстанавливается целиком и полностью из данных, считанных из входного потока `InputStream`¹.

- 27.** (1) Создайте класс, реализующий `Serializable`, содержащий ссылку на объект другого класса, реализующего `Serializable`. Создайте экземпляры первого класса, сериализуйте его на диск, затем восстановите и убедитесь в том, что процесс работает правильно.

Поиск класса

А что необходимо для восстановления объекта после проведения сериализации? Например, предположим, что вы создали объект, сериализовали его и отправили в виде файла или через сетевое соединение на другой компьютер. Сумеет ли программа на другом компьютере реконструировать объект, опираясь только на те данные, что вы записали в файл в процессе сериализации?

Лучший способ получить ответ на этот вопрос (как обычно, впрочем) — провести эксперимент. Следующий файл располагается в подкаталоге данной главы:

```
//: io/Alie.n.java
// Сериализуемый класс.
import java.io.*;
public class Alie.n implements Serializable {} ///:~
```

Файл, который создает и сериализует объект `Alie.n`, помещается в тот же каталог:

```
//: io/FreezeAlie.n.java
// Создание файла с данными сериализации.
import java.io.*;

public class FreezeAlie.n {
    public static void main(String[] args) throws Exception {
        ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream("X.file"));
```

¹ В этом разделе Эккель рассказывает о сериализации *по умолчанию*, где для успешного последующего восстановления объекта требуется соблюдение трех условий, которые здесь не оговариваются. Во-первых, классы, реализующие интерфейс `Serializable`, обязаны иметь конструктор без аргументов (здесь это конструктор по умолчанию `Worm()`). Этот конструктор *будет вызван*, когда объект восстанавливается из файла с расширением `SER` (в такой файл записываются сериализуемые пакетом Beans Development Kit (BeanBox) компоненты JavaBean). Во-вторых, если `Serializable` уже реализован в суперклассе, то его реализация не требуется в подклассе, то есть способность к сериализации наследуется. И последнее, по умолчанию сериализуются только не статические и не `transient`-поля (см. раздел ниже «Ключевое слово `transient`»). Поля сериализуемых классов целиком и полностью восстанавливаются из потока, но предполагается и случай, когда подтипы несериализуемых классов должны сериализоваться. Это возможно только при наличии расширяющего несериализуемый класс конструктора без аргументов. Во время десериализации поля несериализуемого класса будут инициализированы через объявленный открытым (`public`) или защищенным (`protected`) конструктор без аргументов. Этот конструктор должен быть доступен из сериализуемого подкласса. В противном случае будет получена ошибка времени выполнения. Все это подробно описано в документации Java (`java.io.Serializable.java`) и в руководстве *Java tutorial* от Sun (файл `\javabeans\persistence\index.html`). — *Примеч. ред.*


```

    Alien quellek = new Alien();
    out.writeObject(quellek);
}
} ///:~

```

Вместо того чтобы перехватывать и обрабатывать исключения, программа действует крайне небрежно — она пересылает исключения за пределы метода `main()`, поэтому информация о них будет выводиться в консольном окне.

После того как вы скомпилируете и запустите этот код, в каталоге `io` появится файл с именем `x.file`. Следующая программа находится в подкаталоге `xfiles`:

```

//: io/xfiles/ThawAlien.java
// Попытка восстановления сериализованного файла
// без сохранения класса объекта в этом файле.
// {RunByHand}
import java.io.*;

public class ThawAlien {
    public static void main(String[] args) throws Exception {
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream(new File("../", "X.file")));
        Object mystery = in.readObject();
        System.out.println(mystery.getClass());
    }
} /* Output:
class Alien
*///:~

```

Даже для открытия файла и чтения объекта `mystery` необходим объект `Class` для `Alien`; виртуальная машина Java (JVM) не может найти файл `Alien.class` (если только он не находится через переменную окружения `CLASSPATH`, но в этом примере такого быть не должно). Возникает исключение `ClassNotFoundException`. Таким образом, виртуальная машина Java должна быть способна обнаружить ассоциированный с объектом `Alien` файл `.class`.

Управление сериализацией

Как вы могли убедиться, встроенный механизм сериализации достаточно прост в применении. Но что, если у вас возникли особые требования? Возможно, из соображений безопасности вы не хотите сохранять некоторые части вашего объекта, или же сериализовать какой-либо объект, содержащийся в главном объекте, не имеет смысла, так как при восстановлении он все равно должен создаваться заново.

Чтобы управлять процессом сериализации, реализуйте в своем классе интерфейс `Externalizable` вместо интерфейса `Serializable`. Этот интерфейс расширяет оригинальный интерфейс `Serializable` и добавляет в него два метода, `writeExternal()` и `readExternal()`, которые автоматически вызываются в процессе сериализации и восстановления объектов, позволяя вам попутно выполнить специфические действия.

Ниже приведен простой пример реализации `Externalizable`. Заметьте также, что классы `Blip1` и `Blip2` практически одинаковы — разные у них только названия, да и те из серии «Найдите 10 отличий»:

```

//: io/Blips.java
// Простая реализация интерфейса Externalizable - с ловушкой.
import java.io.*;
import static net.mindview.util.Print.*;

class Blip1 implements Externalizable {
    public Blip1() {
        print("Blip1 Constructor");
    }
    public void writeExternal(ObjectOutput out)
        throws IOException {
        print("Blip1.writeExternal");
    }
    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException {
        print("Blip1.readExternal");
    }
}

class Blip2 implements Externalizable {
    Blip2() {
        print("Blip2 Constructor");
    }
    public void writeExternal(ObjectOutput out)
        throws IOException {
        print("Blip2.writeExternal");
    }
    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException {
        print("Blip2.readExternal");
    }
}

public class Blips {
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
        print("Constructing objects:");
        Blip1 b1 = new Blip1();
        Blip2 b2 = new Blip2();
        ObjectOutputStream o = new ObjectOutputStream(
            new FileOutputStream("Blips.out"));
        print("Saving objects:");
        o.writeObject(b1);
        o.writeObject(b2);
        o.close();
        // Получаем обратно:
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream("Blips.out"));
        print("Recovering b1:");
        b1 = (Blip1)in.readObject();
        // БЕДА! Выдается исключение:
        /// print("Recovering b2:");
        /// b2 = (Blip2)in.readObject();
    }
} /* Output:
Constructing objects:
Blip1 Constructor
Blip2 Constructor

```

```
Saving objects:
Blip1.writeExternal
Blip2.writeExternal
Recovering b1:
Blip1 Constructor
Blip1.readExternal
*///:~
```

Почему же в программе не восстанавливается объект `Blip2`? При попытке его восстановления в программе возбуждается исключение. Видите ли вы разницу между классами `Blip1` и `Blip2`? Конструктор класса `Blip1` является открытым (`public`), в то время как конструктор класса `Blip2` таковым не является, и именно это приводит к исключению в процессе восстановления. Попробуйте объявить конструктор класса `Blip2` открытым и удалить комментарии `//!` и вы увидите, что все работает, как и было запланировано.

При восстановлении `b1` вызывается конструктор по умолчанию класса `Blip1`. Ситуация отличается от восстановления объекта, реализующего интерфейс `Serializable`, при котором объект конструируется исключительно на основе сохраненных данных, без вызова конструкторов. В случае с объектом `Externalizable` происходит нормальный процесс реконструирования (включая инициализацию в точке определения), и далее вызывается метод `readExternal()`. Помните об этом при реализации объектов `Externalizable` — в особенности обратите внимание на то, что вызывается конструктор по умолчанию.

В следующем примере показано, что надо сделать для полноты операций сохранения и восстановления объекта `Externalizable`:

```
//: io/Blip3.java
// Восстановление объекта Externalizable.
import java.io.*;
import static net.mindview.util.Print.*;

public class Blip3 implements Externalizable {
    private int i;
    private String s; // Инициализация не проводится
    public Blip3() {
        print("Blip3 Constructor");
        // Ни s, ни i не инициализируются
    }
    public Blip3(String x, int a) {
        print("Blip3(String x, int a)");
        s = x;
        i = a;
        // s и i инициализируются только в конструкторе,
        // не являющемся конструктором по умолчанию
    }
    public String toString() { return s + i; }
    public void writeExternal(ObjectOutput out)
    throws IOException {
        print("Blip3.writeExternal");
        // Обязательные действия:
        out.writeObject(s);
        out.writeInt(i);
    }
    public void readExternal(ObjectInput in)
```

продолжение ➤

```

throws IOException, ClassNotFoundException {
    print("Blip3.readExternal");
    // Обязательные действия:
    s = (String)in.readObject();
    i = in.readInt();
}
public static void main(String[] args)
throws IOException, ClassNotFoundException {
    print("Constructing objects:");
    Blip3 b3 = new Blip3("A String ", 47);
    print(b3);
    ObjectOutputStream o = new ObjectOutputStream(
        new FileOutputStream("Blip3.out"));
    print("Saving object:");
    o.writeObject(b3);
    o.close();
    // Теперь получаем обратно:
    ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("Blip3.out"));
    print("Recovering b3:");
    b3 = (Blip3)in.readObject();
    print(b3);
}
} /* Output:
Constructing objects:
Blip3(String x, int a)
A String 47
Saving object:
Blip3.writeExternal
Recovering b3:
Blip3 Constructor
Blip3.readExternal
A String 47
*///:~

```

Поля *s* и *i* заполняются значениями только во втором конструкторе, но не в конструкторе по умолчанию. Это значит, что если вы не проинициализируете переменные *s* и *i* в методе `readExternal()`, то *s* останется ссылкой `null`, а *i* будет равно нулю (так как при создании объекта его память обнуляется). Если закомментируете по две строки после комментариев Обязательные действия и запустите программу, то обнаружите, что так оно и будет: в восстановленном объекте ссылка *s* имеет значение `null`, а целое *i* равно нулю.

Если вы наследуете от объекта, реализующего интерфейс `Externalizable`, то при выполнении сериализации следует вызывать методы базового класса `writeExternal()` и `readExternal()`, чтобы правильно сохранить и восстановить компоненты базового класса.

Итак, чтобы сериализация выполнялась правильно, нужно не только записать всю значимую информацию в методе `writeExternal()` (для объектов `Externalizable` не существует автоматической записи объектов-членов), но и восстановить ее затем в методе `readExternal()`. На первый взгляд может показаться, что из-за вызова конструктора по умолчанию какие-то действия по записи и восстановлению объектов `Externalizable` происходят сами по себе. Но это не так.

- 28.** (2) Скопируйте файл `Blips.java` и переименуйте его в файл `BlipCheck.java`, а класс `Blip2` переименуйте в `BlipCheck` (также объявите его как `public` и одновременно удалите

спецификатор `public` класса `Blips`). Уберите комментарии `///
тированные строки программы тоже выполнялись. После этого закомментируйте конструктор по умолчанию класса BlipCheck. Запустите полученную программу и объясните, почему все работает. Учтите, что после компиляции программа все равно должна запускаться командой java Blips, поскольку метод main() все еще находится в этом модуле.`

29. (2) В программе `Blip3.java` закомментируйте две строки после выражения **Обязательные действия** и запустите программу. Объясните причину, по которой результат ее работы отличается от того случая, когда обе закомментированные строки выполняются.

Ключевое слово `transient`

При управлении процессом сериализации возможно возникновение ситуации, когда автоматическое сохранение и восстановление некоторого подобъекта механизмом сериализации Java нежелательно — например, если в объекте хранится конфиденциальная информация (пароль и т. д.). Даже если информация в объекте описана как закрытая (`private`), это не спасает ее от сериализации, после которой возможно извлечь секретные данные из файла или из перехваченного сетевого пакета.

Первый способ предотвратить сериализацию некоторой важной части объекта — реализовать интерфейс `Externalizable`, что уже было показано. Тогда автоматически вообще ничего не записывается, и вы можете явно указать, какие части объекта должны сериализоваться внутри метода `writeExternal`.

Однако работать с объектами `Serializable` удобнее, поскольку сериализация для них проходит полностью автоматически. Для управления процессом можно использовать ключевое слово `transient`, которое означает: «Это поле не надо ни сохранять, ни восстанавливать — это моя прерогатива».

Например, рассмотрим объект `Logon` с информацией о сеансе входа в систему, при котором пользователь вводит имя и пароль. Предположим, что после проверки информации ее необходимо сохранить выборочно, без пароля. Для этого проще всего реализовать интерфейс `Serializable` и объявить поле с паролем `password` как `transient`. Вот как это будет выглядеть:

```
//: io/Logon.java
// Демонстрация ключевого слова transient.
import java.util.concurrent.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Logon implements Serializable {
    private Date date = new Date();
    private String username;
    private transient String password;
    public Logon(String name, String pwd) {
        username = name;
        password = pwd;
    }
    public String toString() {
```

продолжение ➤

```

    return "logon info: \n  username: " + username +
           "\n  date: " + date + "\n  password: " + password;
}
public static void main(String[] args) throws Exception {
    Logon a = new Logon("Hulk", "myLittlePony");
    print("logon a = " + a);
    ObjectOutputStream o = new ObjectOutputStream(
        new FileOutputStream("Logon.out"));
    o.writeObject(a);
    o.close();
    TimeUnit.SECONDS.sleep(1); // Задержка
    // Теперь получаем объект обратно:
    ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("Logon.out"));
    print("Recovering object at " + new Date());
    a = (Logon)in.readObject();
    print("logon a = " + a);
}
} /* Output: (Sample)
logon a = logon info:
  username: Hulk
  date: Sat Nov 19 15:03:26 MST 2005
  password: myLittlePony
Recovering object at Sat Nov 19 15:03:28 MST 2005
logon a = logon info:
  username: Hulk
  date: Sat Nov 19 15:03:26 MST 2005
  password: null
*///:~

```

Поля `date` и `username` объявлены без модификатора `transient`, поэтому сериализация для них проводится автоматически. Однако поле `password` описано как `transient` и поэтому не сбрасывается на диск; кроме того, механизм сериализации его игнорирует при восстановлении. Когда объект восстанавливается, поле `password` равно `null`. Заметьте, что хотя метод `toString()` собирает объект `String` с использованием перегруженного оператора `+`, ссылка `null` автоматически преобразуется в строку `null`.

Также видно, что поле `date` сохраняется на диске и при восстановлении его значение не меняется.

Так как объекты `Externalizable` по умолчанию не сохраняют полей, ключевое слово `transient` для них не имеет смысла. Оно применяется только для объектов `Serializable`.

Альтернатива для `Externalizable`

Если вас по каким-либо причинам не прельщает реализация интерфейса `Externalizable`, существует и другой подход. Вы можете реализовать интерфейс `Serializable` и *добавить* (заметьте: именно «добавить», а не «переопределить» или «реализовать») методы с именами `writeObject()` и `readObject()`. Они автоматически будут вызваны при сериализации и восстановлении объектов. То есть если вы предоставите эти два метода, они будут использоваться вместо сериализации по умолчанию.

Эти методы должны иметь четко определенную сигнатуру:

```

private void writeObject(ObjectOutputStream stream)
throws IOException;

```

```
private void readObject(ObjectInputStream stream)
throws IOException, ClassNotFoundException
```

С точки зрения архитектуры программы этот подход выглядит странно. Во-первых, можно подумать, что раз уж эти методы не являются частью базового класса и не определены в интерфейсе `Serializable`, они должны определяться в собственном интерфейсе. Но поскольку эти методы описаны как закрытые (`private`), вызывать их могут лишь члены их собственного класса. Однако члены обычных классов их не вызывают, вместо этого вызов исходит из методов `writeObject()` и `readObject()` классов `ObjectInputStream` и `ObjectOutputStream`. (Обратите внимание на мою сдержанность: я не раздражаюсь длинной тирадой об использовании повторяющихся имен. Скажу одно: невразумительно.) Каким образом классы `ObjectInputStream` и `ObjectOutputStream` обращаются к закрытым членам другого класса? Будем считать, что это часть таинства сериализации¹.

Все методы, описанные в интерфейсе, автоматически являются открытыми (`public`), поэтому если методы `writeObject()` и `readObject()` должны быть закрытыми (`private`), они не могут быть частью какого-либо интерфейса. Однако вам необходимо точно соблюдать сигнатуры, и все происходящее очень похоже на реализацию интерфейса.

Судя по всему, при вызове метода `ObjectOutputStream.writeObject()` передаваемый ему объект `Serializable` анализируется (вне всяких сомнений, с использованием механизма отражения), и в нем ищется метод `writeObject()`. Если такой метод существует, процесс стандартной сериализации пропускается и вызывается метод объекта `writeObject()`. Примерно те же действия происходят и при восстановлении объекта.

Существует еще один нюанс. В вашем собственном методе `writeObject()` можно выполнить действие `writeObject()` по умолчанию; для этого вызывается метод `defaultWriteObject()`. Аналогично, в методе `readObject()` можно вызвать метод стандартного восстановления `defaultReadObject()`. В следующем примере показано, как производится пользовательское управление хранением и восстановлением объектов `Serializable`:

```
//: io/SerialCtl.java
// Управление сериализацией с добавлением
// собственных методов writeObject() и readObject().
import java.io.*;

public class SerialCtl implements Serializable {
    private String a;
    private transient String b;
    public SerialCtl(String aa, String bb) {
        a = "Not Transient: " + aa;
        b = "Transient: " + bb;
    }
    public String toString() { return a + "\n" + b; }
    private void writeObject(ObjectOutputStream stream)
    throws IOException {
        stream.defaultWriteObject();
        stream.writeObject(b);
    }
}
```

продолжение ➤

¹ В разделе «Интерфейсы и информация типов» главы 14 показано, как можно обращаться к закрытым методам из-за границ класса.

```

private void readObject(ObjectInputStream stream)
throws IOException, ClassNotFoundException {
    stream.defaultReadObject();
    b = (String)stream.readObject();
}
public static void main(String[] args)
throws IOException, ClassNotFoundException {
    SerialCt1 sc = new SerialCt1("Test1", "Test2");
    System.out.println("Before:\n" + sc);
    ByteArrayOutputStream buf= new ByteArrayOutputStream();
    ObjectOutputStream o = new ObjectOutputStream(buf);
    o.writeObject(sc);
    // Теперь получаем обратно:
    ObjectInputStream in = new ObjectInputStream(
        new ByteArrayInputStream(buf.toByteArray()));
    SerialCt1 sc2 = (SerialCt1)in.readObject();
    System.out.println("After:\n" + sc2);
}
} /* Output:
Before:
Not Transient: Test1
Transient: Test2
After:
Not Transient: Test1
Transient: Test2
*///:~

```

В этом примере одна из строк класса объявлена как `transient`, с целью продемонстрировать, что такие поля при вызове метода `defaultWriteObject()` не сохраняются. Эта строка сохраняется и восстанавливается программой явно. Поля класса инициализируются в конструкторе, а не в точке определения, чтобы показать, что они не инициализируются каким-либо особым механизмом в процессе восстановления.

Если вы собираетесь использовать встроенный механизм сериализации для записи полей объекта (без объявления `transient`), нужно при записи объекта в первую очередь вызвать метод `defaultWriteObject()`, а при восстановлении объекта — метод `defaultReadObject()`. Это вообще загадочные методы. Например, если вы вызовете метод `defaultWriteObject()` для потока `ObjectOutputStream` и не передадите ему аргументов, он все же как-то узнает, какой объект надо записать, где находится ссылка на него и как записать все его не-`transient`, составляющие. Мистика.

При сохранении и восстановлении `transient`-объектов используется более знакомый код. И все же посмотрите, что происходит. В методе `main()` создается объект `SerialCt1`, который затем сериализуется потоком `ObjectOutputStream`. (При этом для вывода используется буфер, а не файл — для потока `ObjectOutputStream` это одно и то же.) Непосредственно сериализация выполняется в следующей строке:

```
o.writeObject(sc);
```

Метод `writeObject()` должен определить, имеется ли в объекте `sc` свой собственный метод `writeObject()`. (При этом проверка на наличие интерфейса или класса не проводится — их нет, — поиск метода проводится с помощью отражения.) Если поиск успешен, найденный метод задействуется в процедуре сериализации. Примерно такой же подход наблюдается и при восстановлении объекта методом `readObject()`. Возможно, решить задачу можно было только так, но все равно это очень странно.

Версии

Иногда необходимо изменить версию сохраненного класса (объекты исходного класса могут храниться, например, в базе данных). Поддержка для этого есть, но делать это следует только в особых ситуациях, и желательно глубокое понимание вопросов сериализации, какого данная книга не предоставляет. Тема подробно рассматривается в документации JDK, которую можно загрузить на сайте <http://java.sun.com>.

Обратите внимание, что в документации JDK многие комментарии начинаются так:

Предупреждение. *Сериализованные объекты этого класса не будут совместимы с последующими выпусками библиотеки Swing. Текущий механизм сериализации годится только для кратковременного хранения или для удаленных вызовов из приложений...*

Все это происходит из-за того, что на данный момент механизм контроля версий недостаточно совершен для эффективной поддержки большинства ситуаций, особенно для хранения компонентов JavaBean. Разработчики языка продолжают улучшать его архитектуру, в чем и состоит смысл таких предупреждений.

Долговременное хранение

Идея привлечь технологию сериализации для того, чтобы сохранить состояние вашей программы и в будущем восстановить его, выглядит соблазнительно. Но прежде, чем делать это, необходимо ответить на несколько вопросов. Что произойдет, если сохранить два объекта, в которых имеется ссылка на некоторый общий третий объект? Когда вы восстановите эти объекты, сколько экземпляров третьего объекта появится в программе? Что, если вы сохраните объекты в отдельных файлах, а затем десериализуете их в разных частях программы?

Следующий пример демонстрирует возможные проблемы:

```
//: io/MyWorld.java
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class House implements Serializable {}

class Animal implements Serializable {
    private String name;
    private House preferredHouse;
    Animal(String nm, House h) {
        name = nm;
        preferredHouse = h;
    }
    public String toString() {
        return name + "[" + super.toString() +
            "], " + preferredHouse + "\n";
    }
}

public class MyWorld {
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
        House house = new House();
```

продолжение ➤

```

List<Animal> animals = new ArrayList<Animal>();
animals.add(new Animal("Bosco the dog", house));
animals.add(new Animal("Ralph the hamster", house));
animals.add(new Animal("Molly the cat", house));
print("animals: " + animals);
ByteArrayOutputStream buf1 =
    new ByteArrayOutputStream();
ObjectOutputStream o1 = new ObjectOutputStream(buf1);
o1.writeObject(animals);
o1.writeObject(animals); // Записываем второй набор
// Запись в другой поток:
ByteArrayOutputStream buf2 =
    new ByteArrayOutputStream();
ObjectOutputStream o2 = new ObjectOutputStream(buf2);
o2.writeObject(animals);
// Теперь получаем объекты обратно:
ObjectInputStream in1 = new ObjectInputStream(
    new ByteArrayInputStream(buf1.toByteArray()));
ObjectInputStream in2 = new ObjectInputStream(
    new ByteArrayInputStream(buf2.toByteArray()));
List
    animals1 = (List)in1.readObject(),
    animals2 = (List)in1.readObject(),
    animals3 = (List)in2.readObject();
print("animals1: " + animals1);
print("animals2: " + animals2);
print("animals3: " + animals3);
}
} /* Output: (Sample)
animals: [Bosco the dog[Animal@adbbf1], House@42e816
, Ralph the hamster[Animal@9304b1], House@42e816
, Molly the cat[Animal@190d11], House@42e816
]
animals1: [Bosco the dog[Animal@de6f34], House@156ee8e
, Ralph the hamster[Animal@47b480], House@156ee8e
, Molly the cat[Animal@19b49e6], House@156ee8e
]
animals2: [Bosco the dog[Animal@de6f34], House@156ee8e
, Ralph the hamster[Animal@47b480], House@156ee8e
, Molly the cat[Animal@19b49e6], House@156ee8e
]
animals3: [Bosco the dog[Animal@10d448], House@e0e1c6
, Ralph the hamster[Animal@6ca1c], House@e0e1c6
, Molly the cat[Animal@1bf216a], House@e0e1c6
]
*///:~

```

В этом примере интересно еще то, что в нем показано, как использовать механизм сериализации и байтовый массив для «глубокого копирования» любого объекта с интерфейсом `Serializable`. (Глубокое копирование — это создание дубликата всего графа объектов, а не просто основного объекта и его ссылок.)

Объекты `Animal` содержат поля типа `House`. В методе `main()` создается список `List` с несколькими объектами `Animal`, его дважды записывают в один поток и еще один раз — в отдельный поток. Когда эти списки восстанавливают и распечатывают, получается

результат, показанный для одного запуска (объекты при каждом запуске программы будут располагаться в различных областях памяти).

Конечно, нет ничего удивительного в том, что восстановленные объекты и их оригиналы имеют разные адреса. Но заметьте, что адреса в восстановленных объектах `animals1` и `animals2` совпадают, вплоть до повторения ссылок на объект `House`, разделяемый обоими списками. С другой стороны, при восстановлении списка `animals3` система не имеет представления о том, что находящиеся в них объекты уже были восстановлены и имеются в программе, поэтому она создает новую сеть объектов.

Если все данные сериализуются в один выходной поток, у вас есть гарантия того, что сохраненная вами сеть объектов затем полностью восстановится в первоначальном виде, без лишних дубликатов объектов. Конечно, состояние объекта может измениться между первой и последней записью, но это уже на вашей совести — сохраненные объекты останутся в том состоянии, в котором вы их записали (с теми связями, что у них были на момент сериализации).

Если уж необходимо зафиксировать состояние системы, безопаснее всего сделать это в рамках «атомарной» операции. Если вы сохраняете что-то, затем выполняете какие-то действия, снова сохраняете данные и т. д., вам не удастся безопасно зафиксировать состояние системы. Вместо этого следует поместить все объекты, образующие состояние системы, в контейнер и сохранить этот контейнер *единой* операцией. После можно восстановить его вызовом одного метода.

Следующий пример — имитатор системы автоматизированного проектирования (CAD), в котором используется такой подход. Вдобавок в нем сохраняются статические (`static`) поля — если вы взглянете на документацию JDK, то увидите, что класс `Class` реализует интерфейс `Serializable`, поэтому для сохранения статических данных достаточно сохранить объект `Class`. Это достаточно разумное решение.

```
//: io/StoreCADState.java
// Сохранение и восстановление состояния вымышленной системы.
import java.io.*;
import java.util.*;
```

```
abstract class Shape implements Serializable {
    public static final int RED = 1, BLUE = 2, GREEN = 3;
    private int xPos, yPos, dimension;
    private static Random rand = new Random(47);
    private static int counter = 0;
    public abstract void setColor(int newColor);
    public abstract int getColor();
    public Shape(int xVal, int yVal, int dim) {
        xPos = xVal;
        yPos = yVal;
        dimension = dim;
    }
    public String toString() {
        return getClass() +
            "color[" + getColor() + "] xPos[" + xPos +
            "] yPos[" + yPos + "] dim[" + dimension + "]\n";
    }
    public static Shape randomFactory() {
        int xVal = rand.nextInt(100);
```

продолжение ➞

```

    int yVal = rand.nextInt(100);
    int dim = rand.nextInt(100);
    switch(counter++ % 3) {
        default:
        case 0: return new Circle(xVal, yVal, dim);
        case 1: return new Square(xVal, yVal, dim);
        case 2: return new Line(xVal, yVal, dim);
    }
}
}

class Circle extends Shape {
    private static int color = RED;
    public Circle(int xVal, int yVal, int dim) {
        super(xVal, yVal, dim);
    }
    public void setColor(int newColor) { color = newColor; }
    public int getColor() { return color; }
}

class Square extends Shape {
    private static int color;
    public Square(int xVal, int yVal, int dim) {
        super(xVal, yVal, dim);
        color = RED;
    }
    public void setColor(int newColor) { color = newColor; }
    public int getColor() { return color; }
}

class Line extends Shape {
    private static int color = RED;
    public static void
    serializeStaticState(ObjectOutputStream os)
    throws IOException { os.writeInt(color); }
    public static void
    deserializeStaticState(ObjectInputStream os)
    throws IOException { color = os.readInt(); }
    public Line(int xVal, int yVal, int dim) {
        super(xVal, yVal, dim);
    }
    public void setColor(int newColor) { color = newColor; }
    public int getColor() { return color; }
}

public class StoreCADState {
    public static void main(String[] args) throws Exception {
        List<Class<? extends Shape>> shapeTypes =
            new ArrayList<Class<? extends Shape>>();
        // Добавляем ссылки на объекты Class:
        shapeTypes.add(Circle.class);
        shapeTypes.add(Square.class);
        shapeTypes.add(Line.class);
        List<Shape> shapes = new ArrayList<Shape>();
        // Создаем несколько фигур:
        for(int i = 0; i < 10; i++)
            shapes.add(Shape.randomFactory());
        // Устанавливаем все статические цвета в зеленый (GREEN):

```

```

    for(int i = 0; i < 10; i++)
        ((Shape)shapes.get(i)).setColor(Shape.GREEN);
    // Сохраняем вектор состояния:
    ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("CADState.out"));
    out.writeObject(shapeTypes);
    Line.serializeStaticState(out);
    out.writeObject(shapes);
    // Отображение набора фигур:
    System.out.println(shapes);
}
} /* Output:
[class Circlecolor[3] xPos[58] yPos[55] dim[93]
, class Squarecolor[3] xPos[61] yPos[61] dim[29]
, class Linecolor[3] xPos[68] yPos[0] dim[22]
, class Circlecolor[3] xPos[7] yPos[88] dim[28]
, class Squarecolor[3] xPos[51] yPos[89] dim[9]
, class Linecolor[3] xPos[78] yPos[98] dim[61]
, class Circlecolor[3] xPos[20] yPos[58] dim[16]
, class Squarecolor[3] xPos[40] yPos[11] dim[22]
, class Linecolor[3] xPos[4] yPos[83] dim[6]
, class Circlecolor[3] xPos[75] yPos[10] dim[42]
]
*///:~

```

Класс `Shape` реализует интерфейс `Serializable`, поэтому все унаследованные от него классы по определению реализуют `Serializable`. В каждой фигуре `Shape` содержатся некоторые данные, и в каждом унаследованном от `Shape` классе имеется статическое (`static`) поле, которое определяет цвет фигуры. (Если бы мы поместили статическое поле в базовый класс, то получили бы одно поле для всех фигур, поскольку статические поля в производных классах не копируются.) Методы базового класса можно переопределить, чтобы установить цвет фигуры (статические методы не используют динамическое связывание, поэтому они обычные, а не абстрактные). Метод `randomFactory()` создает при каждом вызове новую фигуру, используя случайные значения для данных `Shape`.

Классы `Circle` и `Square` — простые подклассы фигуры `Shape`; разница между ними только в способе инициализации поля `color`: окружность (`Circle`) задает значение этого поля в месте определения, а прямоугольник (`Square`) инициализирует его в конструкторе. Класс `Line` мы обсудим чуть позже.

В методе `main()` один список `ArrayList` используется для хранения объектов `Class`, а другой — для хранения фигур.

Восстановление объектов выполняется очень просто:

```

//: io/RecoverCADState.java
// Restoring the state of the pretend CAD system.
// {RunFirst: StoreCADState}
import java.io.*;
import java.util.*;

public class RecoverCADState {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) throws Exception {

```

продолжение ➤

```

ObjectInputStream in = new ObjectInputStream(
    new FileInputStream("CADState.out"));
// Читаем объекты в том порядке, в котором они были записаны:
List<Class<? extends Shape>> shapeTypes =
    (List<Class<? extends Shape>>)in.readObject();
Line.deserializeStaticState(in);
List<Shape> shapes = (List<Shape>)in.readObject();
System.out.println(shapes);
}
} /* Output:
[class Circlecolor[1] xPos[58] yPos[55] dim[93]
, class Squarecolor[0] xPos[61] yPos[61] dim[29]
, class Linecolor[3] xPos[68] yPos[0] dim[22]
, class Circlecolor[1] xPos[7] yPos[88] dim[28]
, class Squarecolor[0] xPos[51] yPos[89] dim[9]
, class Linecolor[3] xPos[78] yPos[98] dim[61]
, class Circlecolor[1] xPos[20] yPos[58] dim[16]
, class Squarecolor[0] xPos[40] yPos[11] dim[22]
, class Linecolor[3] xPos[4] yPos[83] dim[6]
, class Circlecolor[1] xPos[75] yPos[10] dim[42]
]
*///:~

```

Нетрудно заметить, что значения переменных `xPos`, `yPos` и `dim` сохранились и были успешно восстановлены, однако при восстановлении статической информации произошло что-то странное. При записи все статические поля `color` имели значение 3, но восстановление дало другие результаты. В окружностях значением стала единица (согласно определению константа `RED`), а в прямоугольниках поля `color` вообще равны нулю (помните, в этих объектах инициализация проходит в конструкторе). Такое впечатление, что статические поля вообще не сериализовались! Да, это именно так — хотя класс `Class` и реализует интерфейс `Serializable`, происходит это не так, как нам хотелось бы. Итак, если вам понадобится сохранить статические значения, делайте это самостоятельно.

Именно для этой цели предназначены методы `serializeStaticState()` и `deserializeStaticState()` класса `Line`. Вы можете видеть, как они вызываются в процессе сохранения и восстановления системы. (Заметьте, порядок действий при сохранении информации должен соблюдаться и при ее десериализации.) Поэтому для правильного выполнения программы `CADState.java` необходимо сделать следующее.

1. Добавить статические методы `serializeStaticState()` и `deserializeStaticState()` во все фигуры `Shape`.
2. Убрать из программы список `shapeTypes` и весь код, относящийся к нему.
3. При сериализации и восстановлении вызывать новые методы для сохранения статической информации.

Также стоит учесть аспект безопасности — ведь сериализация сохраняет и закрытые (`private`) поля. Если в вашем объекте имеются поля с конфиденциальной информацией, ее необходимо пометить как `transient`. Но в таком случае придется подумать о безопасном способе хранения такой информации, ведь при восстановлении объекта необходимо восстанавливать все его данные.

30. (1) Внесите в программу `CADState.java` изменения, описанные в тексте.

XML

У механизма сериализации объектов есть одно важное ограничение: он работает только с объектами Java, то есть десериализовать такие объекты можно только в программах Java. Более универсальное решение заключается в преобразовании данных в формат XML, чтобы их можно было использовать на разных платформах и языках.

Из-за популярности этой технологии существует много вариантов программирования XML, включая библиотеки `javax.xml.*`, входящие в поставку JDK. Я решил использовать свободно распространяемую библиотеку ХОМ Эллиота Расти Харольда (библиотеку и документацию можно загрузить на сайте www.xom.nu), потому что это самый простой и прямолинейный способ создания и модификации XML в языке Java. Кроме того, ХОМ уделяет особое внимание корректности данных XML.

Предположим, у вас имеются объекты `Person` с именами и фамилиями, которые вам хотелось бы сериализовать в XML. Приведенный ниже класс `Person` содержит метод `getXML()`, который использует ХОМ для преобразования данных `Person` в объект XML `Element`, и конструктор, который получает `Element` и извлекает данные `Person` (обратите внимание: примеры XML находятся в отдельном подкаталоге):

```
//: xml/Person.java
// Использование библиотеки ХОМ для записи и чтения XML
// {Requires: nu.xom.Node; установите библиотеку
// ХОМ с сайта http://www.xom.nu }
import nu.xom.*;
import java.io.*;
import java.util.*;

public class Person {
    private String first, last;
    public Person(String first, String last) {
        this.first = first;
        this.last = last;
    }
    // Создание объекта XML Element по объекту Person:
    public Element getXML() {
        Element person = new Element("person");
        Element firstName = new Element("first");
        firstName.appendChild(first);
        Element lastName = new Element("last");
        lastName.appendChild(last);
        person.appendChild(firstName);
        person.appendChild(lastName);
        return person;
    }
    // Конструктор для восстановления Person из объекта XML Element:
    public Person(Element person) {
        first= person.getFirstChildElement("first").getValue();
        last = person.getFirstChildElement("last").getValue();
    }
    public String toString() { return first + " " + last; }
    // Преобразование в удобочитаемую форму:
    public static void
    format(OutputStream os, Document doc) throws Exception {
```

продолжение ➤

```

    Serializer serializer= new Serializer(os,"ISO-8859-1");
    serializer.setIndent(4);
    serializer.setMaxLength(60);
    serializer.write(doc);
    serializer.flush();
}
public static void main(String[] args) throws Exception {
    List<Person> people = Arrays.asList(
        new Person("Dr. Bunsen", "Honeydew"),
        new Person("Gonzo", "The Great"),
        new Person("Phillip J.", "Fry"));
    System.out.println(people);
    Element root = new Element("people");
    for(Person p : people)
        root.appendChild(p.getXML());
    Document doc = new Document(root);
    format(System.out, doc);
    format(new BufferedOutputStream(new FileOutputStream(
        "People.xml")), doc);
}
} /* Output:
[Dr. Bunsen Honeydew, Gonzo The Great, Phillip J. Fry]
<?xml version="1.0" encoding="ISO-8859-1"?>
<people>
  <person>
    <first>Dr. Bunsen</first>
    <last>Honeydew</last>
  </person>
  <person>
    <first>Gonzo</first>
    <last>The Great</last>
  </person>
  <person>
    <first>Phillip J.</first>
    <last>Fry</last>
  </person>
</people>
*///:~

```

Смысл методов XOM достаточно очевиден, а их описания можно найти в документации XOM.

XOM также содержит класс `Serializer`, который используется в методе `format()` для преобразования XML в удобочитаемую форму. Чтобы все заработало, достаточно вызвать `toXML()`, так что `Serializer` — весьма удобный инструмент.

Десериализация объектов `Person` из файла XML тоже выполняется очень просто:

```

//: xml/People.java
// {Requires: nu.xom.Node; установите библиотеку
// XOM с сайта http://www.xom.nu }
// {RunFirst: Person}
import nu.xom.*;
import java.util.*;

public class People extends ArrayList<Person> {
    public People(String fileName) throws Exception {
        Document doc = new Builder().build(fileName);

```



```
Elements elements =
    doc.getRootElement().getChildElements();
for(int i = 0; i < elements.size(); i++)
    add(new Person(elements.get(i)));
}
public static void main(String[] args) throws Exception {
    People p = new People("People.xml");
    System.out.println(p);
}
} /* Output:
[Dr. Bunsen Honeydew, Gonzo The Great, Phillip J. Fry]
*///:~
```

Конструктор `People` открывает и читает файл с использованием метода `XOM Builder.build()`, после чего метод `getChildElements()` создает список `Elements` (не стандартный контейнер `Java List`, а объект, содержащий только методы `size()` и `get()`, — Харольд не хотел принуждать разработчиков использовать `Java SE5`, но при этом желал иметь контейнер, безопасный по отношению к типам). Каждый объект `Element` в этом списке представляет объект `Person`, поэтому он передается второму конструктору `Person`. Для этого необходимо заранее знать точную структуру файла XML, но это типично для подобных задач. Если структура данных не совпадает с ожидаемой, `XOM` выдает исключение. Также можно написать более сложный код, который будет анализировать документ XML вместо того, чтобы строить какие-то предположения о нем, — такое решение используется, если у вас не хватает конкретной информации о структуре входного потока XML.

Чтобы эти примеры компилировались, необходимо разместить файлы JAR из поставки `XOM` в каталоге, входящем в `CLASSPATH`.

За дополнительной информацией о работе с XML в Java и использовании библиотеки `XOM` обращайтесь на сайт www.xom.nu.

31. (2) Добавьте адресную информацию в примеры `Person.java` и `People.java`.
32. (4) Используя контейнер `Map<String,Integer>` и класс `net.mindview.util.TextFile`, напишите программу для подсчета вхождений слов в файл (передайте `"\\w+"` во втором аргументе конструктора `TextFile`). Сохраните результаты в файле XML.

Предпочтения

В пакете `JDK 1.4` появился программный интерфейс `Preferences API` для работы с *предпочтениями* (`preferences`). Предпочтения гораздо более тесно связаны с долгосрочным хранением, чем механизм сериализации объектов, поскольку они позволяют автоматически сохранять и восстанавливать вашу информацию. Однако их использование сдерживается маленькими и ограниченными наборами данных — хранить в них можно только примитивы и строки, и длина строки не должна превышать 8 Кбайт (не то чтобы очень мало, но делать с этим что-то серьезное вряд ли стоит). Как и предполагает название нового API, предпочтения предназначены для хранения и получения информации о предпочтениях пользователя и конфигурации программы.

Предпочтения — это набор пар «ключ-значение» (как в карте), образующих иерархию узлов. Хотя иерархия узлов и годится для построения сложных структур, чаще всего

создают один узел, названный так же, как и класс, и хранят информацию в нем. Вот простой пример:

```
//: io/PreferencesDemo.java
import java.util.prefs.*;
import static net.mindview.util.Print.*;

public class PreferencesDemo {
    public static void main(String[] args) throws Exception {
        Preferences prefs = Preferences
            .userNodeForPackage(PreferencesDemo.class);
        prefs.put("Location", "Oz");
        prefs.put("Footwear", "Ruby Slippers");
        prefs.putInt("Companions", 4);
        prefs.putBoolean("Are there witches?", true);
        int usageCount = prefs.getInt("UsageCount", 0);
        usageCount++;
        prefs.putInt("UsageCount", usageCount);
        for(String key : prefs.keys())
            print(key + ": " + prefs.get(key, null));
        // Вы всегда должны предоставлять значение по умолчанию:
        print("How many companions does Dorothy have? " +
            prefs.getInt("Companions", 0));
    }
} /* Output: (Sample)
Location: Oz
Footwear: Ruby Slippers
Companions: 4
Are there witches?: true
UsageCount: 53
How many companions does Dorothy have? 4
*///:~
```

Здесь вызывался метод `userNodeForPackage()`, но с тем же успехом можно было бы заменить его методом `systemNodeForPackage()`; это дело вкуса, хотя префикс `user` обычно используется для хранения индивидуальных предпочтений пользователя, а `system` — для хранения информации общего плана о настройках установки. Так как метод `main()` является статическим, для идентификации узла применен класс `PreferencesDemo.class`, хотя в нестатических методах вы обычно будете вызывать метод `getClass()`. Использовать текущий класс для идентификации узла не обязательно, но чаще всего именно так и поступают.

Как только вы создали узел, он становится пригодным для хранения или считывания информации. В данном примере в узел помещаются различные данные, после вызывается метод `keys()`. Последний возвращает массив строк `String[]`, что может быть непригодно, если вы вспомните, как работает метод `keys()` в коллекциях. Здесь этот массив преобразуется в список (`List`), с помощью которого мы получаем итератор (`Iterator`), который и выводит на печать ключи и значения. Обратите внимание на второй аргумент метода `get()`. Это значение по умолчанию, которое будет возвращено, если для данного ключа не будет найдено значение. При переборе множества ключей каждому из них заведомо сопоставлено значение, поэтому использование `null` как значения по умолчанию безопасно, но чаще выборка осуществляется по именованному ключу:

```
prefs.getInt("Companions", 0));
```

В таких ситуациях стоит предоставить осмысленное значение по умолчанию. Типичная идиома выглядит так:

```
int usageCount = prefs.getInt("UsageCount", 0);
usageCount++;
prefs.putInt("UsageCount", usageCount);
```

В этом случае при первом запуске программы значение `usageCount` равно нулю, а при последующих запусках оно должно измениться.

Когда вы запустите программу `PreferencesDemo.java`, вы увидите, что значение `usageCount` действительно увеличивается при каждом запуске программы. Но где же хранятся данные? После первого запуска программы не появляется никаких локальных файлов. Система предпочтений привлекает для хранения данных подходящие системные возможности, которые в различных операционных системах различны. В Windows используется реестр (поскольку он и так представляет собой иерархию узлов с набором пар «ключ-значение»). Вся суть заключается в том, что информация сохраняется автоматически и вам не приходится беспокоиться о том, как это работает в различных системах.

В программном интерфейсе предпочтений гораздо больше возможностей, чем было здесь показано. За более подробным описанием обратитесь к документации JDK, эта тема там изложена достаточно подробно.

- 33. (2)** Напишите программу, которая выводит текущее значение, связанное с каталогом, и запрашивает новое значение. Используйте API предпочтений для хранения значения.

Резюме

Библиотека ввода-вывода Java подходит для выполнения основных операций: она позволяет проводить чтение данных и запись их на консоль, в файл, в буфер памяти, даже в сетевое соединение Интернета. Применяя наследование, можно создавать новые типы объектов для ввода и вывода данных. Вы даже можете расширить виды объектов, принимаемых потоком, переопределяя метод `toString()`, который автоматически вызывается при передаче объекта методу, ожидающему получить строку (`String`) (ограниченное «автоматическое преобразование типов» Java).

Но есть и вопросы, на которые вы не найдете ответов в интерактивной документации и библиотеке ввода-вывода Java. Например, было бы замечательно, если бы при попытке перезаписи файла возбуждалось исключение — многие программные системы позволяют указать, что файл должен открываться для вывода только в том случае, если файла с таким именем еще не существует. В языке Java проверка существования файла возможна лишь с помощью объекта `File`, поскольку если вы откроете файл как `FileOutputStream` или `FileWriter`, старый файл всегда будет перезаписан.

Библиотека ввода-вывода вызывает смешанные чувства; с одной стороны, она берет на себя значительный объем работы и обеспечивает переносимость между платформами. Но если вы не вполне понимаете суть работы паттерна «Декоратор», архитектура

библиотеки будет оставаться малопонятной, и вам придется приложить немалые усилия для ее изучения и освоения. Кроме того, библиотеку не назовешь полной: например, только отсутствие необходимых средств заставило меня писать инструменты, подобные `TextFile` (новый класс `Java SE PrintWriter` является шагом в правильном направлении, но это лишь частичное решение). В `Java SE5` было добавлено еще одно серьезное усовершенствование: механизм форматирования вывода, поддерживаемый практически во всех остальных языках.

Впрочем, как только вы *действительно* проникаетесь идеей паттерна «Декоратор» и начинаете использовать библиотеку в ситуациях, где требуется вся ее гибкость, она начинает приносить все большую пользу, и несколько лишних строк кода покажутся сущей мелочью.

19

Перечислимые типы

Ключевое слово `enum` создает новый тип с ограниченным набором именованных значений, и работать с этими значениями можно как с обычными компонентами программы. Данная возможность иногда оказывается чрезвычайно полезной¹.

Перечисления (enumerations) были кратко представлены в конце главы 5. Теперь, когда вы более глубоко понимаете Java, мы можем поближе познакомиться с перечислениями Java SE5. Вы увидите, что перечисления открывают ряд очень интересных возможностей, но эта глава также поможет лучше понять другие возможности языка, которые уже рассматривались ранее, — такие, как обобщения и отражение. Также мы рассмотрим несколько новых паттернов проектирования.

Основные возможности перечислений

Как показано в главе 5, вы можете получить список констант перечисления, вызвав для `enum` метод `values()`. Метод `values()` создает массив констант перечисления, следующих в порядке их объявления, так что полученный массив может использоваться (например) в цикле `foreach`.

При создании перечисления компилятор генерирует соответствующий класс. Этот класс автоматически наследует от класса `java.lang.Enum`; некоторые полезные возможности базового класса продемонстрированы в следующем примере:

```
//: enumerated/EnumClass.java
// Возможности класса Enum
import static net.mindview.util.Print.*;

enum Shrubbery { GROUND, CRAWLING, HANGING }

public class EnumClass {
    public static void main(String[] args) {
```

продолжение ➤

¹ Джошуа Блош оказал огромную помощь в написании этой главы.

```

for(Shrubbery s : Shrubbery.values()) {
    print(s + " ordinal: " + s.ordinal());
    printnb(s.compareTo(Shrubbery.CRAWLING) + " ");
    printnb(s.equals(Shrubbery.CRAWLING) + " ");
    print(s == Shrubbery.CRAWLING);
    print(s.getDeclaringClass());
    print(s.name());
    print("-----");
}
// Получить значение из перечисления по строковому имени:
for(String s : "HANGING CRAWLING GROUND".split(" ")) {
    Shrubbery shrub = Enum.valueOf(Shrubbery.class, s);
    print(shrub);
}
}
} /* Output:
GROUND ordinal: 0
-1 false false
class Shrubbery
GROUND
-----
CRAWLING ordinal: 1
0 true true
class Shrubbery
CRAWLING
-----
HANGING ordinal: 2
1 false false
class Shrubbery
HANGING
-----
HANGING
CRAWLING
GROUND
*///:~

```

Метод `ordinal()` возвращает значение `int`, определяющее порядок объявления экземпляров перечисления начиная с нуля. Экземпляры всегда можно безопасно сравнивать с оператором `==`, а методы `equals()` и `hashCode()` автоматически создаются за вас. Класс `Enum` реализует интерфейс `Comparable`, поэтому в нем присутствует метод `compareTo()`; кроме того, реализуется интерфейс `Serializable`.

Вызывая метод `getDeclaringClass()` для экземпляра перечисления, вы узнаете вмещающий класс `enum`.

Метод `name()` возвращает имя точно в таком виде, в каком оно было объявлено; эту же информацию возвращает метод `toString()`. Статический метод `valueOf()` возвращает экземпляр перечисления, соответствующий переданному строковому имени, или возбуждает исключение при отсутствии подходящего экземпляра.

Статическое импортирование и перечисления

Рассмотрим слегка измененную версию примера `Burrito.java` из главы 5:

```

//: enumerated/Spiciness.java
package enumerated;

```

```
public enum Spiciness {
    NOT, MILD, MEDIUM, HOT, FLAMING
} ///:~

//: enumerated/Burrito.java
package enumerated;
import static enumerated.Spiciness.*;

public class Burrito {
    Spiciness degree;
    public Burrito(Spiciness degree) { this.degree = degree;}
    public String toString() { return "Burrito is " + degree;}
    public static void main(String[] args) {
        System.out.println(new Burrito(NOT));
        System.out.println(new Burrito(MEDIUM));
        System.out.println(new Burrito(HOT));
    }
} /* Output:
Burrito is NOT
Burrito is MEDIUM
Burrito is HOT
*///:~
```

Директива `import static` вводит все идентификаторы экземпляров перечисления в локальное пространство имен, чтобы они не требовали уточнения. А может, лучше явно уточнять все экземпляры перечисления? Вероятно, это зависит от сложности кода. Компилятор не позволит использовать неправильный тип, так что вопрос лишь в сложности чтения кода. Вероятно, во многих ситуациях упрощенная запись не создаст проблем, однако решение следует принимать в каждом конкретном случае.

Обратите внимание: этот прием не может использоваться в том случае, если перечисление определяется в том же файле или в пакете по умолчанию.

Добавление методов к перечислению

Перечисление не может использоваться при наследовании, но в остальном оно работает как обычный класс. Это означает, что в перечисление можно добавлять новые методы. Более того, перечисление даже может содержать метод `main()`.

Допустим, вы решили определить для перечисления другое описание — вместо реализации `toString()` по умолчанию, которая просто выдает имя экземпляра перечисления. Для этого можно предоставить конструктор для сохранения расширенной информации, а также дополнительные методы для передачи расширенного описания:

```
//: enumerated/OzWitch.java
// Ведьмы из страны Оз.
import static net.mindview.util.Print.*;

public enum OzWitch {
    // Определения экземпляров должны предшествовать
    // определениям методов:
    WEST("Miss Gulch, aka the Wicked Witch of the West"),
    NORTH("Glinda, the Good Witch of the North"),
```

продолжение ➤

```

EAST("Wicked Witch of the East, wearer of the Ruby " +
    "Slippers, crushed by Dorothy's house"),
SOUTH("Good by inference, but missing");
private String description;
// Конструктор должен быть объявлен закрытым
// или с доступом уровня пакета:
private OzWitch(String description) {
    this.description = description;
}
public String getDescription() { return description; }
public static void main(String[] args) {
    for(OzWitch witch : OzWitch.values())
        print(witch + ": " + witch.getDescription());
}
} /* Output:
WEST: Miss Gulch, aka the Wicked Witch of the West
NORTH: Glinda, the Good Witch of the North
EAST: Wicked Witch of the East, wearer of the Ruby
Slippers, crushed by Dorothy's house
SOUTH: Good by inference, but missing
*///:~

```

Обратите внимание: если вы собираетесь определять методы, то последовательность экземпляров перечисления должна завершаться символом «;». Кроме того, по правилам Java перечисление должно начинаться с определения экземпляров. При попытке определить их после методов или полей компилятор выдает ошибку.

Конструктор и методы строятся по той же схеме, что и в обычном классе, потому что перечисление (с некоторыми ограничениями) и является обычным классом. Таким образом, с перечислениями можно делать практически все, что угодно, хотя чаще они остаются относительно простыми.

В приведенном примере конструктор объявлен закрытым, но уровень доступа мало на что влияет — конструктор может использоваться только для создания экземпляров, объявленных в определении перечисления; после завершения определения перечисления компилятор не позволит создавать новые экземпляры.

Переопределение методов перечисления

Также к созданию нестандартных строковых значений для перечислений можно подойти иначе. В следующем примере будут использоваться имена экземпляров, отформатированные для удобства чтения. Переопределение метода `toString()` для перечисления ничем не отличается от его переопределения для обычного класса:

```

//: enumerated/SpaceShip.java
public enum SpaceShip {
    SCOUT, CARGO, TRANSPORT, CRUISER, BATTLESHIP, MOTHERSHIP;
    public String toString() {
        String id = name();
        String lower = id.substring(1).toLowerCase();
        return id.charAt(0) + lower;
    }
    public static void main(String[] args) {
        for(SpaceShip s : values()) {

```



```

        System.out.println(s);
    }
}
} /* Output:
Scout
Cargo
Transport
Cruiser
Battleship
Mothership
*///:~

```

Метод `toString()` получает имя объекта `SpaceShip`, вызывая метод `name()`, и изменяет результат так, чтобы прописной была только первая буква.

Перечисления в командах switch

Одна из самых удобных возможностей перечисления связана с их использованием в командах `switch`. Обычно `switch` работает только с целочисленными значениями, но перечисления имеют четко определенный целочисленный порядок, а очередность экземпляров может быть получена методом `ordinal()` (очевидно, компилятор проделывает нечто подобное), поэтому перечисления могут использоваться в командах `switch`.

Хотя обычно экземпляр перечисления должен уточняться его типом, в секциях `case` это не обязательно. В следующем примере перечисление используется для создания простейшего конечного автомата:

```

//: enumerated/TrafficLight.java
// Перечисления в командах switch.
import static net.mindview.util.Print.*;

// Определение перечислимого типа:
enum Signal { GREEN, YELLOW, RED, }

public class TrafficLight {
    Signal color = Signal.RED;
    public void change() {
        switch(color) {
            // Обратите внимание: в секции case
            // не обязательно использовать запись Signal.RED:
            case RED:    color = Signal.GREEN;
                        break;
            case GREEN: color = Signal.YELLOW;
                        break;
            case YELLOW: color = Signal.RED;
                        break;
        }
    }
    public String toString() {
        return "The traffic light is " + color;
    }
    public static void main(String[] args) {
        TrafficLight t = new TrafficLight();
    }
}

```

продолжение ➤

```

    for(int i = 0; i < 7; i++) {
        print(t);
        t.change();
    }
}
} /* Output:
The traffic light is RED
The traffic light is GREEN
The traffic light is YELLOW
The traffic light is RED
The traffic light is GREEN
The traffic light is YELLOW
The traffic light is RED
*///:~

```

Компилятор не жалуется на отсутствие секции `default` в команде `switch`, но не потому, что команда содержит секцию `case` для каждого экземпляра `Signal`. Если закомментировать одну из секций `case`, он все равно жаловаться не будет. А это означает, что вы должны быть внимательны и следить за тем, чтобы в команде были учтены все возможные случаи. С другой стороны, при вызове `return` в секциях `case` компилятор будет жаловаться на отсутствие `default` — даже если в команде учтены все возможные значения перечисления.

1. (2) Используйте директиву статического импортирования и измените пример `TrafficLight.java` так, чтобы экземпляры перечисления могли использоваться без уточнения.

Странности `values()`

Как упоминалось ранее, все классы перечислений создаются за вас компилятором и расширяют класс `Enum`. Но если присмотреться к классу `Enum`, вы увидите, что метода `values()` в нем нет, хотя мы его и использовали. Нет ли других «скрытых» методов? Чтобы получить ответ на этот вопрос, можно написать маленькую программу с использованием отражения:

```

//: enumerated/Reflection.java
// Анализ перечислений с использованием отражения.
import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

enum Explore { HERE, THERE }

public class Reflection {
    public static Set<String> analyze(Class<?> enumClass) {
        print("----- Analyzing " + enumClass + " -----");
        print("Interfaces:");
        for(Type t : enumClass.getGenericInterfaces())
            print(t);
        print("Base: " + enumClass.getSuperclass());
        print("Methods: ");
        Set<String> methods = new TreeSet<String>();
        for(Method m : enumClass.getMethods())

```

```

        methods.add(m.getName());
    print(methods);
    return methods;
}
public static void main(String[] args) {
    Set<String> exploreMethods = analyze(Explore.class);
    Set<String> enumMethods = analyze(Enum.class);
    print("Explore.containsAll(Enum)? " +
        exploreMethods.containsAll(enumMethods));
    printnb("Explore.removeAll(Enum): ");
    exploreMethods.removeAll(enumMethods);
    print(exploreMethods);
    // Decompile the code for the enum:
    OSExecute.command("javap Explore");
}
} /* Output:
----- Analyzing class Explore -----
Interfaces:
Base: class java.lang.Enum
Methods:
[compareTo, equals, getClass, getDeclaringClass, hashCode,
name, notify, notifyAll, ordinal, toString, valueOf,
values, wait]
----- Analyzing class java.lang.Enum -----
Interfaces:
java.lang.Comparable<E>
interface java.io.Serializable
Base: class java.lang.Object
Methods:
[compareTo, equals, getClass, getDeclaringClass, hashCode,
name, notify, notifyAll, ordinal, toString, valueOf, wait]
Explore.containsAll(Enum)? true
Explore.removeAll(Enum): [values]
Compiled from "Reflection.java"
final class Explore extends java.lang.Enum{
    public static final Explore HERE;
    public static final Explore THERE;
    public static final Explore[] values();
    public static Explore valueOf(java.lang.String);
    static {};
}
*///:~

```

Итак, `values()` — статический метод, добавляемый компилятором. Мы видим, что в процессе создания перечисления в `Explore` также добавляется метод `valueOf()`. Это создает некоторую путаницу, потому что метод `valueOf()` также присутствует в классе `Enum`, но этот метод имеет два аргумента, а добавленный метод — только один. Однако использованный метод `Set` учитывает только имена методов, а не сигнатуры, так что после вызова `Explore.removeAll(Enum)` остается только `[values]`.

Из результатов видно, что класс `Explore` объявляется компилятором как `final`, поэтому он не может использоваться для создания производных классов. Также имеется статическая секция инициализации, которая, как вы вскоре увидите, может переопределяться.

Из-за стирания (см. главу 15) декомпилятор не располагает полной информацией об `Enum`, поэтому в качестве базового класса `Explore` указывается обобщение `Enum` (вместо специализации `Enum<Explore>`).

Так как `values()` является статическим методом, который вставляется в определение перечисления компилятором, при восходящем преобразовании перечислимого типа к `Enum` метод `values()` будет недоступен. Однако `Class` содержит метод `getEnumConstants()`, так что даже при отсутствии `values()` в интерфейсе `Enum` вы все равно сможете получить экземпляры перечисления через объект `Class`:

```
//: enumerated/UpcastEnum.java
// При восходящем преобразовании перечисления
// метод values() недоступен.

enum Search { HITHER, YON }

public class UpcastEnum {
    public static void main(String[] args) {
        Search[] vals = Search.values();
        Enum e = Search.HITHER; // Восходящее преобразование
        // e.values();           // Enum не содержит метода values()
        for(Enum en : e.getClass().getEnumConstants())
            System.out.println(en);
    }
} /* Output:
HITHER
YON
*///:~
```

Так как `getEnumConstants()` является методом `Class`, его можно вызвать для класса, не являющегося перечислением:

```
//: enumerated/NonEnum.java

public class NonEnum {
    public static void main(String[] args) {
        Class<Integer> intClass = Integer.class;
        try {
            for(Object en : intClass.getEnumConstants())
                System.out.println(en);
        } catch(Exception e) {
            System.out.println(e);
        }
    }
} /* Output:
java.lang.NullPointerException
*///:~
```

Однако метод возвращает `null`, так что при попытке использования результата возбуждается исключение.

Реализация, а не наследование

Мы установили, что все перечисления расширяют `java.lang.Enum`. Так как Java не поддерживает множественное наследование, это означает, что перечисление не может быть создано посредством наследования:

```
enum NotPossible extends Pet { ... // Не работает
```

Однако можно создать перечисление, реализующее один или несколько интерфейсов:

```
//: enumerated/cartoons/EnumImplementation.java
// Перечисление может реализовать интерфейс
package enumerated.cartoons;
import java.util.*;
import net.mindview.util.*;
enum CartoonCharacter
implements Generator<CartoonCharacter> {
    SLAPPY, SPANKY, PUNCHY, SILLY, BOUNCY, NUTTY, BOB;
    private Random rand = new Random(47);
    public CartoonCharacter next() {
        return values()[rand.nextInt(values().length)];
    }
}

public class EnumImplementation {
    public static <T> void printNext(Generator<T> rg) {
        System.out.print(rg.next() + ", ");
    }
    public static void main(String[] args) {
        // Выберите любой экземпляр:
        CartoonCharacter cc = CartoonCharacter.BOB;
        for(int i = 0; i < 10; i++)
            printNext(cc);
    }
} /* Output:
BOB, PUNCHY, BOB, SPANKY, NUTTY, PUNCHY, SLAPPY, NUTTY,
NUTTY, SLAPPY,
*///:~
```

Результат выглядит немного странно, потому что для вызова метода необходимо иметь экземпляр `enum`, для которого он вызывается. Однако `CartoonCharacter` теперь может приниматься любым методом, получающим `Generator`, например `printNext()`.

2. (2) Вместо того чтобы реализовать интерфейс, сделайте `next()` статическим методом. Какими преимуществами и недостатками обладает это решение?

Случайный выбор

Во многих примерах этой главы (как, например, в `CartoonCharacter.next()`) используется случайный выбор из экземпляров перечисления. Эту задачу можно перевести на более общий уровень с использованием обобщений и поместить результат в библиотеку:

```
//: net/mindview/util/Enums.java
package net.mindview.util;
import java.util.*;

public class Enums {
    private static Random rand = new Random(47);
    public static <T extends Enum<T>> T random(Class<T> ec) {
        return random(ec.getEnumConstants());
    }
    public static <T> T random(T[] values) {
        return values[rand.nextInt(values.length)];
    }
} ///:~
```

Несколько странный синтаксис `<T extends Enum<T>>` описывает `T` как экземпляр перечисления. Передавая `Class<T>`, мы предоставляем доступ к объекту класса для получения массива экземпляров класса. Перегруженному методу `random()` достаточно знать, что он получает `T[]`, потому что ему не нужно выполнять операции `Enum`; все, что от него требуется, — случайный выбор элемента массива. Возвращаемым типом является точный тип `enum`.

Простой тестовый пример для метода `random()`:

```
//: enumerated/RandomTest.java
import net.mindview.util.*;

enum Activity { SITTING, LYING, STANDING, HOPPING,
    RUNNING, DODGING, JUMPING, FALLING, FLYING }

public class RandomTest {
    public static void main(String[] args) {
        for(int i = 0; i < 20; i++)
            System.out.print(Enums.random(Activity.class) + " ");
    }
} /* Output:
STANDING FLYING RUNNING STANDING RUNNING STANDING LYING
DODGING SITTING RUNNING HOPPING HOPPING HOPPING RUNNING
STANDING LYING FALLING RUNNING FLYING LYING
*///:~
```

Вы увидите, что несмотря на небольшие размеры, класс `Enums` предотвращает значительный объем дублирования кода. Дублирование повышает вероятность ошибок, поэтому его устранение является полезным делом.

Использование интерфейсов для организации кода

Невозможность наследования от перечислений иногда раздражает. Для чего может понадобиться использование перечислений при наследовании? Например, для расширения количества элементов в исходном перечислении или для создания подкатегорий на основе подтипов.

Категории можно реализовать группировкой элементов внутри интерфейса и созданием перечисления на основе этого интерфейса. Допустим, вы хотите создать перечисления для разных видов еды, но при этом каждое из них должно оставаться частным случаем типа `Food`. Вот как может выглядеть решение:

```
//: enumerated/menu/Food.java
// Деление перечислений на категории.
package enumerated.menu;

public interface Food {
    enum Appetizer implements Food {
        SALAD, SOUP, SPRING_ROLLS;
    }
    enum MainCourse implements Food {
        LASAGNE, BURRITO, PAD_THAI,
        LENTILS, HUMMOUS, VINDALOO;
    }
    enum Dessert implements Food {
```

```

    TIRAMISU, GELATO, BLACK_FOREST_CAKE,
    FRUIT, CREME_CARAMEL;
}
enum Coffee implements Food {
    BLACK_COFFEE, DECAF_COFFEE, ESPRESSO,
    LATTE, CAPPUCCINO, TEA, HERB_TEA;
}
} ///:~

```

Так как единственной формой создания подтипов для перечислений является реализация интерфейса, каждое вложенное перечисление реализует окружающий интерфейс `Food`. Теперь можно смело утверждать, что «все экземпляры перечислений являются подтипом `Food`», как видно из следующего примера:

```

//: enumerated/menu/TypeOfFood.java
package enumerated.menu;
import static enumerated.menu.Food.*;

public class TypeOfFood {
    public static void main(String[] args) {
        Food food = Appetizer.SALAD;
        food = MainCourse.LASAGNE;
        food = Dessert.GELATO;
        food = Coffee.CAPPUCCINO;
    }
} ///:~

```

Восходящее преобразование в `Food` работает для каждого типа перечисления, реализующего `Food`, так что все они являются подтипами `Food`.

Впрочем, при работе с набором типов от интерфейса меньше пользы, чем от перечисления. Если вы хотите создать «перечисление перечислений», создайте окружающее перечисление, содержащее один экземпляр для каждого перечисления из `Food`:

```

//: enumerated/menu/Course.java
package enumerated.menu;
import net.mindview.util.*;

public enum Course {
    APPETIZER(Food.Appetizer.class),
    MAINCOURSE(Food.MainCourse.class),
    DESSERT(Food.Dessert.class),
    COFFEE(Food.Coffee.class);
    private Food[] values;
    private Course(Class<? extends Food> kind) {
        values = kind.getEnumConstants();
    }
    public Food randomSelection() {
        return Enums.random(values);
    }
} ///:~

```

Каждое из приведенных перечислений получает в аргументе конструктора соответствующий объект `Class`, из которого оно извлекает и сохраняет все интерфейсы перечислений при помощи `getEnumConstants()`. Эти экземпляры позднее используются в `randomSelection()`, так что теперь мы можем создать случайно сгенерированное блюдо, выбирая один элемент `Food` из каждого экземпляра `Course`:

```

//: enumerated/menu/Meal.java
package enumerated.menu;

public class Meal {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            for(Course course : Course.values()) {
                Food food = course.randomSelection();
                System.out.println(food);
            }
            System.out.println("---");
        }
    }
} /* Output:
SPRING_ROLLS
VINDALOO
FRUIT
DECAF_COFFEE
---
SOUP
VINDALOO
FRUIT
TEA
---
SALAD
BURRITO
FRUIT
TEA
---
SALAD
BURRITO
CREME_CARAMEL
LATTE
---
SOUP
BURRITO
TIRAMISU
ESPRESSO
---
*///:~

```

В этом случае «перечисление перечислений» создается для перебора по всем видам `Course`. Позднее, в примере `VendingMachine.java`, будет продемонстрирован другой подход к делению на категории, определяющийся другими ограничениями.

Другое, более компактное решение проблемы деления на категории основано на вложении перечислений в перечисления:

```

//: enumerated/SecurityCategory.java
// Более компактный способ создания категорий
// в перечислениях.
import net.mindview.util.*;

enum SecurityCategory {
    STOCK(Security.Stock.class), BOND(Security.Bond.class);
    Security[] values;
    SecurityCategory(Class<? extends Security> kind) {
        values = kind.getEnumConstants();
    }
}

```



```

interface Security {
    enum Stock implements Security { SHORT, LONG, MARGIN }
    enum Bond implements Security { MUNICIPAL, JUNK }
}
public Security randomSelection() {
    return Enums.random(values);
}
public static void main(String[] args) {
    for(int i = 0; i < 10; i++) {
        SecurityCategory category =
            Enums.random(SecurityCategory.class);
        System.out.println(category + ": " +
            category.randomSelection());
    }
}
} /* Output:
BOND: MUNICIPAL
BOND: MUNICIPAL
STOCK: MARGIN
STOCK: MARGIN
BOND: JUNK
STOCK: SHORT
STOCK: LONG
STOCK: LONG
BOND: MUNICIPAL
BOND: JUNK
*///:~

```

Интерфейс `Security` необходим для объединения внутренних перечислений под общим типом. Далее они классифицируются по перечислениям в `SecurityCategory`.

Если пойти по этому пути в примере с `Food`, то результат будет выглядеть так:

```

//: enumerated/menu/Meal2.java
package enumerated.menu;
import net.mindview.util.*;

public enum Meal2 {
    APPETIZER(Food.Appetizer.class),
    MAINCOURSE(Food.MainCourse.class),
    DESSERT(Food.Dessert.class),
    COFFEE(Food.Coffee.class);
    private Food[] values;
    private Meal2(Class<? extends Food> kind) {
        values = kind.getEnumConstants();
    }
    public interface Food {
        enum Appetizer implements Food {
            SALAD, SOUP, SPRING_ROLLS;
        }
        enum MainCourse implements Food {
            LASAGNE, BURRITO, PAD_THAI,
            LENTILS, HUMMOUS, VINDALOO;
        }
        enum Dessert implements Food {
            TIRAMISU, GELATO, BLACK_FOREST_CAKE,
            FRUIT, CREME_CARAMEL;
        }
    }
}

```

продолжение ➤

```

enum Coffee implements Food {
    BLACK_COFFEE, DECAF_COFFEE, ESPRESSO,
    LATTE, CAPPUCCINO, TEA, HERB_TEA;
}
}
public Food randomSelection() {
    return Enums.random(values);
}
public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
        for(Meal2 meal : Meal2.values()) {
            Food food = meal.randomSelection();
            System.out.println(food);
        }
        System.out.println("---");
    }
}
} /* Same output as Meal.java *///:~

```

Конечным результатом является всего лишь реорганизация кода, но в некоторых случаях его структура становится более четкой.

3. (1) Добавьте новую категорию `Course` в `Course.java` и продемонстрируйте, что она работает в `Meal.java`.
4. (1) Повторите предыдущее упражнение для `Meal2.java`.
5. (4) Измените пример `control/VowelsAndConsonants.java` так, чтобы в нем использовались три типа перечислений: `VOWEL`, `SOMETIMES_A_VOWEL` и `CONSONANT`. Конструктор перечисления должен получать буквы, описывающие эту конкретную категорию. Подсказка: используйте списки аргументов переменной длины и помните, что массив создается автоматически.
6. (3) Предоставляет ли вложение `Appetizer`, `MainCourse`, `Dessert` и `Coffee` в `Food` какие-либо преимущества по сравнению с их реализацией в виде автономных перечислений, реализующих `Food`?

Использование EnumSet вместо флагов

Множество (`Set`) — разновидность коллекции, в которой запрещены повторяющиеся объекты. Конечно, перечисление требует, чтобы все его члены были уникальными, так что на первый взгляд оно обладает поведением множества, но из-за невозможности добавления и удаления элементов пользы от такого «множества» немного. Класс `EnumSet` был добавлен в Java SE5, где в сочетании с перечислениями он должен был создать замену для традиционных «битовых флагов» на базе `int`. Наборы флагов обычно используются для передачи информации с состояниями «вкл/выкл», но в конечном итоге вам приходится работать с физическими битами, а не с концепциями, что часто приводит к появлению запутанного кода.

Главной целью при проектировании класса `EnumSet` была высокая скорость, потому что он должен успешно конкурировать с наборами битовых флагов (операции с которыми обычно выполняются намного быстрее, чем операции с `HashSet`). Во внутренней

реализации он представляется (если это возможно) одним значением long, которое интерпретируется как битовый вектор, поэтому все операции выполняются исключительно быстро и эффективно. В распоряжении разработчика появляется более выразительный механизм представления двоичных состояний и ему не нужно беспокоиться о производительности.

Элементы EnumSet должны находиться в одном перечислении. В следующем примере используется перечисление мест, в которых установлены датчики сигнализации:

```
//: enumerated/AlarmPoints.java
package enumerated;
public enum AlarmPoints {
    STAIR1, STAIR2, LOBBY, OFFICE1, OFFICE2, OFFICE3,
    OFFICE4, BATHROOM, UTILITY, KITCHEN
} ///:~
```

Контейнер EnumSet может использоваться для отслеживания состояния датчиков:

```
//: enumerated/EnumSets.java
// Операции с EnumSet
package enumerated;
import java.util.*;
import static enumerated.AlarmPoints.*;
import static net.mindview.util.Print.*;

public class EnumSets {
    public static void main(String[] args) {
        EnumSet<AlarmPoints> points =
            EnumSet.noneOf(AlarmPoints.class); // Пустое множество
        points.add(BATHROOM);
        print(points);
        points.addAll(EnumSet.of(STAIR1, STAIR2, KITCHEN));
        print(points);
        points = EnumSet.allOf(AlarmPoints.class);
        points.removeAll(EnumSet.of(STAIR1, STAIR2, KITCHEN));
        print(points);
        points.removeAll(EnumSet.range(OFFICE1, OFFICE4));
        print(points);
        points = EnumSet.complementOf(points);
        print(points);
    }
} /* Output:
[BATHROOM]
[STAIR1, STAIR2, BATHROOM, KITCHEN]
[LOBBY, OFFICE1, OFFICE2, OFFICE3, OFFICE4, BATHROOM,
UTILITY]
[LOBBY, BATHROOM, UTILITY]
[STAIR1, STAIR2, OFFICE1, OFFICE2, OFFICE3, OFFICE4,
KITCHEN]
*///:~
```

Директива статического импортирования упрощает использование констант перечисления. Имена методов достаточно очевидны, а подробные описания можно найти в документации JDK. При обращении к документации обнаруживается нечто интересное — метод of() перегружен как со списком аргументов переменной длины, так и с отдельными методами, получающими от двух до пяти явно заданных аргументов. Это один из признаков ориентированности EnumSet на производительность, потому

что список аргументов переменной длины тоже решает проблему, но слегка уступает по эффективности явной передаче аргументов. Таким образом, при вызове `of()` с передачей от двух до пяти аргументов вы получите (чуть более быстрые) вызовы с явной передачей, а при вызове с одним или более чем пятью аргументами будет вызвана версия `of()` со списком аргументов переменной длины. Учтите, что при вызове с одним аргументом компилятор не будет конструировать массив аргументов, так что вызов этой версии обходится без лишних затрат.

Объекты `EnumSet` строятся на базе 64-разрядных значений `long`, а состояние каждого экземпляра перечисления обозначается одним битом. Это означает, что множество `EnumSet` может представлять до 64 элементов без выхода за пределы одного `long`. А что произойдет, если перечисление содержит более 64 элементов?

```
//: enumerated/BigEnumSet.java
import java.util.*;
```

```
public class BigEnumSet {
    enum Big { A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10,
        A11, A12, A13, A14, A15, A16, A17, A18, A19, A20, A21,
        A22, A23, A24, A25, A26, A27, A28, A29, A30, A31, A32,
        A33, A34, A35, A36, A37, A38, A39, A40, A41, A42, A43,
        A44, A45, A46, A47, A48, A49, A50, A51, A52, A53, A54,
        A55, A56, A57, A58, A59, A60, A61, A62, A63, A64, A65,
        A66, A67, A68, A69, A70, A71, A72, A73, A74, A75 }
    public static void main(String[] args) {
        EnumSet<Big> bigEnumSet = EnumSet.allOf(Big.class);
        System.out.println(bigEnumSet);
    }
} /* Output:
[A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12,
A13, A14, A15, A16, A17, A18, A19, A20, A21, A22, A23, A24,
A25, A26, A27, A28, A29, A30, A31, A32, A33, A34, A35, A36,
A37, A38, A39, A40, A41, A42, A43, A44, A45, A46, A47, A48,
A49, A50, A51, A52, A53, A54, A55, A56, A57, A58, A59, A60,
A61, A62, A63, A64, A65, A66, A67, A68, A69, A70, A71, A72,
A73, A74, A75]
*///:~
```

Очевидно, у множества `EnumSet` нет проблем с перечислениями, содержащими более 64 элементов. Можно предположить, что при необходимости оно добавляет другое значение `long`.

7. (3) Найдите исходный код класса `EnumSet` и объясните, как он работает.

Использование `EnumMap`

`EnumMap` — специализированная карта, ключи которой хранятся в одном перечислении. Из-за ограничений, действующих для перечислений, внутренняя реализация `EnumMap` может базироваться на массиве. Такие контейнеры работают исключительно быстро, поэтому `EnumMap` можно свободно использовать для поиска значений по ключам из перечислений.

Вызов `put()` разрешен только для ключей, входящих в перечисление, но в остальном контейнер `EnumMap` ведет себя как самая обычная карта.

Следующий пример демонстрирует использование паттерна проектирования «Команда». Этот паттерн начинается с интерфейса, содержащего (обычно) один метод, и создает несколько реализаций с разным поведением этого метода. Вы устанавливаете объекты-команды, а ваша программа вызывает их, когда потребуется:

```
//: enumerated/EnumMaps.java
// Основы EnumMap.
package enumerated;
import java.util.*;
import static enumerated.AlarmPoints.*;
import static net.mindview.util.Print.*;

interface Command { void action(); }

public class EnumMaps {
    public static void main(String[] args) {
        EnumMap<AlarmPoints, Command> em =
            new EnumMap<AlarmPoints, Command>(AlarmPoints.class);
        em.put(KITCHEN, new Command() {
            public void action() { print("Kitchen fire!"); }
        });
        em.put(BATHROOM, new Command() {
            public void action() { print("Bathroom alert!"); }
        });
        for(Map.Entry<AlarmPoints, Command> e : em.entrySet()) {
            printnb(e.getKey() + ": ");
            e.getValue().action();
        }
        try { // If there's no value for a particular key:
            em.get(UTILITY).action();
        } catch(Exception e) {
            print(e);
        }
    }
} /* Output:
BATHROOM: Bathroom alert!
KITCHEN: Kitchen fire!
java.lang.NullPointerException
*///:~
```

Как и в случае с EnumSet, порядок элементов EnumMap определяется порядком их включения в перечисление.

Последняя часть main() показывает, что для каждого элемента перечисления всегда существует ключ, но если не вызвать для этого ключа метод put(), связанное с ним значение равно null.

Важное преимущество EnumMap перед *методами констант* (см. следующий раздел) заключается в том, что EnumMap позволяет изменять объекты значений, тогда как методы констант фиксируются во время компиляции.

Как будет показано позднее в этой главе, контейнеры EnumMap могут использоваться для диспетчеризации в ситуациях с несколькими взаимодействующими перечислениями.

Методы констант

Перечисления Java обладают очень интересной возможностью, которая позволяет назначить каждому экземпляру перечисления свое поведение. Для этого следует определить один или несколько абстрактных методов в составе перечисления, а затем определить методы для каждого экземпляра перечисления. Пример:

```
//: enumerated/ConstantSpecificMethod.java
import java.util.*;
import java.text.*;

public enum ConstantSpecificMethod {
    DATE_TIME {
        String getInfo() {
            return
                DateFormat.getDateInstance().format(new Date());
        }
    },
    CLASSPATH {
        String getInfo() {
            return System.getenv("CLASSPATH");
        }
    },
    VERSION {
        String getInfo() {
            return System.getProperty("java.version");
        }
    };
    abstract String getInfo();
    public static void main(String[] args) {
        for(ConstantSpecificMethod csm : values())
            System.out.println(csm.getInfo());
    }
} /* (Execute to see output) *///:~
```

Для идентификации и вызова методов можно использовать связанный с ними экземпляр перечисления. Такой код часто называется *табличным* (обратите внимание на сходство с упоминавшимся выше паттерном «Команда»).

В объектно-ориентированном программировании с разными классами связывается разное поведение. Так как каждый экземпляр перечисления может иметь собственное поведение, реализованное методами констант, создается впечатление, будто каждый экземпляр относится к отдельному типу. В приведенном примере каждый экземпляр перечисления интерпретируется как «базовый тип» `ConstantSpecificMethod`, но вызов метода `getInfo()` реализует полиморфное поведение.

Тем не менее это лишь внешнее сходство. Вы не сможете полноценно работать с экземплярами перечисления как с разными типами:

```
//: enumerated/NotClasses.java
// {Exec: javap -c LikeClasses}
import static net.mindview.util.Print.*;

enum LikeClasses {
    WINKEN { void behavior() { print("Behavior1"); } },
    BLINKEN { void behavior() { print("Behavior2"); } },
}
```

```

    NOD { void behavior() { print("Behavior3"); } };
    abstract void behavior();
}

public class NotClasses {
    // void f1(LikeClasses.WINKEN instance) {} // Нет
} /* Output:
Compiled from "NotClasses.java"
abstract class LikeClasses extends java.lang.Enum{
public static final LikeClasses WINKEN;

public static final LikeClasses BLINKEN;

public static final LikeClasses NOD;
...
*///:~

```

Как видите, в `f1()` компилятор не позволяет использовать экземпляр перечисления как тип класса, и это вполне логично, если представить код, генерируемый компилятором, — каждый элемент перечисления представляет собой статический `final`-экземпляр `LikeClasses`.

Кроме того, будучи статическими, экземпляры внутренних перечислений по своему поведению отличаются от обычных внутренних классов; вы не можете обращаться к нестатическим полям или методам внешнего класса.

Рассмотрим более интересный пример: модель автомойки. Каждому клиенту предоставляется набор вариантов, для каждого из которых выполняется свое действие `action()`. С каждым вариантом связывается метод константы, а для хранения выбора клиента может использоваться контейнер `EnumSet`:

```

//: enumerated/CarWash.java
import java.util.*;
import static net.mindview.util.Print.*;

public class CarWash {
    public enum Cycle {
        UNDERBODY {
            void action() { print("Spraying the underbody"); }
        },
        WHEELWASH {
            void action() { print("Washing the wheels"); }
        },
        PREWASH {
            void action() { print("Loosening the dirt"); }
        },
        BASIC {
            void action() { print("The basic wash"); }
        },
        HOTWAX {
            void action() { print("Applying hot wax"); }
        },
        RINSE {
            void action() { print("Rinsing"); }
        },
        BLOWDRY {
            void action() { print("Blowing dry"); }
        }
    }
}

```

```

    };
    abstract void action();
}
EnumSet<Cycle> cycles =
    EnumSet.of(Cycle.BASIC, Cycle.RINSE);
public void add(Cycle cycle) { cycles.add(cycle); }
public void washCar() {
    for(Cycle c : cycles)
        c.action();
}
public String toString() { return cycles.toString(); }
public static void main(String[] args) {
    CarWash wash = new CarWash();
    print(wash);
    wash.washCar();
    // Порядок добавления неважен:
    wash.add(Cycle.BLOWDRY);
    wash.add(Cycle.BLOWDRY); // Дубликаты игнорируются
    wash.add(Cycle.RINSE);
    wash.add(Cycle.HOTWAX);
    print(wash);
    wash.washCar();
}
} /* Output:
[BASIC, RINSE]
The basic wash
Rinsing
[BASIC, HOTWAX, RINSE, BLOWDRY]
The basic wash
Applying hot wax
Rinsing
Blowing dry
*///:~

```

Синтаксис определения методов констант фактически совпадает с синтаксисом анонимных внутренних классов, но оказывается более компактным.

В этом примере также продемонстрированы дополнительные характеристики контейнера `EnumSet`. Как и любое множество, он не содержит дубликатов, так что вызовы `add()` с одинаковым аргументом игнорируются (логично, поскольку бит можно «установить» только один раз). Кроме того, порядок добавления экземпляров перечисления не важен — порядок вывода определяется порядком объявления перечисления.

Возможно ли переопределить методы констант вместо реализации абстрактного метода? Да, можно, как показывает следующий пример:

```

//: enumerated/OverrideConstantSpecific.java
import static net.mindview.util.Print.*;

public enum OverrideConstantSpecific {
    NUT, BOLT,
    WASHER {
        void f() { print("Overridden method"); }
    };
    void f() { print("default behavior"); }
    public static void main(String[] args) {
        for(OverrideConstantSpecific ocs : values()) {

```



```

        printnb(ocs + ": ");
        ocs.f();
    }
}
} /* Output:
NUT: default behavior
BOLT: default behavior
WASHER: Overridden method
*///:~

```

Хотя некоторые программные конструкции с перечислениями не работают, в общем случае, с ними можно экспериментировать так, словно они являются классами.

Цепочка обязанностей

В паттерне «Цепочка обязанностей» разработчик создает несколько способов решения некоторой задачи и связывает их в цепочку. Поступающий запрос передается по цепочке до тех пор, пока одно из решений не сможет обработать его.

Простой вариант «Цепочки обязанностей» легко реализуется методами констант. Рассмотрим модель почтового отделения, которое пытается обрабатывать входящую почту по общим правилам, но продолжает применять разные варианты до тех пор, пока не придет к выводу о невозможности доставки. Каждая попытка может рассматриваться как «Стратегия» (еще один паттерн проектирования), а весь список в целом — как «Цепочка обязанностей».

Все начинается с описания почтового отправления. Все разные характеристики, учитываемые при обработке, могут быть представлены перечислениями. Так как объекты mail будут генерироваться случайным образом, для снижения вероятности получить (например) YES в категории GeneralDelivery проще всего создать побольше экземпляров, отличных от YES, так что определения enum на первый взгляд смотрятся немного странно.

В определении Mail присутствует метод randomMail(), создающий случайные объекты почтовых отправок. Метод generator() создает объект Iterable, который использует randomMail() для создания набора объектов — по одному при каждом вызове next() через итератор. Такая конструкция позволяет легко создать цикл foreach вызовом Mail.generator():

```

//: enumerated/PostOffice.java
// Modeling a post office.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

class Mail {
    // Множественные NO снижают вероятность
    // случайного выбора YES:
    enum GeneralDelivery {YES,NO1,NO2,NO3,NO4,NO5}
    enum Scannability {UNSCANNABLE,YES1,YES2,YES3,YES4}
    enum Readability {ILLEGIBLE,YES1,YES2,YES3,YES4}
    enum Address {INCORRECT,OK1,OK2,OK3,OK4,OK5,OK6}
    enum ReturnAddress {MISSING,OK1,OK2,OK3,OK4,OK5}
    GeneralDelivery generalDelivery;
    Scannability scannability;
}

```

продолжение »

```

Readability readability;
Address address;
ReturnAddress returnAddress;
static long counter = 0;
long id = counter++;
public String toString() { return "Mail " + id; }
public String details() {
    return toString() +
        ", General Delivery: " + generalDelivery +
        ", Address Scannability: " + scannability +
        ", Address Readability: " + readability +
        ", Address Address: " + address +
        ", Return address: " + returnAddress;
}
// Generate test Mail:
public static Mail randomMail() {
    Mail m = new Mail();
    m.generalDelivery= Enums.random(GeneralDelivery.class);
    m.scannability = Enums.random(Scannability.class);
    m.readability = Enums.random(Readability.class);
    m.address = Enums.random(Address.class);
    m.returnAddress = Enums.random(ReturnAddress.class);
    return m;
}
public static Iterable<Mail> generator(final int count) {
    return new Iterable<Mail>() {
        int n = count;
        public Iterator<Mail> iterator() {
            return new Iterator<Mail>() {
                public boolean hasNext() { return n-- > 0; }
                public Mail next() { return randomMail(); }
                public void remove() { // Не реализовано
                    throw new UnsupportedOperationException();
                }
            };
        }
    };
}

public class PostOffice {
    enum MailHandler {
        GENERAL_DELIVERY {
            boolean handle(Mail m) {
                switch(m.generalDelivery) {
                    case YES:
                        print("Using general delivery for " + m);
                        return true;
                    default: return false;
                }
            }
        },
        MACHINE_SCAN {
            boolean handle(Mail m) {
                switch(m.scannability) {
                    case UNSCANNABLE: return false;
                    default:
                        switch(m.address) {

```

```

        case INCORRECT: return false;
        default:
            print("Delivering "+ m + " automatically");
            return true;
    }
}
},
VISUAL_INSPECTION {
    boolean handle(Mail m) {
        switch(m.readability) {
            case ILLEGIBLE: return false;
            default:
                switch(m.address) {
                    case INCORRECT: return false;
                    default:
                        print("Delivering " + m + " normally");
                        return true;
                }
        }
    }
},
RETURN_TO_SENDER {
    boolean handle(Mail m) {
        switch(m.returnAddress) {
            case MISSING: return false;
            default:
                print("Returning " + m + " to sender");
                return true;
        }
    }
};
abstract boolean handle(Mail m);
}
static void handle(Mail m) {
    for(MailHandler handler : MailHandler.values())
        if(handler.handle(m))
            return;
    print(m + " is a dead letter");
}
public static void main(String[] args) {
    for(Mail mail : Mail.generator(10)) {
        print(mail.details());
        handle(mail);
        print("*****");
    }
}
} /* Output:
Mail 0, General Delivery: NO2, Address Scanability:
UNSCANNABLE, Address Readability: YES3, Address Address:
OK1, Return address: OK1
Delivering Mail 0 normally
*****
Mail 1, General Delivery: NO5, Address Scanability: YES3,
Address Readability: ILLEGIBLE, Address Address: OK5,
Return address: OK1
Delivering Mail 1 automatically
*****

```

Mail 2, General Delivery: YES, Address Scanability: YES3,
Address Readability: YES1, Address Address: OK1, Return
address: OK5

Using general delivery for Mail 2

Mail 3, General Delivery: NO4, Address Scanability: YES3,
Address Readability: YES1, Address Address: INCORRECT,
Return address: OK4

Returning Mail 3 to sender

Mail 4, General Delivery: NO4, Address Scanability:
UNSCANNABLE, Address Readability: YES1, Address Address:
INCORRECT, Return address: OK2

Returning Mail 4 to sender

Mail 5, General Delivery: NO3, Address Scanability: YES1,
Address Readability: ILLEGIBLE, Address Address: OK4,
Return address: OK2

Delivering Mail 5 automatically

Mail 6, General Delivery: YES, Address Scanability: YES4,
Address Readability: ILLEGIBLE, Address Address: OK4,
Return address: OK4

Using general delivery for Mail 6

Mail 7, General Delivery: YES, Address Scanability: YES3,
Address Readability: YES4, Address Address: OK2, Return
address: MISSING

Using general delivery for Mail 7

Mail 8, General Delivery: NO3, Address Scanability: YES1,
Address Readability: YES3, Address Address: INCORRECT,
Return address: MISSING

Mail 8 is a dead letter

Mail 9, General Delivery: NO1, Address Scanability:
UNSCANNABLE, Address Readability: YES2, Address Address:
OK1, Return address: OK4

Delivering Mail 9 normally

*///:~

«Цепочка обязанностей» выражается перечислением `MailHandler`, причем порядок определений задает последовательность, в которой к каждому почтовому отправлению применяются различные стратегии. Каждая стратегия применяется последовательно до тех пор, пока одна из них не завершится успешно или во всех случаях не будет получен отказ (и тогда делается вывод о невозможности доставки).

8. (6) Измените пример `PostOffice.java` так, чтобы он поддерживал возможность пересылки почты.

9. (5) Измените класс `PostOffice` так, чтобы в нем использовался контейнер `EnumMap`.

Проект¹. Специализированные языки (такие, как `Prolog`) используют *обратное построение цепочки* для решения подобных задач. Взяв за основу код `PostOffice.java`, ис-

¹ Проекты могут использоваться (например) в качестве тем для курсовых работ. Решения для них не приводятся в сборнике.

следуйте эту возможность в таких языках и разработайте программу, которая позволяет легко добавлять в систему новые «правила».

Конечные автоматы

Перечислимые типы идеально подходят для реализации конечных автоматов. Конечный автомат обладает конечным числом состояний. В общем случае автомат переходит между состояниями в зависимости от вводимых данных, но также возможны *переходные состояния*, из которых автомат выходит сразу же после завершения их задач.

Для каждого состояния имеются допустимые сочетания входных действий, причем разные входные действия переводят автомат в разные новые состояния. Так как перечисления ограничивают набор допустимых вариантов, они хорошо подходят для определения разных состояний и входных действий.

С каждым состоянием также обычно связываются некоторые выходные данные.

Хорошим примером конечного автомата является торговый автомат. Сначала мы определяем различные варианты ввода в перечислении:

```
//: enumerated/Input.java
package enumerated;
import java.util.*;

public enum Input {
    NICKEL(5), DIME(10), QUARTER(25), DOLLAR(100),
    TOOTHPASTE(200), CHIPS(75), SODA(100), SOAP(50),
    ABORT_TRANSACTION {
        public int amount() { // Запрещено
            throw new RuntimeException("ABORT.amount()");
        }
    },
    STOP { // Это должен быть последний экземпляр.
        public int amount() { // Запрещено
            throw new RuntimeException("SHUT_DOWN.amount()");
        }
    };
    int value; // В центах
    Input(int value) { this.value = value; }
    Input() {}
    int amount() { return value; }; // В центах
    static Random rand = new Random(47);
    public static Input randomSelection() {
        // Не включая STOP:
        return values()[rand.nextInt(values().length - 1)];
    }
} //:~
```

Обратите внимание: с двумя разновидностями `Input` связана сумма, поэтому в интерфейсе определен метод `amount()`. Однако вызов `amount()` для двух других видов `Input` некорректен, поэтому при вызове `amount()` они возбуждают исключения. Такая конфигурация выглядит несколько странно (определяем метод в интерфейсе, а затем возбуждаем исключение при его вызове для некоторых реализаций), но она является вынужденной из-за ограничений, связанных с перечислениями.

Класс `VendingMachine` реагирует на ввод, сначала классифицируя его по перечислению `Category`. Этот пример также показывает, как перечисления делают код более четким и удобным в сопровождении:

```

//: enumerated/VendingMachine.java
// {Args: VendingMachineInput.txt}
package enumerated;
import java.util.*;
import net.mindview.util.*;
import static enumerated.Input.*;
import static net.mindview.util.Print.*;

enum Category {
    MONEY(NICKEL, DIME, QUARTER, DOLLAR),
    ITEM_SELECTION(TOOTHPASTE, CHIPS, SODA, SOAP),
    QUIT_TRANSACTION(ABORT_TRANSACTION),
    SHUT_DOWN(STOP);
    private Input[] values;
    Category(Input... types) { values = types; }
    private static EnumMap<Input,Category> categories =
        new EnumMap<Input,Category>(Input.class);
    static {
        for(Category c : Category.class.getEnumConstants())
            for(Input type : c.values)
                categories.put(type, c);
    }
    public static Category categorize(Input input) {
        return categories.get(input);
    }
}

public class VendingMachine {
    private static State state = State.RESTING;
    private static int amount = 0;
    private static Input selection = null;
    enum StateDuration { TRANSIENT } // Перечисление для пометки
    enum State {
        RESTING {
            void next(Input input) {
                switch(Category.categorize(input)) {
                    case MONEY:
                        amount += input.amount();
                        state = ADDING_MONEY;
                        break;
                    case SHUT_DOWN:
                        state = TERMINAL;
                    default:
                }
            }
        },
        ADDING_MONEY {
            void next(Input input) {
                switch(Category.categorize(input)) {
                    case MONEY:
                        amount += input.amount();
                        break;
                    case ITEM_SELECTION:
                        selection = input;
                }
            }
        }
    }
}

```

```

        if(amount < selection.amount())
            print("Insufficient money for " + selection);
        else state = DISPENSING;
        break;
    case QUIT_TRANSACTION:
        state = GIVING_CHANGE;
        break;
    case SHUT_DOWN:
        state = TERMINAL;
    default:
    }
}
},
DISPENSING(StateDuration.TRANSIENT) {
    void next() {
        print("here is your " + selection);
        amount -= selection.amount();
        state = GIVING_CHANGE;
    }
},
GIVING_CHANGE(StateDuration.TRANSIENT) {
    void next() {
        if(amount > 0) {
            print("Your change: " + amount);
            amount = 0;
        }
        state = RESTING;
    }
},
TERMINAL { void output() { print("Halted"); } };
private boolean isTransient = false;
State() {}
State(StateDuration trans) { isTransient = true; }
void next(Input input) {
    throw new RuntimeException("Only call " +
        "next(Input input) for non-transient states");
}
void next() {
    throw new RuntimeException("Only call next() for " +
        "StateDuration.TRANSIENT states");
}
void output() { print(amount); }
}
static void run(Generator<Input> gen) {
    while(state != State.TERMINAL) {
        state.next(gen.next());
        while(state.isTransient)
            state.next();
        state.output();
    }
}
public static void main(String[] args) {
    Generator<Input> gen = new RandomInputGenerator();
    if(args.length == 1)
        gen = new FileInputGenerator(args[0]);
    run(gen);
}
}

```

```
// Для простейшей проверки работоспособности:
class RandomInputGenerator implements Generator<Input> {
    public Input next() { return Input.randomSelection(); }
}

// Создание данных Input по файлу строк, разделенных символом ';':
class FileInputGenerator implements Generator<Input> {
    private Iterator<String> input;
    public FileInputGenerator(String fileName) {
        input = new TextFile(fileName, ";").iterator();
    }
    public Input next() {
        if(!input.hasNext())
            return null;
        return Enum.valueOf(Input.class, input.next().trim());
    }
} /* Output:
25
50
75
here is your CHIPS
0
100
200
here is your TOOTHPASTE
0
25
35
Your change: 35
0
25
35
Insufficient money for SODA
35
60
70
75
Insufficient money for SODA
75
Your change: 75
0
Halted
*///:~
```

Так как выбор между экземплярами перечисления чаще всего реализуется командой `switch` (обратите внимание, как язык упрощает выбор вариантов по перечислению), один из самых частых вопросов, которые следует задавать себе при упорядочении нескольких перечислений: «По какому условию должен осуществляться выбор?» В нашем примере для каждого состояния `State` переключение должно осуществляться по базовым категориям входных действий: внесение денег, выбор товара, отмена покупки, выключение автомата. Однако внутри этих категорий возможно внесение разных валют и выбор разных товаров. Перечисление `Category` группирует разные типы `Input`, чтобы метод `categorize()` мог создать соответствующую категорию `Category` в команде `switch`. Метод использует класс `EnumMap` для эффективного и безопасного выполнения поиска.

Изучив класс `VendingMachine`, вы увидите, чем состояния отличаются друг от друга и как они реагируют на входные действия. Также обратите внимание на два переходных состояния; в состоянии `run()` машина ожидает ввода `Input` и продолжает перемещаться между состояниями, пока ее текущее состояние остается переходным.

Для тестирования `VendingMachine` можно воспользоваться двумя способами, с двумя разными объектами `Generator`. `RandomInputGenerator` просто выдает случайные варианты ввода (кроме `SHUT_DOWN`). Работа этого генератора в течение продолжительного времени поможет убедиться в том, что автомат не переходит в недопустимое состояние. `FileInputGenerator` получает файл с описанием ввода в текстовой форме, преобразует его в экземпляры перечисления и создает объекты `Input`.

Вот как выглядит текстовый файл для получения результатов, приведенных выше:

```
//:1 enumerated/VendingMachineInput.txt
QUARTER; QUARTER; QUARTER; CHIPS;
DOLLAR; DOLLAR; TOOTHPASTE;
QUARTER; DIME; ABORT_TRANSACTION;
QUARTER; DIME; SODA;
QUARTER; DIME; NICKEL; SODA;
ABORT_TRANSACTION;
STOP;
///:~
```

Одно из ограничений этой архитектуры заключается в том, что поля `VendingMachine`, с которыми работают экземпляры перечисления `State`, должны быть статическими; это означает, что экземпляр `VendingMachine` может быть только один. Это не такая уж большая проблема, если рассмотреть фактическую реализацию, так как на одном компьютере с большой вероятностью будет запускаться только одно приложение.

10. (7) Измените класс `VendingMachine` (только) с использованием `EnumMap`, чтобы в одной программе можно было создать несколько экземпляров `VendingMachine`.
11. (7) В программном обеспечении настоящего торгового автомата нужно иметь возможность легко добавлять и менять тип продаваемых товаров, так что ограничения, устанавливаемые перечислением для `Input`, не практичны (не забудьте, что перечисления предназначены для ограниченных наборов типов). Измените код `VendingMachine.java` так, чтобы продаваемые товары представлялись классом, а не были частью `Input`, и инициализируйте контейнер `ArrayList` этих объектов из текстового файла (используйте `net.mindview.util.TextFile`).

Проект¹. Разработайте модель торгового автомата с поддержкой интернационализации, чтобы один автомат мог быть легко приспособлен для всех стран.

Множественная диспетчеризация

При использовании нескольких взаимодействующих типов программа основательно усложняется. Для примера возьмем систему для разбора и выполнения математических выражений. Было бы желательно использовать запись вида `Number.plus(Number)`,

¹ Проекты могут использоваться (например) в качестве тем для курсовых работ. Решения для них здесь не приводятся.

`Number.multiply(Number)` и т. д., где `Number` — базовый класс для семейства числовых объектов. Но если вы используете запись `a.plus(b)`, а точный тип `a` или `b` неизвестен, как обеспечить их взаимодействие?

Ответ начинается несколько неожиданно: в Java выполняется только *одиночная диспетчеризация*. Другими словами, если вы выполняете операцию с более чем одним объектом, тип которого неизвестен, Java может активизировать механизм динамического связывания только для одного из этих типов. Это не решает описанную ранее проблему, так что в конечном итоге вам придется определять типы вручную и по сути самостоятельно реализовать поведение динамического связывания.

Решение называется *множественной диспетчеризацией* (multiple dispatching). (В нашем случае это двойная диспетчеризация.) Полиморфизм работает только при вызове методов, поэтому для достижения двойной диспетчеризации должны использоваться два вызова методов: один для определения первого неизвестного типа и второй для определения второго неизвестного типа. При множественной диспетчеризации необходимо иметь виртуальный вызов для каждого из типов — если вы работаете с двумя разными иерархиями, которые взаимодействуют между собой, понадобится виртуальный вызов в каждой иерархии. В общем случае создается такая конфигурация, при которой один вызов метода инициирует более одного вызова виртуального метода, а следовательно, обслуживает более одного типа. Для достижения нужного эффекта понадобится вызов метода для каждой диспетчеризации. Методы в следующем примере (реализующем игру «камень-ножницы-бумага») называются `compete()` и `eval()` и являются членами одного типа. Они приводят к одному из трех возможных результатов¹:

```
//: enumerated/Outcome.java
package enumerated;
public enum Outcome { WIN, LOSE, DRAW } ///:~

//: enumerated/RoShamBo1.java
// Демонстрация множественной диспетчеризации.
package enumerated;
import java.util.*;
import static enumerated.Outcome.*;

interface Item {
    Outcome compete(Item it);
    Outcome eval(Paper p);
    Outcome eval(Scissors s);
    Outcome eval(Rock r);
}

class Paper implements Item {
    public Outcome compete(Item it) { return it.eval(this); }
    public Outcome eval(Paper p) { return DRAW; }
    public Outcome eval(Scissors s) { return WIN; }
    public Outcome eval(Rock r) { return LOSE; }
    public String toString() { return "Paper"; }
}
```

¹ Этот пример несколько лет просуществовал в реализациях на C++ и Java (в книге *Thinking in Patterns*) на сайте www.MindView.net, после чего появился без ссылки на источник в книге других авторов.

```

class Scissors implements Item {
    public Outcome compete(Item it) { return it.eval(this); }
    public Outcome eval(Paper p) { return LOSE; }
    public Outcome eval(Scissors s) { return DRAW; }
    public Outcome eval(Rock r) { return WIN; }
    public String toString() { return "Scissors"; }
}

class Rock implements Item {
    public Outcome compete(Item it) { return it.eval(this); }
    public Outcome eval(Paper p) { return WIN; }
    public Outcome eval(Scissors s) { return LOSE; }
    public Outcome eval(Rock r) { return DRAW; }
    public String toString() { return "Rock"; }
}

public class RoShamBo1 {
    static final int SIZE = 20;
    private static Random rand = new Random(47);
    public static Item newItem() {
        switch(rand.nextInt(3)) {
            default:
            case 0: return new Scissors();
            case 1: return new Paper();
            case 2: return new Rock();
        }
    }
    public static void match(Item a, Item b) {
        System.out.println(
            a + " vs. " + b + ": " + a.compete(b));
    }
    public static void main(String[] args) {
        for(int i = 0; i < SIZE; i++)
            match(newItem(), newItem());
    }
} /* Output:
Rock vs. Rock: DRAW
Paper vs. Rock: WIN
Paper vs. Rock: WIN
Paper vs. Rock: WIN
Scissors vs. Paper: WIN
Scissors vs. Scissors: DRAW
Scissors vs. Paper: WIN
Rock vs. Paper: LOSE
Paper vs. Paper: DRAW
Rock vs. Paper: LOSE
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
Rock vs. Scissors: WIN
Rock vs. Paper: LOSE
Paper vs. Rock: WIN
Scissors vs. Paper: WIN
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
*///:~

```

`Item` — интерфейс для типов, которые будут задействованы во множественной диспетчеризации. Метод `RoShamBo1.match()` берет два объекта `Item` и запускает процесс двойной диспетчеризации, вызывая функцию `Item.compete()`. Виртуальный механизм определяет тип `a`, поэтому активизируется функция `compete()` конкретного типа `a`. Функция `compete()` выполняет вторую диспетчеризацию, вызывая `eval()` для оставшегося типа. Передача самого себя (`this`) в аргументе `eval()` приводит к вызову перегруженной функции `eval()`, сохраняющему информацию типа первой диспетчеризации. При завершении второй диспетчеризации вы знаете точный тип обоих объектов `Item`.

Организация множественной диспетчеризации требует изрядных хлопот, но ее преимуществом является элегантность синтаксиса при вызове — вместо того, чтобы писать неуклюжий код для определения типа одного или нескольких объектов во время вызова, вы просто заявляете: «Вы, двое! Мне безразлично, к какому типу вы относитесь, правильно взаимодействуйте друг с другом!» Впрочем, прежде чем решаться на множественную диспетчеризацию, убедитесь в том, что подобная элегантность для вас действительно важна.

Диспетчеризация с использованием перечислений

Напрямую преобразовать пример `RoShamBo1.java` в решение на базе перечислений проблематично — экземпляры перечислений не являются типами, поэтому перегруженные методы `eval()` не работают — экземпляры перечислений не могут использоваться как типы аргументов. Впрочем, существует ряд различных подходов к реализации множественной диспетчеризации, использующих возможности перечислений.

В одном из решений конструктор используется для инициализации каждого экземпляра перечисления со «строкой» результатов; так формируется некое подобие таблицы:

```
//: enumerated/RoShamBo2.java
// Переключение между перечислениями.
package enumerated;
import static enumerated.Outcome.*;

public enum RoShamBo2 implements Competitor<RoShamBo2> {
    PAPER(DRAW, LOSE, WIN),
    SCISSORS(WIN, DRAW, LOSE),
    ROCK(LOSE, WIN, DRAW);
    private Outcome vPAPER, vSCISSORS, vROCK;
    RoShamBo2(Outcome paper, Outcome scissors, Outcome rock) {
        this.vPAPER = paper;
        this.vSCISSORS = scissors;
        this.vROCK = rock;
    }
    public Outcome compete(RoShamBo2 it) {
        switch(it) {
            default:
            case PAPER: return vPAPER;
            case SCISSORS: return vSCISSORS;
            case ROCK: return vROCK;
        }
    }
}
```

```

    public static void main(String[] args) {
        RoShamBo.play(RoShamBo2.class, 20);
    }
} /* Output:
ROCK vs. ROCK: DRAW
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
PAPER vs. PAPER: DRAW
PAPER vs. SCISSORS: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. SCISSORS: DRAW
ROCK vs. SCISSORS: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
ROCK vs. PAPER: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
*///:~

```

Когда в `compete()` будут определены оба типа, единственным действием будет возвращение соответствующего объекта `Outcome`. Но также возможен вызов другого метода — даже (например) через объект `Command`, присвоенный в конструкторе.

Программа `RoShamBo2.java` намного компактнее и стройнее исходного примера и ее код намного проще в сопровождении. Обратите внимание: мы все еще используем две диспетчеризации для определения типа обоих объектов. В `RoShamBo1.java` обе диспетчеризации выполнялись с использованием вызовов виртуальных методов, но здесь вызов виртуального метода используется только при первой диспетчеризации. Вторая диспетчеризация основана на `switch`, но такой способ безопасен, потому что перечисление ограничивает выбор в команде `switch`.

Код, управляющий перечислением, был вынесен для возможного использования в других примерах. Сначала интерфейс `Competitor` определяет тип, который соревнуется с другим объектом `Competitor`:

```

//: enumerated/Competitor.java
// Переключение между перечислениями.
package enumerated;

public interface Competitor<T extends Competitor<T>> {
    Outcome compete(T competitor);
} ///:~

```

Затем мы определяем два статических метода (статических, чтобы избежать необходимости явно задавать тип параметра). Сначала `match()` вызывает `compete()` для двух объектов `Competitor`; в этом случае в качестве параметра-типа достаточно использовать `Competitor<T>`. Но в `play()` параметр-тип должен быть одновременно `Enum<T>`, потому что он используется в `Enums.random()` и `Competitor<T>`, а также потому, что он передается `match()`:

```

//: enumerated/RoShamBo.java
// Общий инструментарий для примеров RoShamBo.
package enumerated;
import net.mindview.util.*;

public class RoShamBo {
    public static <T extends Competitor<T>>
    void match(T a, T b) {
        System.out.println(
            a + " vs. " + b + ": " + a.compete(b));
    }
    public static <T extends Enum<T> & Competitor<T>>
    void play(Class<T> rsbClass, int size) {
        for(int i = 0; i < size; i++)
            match(
                Enums.random(rsbClass), Enums.random(rsbClass));
    }
} ///:~

```

Метод play() не имеет возвращаемого значения, в котором задействован параметр-тип T, поэтому может показаться, что в типе Class<T> можно использовать маски (wildcards) вместо начального описания параметра. Однако маски не могут расширять более одного базового типа, поэтому приходится использовать приведенное выше выражение.

Использование методов констант

Так как методы констант позволяют определять разные реализации метода для разных экземпляров перечисления, на первый взгляд они идеально решают проблему настройки множественной диспетчеризации. Но даже несмотря на возможность такого определения разного поведения, экземпляры перечисления не являются типами и поэтому не могут использоваться в качестве типов аргументов в сигнатурах методов. Лучшее, что можно сделать в этом случае, — использовать конструкцию switch:

```

//: enumerated/RoShamBo3.java
// Использовать методов констант.
package enumerated;
import static enumerated.Outcome.*;

public enum RoShamBo3 implements Competitor<RoShamBo3> {
    PAPER {
        public Outcome compete(RoShamBo3 it) {
            switch(it) {
                default: // Чтобы компилятор не жаловался
                case PAPER: return DRAW;
                case SCISSORS: return LOSE;
                case ROCK: return WIN;
            }
        }
    },
    SCISSORS {
        public Outcome compete(RoShamBo3 it) {
            switch(it) {
                default:
                case PAPER: return WIN;
                case SCISSORS: return DRAW;
                case ROCK: return LOSE;
            }
        }
    }
}

```

```

    }
  }
},
ROCK {
  public Outcome compete(RoShamBo3 it) {
    switch(it) {
      default:
      case PAPER: return LOSE;
      case SCISSORS: return WIN;
      case ROCK: return DRAW;
    }
  }
};
public abstract Outcome compete(RoShamBo3 it);
public static void main(String[] args) {
  RoShamBo.play(RoShamBo3.class, 20);
}
} /* Same output as RoShamBo2.java *///:~

```

Такое решение функционально и вполне разумно, но `RoShamBo2.java` требует меньшего объема кода при добавлении нового типа и потому выглядит более прямолинейно.

Впрочем, пример `RoShamBo3.java` можно упростить и сделать более компактным:

```

//: enumerated/RoShamBo4.java
package enumerated;

public enum RoShamBo4 implements Competitor<RoShamBo4> {
  ROCK {
    public Outcome compete(RoShamBo4 opponent) {
      return compete(SCISSORS, opponent);
    }
  },
  SCISSORS {
    public Outcome compete(RoShamBo4 opponent) {
      return compete(PAPER, opponent);
    }
  },
  PAPER {
    public Outcome compete(RoShamBo4 opponent) {
      return compete(ROCK, opponent);
    }
  };
  Outcome compete(RoShamBo4 loser, RoShamBo4 opponent) {
    return ((opponent == this) ? Outcome.DRAW
      : ((opponent == loser) ? Outcome.WIN
        : Outcome.LOSE));
  }
  public static void main(String[] args) {
    RoShamBo.play(RoShamBo4.class, 20);
  }
} /* Same output as RoShamBo2.java *///:~

```

Здесь вторая диспетчеризация выполняется версией `compete()` с двумя аргументами, которая выполняет серию сравнений, а следовательно, сходна с действием `switch`. Она компактнее, но выглядит не так понятно. В больших системах эта сложность может оказаться критичной.

Диспетчеризация с EnumMap

«Полноценную» двойную диспетчеризацию можно реализовать с использованием класса EnumMap, который специально спроектирован для очень эффективной работы с перечислениями. Так как целью является переключение по двум неизвестным типам, карта EnumMap с элементами EnumMap может использоваться для достижения двойной диспетчеризации:

```
//: enumerated/RoShamBo5.java
// Множественная диспетчеризация на базе
// карты EnumMap с элементами EnumMap.
package enumerated;
import java.util.*;
import static enumerated.Outcome.*;

enum RoShamBo5 implements Competitor<RoShamBo5> {
    PAPER, SCISSORS, ROCK;
    static EnumMap<RoShamBo5, EnumMap<RoShamBo5, Outcome>>
        table = new EnumMap<RoShamBo5,
            EnumMap<RoShamBo5, Outcome>>(RoShamBo5.class);
    static {
        for(RoShamBo5 it : RoShamBo5.values())
            table.put(it,
                new EnumMap<RoShamBo5, Outcome>(RoShamBo5.class));
        initRow(PAPER, DRAW, LOSE, WIN);
        initRow(SCISSORS, WIN, DRAW, LOSE);
        initRow(ROCK, LOSE, WIN, DRAW);
    }
    static void initRow(RoShamBo5 it,
        Outcome vPAPER, Outcome vSCISSORS, Outcome vROCK) {
        EnumMap<RoShamBo5, Outcome> row =
            RoShamBo5.table.get(it);
        row.put(RoShamBo5.PAPER, vPAPER);
        row.put(RoShamBo5.SCISSORS, vSCISSORS);
        row.put(RoShamBo5.ROCK, vROCK);
    }
    public Outcome compete(RoShamBo5 it) {
        return table.get(this).get(it);
    }
    public static void main(String[] args) {
        RoShamBo5.play(RoShamBo5.class, 20);
    }
} /* Same output as RoShamBo2.java *///:~
```

Карта EnumMap инициализируется в секции static; табличная структура видна на примере вызовов initRow(). Обратите внимание на метод compete(), в котором обе диспетчеризации выполняются в одной команде.

Использование двумерного массива

Решение можно дополнительно упростить — достаточно заметить, что каждый экземпляр перечисления имеет фиксированное значение (определяемое порядком объявления), которое может быть получено вызовом ordinal(). Двумерный массив, отображающий Competitor на Outcome, дает самое компактное и прямолинейное решение (и вероятно, самое быстрое, хотя следует помнить, что EnumMap использует внутренний массив):


```
//: enumerated/RoShamBo6.java
// Перечисления используют "таблицы"
// вместо множественной диспетчеризации.
package enumerated;
import static enumerated.Outcome.*;

enum RoShamBo6 implements Competitor<RoShamBo6> {
    PAPER, SCISSORS, ROCK;
    private static Outcome[][] table = {
        { DRAW, LOSE, WIN }, // PAPER
        { WIN, DRAW, LOSE }, // SCISSORS
        { LOSE, WIN, DRAW }, // ROCK
    };
    public Outcome compete(RoShamBo6 other) {
        return table[this.ordinal()][other.ordinal()];
    }
    public static void main(String[] args) {
        RoShamBo.play(RoShamBo6.class, 20);
    }
} ///:~
```

Данные `table` следуют точно в таком же порядке, как и при вызовах `initRow()` в предыдущем примере.

Этот пример выгодно выделяется на фоне предыдущих примеров своей компактностью, он понятен и прост в сопровождении. С другой стороны, он не так «безопасен», как предыдущие примеры, потому что в нем используется массив. При больших массивах можно ошибиться с размером, и если ваши тесты не покрывают все возможные комбинации, какие-то ошибки могут быть упущены.

Все эти решения используют разные виды таблиц, но вам стоит поближе познакомиться со способами представления табличных структур и найти тот, который лучше всего подходит для ваших целей. Учтите, что при всей компактности последнего решения его гибкость оставляет желать лучшего, потому что оно выдает только постоянный результат для постоянных входных данных. Впрочем, ничто не мешает вам создать таблицу, выдающую объект функции. В некоторых видах задач концепция «табличного кода» оказывается чрезвычайно мощной.

Резюме

Хотя перечислимые типы сами по себе не особенно сложны, я решил разместить эту главу во второй половине книги из-за интересных возможностей, которые открываются при использовании перечислений в сочетании с такими возможностями, как полиморфизм, обобщения и отражения.

Перечисления Java намного функциональнее перечислений C или C++, хотя они все еще остаются «второстепенной» возможностью, без которой язык обходился (пусть и с некоторыми неудобствами) в течение многих лет. Тем не менее в этой главе представлены некоторые важные применения этой «второстепенной» возможности — иногда перечисления предоставляют все необходимое для элегантного, наглядного решения задачи. А как я указывал, элегантность важна, а наглядность может определять различия

между успешным решением и решением, которое завершается неудачей из-за того, что оно оказалось непонятным для других.

К сожалению, в Java 1.0 термин «перечисление» (enumeration) использовался вместо общепринятого термина «итератор» для обозначения объекта, выбирающего каждый элемент последовательности (см. главу 17). С тех пор ошибка была исправлена, но конечно, интерфейс `Enumeration` нельзя было просто убрать, поэтому он до сих пор продолжает встречаться в старом (а иногда и новом!) коде, библиотеках и документации.

20

Аннотации

Аннотации (также называемые метаданными) — формализованный механизм включения в код информации, которую позднее можно легко и удобно использовать в программе¹.

Появление аннотаций отчасти объясняется общей тенденцией к объединению метаданных с файлами исходного кода (вместо их хранения во внешних документах). Другая причина — ответ на расширение функциональности других языков (таких, как C#).

Аннотации стали одним из самых важных языковых нововведений Java SE. Они предоставляют информацию, которая необходима для полного описания программы, но не может быть выражена в Java. Таким образом, аннотации позволяют сохранить дополнительную информацию о программе в формате, который может быть проверен компилятором. Аннотации могут использоваться для генерирования файлов описаний и даже определений новых классов; они избавляют разработчика от хлопот с написанием «шаблонного» кода. При помощи аннотаций можно держать метаданные в исходном коде Java и пользоваться преимуществами более понятного кода, проверки типов во время компиляции и API, упрощающего построение средств обработки аннотаций. И хотя в Java SE5 есть несколько предопределенных видов метаданных, в общем случае вы сами решаете, какую информацию добавить и что с ней делать.

Синтаксис аннотаций прост; в основном он сводится к включению в язык знака @. Java SE5 содержит три встроенные аннотации общего назначения, определенные в `java.lang`.

- ❑ `@Override` — означает, что определение метода предназначено для переопределения метода базового класса. При опечатке в имени метода или ошибочной сигнатуре компилятор выдает сообщение об ошибке².
- ❑ `@Deprecated` — заставляет компилятор выдать предупреждение при использовании этого элемента.

¹ Джереми Мейер приехал в Crested Butte и провел две недели со мной, совместно работая над этой главой. Его помощь была неоценимой.

² Несомненно, эта аннотация была создана по образцу аналогичной возможности C# (хотя в C# это ключевое слово, а не аннотация; таким образом, при переопределении метода в C# необходимо использовать ключевое слово `override`, тогда как в Java аннотация `@Override` не является обязательной).

- ❑ `@SuppressWarnings` — подавляет неподходящие предупреждения компилятора. Эта аннотация разрешена, но не поддерживалась в ранних версиях Java SE (в которых она игнорировалась).

Четыре дополнительных типа аннотаций поддерживают создание новых аннотаций; вы узнаете о них позднее в этой главе.

Практически везде, где вам приходится создавать дескрипторные классы или интерфейсы, требующие рутинной работы, процесс обычно можно автоматизировать и упростить при помощи аннотаций. Например, большая часть лишней работы в Enterprise JavaBeans (EJB) исключается благодаря разумному применению аннотаций в EJB3.0.

Аннотации могут заменять существующие системы (такие, как XDoclet — независимая библиотека, см. <http://MindView.net/Books/BetterJava>), предназначенные для создания доклетов. Напротив, аннотации являются полноценными языковыми конструкциями, а следовательно, структурируются и подвергаются проверке типов во время компиляции. Хранение всей информации в исходном коде, а не в комментариях делает код более аккуратным и упрощает его сопровождение. Как будет показано в этой главе, использование и расширение инструментария и API аннотаций в сочетании с внешними библиотеками обработки байт-кода открывает нетривиальные возможности анализа и обработки как исходного кода программы, так и байт-кода.

Базовый синтаксис

В следующем примере метод `testExecute()` снабжается аннотацией `@Test`. Сама по себе эта аннотация ничего не делает, но компилятор проследит за тем, чтобы определение аннотации `@Test` присутствовало в пути построения. Как будет показано позднее в этой главе, вы можете написать программу, которая будет выполнять этот метод через механизм отражения.

```
//: annotations/Testable.java
package annotations;
import net.mindview.atunit.*;
public class Testable {
    public void execute() {
        System.out.println("Executing..");
    }
    @Test void testExecute() { execute(); }
} ///:~
```

Методы, снабженные аннотациями, не отличаются от других методов. Аннотация `@Test` в этом примере может использоваться в сочетании с другими модификаторами — такими, как `public`, `static` или `void`. С точки зрения синтаксиса аннотации используются почти так же, как модификаторы.

Определение аннотаций

Ниже приведено определение предыдущей аннотации. Как видите, определения аннотаций во многом похожи на определения интерфейсов. Более того, они компилируются в файлы классов, как и интерфейсы Java:

```
//: net/mindview/atunit/Test.java
// The @Test tag.
package net.mindview.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {} ///:~
```

Если не считать знака @, определение @Test сильно напоминает определение пустого интерфейса. Определение аннотации также требует мета-аннотаций @Target и @Retention. @Target определяет, к чему может применяться данная аннотация (например, к методу или к полю), а @Retention определяет, доступны ли аннотации в исходном коде (SOURCE), в файлах классов (CLASS) или во время выполнения (RUNTIME).

Аннотации обычно содержат элементы, задающие некоторые значения. Программы или утилиты могут использовать эти параметры при обработке аннотаций. Элементы похожи на методы интерфейсов, не считая того, что они могут объявлять значения по умолчанию.

Аннотация без элементов (такая, как приведенная выше аннотация @Test) называется *маркерной аннотацией*.

Ниже приведена простая аннотация для отслеживания сценариев использования проекта. Программисты помечают аннотацией каждый метод или набор методов, соответствующих требованиям конкретного сценария. Руководитель проекта может оценить ход работы над проектом, подсчитав количество реализованных сценариев, а разработчики, сопровождающие проект, смогут легко найти нужный сценарий при необходимости обновления или отладки бизнес-правил в системе.

```
//: annotations/UseCase.java
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface UseCase {
    public int id();
    public String description() default "no description";
} ///:~
```

Обратите внимание: id и description напоминают объявления методов. Так как id проходит проверку типов, осуществляемую компилятором, это надежный механизм связывания поисковой базы данных с документом, содержащим описание сценариев использования, и исходным кодом. Элемент description имеет значение по умолчанию, которое будет использовано обработчиком аннотаций в том случае, если при снабжении метода аннотацией значение не указано.

Следующий класс содержит три метода, помеченных аннотацией @UseCase:

```
//: annotations/PasswordUtils.java
import java.util.*;

public class PasswordUtils {
    @UseCase(id = 47, description =
        "Passwords must contain at least one numeric")
    public boolean validatePassword(String password) {
```

продолжение ➤

```

    return (password.matches("\\w*\\d\\w*"));
}
@UseCase(id = 48)
public String encryptPassword(String password) {
    return new StringBuilder(password).reverse().toString();
}
@UseCase(id = 49, description =
    "New passwords can't equal previously used ones")
public boolean checkForNewPassword(
    List<String> prevPasswords, String password) {
    return !prevPasswords.contains(password);
}
} ///:~

```

Значения элементов аннотаций выражаются парами «имя-значение» в круглых скобках после объявления `@UseCase`. В аннотации `encryptPassword()` не передается значение элемента `description`, поэтому при передаче класса обработчику аннотаций будет использовано значение по умолчанию, определенное в `@interface UseCase`.

Например, такой системой можно воспользоваться для создания «эскиза» вашей системы с последующим заполнением ее функциональности в процессе построения.

Мета-аннотации

В настоящее время в языке Java определены всего три стандартные аннотации (см. выше) и четыре *мета-аннотации*, предназначенные для пометки аннотаций.

@Target	Цель, к которой может применяться аннотация. Допустимые аргументы <code>ElementType</code>: CONSTRUCTOR — объявление конструктора FIELD — объявление поля (включая константы перечислений) LOCAL_VARIABLE — объявление локальной переменной METHOD — объявление метода PACKAGE — объявление пакета PARAMETER — объявление параметра TYPE — объявление класса, интерфейса (включая тип аннотации) или перечисления
@Retention	Продолжительность хранения информации. Допустимые аргументы <code>RetentionPolicy</code>: SOURCE — аннотации игнорируются компилятором CLASS — аннотации доступны в файле класса, но могут игнорироваться виртуальной машиной RUNTIME — аннотации сохраняются виртуальной машиной на стадии выполнения, что позволяет читать их посредством отражения
@Documented	Аннотация включается в Javadocs
@Inherited	Субклассы могут наследовать родительские аннотации

В большинстве случаев вы будете определять собственные аннотации и писать для них собственные обработчики.

Написание обработчиков аннотаций

Без средств чтения аннотации вряд ли принесут больше пользы, чем комментарии. Важной частью процесса работы с аннотациями является создание и использование обработчиков аннотаций. В Java SE5 появились расширения API отражения, упрощающие создание таких средств. Также имеется внешняя программа `apt`, которая помогает разбирать исходный код Java с аннотациями.

Ниже приведен очень простой обработчик аннотаций, который читает класс `PasswordUtils`, снабженный аннотациями, и использует отражение для поиска меток `@UseCase`. По списку значений `id` он выводит перечень найденных сценариев использования и сообщает обо всех отсутствующих сценариях:

```
//: annotations/UseCaseTracker.java
import java.lang.reflect.*;
import java.util.*;

public class UseCaseTracker {
    public static void
    trackUseCases(List<Integer> useCases, Class<?> cl) {
        for(Method m : cl.getDeclaredMethods()) {
            UseCase uc = m.getAnnotation(UseCase.class);
            if(uc != null) {
                System.out.println("Found Use Case:" + uc.id() +
                    " " + uc.description());
                useCases.remove(new Integer(uc.id()));
            }
        }
        for(int i : useCases) {
            System.out.println("Warning: Missing use case-" + i);
        }
    }
    public static void main(String[] args) {
        List<Integer> useCases = new ArrayList<Integer>();
        Collections.addAll(useCases, 47, 48, 49, 50);
        trackUseCases(useCases, PasswordUtils.class);
    }
} /* Output:
Found Use Case:47 Passwords must contain at least one
numeric
Found Use Case:48 no description
Found Use Case:49 New passwords can't equal previously used
ones
Warning: Missing use case-50
*///:~
```

В этом коде используются метод отражения `getDeclaredMethods()` и метод `getAnnotation()` из интерфейса `AnnotatedElement` (этот интерфейс реализуется такими классами, как `Class`, `Method` и `Field`). Он возвращает объект аннотации заданного типа — в данном случае `UseCase`. При отсутствии аннотаций нужного типа у метода возвращается значение `null`. Значения элементов извлекаются вызовами `id()` и `description()`. Помните, что в аннотации метода `encryptPassword()` значение `description` не задано, поэтому при вызове метода `description()` для этой конкретной аннотации обработчик находит значение по умолчанию «no description».

Элементы аннотаций

Метка `@UseCase`, определенная в `UseCase.java`, содержит элемент `id` типа `int` и элемент `description` типа `String`. Ниже перечислены допустимые типы элементов аннотаций.

- ☐ Все примитивы (`int`, `float`, `boolean` и т. д.).
- ☐ `String`.
- ☐ `Class`.
- ☐ `enum`.
- ☐ `Annotation`.
- ☐ Массивы всех перечисленных типов.

При попытке использования любого другого типа компилятор сообщает об ошибке. Использовать классы-«обертки» запрещено, но благодаря автоматической упаковке это не является серьезным ограничением. Также элементы сами по себе могут быть аннотациями. Как вы вскоре убедитесь, вложение аннотаций бывает очень полезным.

Ограничения значений по умолчанию

Компилятор ревностно следит за значениями элементов по умолчанию. Ни один элемент не может иметь неопределенного значения. Это означает, что либо элемент должен иметь значение по умолчанию, либо значения должны предоставляться классом, который использует аннотацию.

Также существует еще одно ограничение: элементы не-примитивных типов не могут получать значение `null` (либо при объявлении в исходном коде, либо при определении значения по умолчанию в интерфейсе аннотации). Это усложняет написание обработчика, который срабатывает по присутствию или отсутствию элемента, потому что каждый элемент фактически присутствует в каждом объявлении аннотации. Проблему можно обойти проверкой конкретных значений — например, пустых строк или отрицательных чисел:

```
//: annotations/SimulatingNull.java
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SimulatingNull {
    public int id() default -1;
    public String description() default "";
} ///:~
```

Это типичная идиома в определениях аннотаций.

Генерирование внешних файлов

Аннотации особенно полезны при работе с фреймворками, требующими дополнительной информации для использования вашего исходного кода.

Такие технологии, как `Enterprise JavaBeans` (до выхода `EJB3`), требовали многочисленных интерфейсов и дескрипторов развертывания, которые представляли собой

«шаблонный» код, определявшийся одинаково для всех компонентов. Веб-службы, библиотеки пользовательских тегов и средства объектно-реляционного отображения — такие, как Toplink и Hibernate, — часто требуют применения дескрипторов XML, внешних по отношению к коду. После определения класса Java программист должен проделать рутинную работу по повторному определению информации — имени, пакета и т. д., — уже существующей в исходном классе. Каждый раз, когда вы используете внешний файл дескриптора, у вас появляются два отдельных источника информации о классе, что нередко порождает проблемы с синхронизацией кода. Кроме того, программисты, работающие над проектом, должны уметь не только писать программы на Java, но и редактировать дескриптор.

Допустим, вы хотите предоставить базовую функциональность объектно-реляционного отображения для автоматизации создания таблицы базы данных, предназначенной для хранения компонентов JavaBean. Вы можете использовать файл дескриптора XML для указания имени класса, всех его членов и информации об отображении на базу данных. Однако благодаря аннотациям всю эту информацию можно хранить в исходном файле JavaBean. Для этого вам понадобятся аннотации, определяющие имя таблицы базы данных, связанной с компонентом, столбцы и типы SQL, соответствующие свойствам компонента.

Следующая аннотация сообщает обработчику аннотаций, что он должен создать таблицу базы данных:

```
//: annotations/database/DBTable.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.TYPE) // Applies to classes only
@Retention(RetentionPolicy.RUNTIME)
public @interface DBTable {
    public String name() default "";
} ///:~
```

Каждый аргумент `ElementType` в аннотации `@Target` устанавливает ограничение, которое сообщает компилятору, что ваша аннотация может применяться только к этому конкретному типу. Вы можете указать одно значение из перечисления `ElementType` или же перечислить любую комбинацию значений, разделяя их запятыми. Чтобы аннотация могла применяться к любому типу `ElementType`, аннотацию `@Target` можно опустить, хотя это встречается нечасто.

Обратите внимание: аннотация `@DBTable` содержит элемент `name()`, чтобы аннотация могла предоставить имя таблицы базы данных, которая будет создана обработчиком.

Аннотации для полей JavaBean выглядят так:

```
//: annotations/database/Constraints.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Constraints {
    boolean primaryKey() default false;
```

продолжение ➤

```

    boolean allowNull() default true;
    boolean unique() default false;
} ///:~

//: annotations/database/SQLString.java
package annotations.database;
import java.lang.annotation.*;
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLString {
    int value() default 0;
    String name() default "";
    Constraints constraints() default @Constraints;
} ///:~

//: annotations/database/SQLInteger.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLInteger {
    String name() default "";
    Constraints constraints() default @Constraints;
} ///:~

```

Аннотация `@Constraints` позволяет обработчику получить метаданные, относящиеся к таблице базы данных. Она представляет лишь небольшое подмножество ограничений, обычно предоставляемых базами данных, но по крайней мере общее представление о происходящем. Элементам `primaryKey()`, `allowNull()` и `unique()` присваиваются разумные значения по умолчанию, так что в большинстве случаев пользователю аннотации не придется вводить слишком много текста.

Два других объявления `@interface` определяют типы SQL. Чтобы эта структура реально работала, необходимо определить аннотации для всех дополнительных типов SQL. В нашем примере двух будет достаточно.

У каждого типа имеется элемент `name()` и элемент `constraints()`. Последний использует механизм вложения аннотаций для включения информации об ограничениях типа столбца. Обратите внимание: для элемента `constraints()` задано значение по умолчанию `@Constraints`. Так как после типа аннотации значения элементов в круглых скобках не указаны, значение по умолчанию для `constraints()` в действительности представляет собой аннотацию `@Constraints` с собственным набором значений по умолчанию. Чтобы создать вложенную аннотацию `@Constraints`, у которой по умолчанию ограничение уникальности истинно, можно определить элемент следующим образом:

```

//: annotations/database/Uniqueness.java
// Пример вложенной аннотации
package annotations.database;

public @interface Uniqueness {
    Constraints constraints()
        default @Constraints(unique=true);
} ///:~

```

Простой компонент с использованием этих аннотаций:

```
//: annotations/database/Member.java
package annotations.database;

@DBTable(name = "MEMBER")
public class Member {
    @SQLString(30) String firstName;
    @SQLString(50) String lastName;
    @SQLInteger Integer age;
    @SQLString(value = 30,
        constraints = @Constraints(primaryKey = true))
    String handle;
    static int memberCount;
    public String getHandle() { return handle; }
    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public String toString() { return handle; }
    public Integer getAge() { return age; }
} ///:~
```

Аннотации класса `@DBTable` задано значение «MEMBER», которое будет использовано в качестве имени таблицы. Свойства `firstName` и `lastName` снабжаются аннотациями `@SQLString` и получают значения 30 и 50 соответственно. Эти аннотации интересны по двум причинам: во-первых, они используют значение по умолчанию для вложенной аннотации `@Constraints`, а во-вторых, они используют сокращенную запись. Если для аннотации определен элемент с именем `value`, то при условии, что это единственный заданный тип элемента, соблюдение синтаксиса пар «имя-значение» не обязательно; можно просто указать значение в круглых скобках. Эта возможность может применяться для любых допустимых типов элементов. Конечно, для этого элемент должен называться «value», но в приведенном примере сокращенная запись позволяет использовать семантически содержательную и удобочитаемую аннотацию:

```
@SQLString(30)
```

Обработчик использует это значение для задания размера создаваемого столбца SQL.

Синтаксис значения по умолчанию удобен, но он быстро усложняется. Взгляните на аннотацию поля `handle`. В нем используется аннотация `@SQLString`, но это поле также должно быть первичным ключом базы данных, поэтому для вложенной аннотации `@Constraint` должен быть установлен тип элемента `primaryKey`. Здесь-то и начинаются проблемы. Теперь для вложенной аннотации приходится использовать длинную форму с парой «ключ-значение», с повторным указанием имени элемента и имени `@interface`. Но так как элемент со специальным именем `value` уже не является единственным заданным элементом, воспользоваться сокращенной записью не удастся. Как видите, результат выглядит некрасиво.

Альтернативные решения

Также существуют другие способы создания аннотаций для этой задачи. Например, можно создать один класс аннотации с именем `@TableColumn`, содержащий элемент-перечисление, определяющий значения `STRING`, `INTEGER`, `FLOAT` и т. д. Тем самым устраняется

необходимость в объявлении `@interface` для каждого типа SQL, но зато теряется возможность уточнения типов дополнительными элементами (размер, точность и т. д.) — что, пожалуй, полезнее.

Также можно воспользоваться элементом `String` для описания фактического типа SQL — например, «`VARCHAR(30)`» или «`INTEGER`». Это позволит уточнять типы, но отображение типов Java на типы SQL привязывается к вашему коду, что нежелательно с точки зрения архитектуры. Вряд ли вам захочется перекомпилировать классы при изменении базы данных; было бы элегантнее просто сообщить обработчику аннотаций, что вы используете другую «разновидность» SQL, чтобы этот факт был учтен при обработке аннотаций.

В третьем возможном решении поле помечается сразу двумя типами аннотаций, `@Constraints` и соответствующего типа SQL (например, `@SQLInteger`). Результат выглядит немного неряшливо, но компилятор позволяет назначить для цели аннотации сколько угодно разных аннотаций. Учтите, что при использовании нескольких аннотаций одна аннотация не может быть использована дважды.

Аннотации не поддерживают наследование

Ключевое слово `extends` не может использоваться с `@interface`. Это обидно, потому что элегантное решение могло бы заключаться в определении аннотации `@TableColumn`, как предлагалось выше, с вложенной аннотацией типа `@SQLType`. В этом случае можно было бы сделать все типы SQL — такие, как `@SQLInteger` и `@SQLString`, — производными от `@SQLType`. Это сократило бы объем вводимого кода и сделало бы синтаксис более аккуратным. Пока никакой информации о поддержке наследования аннотациями в будущих версиях нет, поэтому приведенные выше примеры — лучшее, что можно сделать в данных обстоятельствах.

Реализация обработчика

Ниже приведен пример обработчика аннотаций, который читает файл класса, проверяет его на аннотации базы данных и генерирует команду SQL для создания базы данных:

```
//: annotations/database/TableCreator.java
// Обработчик аннотаций с использованием отражения.
// {Args: annotations.database.Member}
package annotations.database;
import java.lang.annotation.*;
import java.lang.reflect.*;
import java.util.*;

public class TableCreator {
    public static void main(String[] args) throws Exception {
        if(args.length < 1) {
            System.out.println("arguments: annotated classes");
            System.exit(0);
        }
        for(String className : args) {
            Class<?> cl = Class.forName(className);
            DBTable dbTable = cl.getAnnotation(DBTable.class);
```

```

if(dbTable == null) {
    System.out.println(
        "No DBTable annotations in class " + className);
    continue;
}
String tableName = dbTable.name();
// Если имя не указано, использовать имя Class:
if(tableName.length() < 1)
    tableName = cl.getName().toUpperCase();
List<String> columnDefs = new ArrayList<String>();
for(Field field : cl.getDeclaredFields()) {
    String columnName = null;
    Annotation[] anns = field.getDeclaredAnnotations();
    if(anns.length < 1)
        continue; // Не является столбцом таблицы базы данных
    if(anns[0] instanceof SQLInteger) {
        SQLInteger sInt = (SQLInteger) anns[0];
        // Использовать имя поля, если имя не указано
        if(sInt.name().length() < 1)
            columnName = field.getName().toUpperCase();
        else
            columnName = sInt.name();
        columnDefs.add(columnName + " INT" +
            getConstraints(sInt.constraints()));
    }
    if(anns[0] instanceof SQLString) {
        SQLString sString = (SQLString) anns[0];
        // Использовать имя поля, если имя не указано
        if(sString.name().length() < 1)
            columnName = field.getName().toUpperCase();
        else
            columnName = sString.name();
        columnDefs.add(columnName + " VARCHAR(" +
            sString.value() + ")" +
            getConstraints(sString.constraints()));
    }
    StringBuilder createCommand = new StringBuilder(
        "CREATE TABLE " + tableName + "(");
    for(String columnDef : columnDefs)
        createCommand.append("\n    " + columnDef + ",");
    // Удалить завершающую запятую
    String tableCreate = createCommand.substring(
        0, createCommand.length() - 1) + ");";
    System.out.println("Table Creation SQL for " +
        className + " is :\n" + tableCreate);
}
}
}

private static String getConstraints(Constraints con) {
    String constraints = "";
    if(!con.allowNull())
        constraints += " NOT NULL";
    if(con.primaryKey())
        constraints += " PRIMARY KEY";
    if(con.unique())
        constraints += " UNIQUE";
    return constraints;
}

```

```

} /* Output:
Table Creation SQL for annotations.database.Member is :
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30));
Table Creation SQL for annotations.database.Member is :
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50));
Table Creation SQL for annotations.database.Member is :
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50),
    AGE INT);
Table Creation SQL for annotations.database.Member is :
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50),
    AGE INT,
    HANDLE VARCHAR(30) PRIMARY KEY);
*///:~

```

Метод `main()` перебирает имена классов, заданные в командной строке. Каждый класс загружается методом `forName()` и проверяется на наличие аннотации `@DBTable` методом `getAnnotation(DBTable.class)`. Если аннотация присутствует, программа находит и сохраняет имя таблицы. Затем все поля класса загружаются и проверяются методом `getDeclaredAnnotations()`. Этот метод возвращает массив всех аннотаций, определенных для метода. Оператор `instanceof` проверяет, относятся ли эти аннотации к типу `@SQLInteger` или `@SQLString`; в каждом случае строится соответствующий фрагмент `String` с именем столбца таблицы. Так как для интерфейсов аннотаций наследование не поддерживается, использование `getDeclaredAnnotations()` является единственным способом моделирования полиморфного поведения.

Вложенная аннотация `@Constraint` передается методу `getConstraints()`, который строит объект `String` с ограничениями SQL.

Необходимо заметить, что продемонстрированный способ определения объектно-реляционных отображений выглядит немного наивно. Аннотация типа `@DBTable`, получающая имя таблицы в параметре, заставляет вас перекомпилировать код Java, если вдруг потребуются сменить имя таблицы, что, возможно, нежелательно. Существует много фреймворков для отображения объектов на реляционные базы данных, и все большая их часть использует аннотации.

1. (2) Реализуйте дополнительные типы SQL в примере с базой данных.

Проект¹. Измените пример с базой данных так, чтобы он подключался к реальной базе данных и взаимодействовал с ней средствами JDBC.

Проект. Измените пример с базой данных так, чтобы он генерировал файлы в формате XML (вместо кода SQL).

¹ Проекты могут использоваться (например) в качестве тем для курсовых работ. Решения для них здесь не приводятся.

Использование apt для обработки аннотаций

Apt (Annotation Processing Tool) — первая версия программы Sun, упрощающей обработку аннотаций. Так как программа появилась относительно недавно, она еще немного примитивна, но некоторые из ее возможностей упростят вашу работу.

Как и *javac*, программа *apt* работает с исходными файлами Java, а не с откомпилированными классами. По умолчанию *apt* компилирует исходные файлы после завершения их обработки. Это удобно в том случае, если новые исходные файлы создаются автоматически в процессе построения. Более того, *apt* проверяет вновь созданные исходные файлы на наличие аннотаций и компилирует их за один проход.

Когда ваш обработчик аннотаций создает новый исходный файл, этот файл проверяется на аннотации в новом *раунде* (как это названо в документации) обработки. Программа продолжает обработку раунд за раундом, пока новые исходные файлы не перестанут появляться. После этого она компилирует все исходные файлы.

Каждой написанной вами аннотации необходим собственный обработчик, но программа *apt* позволяет легко группировать несколько обработчиков. Она позволяет выбрать для обработки сразу несколько классов, что гораздо проще самостоятельного перебора классов *File*. Вы также можете добавлять слушателей, получающих оповещения о завершении очередного раунда обработки аннотаций.

На момент написания книги *apt* не может использоваться как задача Ant (см. приложение <http://MindView.net/Books/BetterJava>), но разумеется, может запускаться в Ant как внешняя задача. Для компиляции обработчиков аннотаций, приведенных в этом разделе, в *CLASSPATH* должна присутствовать библиотека *tools.jar*, которая также содержит интерфейсы *com.sun.mirror.**.

Работа *apt* основана на использовании интерфейса *AnnotationProcessorFactory*, создающего нужный обработчик для каждой найденной аннотации. При запуске *apt* указывается либо класс-фабрика, либо путь, по которому находятся необходимые фабрики. Если этого не сделать, *apt* запускает сложный процесс поиска, подробности которого можно найти в разделе «Developing an Annotation Processor» документации Sun.

При создании обработчика аннотаций для *apt* нельзя использовать средства отражения Java, потому что вы работаете с исходным кодом, а не с откомпилированными классами¹. «Зеркальный»² *mirror API* решает эту проблему, предоставляя средства для просмотра методов, полей и типов в некомпилитованном исходном коде.

Следующая аннотация может использоваться для извлечения открытых методов из класса и преобразования их в интерфейс:

```
//: annotations/ExtractInterface.java
// Обработка аннотаций на базе APT.
package annotations;
import java.lang.annotation.*;
```

продолжение ➤

¹ Впрочем, при использовании нестандартного ключа *-XclassesAsDecls* можно работать с аннотациями из откомпилированных классов.

² Проектировщики Java намекают, что зеркало — то, в чем можно найти отражение.

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface ExtractInterface {
    public String value();
} ///:~
```

В аргументе `RetentionPolicy` выбрано значение `SOURCE`, потому что сохранять эту аннотацию в файле класса после выделения интерфейса бессмысленно. Следующий класс предоставляет открытый метод, который может стать частью полезного интерфейса:

```
//: annotations/Multiplier.java
// Обработка аннотаций на базе APT.
package annotations;

@ExtractInterface("IMultiplier")
public class Multiplier {
    public int multiply(int x, int y) {
        int total = 0;
        for(int i = 0; i < x; i++)
            total = add(total, y);
        return total;
    }
    private int add(int x, int y) { return x + y; }
    public static void main(String[] args) {
        Multiplier m = new Multiplier();
        System.out.println("11*16 = " + m.multiply(11, 16));
    }
} /* Output:
11*16 = 176
*///:~
```

Класс `Multiplier` (работающий только с положительными целыми числами) содержит метод `multiply()`, который многократно вызывает закрытый метод `add()` для выполнения умножения. Метод `add()` не является открытым, поэтому он не входит в интерфейс. В аннотации указано значение `IMultiplier` — имя создаваемого интерфейса.

Теперь нам понадобится обработчик, непосредственно выполняющий извлечение:

```
//: annotations/InterfaceExtractorProcessor.java
// Обработка аннотаций на базе APT.
// {Exec: apt -factory
// annotations.InterfaceExtractorProcessorFactory
// Multiplier.java -s ../annotations}
package annotations;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import java.io.*;
import java.util.*;

public class InterfaceExtractorProcessor
    implements AnnotationProcessor {
    private final AnnotationProcessorEnvironment env;
    private ArrayList<MethodDeclaration> interfaceMethods =
        new ArrayList<MethodDeclaration>();
    public InterfaceExtractorProcessor(
        AnnotationProcessorEnvironment env) { this.env = env; }
    public void process() {
```



```

for(TypeDeclaration typeDecl :
    env.getSpecifiedTypeDeclarations()) {
    ExtractInterface annot =
        typeDecl.getAnnotation(ExtractInterface.class);
    if(annot == null)
        break;
    for(MethodDeclaration m : typeDecl.getMethods())
        if(m.getModifiers().contains(Modifier.PUBLIC) &&
            !(m.getModifiers().contains(Modifier.STATIC)))
            interfaceMethods.add(m);
    if(interfaceMethods.size() > 0) {
        try {
            PrintWriter writer =
                env.getFile().createSourceFile(annot.value());
            writer.println("package " +
                typeDecl.getPackage().getQualifiedName() + ";");
            writer.println("public interface " +
                annot.value() + " {");
            for(MethodDeclaration m : interfaceMethods) {
                writer.print("    public ");
                writer.print(m.getReturnType() + " ");
                writer.print(m.getSimpleName() + " (");
                int i = 0;
                for(ParameterDeclaration parm :
                    m.getParameters()) {
                    writer.print(parm.getType() + " " +
                        parm.getSimpleName());
                    if(++i < m.getParameters().size())
                        writer.print(", ");
                }
                writer.println(");");
            }
            writer.println("}");
            writer.close();
        } catch(IOException ioe) {
            throw new RuntimeException(ioe);
        }
    }
}
}
} ///:~

```

Вся работа выполняется в методе `process()`. Класс `MethodDeclaration` и его метод `getModifiers()` используются для идентификации открытых методов (хотя статические методы при этом игнорируются) обрабатываемого класса. Если такие методы будут обнаружены, они сохраняются в контейнере `ArrayList` и используются для создания методов нового определения интерфейса в файле `.java`.

Обратите внимание на передачу конструктору объекта `AnnotationProcessorEnvironment`. К нему можно обращаться с запросами всех типов (определений классов), обрабатываемых программой *apt*, а также использовать его для получения объекта `Messenger` и объекта `Filer`. Объект `Messenger` позволяет организовать передачу сообщений пользователю (например, сообщений об ошибках, произошедших во время обработки, и об их местонахождении в исходном коде). Объект `Filer` представляет собой разновидность `PrintWriter` для создания новых файлов. Главная причина для использования объекта

`Filer` вместо обычного `PrintWriter` заключается в том, что он позволяет *apt* отслеживать все создаваемые новые файлы, чтобы их можно было проверить на аннотации и откомпилировать при необходимости.

Также можно заметить, что метод `createSourceFile()` открывает обычный выходной поток с правильным именем для класса или интерфейса Java. Поддержка создания языковых конструкций Java отсутствует, так что исходный код Java приходится генерировать с использованием примитивных методов `print()` и `println()`. А это значит, что вы должны следить за парными скобками и синтаксической корректностью своего кода.

Метод `process()` вызывается программой *apt*, которой необходима фабрика, предоставляющая нужный обработчик:

```
//: annotations/InterfaceExtractorProcessorFactory.java
// Обработка аннотаций на базе APT.
package annotations;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import java.util.*;

public class InterfaceExtractorProcessorFactory
    implements AnnotationProcessorFactory {
    public AnnotationProcessor getProcessorFor(
        Set<AnnotationTypeDeclaration> atds,
        AnnotationProcessorEnvironment env) {
        return new InterfaceExtractorProcessor(env);
    }
    public Collection<String> supportedAnnotationTypes() {
        return
            Collections.singleton("annotations.ExtractInterface");
    }
    public Collection<String> supportedOptions() {
        return Collections.emptySet();
    }
} ///:~
```

Интерфейс `AnnotationProcessorFactory` содержит всего три метода. Обработчик предоставляется методом `getProcessorFor()`, получающим множество (`Set`) объявлений типов (классы Java, для которых запускается *apt*) и объект `AnnotationProcessorEnvironment`, который, как вы уже видели, передается обработчику. Два других метода, `supportedAnnotationTypes()` и `supportedOptions()`, позволяют проверить наличие обработчиков для всех аннотаций, найденных *apt*, и поддержку всех параметров, заданных в командной строке. Метод `getProcessorFor()` особенно важен, потому что если вы не вернете полное имя класса своего типа аннотации в коллекции `String`, *apt* предупредит об отсутствии необходимого обработчика и завершится, ничего не сделав.

Обработчик и фабрика находятся в пакете `annotations`, поэтому для приведенной выше структуры каталогов командная строка встраивается в тег комментария «Ехес» в начале `InterfaceExtractorProcessor.java`. Этот комментарий приказывает *apt* использовать класс-фабрику, определенный выше, и обработать файл `Multiplier.java`. Ключ `-s` указывает, что все новые файлы должны создаваться в каталоге `annotations`. Сгенерированный файл `IMultiplier.java`, как нетрудно догадаться по командам `println()` в приведенном обработчике, выглядит так:

```
package annotations;  
public interface IMultiplier {  
    public int multiply (int x, int y);  
}
```

Этот файл также будет откомпилирован *apt*, поэтому вы найдете файл `IMultiplier.class` в том же каталоге.

2. (3) Добавьте поддержку деления в программу извлечения интерфейса.

Использование паттерна «Посетитель» с apt

Обработка аннотаций может оказаться довольно сложной задачей. В приведенном выше примере задействован относительно простой обработчик аннотаций, и он интерпретирует всего одну аннотацию, но даже в этом случае решение выглядит достаточно нетривиально. Чтобы избежать катастрофического нарастания сложности при добавлении аннотаций и обработчиков, *mirror API* предоставляет классы для поддержки паттерна «Посетитель». Это один из классических паттернов проектирования из книги «Приемы объектно-ориентированного проектирования»; также дополнительную информацию о нем можно найти в книге «Thinking in Patterns».

Паттерн «Посетитель» перебирает структуру данных или коллекцию объектов, выполняя операцию с каждым элементом. Структура данных не обязана быть упорядоченной, а операция, выполняемая с каждым объектом, выбирается в соответствии с его типом. Таким образом операции отделяются от самих объектов, то есть новые операции могут добавляться без добавления методов в определения класса.

Описанная схема хорошо подходит для обработки аннотаций, потому что класс Java может рассматриваться как коллекция объектов (`TypeDeclaration`, `FieldDeclaration`, `MethodDeclaration` и т. д.). При использовании программы *apt* с паттерном «Посетитель» вы предоставляете класс `Visitor`, который содержит метод для обработки каждого обнаруженного типа объявлений. Таким образом реализуется соответствующее поведение для аннотаций методов, классов, полей и т. д.

Ниже снова приведена программа-генератор таблиц SQL — на этот раз с использованием фабрики и обработчика, в котором задействован паттерн «Посетитель»:

```
//: annotations/database/TableCreationProcessorFactory.java  
// Пример с базой данных, использующий паттерн "Посетитель".  
// {Exec: apt -factory  
// annotations.database.TableCreationProcessorFactory  
// database/Member.java -s database}  
package annotations.database;  
import com.sun.mirror.apt.*;  
import com.sun.mirror.declaration.*;  
import com.sun.mirror.util.*;  
import java.util.*;  
import static com.sun.mirror.util.DeclarationVisitors.*;
```

```
public class TableCreationProcessorFactory  
    implements AnnotationProcessorFactory {  
    public AnnotationProcessor getProcessorFor(  

```

продолжение ➤

```

    Set<AnnotationTypeDeclaration> atds,
    AnnotationProcessorEnvironment env) {
    return new TableCreationProcessor(env);
}
public Collection<String> supportedAnnotationTypes() {
    return Arrays.asList(
        "annotations.database.DBTable",
        "annotations.database.Constraints",
        "annotations.database.SQLString",
        "annotations.database.SQLInteger");
}
public Collection<String> supportedOptions() {
    return Collections.emptySet();
}
private static class TableCreationProcessor
    implements AnnotationProcessor {
    private final AnnotationProcessorEnvironment env;
    private String sql = "";
    public TableCreationProcessor(
        AnnotationProcessorEnvironment env) {
        this.env = env;
    }
    public void process() {
        for(TypeDeclaration typeDecl :
            env.getSpecifiedTypeDeclarations()) {
            typeDecl.accept(getDeclarationScanner(
                new TableCreationVisitor(), NO_OP));
            sql = sql.substring(0, sql.length() - 1) + " ";
            System.out.println("creation SQL is :\n" + sql);
            sql = "";
        }
    }
    private class TableCreationVisitor
        extends SimpleDeclarationVisitor {
        public void visitClassDeclaration(
            ClassDeclaration d) {
            DBTable dbTable = d.getAnnotation(DBTable.class);
            if(dbTable != null) {
                sql += "CREATE TABLE ";
                sql += (dbTable.name().length() < 1)
                    ? d.getSimpleName().toUpperCase()
                    : dbTable.name();
                sql += " (" ;
            }
        }
        public void visitFieldDeclaration(
            FieldDeclaration d) {
            String columnName = "";
            if(d.getAnnotation(SQLInteger.class) != null) {
                SQLInteger sInt = d.getAnnotation(
                    SQLInteger.class);
                // Использовать имя поля, если имя не указано
                if(sInt.name().length() < 1)
                    columnName = d.getSimpleName().toUpperCase();
                else
                    columnName = sInt.name();
                sql += "\n    " + columnName + " INT" +
                    getConstraints(sInt.constraints()) + "," ;
            }
        }
    }
}

```

```

        if(d.getAnnotation(SQLString.class) != null) {
            SQLString sString = d.getAnnotation(
                SQLString.class);
            // Use field name if name not specified.
            if(sString.name().length() < 1)
                columnName = d.getSimpleName().toUpperCase();
            else
                columnName = sString.name();
            sql += "\n    " + columnName + " VARCHAR(" +
                sString.value() + ")" +
                getConstraints(sString.constraints()) + ",";
        }
    }
    private String getConstraints(Constraints con) {
        String constraints = "";
        if(!con.allowNull())
            constraints += " NOT NULL";
        if(con.primaryKey())
            constraints += " PRIMARY KEY";
        if(con.unique())
            constraints += " UNIQUE";
        return constraints;
    }
}
}
} ///:~

```

Результат выглядит так же, как в предыдущем примере DBTable.

Обработчик и посетитель в этом примере представлены внутренними классами. Обратите внимание: метод process() только добавляет класс посетителя и инициализирует строку SQL.

Оба параметра getDeclarationScanner() определяют посетителей; первый используется перед посещением каждого объявления, а второй — после. Этому обработчику нужен только первый посетитель, поэтому во втором параметре передается NO_OP — статическое поле интерфейса DeclarationVisitor для посетителя, который не делает ничего.

Класс TableCreationVisitor расширяет SimpleDeclarationVisitor, переопределяя два метода, visitClassDeclaration() и visitFieldDeclaration(). SimpleDeclarationVisitor — адаптер, реализующий все методы интерфейса DeclarationVisitor, так что вы можете сосредоточиться на тех методах, которые вам нужны. В методе visitClassDeclaration() объект ClassDeclaration проверяется на аннотацию DBTable, и если она присутствует, инициализируется первая часть строки SQL. В методе visitFieldDeclaration() объявление поля проверяется на аннотации полей, а информация извлекается таким же способом, как в примере, приведенном ранее в этой главе.

На первый взгляд такая схема выглядит излишне сложной, но такое решение лучше масштабируется. Если сложность обработчика аннотаций возрастет, написание автономного обработчика (как в предыдущем примере) также станет весьма непростым делом.

3. (2) Добавьте поддержку других типов SQL в TableCreationProcessorFactory.java.

Использование аннотаций при модульном тестировании

Модульным тестированием (unit testing) называется практика создания тестов для каждого метода класса для регулярного тестирования компонентов класса на правильность поведения. Самый популярный инструмент модульного тестирования для Java называется JUnit; на момент написания книги система JUnit находилась в процессе обновления до версии 4, в которой должны были появиться аннотации¹. Одной из главных проблем версий JUnit до появления аннотаций был объем подготовительного кода, необходимого для создания и выполнения тестов JUnit. Со временем он сократился, но поддержка аннотаций становится шагом на пути к заданной цели — «самой простой системе модульного тестирования, которая работает».

В версиях JUnit до появления аннотаций для хранения модульных тестов приходилось создавать отдельный класс. С аннотациями модульные тесты можно включить в тестируемый класс, что позволяет свести к минимуму затраты времени и сил на модульное тестирование. У такого подхода есть дополнительное преимущество: он позволяет тестировать закрытые методы так же легко, как и открытые.

Так как эта среда тестирования базируется на аннотациях, она называется *@Unit*. Простейшая форма тестирования (с которой вам, вероятно, в основном придется иметь дело) использует аннотацию *@Test* для пометки тестируемых методов. Одна из разновидностей тестовых методов не получает аргументов и возвращает логический признак успеха или неудачи. Вы можете присваивать тестовым методам любые имена по своему усмотрению. Кроме того, тестовые методы *@Unit* могут обладать любым уровнем доступа, включая *private*.

Чтобы использовать *@Unit*, достаточно импортировать библиотеку *net.mindview.atunit*², пометить соответствующие методы и поля маркерами *@Unit* (о которых вы больше узнаете в следующих примерах), после чего заставить вашу систему построения выполнить *@Unit* для полученного класса. Простой пример:

```
//: annotations/AtUnitExample1.java
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample1 {
    public String methodOne() {
        return "This is methodOne";
    }
    public int methodTwo() {
        System.out.println("This is methodTwo");
        return 2;
    }
}
```

¹ Когда-то я думал о создании «улучшенного варианта JUnit» на основании идей, представленных в этом разделе. Однако оказалось, что многие из представленных идей были реализованы в JUnit4, так что сейчас проще пользоваться тем, что есть.

² Библиотека включена в пакет кода этой книги (*www.MindView.net*).

```

@Test boolean methodOneTest() {
    return methodOne().equals("This is methodOne");
}
@Test boolean m2() { return methodTwo() == 2; }
@Test private boolean m3() { return true; }
// Shows output for failure:
@Test boolean failureTest() { return false; }
@Test boolean anotherDisappointment() { return false; }
public static void main(String[] args) throws Exception {
    OSExecute.command(
        "java net.mindview.atunit.AtUnit AtUnitExample1");
}
} /* Output:
annotations.AtUnitExample1
. methodOneTest
. m2 This is methodTwo

. m3
. failureTest (failed)
. anotherDisappointment (failed)
(5 tests)

>>> 2 FAILURES <<<
annotations.AtUnitExample1: failureTest
annotations.AtUnitExample1: anotherDisappointment
*///:~

```

Тестируемые классы должны быть помещены в пакеты.

Аннотация `@Test` перед методами `methodOneTest()`, `m2()`, `m3()`, `failureTest()` и `anotherDisappointment()` приказывает `@Unit` выполнить эти методы как модульные тесты. Она также гарантирует, что эти методы не получают аргументов и возвращают `boolean` или `void`. При написании модульных тестов вам остается лишь проверить, успешно ли выполнялся тест и вернул ли он `true` или `false` соответственно (для методов, возвращающих `boolean`).

Читатели, знакомые с JUnit, обратят внимание на более содержательный вывод `@Unit` — вы видите, какой тест выполняется в данный момент, поэтому результаты приносят больше пользы, а в конце выводится информация о классах и тестах, выполнение которых завершилось неудачей.

Тестовые методы не обязательно встраивать в ваши классы, если вас это почему-либо не устраивает. Простейший способ создания не-встроенных тестов основан на наследовании:

```

//: annotations/AtUnitExternalTest.java
// Создание не-встроенных тестов.
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExternalTest extends AtUnitExample1 {
    @Test boolean _methodOne() {
        return methodOne().equals("This is methodOne");
    }
    @Test boolean _methodTwo() { return methodTwo() == 2; }
}

```

продолжение ➤

```

public static void main(String[] args) throws Exception {
    OSExecute.command(
        "java net.mindview.atunit.AtUnit AtUnitExternalTest");
}
} /* Output:
annotations.AtUnitExternalTest
. _methodOne
. _methodTwo This is methodTwo

OK (2 tests)
*///:~

```

Пример также демонстрирует полезность гибкой схемы выбора имен (в отличие от требования JUnit, согласно которому все тесты должны начинаться с префикса «test»). В приведенном примере методам `@Test`, выполняющим непосредственное тестирование другого метода, присваивается имя, состоящее из символа подчеркивания и имени тестируемого метода (я не утверждаю, что это идеальное решение, а только демонстрирую возможность).

Также для создания не-встроенных тестов можно воспользоваться композицией:

```

//: annotations/AtUnitComposition.java
// Создание не-встроенных тестов.
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitComposition {
    AtUnitExample1 testObject = new AtUnitExample1();
    @Test boolean _methodOne() {
        return
            testObject.methodOne().equals("This is methodOne");
    }
    @Test boolean _methodTwo() {
        return testObject.methodTwo() == 2;
    }
    public static void main(String[] args) throws Exception {
        OSExecute.command(
            "java net.mindview.atunit.AtUnit AtUnitComposition");
    }
} /* Output:
annotations.AtUnitComposition
. _methodOne
. _methodTwo This is methodTwo

OK (2 tests)
*///:~

```

Для каждого теста создается новый объект `testObject`, так как для каждого теста создается объект `AtUnitComposition`.

В отличие от JUnit здесь нет специальных методов `assert`, но вторая форма `@Test` позволяет вернуть `void` (или `boolean`, если вы предпочитаете возвращать `true` или `false`). Для проверки успешного выполнения можно использовать команды Java `assert`. Обычно они должны включаться ключом `-ea` командной строки `java`, но `@Unit` включает их автоматически. Для обозначения неудачи даже можно воспользоваться исключением. Одной из целей проектирования `@Unit` был минимальный объем дополнительного

синтаксиса, а `assert` и исключения — все, что необходимо для передачи информации об ошибках. Неудачный результат `assert` или возбуждение исключения тестовым методом интерпретируется как неудавшийся тест, но `@Unit` в этом случае не прерывает выполнение — оно продолжается, пока не будут выполнены все тесты.

Пример:

```
//: annotations/AtUnitExample2.java
// В методах @Test могут использоваться
// команды assert и исключения.
package annotations;
import java.io.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample2 {
    public String methodOne() {
        return "This is methodOne";
    }
    public int methodTwo() {
        System.out.println("This is methodTwo");
        return 2;
    }
    @Test void assertExample() {
        assert methodOne().equals("This is methodOne");
    }
    @Test void assertFailureExample() {
        assert 1 == 2: "What a surprise!";
    }
    @Test void exceptionExample() throws IOException {
        new FileInputStream("nofile.txt"); // Возбуждает исключение
    }
    @Test boolean assertAndReturn() {
        // Assert с сообщением:
        assert methodTwo() == 2: "methodTwo must equal 2";
        return methodOne().equals("This is methodOne");
    }
    public static void main(String[] args) throws Exception {
        OSExecute.command(
            "java net.mindview.atunit.AtUnit AtUnitExample2");
    }
} /* Output:
annotations.AtUnitExample2
. assertExample
. assertFailureExample java.lang.AssertionError: What a
surprise!
(failed)
. exceptionExample java.io.FileNotFoundException:
nofile.txt (The system cannot find the file specified)
(failed)
. assertAndReturn This is methodTwo

(4 tests)

>>> 2 FAILURES <<<
annotations.AtUnitExample2: assertFailureExample
annotations.AtUnitExample2: exceptionExample
*///:~
```

Пример использования не-встроенных тестов с проверками `assert` и выполнением простых тестов с `java.util.HashSet`:

```
//: annotations/HashSetTest.java
package annotations;
import java.util.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class HashSetTest {
    HashSet<String> testObject = new HashSet<String>();
    @Test void initialization() {
        assert testObject.isEmpty();
    }
    @Test void _contains() {
        testObject.add("one");
        assert testObject.contains("one");
    }
    @Test void _remove() {
        testObject.add("one");
        testObject.remove("one");
        assert testObject.isEmpty();
    }
    public static void main(String[] args) throws Exception {
        OSExecute.command(
            "java net.mindview.atunit.AtUnit HashSetTest");
    }
} /* Output:
annotations.HashSetTest
. initialization
. _remove
. _contains
OK (3 tests)
*///:~
```

Вариант с наследованием выглядит проще (при отсутствии других ограничений).

4. (3) Убедитесь в том, что перед каждым тестом создается новый объект `testObject`.
5. (1) Измените предыдущее упражнение так, чтобы в нем использовалось наследование.
6. (1) Протестируйте контейнер `LinkedList` по образцу `HashSetTest.java`.
7. (1) Измените предыдущее упражнение так, чтобы в нем использовалось наследование.

Для каждого модульного теста `@Unit` создает объект тестируемого класса с использованием конструктора по умолчанию. Для этого объекта вызывается тест, после чего объект уничтожается, чтобы предотвратить возможное проникновение побочных эффектов в другие модульные тесты. Объекты создаются конструктором по умолчанию. Если у вас нет конструктора по умолчанию или вам требуется более сложное построение объектов, создайте статический метод для построения объекта и присоедините аннотацию `@TestObjectCreate`:

```
//: annotations/AtUnitExample3.java
package annotations;
import net.mindview.atunit.*;
```

```
import net.mindview.util.*;

public class AtUnitExample3 {
    private int n;
    public AtUnitExample3(int n) { this.n = n; }
    public int getN() { return n; }
    public String methodOne() {
        return "This is methodOne";
    }
    public int methodTwo() {
        System.out.println("This is methodTwo");
        return 2;
    }
    @TestObjectCreate static AtUnitExample3 create() {
        return new AtUnitExample3(47);
    }
    @Test boolean initialization() { return n == 47; }
    @Test boolean methodOneTest() {
        return methodOne().equals("This is methodOne");
    }
    @Test boolean m2() { return methodTwo() == 2; }
    public static void main(String[] args) throws Exception {
        OSExecute.command(
            "java net.mindview.atunit.AtUnit AtUnitExample3");
    }
} /* Output:
annotations.AtUnitExample3
. initialization
. methodOneTest
. m2 This is methodTwo
```

OK (3 tests)

*///:~

Метод `@TestObjectCreate` должен быть статическим и должен возвращать объект тестируемого типа — программа `@Unit` проследит за выполнением этого требования.

Иногда для поддержки модульного тестирования нужны дополнительные поля. Аннотация `@TestProperty` может использоваться для пометки полей, которые используются только для модульного тестирования (чтобы их можно было удалить перед отправкой продукта клиенту). Следующий пример читает значения из объекта `String`, разбиваемого методом `String.split()`. Входные данные используются для создания тестовых объектов:

```
//: annotations/AtUnitExample4.java
package annotations;
import java.util.*;
import net.mindview.atunit.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;
public class AtUnitExample4 {
    static String theory = "All brontosauruses " +
        "are thin at one end, much MUCH thicker in the " +
        "middle, and then thin again at the far end.";
    private String word;
    private Random rand = new Random(); // Случайная раскрутка
    public AtUnitExample4(String word) { this.word = word; }
```

продолжение ➤

```

public String getWord() { return word; }
public String scrambleWord() {
    List<Character> chars = new ArrayList<Character>();
    for(Character c : word.toCharArray())
        chars.add(c);
    Collections.shuffle(chars, rand);
    StringBuilder result = new StringBuilder();
    for(char ch : chars)
        result.append(ch);
    return result.toString();
}
@TestProperty static List<String> input =
    Arrays.asList(theory.split(" "));
@TestProperty
    static Iterator<String> words = input.iterator();
@TestObjectCreate static AtUnitExample4 create() {
    if(words.hasNext())
        return new AtUnitExample4(words.next());
    else
        return null;
}
@Test boolean words() {
    print("'" + getWord() + "'");
    return getWord().equals("are");
}
@Test boolean scramble1() {
    // Замена инициализацией конкретным значением
    // для получения воспроизводимых результатов:
    rand = new Random(47);
    print("'" + getWord() + "'");
    String scrambled = scrambleWord();
    print(scrambled);
    return scrambled.equals("lAl");
}
@Test boolean scramble2() {
    rand = new Random(74);
    print("'" + getWord() + "'");
    String scrambled = scrambleWord();
    print(scrambled);
    return scrambled.equals("tsaeborornussu");
}
public static void main(String[] args) throws Exception {
    System.out.println("starting");
    OSExecute.command(
        "java net.mindview.atunit.AtUnit AtUnitExample4");
}
} /* Output:
starting
annotations.AtUnitExample4
    . scramble1 'All'
lAl

    . scramble2 'brontosaurus'
tsaeborornussu

    . words 'are'

OK (3 tests)
*///:~

```

Аннотация `@TestProperty` также может использоваться для пометки методов, которые могут использоваться при тестировании, но сами тестами не являются.

Обратите внимание: программа зависит от порядка выполнения тестов, что в общем случае нежелательно.

Если при создании тестового объекта выполняется инициализация, требующая завершающих действий на более поздней стадии, можно добавить статический метод `@TestObjectCleanup`, который отработает после завершения работы с тестовым объектом. В следующем примере `@TestObjectCreate` открывает файл для создания каждого тестового объекта, поэтому перед уничтожением тестового объекта файл необходимо закрыть:

```
//: annotations/AtUnitExample5.java
package annotations;
import java.io.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample5 {
    private String text;
    public AtUnitExample5(String text) { this.text = text; }
    public String toString() { return text; }
    @TestProperty static PrintWriter output;
    @TestProperty static int counter;
    @TestObjectCreate static AtUnitExample5 create() {
        String id = Integer.toString(counter++);
        try {
            output = new PrintWriter("Test" + id + ".txt");
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
        return new AtUnitExample5(id);
    }
    @TestObjectCleanup static void
    cleanup(AtUnitExample5 tobj) {
        System.out.println("Running cleanup");
        output.close();
    }
    @Test boolean test1() {
        output.print("test1");
        return true;
    }
    @Test boolean test2() {
        output.print("test2");
        return true;
    }
    @Test boolean test3() {
        output.print("test3");
        return true;
    }
    public static void main(String[] args) throws Exception {
        OSExecute.command(
            "java net.mindview.atunit.AtUnit AtUnitExample5");
    }
} /* Output:
annotations.AtUnitExample5
. test1
```

продолжение ➤

```
Running cleanup
  . test2
Running cleanup
  . test3
Running cleanup
OK (3 tests)
*///:~
```

Из результатов видно, что завершающий метод автоматически выполняется после каждого теста.

Использование @Unit с обобщениями

Обобщения представляют особую проблему, потому что «обобщенное тестирование» невозможно — при тестировании должен быть задан конкретный параметр-тип или набор параметров. Проблема решается просто: тестовый класс создается наследованием от заданной версии обобщенного класса. Ниже приведена простая реализация стека:

```
//: annotations/StackL.java
// Стек на базе LinkedList.
package annotations;
import java.util.*;

public class StackL<T> {
    private LinkedList<T> list = new LinkedList<T>();
    public void push(T v) { list.addFirst(v); }
    public T top() { return list.getFirst(); }
    public T pop() { return list.removeFirst(); }
} ///:~
```

Чтобы протестировать версию для String, определите тестовый класс производным от StackL<String>:

```
//: annotations/StackLStringTest.java
// Применение @Unit к обобщениям.
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class StackLStringTest extends StackL<String> {
    @Test void _push() {
        push("one");
        assert top().equals("one");
        push("two");
        assert top().equals("two");
    }
    @Test void _pop() {
        push("one");
        push("two");
        assert pop().equals("two");
        assert pop().equals("one");
    }
    @Test void _top() {
        push("A");
        push("B");
        assert top().equals("B");
    }
}
```

```

    assert top().equals("B");
}
public static void main(String[] args) throws Exception {
    OSExecute.command(
        "java net.mindview.atunit.AtUnit StackLStringTest");
}
} /* Output:
annotations.StackLStringTest
    . _push
    . _pop
    . _top
OK (3 tests)
*///:~

```

Единственный потенциальный недостаток наследования — потеря возможности обращения к закрытым методам тестируемого класса. Если это создает проблемы, либо сделайте соответствующий метод защищенным, либо добавьте не-закрытый метод `@TestProperty`, который вызывает закрытый метод (метод `@TestProperty` будет исключен из окончательного кода программой `AtUnitRemover`, которая будет представлена позднее в этой главе).

8. (2) Создайте класс с закрытым методом и добавьте не-закрытый метод `@TestProperty` так, как описано выше. Вызовите этот метод в своем тестовом коде.
9. (2) Напишите простейшие тесты `@Unit` для `HashMap`.
10. (2) Выберите какой-нибудь пример из книги и добавьте в него тесты `@Unit`.

«Семейства» не нужны

У `@Unit` есть одно важное преимущество перед `JUnit`: в `@Unit` не нужны «семейства» (suites). В `JUnit` необходимо каким-то образом сообщить программе, что же следует протестировать; так появилось понятие «семейства» для группировки тестов, чтобы программа `JUnit` могла найти их и выполнить тесты.

`@Unit` просто ищет файлы классов, содержащие соответствующие аннотации, после чего выполняет методы `@Test`. Работая над системой тестирования `@Unit`, я стремился сделать ее максимально прозрачной, чтобы люди могли начать пользоваться ею, просто добавляя методы `@Test`, без дополнительного кода или знаний, необходимых для `JUnit` и многих других сред модульного тестирования. Тесты писать и так непросто, поэтому `@Unit` старается по возможности упростить эту работу — чтобы повысить вероятность того, что программист действительно *напишет* положенные тесты.

Реализация `@Unit`

Сначала необходимо определить все типы аннотаций. Они представляют собой простые метки без полей. Аннотация `@Test` была определена в начале главы, а остальные аннотации выглядят так:

```

//: net/mindview/atunit/TestObjectCreate.java
// Аннотация @Unit @TestObjectCreate.
package net.mindview.atunit;
import java.lang.annotation.*;

```

продолжение ➤

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestObjectCreate {} ///:~
```

```
//: net/mindview/atunit/TestObjectCleanup.java
// Аннотация @Unit @TestObjectCleanup.
package net.mindview.atunit;
import java.lang.annotation.*;
```

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestObjectCleanup {} ///:~
```

```
//: net/mindview/atunit/TestProperty.java
// Аннотация @Unit @TestProperty.
package net.mindview.atunit;
import java.lang.annotation.*;
```

```
// Поля и методы могут помечаться как свойства:
@Target({ElementType.FIELD, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface TestProperty {} ///:~
```

Для всех тестов задан режим `RetentionPolicy.RUNTIME`, потому что система `@Unit` должна распознать тесты в откомпилированном коде.

Чтобы реализовать систему выполнения тестов, мы воспользуемся отражением для извлечения аннотаций. На основании полученной информации программа решает, как следует строить тестовые объекты и выполнять тесты. С аннотациями код получается на удивление компактным и прямолинейным:

```
//: net/mindview/atunit/AtUnit.java
// Среда модульного тестирования на базе аннотаций.
// {RunByHand}
package net.mindview.atunit;
import java.lang.reflect.*;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;
public class AtUnit implements ProcessFiles.Strategy {
    static Class<?> testClass;
    static List<String> failedTests= new ArrayList<String>();
    static long testsRun = 0;
    static long failures = 0;
    public static void main(String[] args) throws Exception {
        ClassLoader.getSystemClassLoader()
            .setDefaultAssertionStatus(true); // Включить assert
        new ProcessFiles(new AtUnit(), "class").start(args);
        if(failures == 0)
            print("OK (" + testsRun + " tests)");
        else {
            print("(" + testsRun + " tests)");
            print("\n>>> " + failures + " FAILURE" +
                (failures > 1 ? "S" : "") + " <<<");
            for(String failed : failedTests)
                print(" " + failed);
        }
    }
}
```



```

public void process(File cFile) {
    try {
        String cName = ClassNameFinder.thisClass(
            BinaryFile.read(cFile));
        if(!cName.contains("."))
            return; // Ignore unpackaged classes
        testClass = Class.forName(cName);
    } catch(Exception e) {
        throw new RuntimeException(e);
    }
    TestMethods testMethods = new TestMethods();
    Method creator = null;
    Method cleanup = null;
    for(Method m : testClass.getDeclaredMethods()) {
        testMethods.addIfTestMethod(m);
        if(creator == null)
            creator = checkForCreatorMethod(m);
        if(cleanup == null)
            cleanup = checkForCleanupMethod(m);
    }
    if(testMethods.size() > 0) {
        if(creator == null)
            try {
                if(!Modifier.isPublic(testClass
                    .getDeclaredConstructor().getModifiers())) {
                    print("Error: " + testClass +
                        " default constructor must be public");
                    System.exit(1);
                }
            } catch(NoSuchMethodException e) {
                // Синтезированный конструктор по умолчанию; OK
            }
        print(testClass.getName());
    }
    for(Method m : testMethods) {
        printnb("  " + m.getName() + " ");
        try {
            Object testObject = createTestObject(creator);
            boolean success = false;
            try {
                if(m.getReturnType().equals(boolean.class))
                    success = (Boolean)m.invoke(testObject);
                else {
                    m.invoke(testObject);
                    success = true; // If no assert fails
                }
            } catch(InvocationTargetException e) {
                // Фактическое исключение внутри e:
                print(e.getCause());
            }
            print(success ? "" : "(failed)");
            testsRun++;
            if(!success) {
                failures++;
                failedTests.add(testClass.getName() +
                    ": " + m.getName());
            }
        }
        if(cleanup != null)
            cleanup.invoke(testObject, testObject);
    }
}

```

продолжение ➤

```

    } catch(Exception e) {
        throw new RuntimeException(e);
    }
}
}
static class TestMethods extends ArrayList<Method> {
    void addIfTestMethod(Method m) {
        if(m.getAnnotation(Test.class) == null)
            return;
        if(!(m.getReturnType().equals(boolean.class) ||
            m.getReturnType().equals(void.class)))
            throw new RuntimeException("@Test method " +
                " must return boolean or void");
        m.setAccessible(true); // На случай, если метод закрытый и т.д.
        add(m);
    }
}
private static Method checkForCreatorMethod(Method m) {
    if(m.getAnnotation(TestObjectCreate.class) == null)
        return null;
    if(!m.getReturnType().equals(testClass))
        throw new RuntimeException("@TestObjectCreate " +
            "must return instance of Class to be tested");
    if((m.getModifiers() &
        java.lang.reflect.Modifier.STATIC) < 1)
        throw new RuntimeException("@TestObjectCreate " +
            "must be static.");
    m.setAccessible(true);
    return m;
}
private static Method checkForCleanupMethod(Method m) {
    if(m.getAnnotation(TestObjectCleanup.class) == null)
        return null;
    if(!m.getReturnType().equals(void.class))
        throw new RuntimeException("@TestObjectCleanup " +
            "must return void");
    if((m.getModifiers() &
        java.lang.reflect.Modifier.STATIC) < 1)
        throw new RuntimeException("@TestObjectCleanup " +
            "must be static.");
    if(m.getParameterTypes().length == 0 ||
        m.getParameterTypes()[0] != testClass)
        throw new RuntimeException("@TestObjectCleanup " +
            "must take an argument of the tested type.");
    m.setAccessible(true);
    return m;
}
private static Object createTestObject(Method creator) {
    if(creator != null) {
        try {
            return creator.invoke(testClass);
        } catch(Exception e) {
            throw new RuntimeException("Couldn't run " +
                "@TestObject (creator) method.");
        }
    }
    else { // Использовать конструктор по умолчанию:
        try {
            return testClass.newInstance();
        }
    }
}

```

```

    } catch(Exception e) {
        throw new RuntimeException("Couldn't create a " +
            "test object. Try using a @TestObject method.");
    }
}
}
} ///:~

```

AtUnit.java использует класс ProcessFiles из пакета net.mindview.util. Класс AtUnit реализует интерфейс ProcessFiles.Strategy, включающий метод process(). При этом экземпляр AtUnit может передаваться конструктору ProcessFiles. Второй аргумент конструктора приказывает ProcessFiles искать все файлы с расширением «class».

Если аргумент командной строки не указан, программа обходит текущее дерево каталогов. Также можно передать несколько аргументов, которые могут определять либо файлы классов (с расширением .class или без него), либо каталоги. Так как @Unit автоматически находит тестируемые классы и методы, механизм «семейств» не нужен¹.

Одна из проблем, которую должна решить программа AtUnit.java при обнаружении файлов классов, заключается в том, что из имени файла класса невозможно легко вывести уточненное имя класса (включая пакет). Чтобы получить эту информацию, необходимо проанализировать файл класса — задача нетривиальная, но решаемая². Итак, при обнаружении файла .class этот файл открывается, его двоичные данные читаются и передаются методу ClassNameFinder.thisClass(). Здесь происходит переход в область манипуляций с байт-кодом, так как мы занимаемся фактическим анализом содержимого файла класса:

```

//: net/mindview/atunit/ClassNameFinder.java
package net.mindview.atunit;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class ClassNameFinder {
    public static String thisClass(byte[] classBytes) {
        Map<Integer,Integer> offsetTable =
            new HashMap<Integer,Integer>();
        Map<Integer,String> classNameTable =
            new HashMap<Integer,String>();
        try {
            DataInputStream data = new DataInputStream(
                new ByteArrayInputStream(classBytes));
            int magic = data.readInt(); // 0xcafebabe
            int minorVersion = data.readShort();
            int majorVersion = data.readShort();
            int constant_pool_count = data.readShort();
            int[] constant_pool = new int[constant_pool_count];
            for(int i = 1; i < constant_pool_count; i++) {

```

продолжение ➤

¹ Неясно, почему конструктор по умолчанию тестируемого класса должен быть объявлен открытым (public), но в противном случае вызов newInstance() просто «зависает» (без возбуждения исключения).

² Мы с Джереми Мейером потратили целый день на ее решение.

```

int tag = data.read();
int tableSize;
switch(tag) {
    case 1: // UTF
        int length = data.readShort();
        char[] bytes = new char[length];
        for(int k = 0; k < bytes.length; k++)
            bytes[k] = (char)data.read();
        String className = new String(bytes);
        classNameTable.put(i, className);
        break;
    case 5: // LONG
    case 6: // DOUBLE
        data.readLong(); // Отбросить 8 байт
        i++; // Необходим дополнительный пропуск
        break;
    case 7: // CLASS
        int offset = data.readShort();
        offsetTable.put(i, offset);
        break;
    case 8: // STRING
        data.readShort(); // Отбросить 2 байта
        break;
    case 3: // INTEGER
    case 4: // FLOAT
    case 9: // FIELD_REF
    case 10: // METHOD_REF
    case 11: // INTERFACE_METHOD_REF
    case 12: // NAME_AND_TYPE
        data.readInt(); // Отбросить 4 байта;
        break;
    default:
        throw new RuntimeException("Bad tag " + tag);
}
}
short access_flags = data.readShort();
int this_class = data.readShort();
int super_class = data.readShort();
return classNameTable.get(
    offsetTable.get(this_class)).replace('/', '.');
} catch(Exception e) {
    throw new RuntimeException(e);
}
}
// Демонстрация:
public static void main(String[] args) throws Exception {
    if(args.length > 0) {
        for(String arg : args)
            print(thisClass(BinaryFile.read(new File(arg))));
    } else
        // Обход всего дерева:
        for(File klass : Directory.walk(".", ".*\\.class"))
            print(thisClass(BinaryFile.read(klass)));
}
} ///:~

```

Хотя привести здесь полное объяснение не удастся, каждый файл класса имеет определенный формат, и я попытался назначить осмысленные имена полей блокам данных,

выбираемым из `ByteArrayInputStream`; также вы видите размер каждого компонента по длине данных, читаемых из входного потока. Например, первые 32 бита любого файла класса всегда содержат «волшебное число» `0xcafefabe`, а следующие два поля `short` содержат информацию версии. Пул констант содержит константы программы и также имеет переменный размер; следующее поле `short` сообщает его размер для выделения массива соответствующего размера. Каждая запись в пуле констант может представлять значение фиксированного или переменного размера, поэтому мы должны проанализировать начальный маркер каждой записи, чтобы решить, как ее обрабатывать, — отсюда и команда `switch`. Здесь мы не пытаемся точно проанализировать все данные в файле класса, а всего лишь ищем и сохраняем интересные нас блоки, поэтому большой объем данных просто отбрасывается. Информация о классе хранится в `classNameTable` и `offsetTable`. После чтения пула констант следует поле `this_class`, которое представляет собой индекс в таблице `classNameTable` для получения имени класса.

Возвращаемся к `AtUnit.java`: метод `process()` теперь знает имя класса и может проверить, содержит ли оно символ «.», который означает, что класс принадлежит пакету. Классы без пакетов игнорируются. Если класс содержится в пакете, стандартный загрузчик используется для загрузки класса методом `Class.forName()`. Теперь класс можно проанализировать на аннотации `@Unit`.

Нас интересуют только три вещи: методы `@Test`, хранящиеся в списке `TestMethods`, а также методы `@TestObjectCreate` и `@TestObjectCleanup`. Для их обнаружения используются вызовы методов, приведенные в коде, которые занимаются поиском аннотаций.

Если методы `@Test` будут найдены, программа выводит имя класса, чтобы разработчик понимал, что происходит, после чего выполняется каждый тест. Для этого выводится имя метода, после чего вызывается метод `createTestObject()`, который использует метод `@TestObjectCreate`, если он существует, или конструктор по умолчанию в противном случае. После создания тестового объекта для него вызывается тестовый метод. Если тест возвращает `boolean`, результат сохраняется, а если нет — успех предполагается при отсутствии исключений (которые могут возникать при неудачном вызове `assert` или любой другой причине). Если при выполнении возбуждается исключение, программа выводит информацию о нем. Счетчик ошибок увеличивается, а имена класса и метода добавляются в `failedTests` для вывода в конце выполнения.

11. (5) Добавьте в `@Unit` аннотацию `@TestNote`, чтобы соответствующий комментарий просто выводился во время тестирования.

Удаление тестового кода

Во многих проектах тестовый код можно оставить в конечном продукте (особенно если все тестовые методы объявлены закрытыми), но в некоторых случаях желательно удалить тестовый код — либо чтобы сократить размер продукта, либо чтобы клиент не смог получить к нему доступ.

Для этого требуются слишком сложные манипуляции с байт-кодом, которые неудобно выполнять вручную. Однако библиотека с открытым кодом `Javassist`¹ переводит

¹ Спасибо доктору Сигэру Чибэ за создание этой библиотеки и помощь с созданием `AtUnitRemover.java`.

манипуляции с байт-кодом в область возможного. Следующая программа получает в первом аргументе необязательный флаг -r, который удаляет аннотации @Test (без него аннотации @Test просто выводятся). Также для перебора файлов и каталогов используется вспомогательный класс ProcessFiles:

```
//: net/mindview/atunit/AtUnitRemover.java
// Вывод аннотаций @Unit в откомпилированных файлах классов. Если
// в первом аргументе передается ключ "-r", аннотации @Unit удаляются.
// {Args: ..}
// {Requires: javassist.bytecode.ClassFile;
// You must install the Javassist library from
// http://sourceforge.net/projects/jboss/ }
package net.mindview.atunit;
import javassist.*;
import javassist.expr.*;
import javassist.bytecode.*;
import javassist.bytecode.annotation.*;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;
public class AtUnitRemover
implements ProcessFiles.Strategy {
    private static boolean remove = false;
    public static void main(String[] args) throws Exception {
        if(args.length > 0 && args[0].equals("-r")) {
            remove = true;
            String[] nargs = new String[args.length - 1];
            System.arraycopy(args, 1, nargs, 0, nargs.length);
            args = nargs;
        }
        new ProcessFiles(
            new AtUnitRemover(), "class").start(args);
    }
    public void process(File cFile) {
        boolean modified = false;
        try {
            String cName = ClassNameFinder.thisClass(
                BinaryFile.read(cFile));
            if(!cName.contains("."))
                return; // Классы вне пакетов игнорируются
            ClassPool cPool = ClassPool.getDefault();
            CtClass ctClass = cPool.get(cName);
            for(CtMethod method : ctClass.getDeclaredMethods()) {
                MethodInfo mi = method.getMethodInfo();
                AnnotationsAttribute attr = (AnnotationsAttribute)
                    mi.getAttribute(AnnotationsAttribute.visibleTag);
                if(attr == null) continue;
                for(Annotation ann : attr.getAnnotations()) {
                    if(ann.getTypeName()
                        .startsWith("net.mindview.atunit")) {
                        print(ctClass.getName() + " Method: "
                            + mi.getName() + " " + ann);
                        if(remove) {
                            ctClass.removeMethod(method);
                            modified = true;
                        }
                    }
                }
            }
        }
    }
}
```

```
    }  
  }  
}  
// В этой версии поля не удаляются (см. текст).  
if(modified)  
    ctClass.toByteArray(new DataOutputStream(  
        new FileOutputStream(cFile)));  
ctClass.detach();  
} catch(Exception e) {  
    throw new RuntimeException(e);  
}  
}  
}  
} ///:~
```

`ClassPool` — своего рода «картинка» всех модифицируемых классов в системе. Процесс извлечения объектов `CtClass` из `ClassPool` напоминает загрузку классов в JVM с использованием загрузчика классов и `Class.forName()`.

Объект `CtClass` содержит байт-код объекта класса; он позволяет получить информацию о классе и манипулировать с кодом этого класса. Здесь мы вызываем `getDeclaredMethods()` (по аналогии с механизмом отражения Java) и получаем объект `MethodInfo` для каждого объекта `CtMethod`. Теперь можно переходить к просмотру аннотаций. Если метод содержит аннотацию из пакета `net.mindview.atunit`, то этот метод удаляется.

Если класс был модифицирован, то исходный файл класса заменяется новым классом.

На момент написания книги функциональность «удаления» была только добавлена в `Javassist`¹, и мы обнаружили, что удаление полей `@TestProperty` оказывается более сложной задачей, чем удаление методов. На эти поля могут ссылаться статические операции инициализации, поэтому их нельзя просто удалить. Приведенная выше версия кода удаляет только методы `@Unit`. Впрочем, следите за обновлениями на сайте `Javassist`; возможно, удаление полей со временем станет возможным. А пока следует заметить, что метод внешнего тестирования, продемонстрированный в `AtUnitExternalTest.java`, позволяет удалить все тесты простым удалением файла класса, созданного тестовым кодом.

Резюме

Аннотации стали одним из полезных нововведений в Java. Это структурированный, обеспечивающий проверку типов механизм добавления метаданных, который не загромождает код и не затрудняет его чтение. Аннотации помогают избавиться от рутины, связанной с написанием дескрипторов развертывания и других генерируемых файлов. Замена тега `Javadoc @deprecated` аннотацией `@Deprecated` — лишь один из признаков того, насколько лучше аннотации подходят для передачи информации о классах, чем комментарии.

В поставку Java SE5 включен лишь небольшой набор аннотаций. Это означает, что если найти нужную библиотеку не удастся, вам придется самостоятельно создавать

¹ Доктор Сигэру Чибэ очень любезно добавил метод `CtClass.removeMethod()` по нашей просьбе.

аннотации и связанную с ними логику. При помощи программы *apt* вы можете автоматически откомпилировать только что сгенерированные файлы; это упрощает процесс построения, но в настоящее время *mirror API* фактически содержит лишь базовую функциональность, которая помогает идентифицировать элементы определений классов Java. Как было показано выше, манипуляции с байт-кодом можно запрограммировать вручную или же воспользоваться библиотекой *Javassist*.

Безусловно, со временем ситуация улучшится, и поставщики API и фреймворков начнут включать аннотации в свои инструментарии. Как нетрудно представить по системе *@Unit*, скорее всего, аннотации окажут значительное влияние на практику программирования на Java.

21

Параллельное выполнение

До настоящего момента мы занимались последовательным программированием, при котором все действия в программе выполняются одно за другим.

Последовательное программирование способно решить многие задачи. Однако в некоторых областях бывает удобно (и даже необходимо) выполнять несколько частей программы параллельно, чтобы создать иллюзию их параллельного выполнения — или, если компьютер оснащен несколькими процессорами, действительно выполнять их одновременно.

Параллельное программирование способно серьезно улучшить скорость выполнения и/или предоставить упрощенную модель проектирования для некоторых видов программ. Тем не менее освоение теории и основных приемов параллельного программирования выходит за рамки всего, что вы узнали в этой книге; эта тема лежит где-то между промежуточным и высоким уровнем. Данная глава содержит вводный материал, и вы не можете считать себя хорошим специалистом по параллельному программированию, даже если полностью поймете весь материал.

Как вы вскоре увидите, настоящие проблемы с параллельным выполнением начинаются тогда, когда одновременно выполняемые задачи начинают вмешиваться в дела друг друга. Такое вмешательство бывает настолько тонким и неочевидным, что параллельное выполнение можно с вескими основаниями считать условно-недетерминированным. Другими словами, теоретически возможно написать параллельные программы, которые благодаря внимательности программиста и анализу кода всегда работают корректно. Однако на практике гораздо проще писать параллельные программы, которые на первый взгляд работают корректно, но при некоторых условиях порождают ошибки. Возможно, эти условия никогда не возникнут или же будут возникать так редко, что никогда не встретятся при тестировании. Более того, нет гарантий, что вы сможете написать тестовый код, порождающий сбои в вашей параллельной программе. Ошибки будут происходить в очень редких случаях и в результате обернутся жалобами пользователей. Это одна из самых веских причин для изучения параллельного программирования: если вы не будете уделять ему внимания, то, скорее всего, сами же от этого и пострадаете.

Итак, параллельное программирование сопряжено с неизбежным риском. Если вас это немного пугает, пожалуй, это даже хорошо. Хотя в Java SE5 в области параллельности были достигнуты значительные усовершенствования, до сих пор не существует никакой «страховки» вроде проверки во время компиляции или контролируемых исключений, сообщающих о допущенных ошибках. В области параллельного программирования вы сами отвечаете за все, и только повышенная подозрительность в сочетании с настойчивостью позволит вам писать надежный многопоточный код на Java.

Некоторые люди считают, что параллельное программирование — слишком сложная тема для включения в книгу, которая только знакомит читателя с языком. Они полагают, что эта тема должна излагаться отдельно, а для тех редких случаев, когда она проявляется в повседневном программировании (например, построении графических интерфейсов), можно подобрать специальные идиомы. Зачем излагать такую сложную тему, если ее можно обойти?

Если бы так... К сожалению, вы не властны над тем, где и как в ваших программах Java проявятся программные потоки. Даже если вы никогда не запускаете поток самостоятельно, это не значит, что вам удастся избежать написания многопоточного кода. Например, веб-системы — одна из самых распространенных областей для применения Java, а базовый класс *сервлета* изначально является многопоточным — и это неизбежно, потому что веб-серверы часто оснащаются несколькими процессорами, а параллельность является идеальным способом эффективного использования дополнительных процессоров. Каким бы простым ни казался сервлет, для правильного использования сервлетов необходимо понимать проблемы параллельного программирования. Это относится и к программированию графического интерфейса, как вы увидите в главе 22. Хотя в библиотеках Swing и SWT предусмотрены механизмы потоковой безопасности, вряд ли кто-то сможет правильно пользоваться ими без понимания параллельности.

Java является многопоточным языком, и проблемы параллельного программирования присутствуют независимо от того, знаете вы о них или нет. В результате сейчас используется много Java-приложений, которые либо работают по случайности, либо работают большую часть времени, а иногда непостижимым образом «ломаются» от скрытых дефектов, связанных с параллельным выполнением. Иногда такие поломки приводят к потере ценных данных, и если вы не будете хотя бы в общих чертах представлять проблемы параллельности, то можете прийти к ошибочному выводу, что проблема кроется не в вашей программе, а где-то еще. Такие проблемы также могут выявляться или усиливаться при перемещении программы в многопроцессорную систему. По сути, разбираясь в параллелизме, вы будете знать, что даже правильная на первый взгляд программа может проявлять неправильное поведение.

Параллельное программирование напоминает изучение нового языка — или по крайней мере новых языковых концепций. Понять логику параллельного программирования так же сложно, как понять логику объектно-ориентированного программирования. С некоторыми усилиями можно понять базовый механизм, но для подлинного понимания вопроса требуется углубленное изучение. Эта глава закладывает прочный фундамент в области основ, чтобы вы понимали концепции и могли писать несложные многопоточные программы. Тем не менее для написания любой сколько-нибудь нетривиальной программы вам придется изучать специализированную литературу по теме.

Многогранная параллельность

Главная причина сложностей с изучением параллельного программирования заключается в том, что оно применяется для решения самых разнообразных задач и реализуется разными методами, поэтому между разными случаями не существует четкого соответствия. В результате вам приходится разбираться во всех нюансах и частных случаях, чтобы эффективно использовать параллельное программирование.

Задачи, решаемые при помощи параллельного программирования, можно условно разделить на категории «скорости» и «управляемости структуры».

Ускорение выполнения

Со скоростью на первый взгляд дело обстоит просто: если вы хотите, чтобы программа выполнялась быстрее, разбейте ее на части и запустите каждую часть на отдельном процессоре. Параллельность является основным инструментом многопроцессорного программирования. В наши дни, когда закон Мура постепенно перестает действовать (по крайней мере для традиционных микросхем), ускорение проявляется в форме многоядерных процессоров, нежели в ускорении отдельных чипов. Чтобы ваша программа работала быстрее, вы должны научиться эффективно использовать дополнительные процессоры, и это одна из возможностей, которую параллельное программирование может вам предоставить.

На многопроцессорном компьютере задачи могут распределяться по разным процессорам, что приводит к радикальному возрастанию скорости. Это явление характерно для мощных многопроцессорных веб-серверов, которые могут распределять большое количество пользовательских запросов по разным процессорам в программе, назначающей отдельный поток для каждого запроса.

Однако параллельность часто повышает производительность программ, выполняющихся на *одном* процессоре.

На первый взгляд это нелогично. Если задуматься, выполнение многопоточной программы на одном процессоре неизбежно должно сопровождаться *повышенными* затратами ресурсов по сравнению с последовательным выполнением всех частей программы, из-за так называемых переключений контекста (перехода от одной задачи к другой). Казалось бы, эффективнее выполнить все части программы как одну задачу и избавиться от затрат на переключение контекста.

Однако при этом необходимо учитывать проблему *блокирования*. Если одна задача в программе не может продолжать выполнение из-за какого-то условия, неподконтрольного программе (обычно ввода-вывода), говорят, что эта задача или программный поток *блокируется* (blocks). Традиционная программа останавливается до изменения внешнего условия. Однако в программе, написанной с учетом параллельности, во время блокировки одной задачи могут продолжать выполняться другие задачи, так что программа не будет простаивать. Итак, с точки зрения производительности параллельность на однопроцессорной машине имеет смысл только в том случае, если некоторые задачи могут блокироваться.

Очень распространенным примером повышения быстродействия на однопроцессорной системе является модель *событийного программирования*. Возьмем программу, которая выполняет некоторую продолжительную операцию и перестает реагировать на действия пользователя. Конечно, можно предусмотреть кнопку прерывания операции, но опрашивать ее состояние в каждом написанном вами фрагменте кода явно нежелательно. Такие проверки загромождают код, вдобавок ничего не гарантирует, что программист не забудет выполнить проверку. Без применения параллельности реакцию пользовательского интерфейса можно обеспечить только одним способом: периодической проверкой ввода во всех задачах. Создание отдельного потока выполнения для реакции на действия пользователя (даже при том, что этот поток большую часть времени будет оставаться заблокированным) гарантирует определенную способность программы к реагированию на внешние воздействия.

Программа должна продолжать свою работу и в то же время она должна вернуть управление пользовательскому интерфейсу для обработки действий пользователя. Однако традиционный механизм выполнения не предоставляет такой возможности. Все выглядит так, словно процессор должен делать два дела одновременно — но именно такую иллюзию создает параллельное программирование (а в многопроцессорной системе это больше чем иллюзия).

Один из простейших механизмов реализации параллельности — *процессы* уровня операционной системы. Процесс представляет собой самостоятельную программу, выполняемую в собственном адресном пространстве. Многозадачная операционная система может одновременно выполнять более одного процесса (программы), переключая процессор между процессами; внешне все выглядит так, словно каждый процесс постепенно двигается вперед без прерывания. Процессы очень удобны, потому что операционная система обычно изолирует их друг от друга; таким образом, один процесс не может помешать выполнению другого процесса, благодаря чему программирование оказывается относительно простым. Напротив, параллельные системы (вроде той, что используется в Java) совместно используют ресурсы — память, ввод-вывод и т. д., поэтому основные трудности при написании многопоточных программ связаны с координацией использования ресурсов между задачами разных потоков, чтобы в любой момент времени ресурс был доступен только для одной задачи.

Рассмотрим простой пример использования процессов операционной системы. Во время работы над книгой я часто создавал несколько резервных копий текущего состояния материалов. Одна копия сохранялась в локальном каталоге, другая на карте памяти, третья на Zip-диске и четвертая на сервере FTP. Чтобы автоматизировать этот процесс, я написал маленькую программу (на Python, но концепции остаются теми же), которая архивирует материалы в файл, имя которого включает номер версии, а затем выполняет копирование. Изначально все копии создавались последовательно, а каждая новая операция копирования начиналась только после завершения предыдущей. Но потом я понял, что операции копирования занимают разное время в зависимости от скорости ввода-вывода носителя. Так как я работаю в многозадачной операционной системе, я мог запустить каждую операцию копирования в отдельном процессе, а затем выполнять их параллельно, чтобы ускорить выполнение всей программы. Если один процесс блокировался, другой мог двигаться вперед.

Это идеальный пример параллельного выполнения. Каждая задача выполняется как самостоятельный процесс в своем адресном пространстве, поэтому любые конфликты

между задачами исключены. Что еще важнее, задачам не нужно взаимодействовать друг с другом — они полностью независимы. Операционная система берет на себя все технические подробности и следит за выполнением копирования. Никакого риска, а выполнение программы ускоряется практически «бесплатно».

Некоторые специалисты доходят до того, что представляют процессы как единственно разумный подход к организации параллельного выполнения¹. К сожалению, применимость процессов в области параллельного выполнения обычно ограничивается факторами, относящимися к их количеству и непроизводительным затратам.

Некоторые языки программирования спроектированы так, чтобы изолировать параллельно выполняемые задачи друг от друга. Обычно в так называемых *функциональных языках* любой вызов функции не создает побочных эффектов (а следовательно, не может помешать другим функциям), что позволяет выполнять его в независимой задаче. Один из таких языков — Erlang — включает безопасные механизмы «общения» между задачами. Если выясняется, что часть вашей программы должна интенсивно использовать параллельность и у вас возникает слишком много проблем с построением этой части, возможно, ее следует написать на специализированном языке параллельных взаимодействий — таком, как Erlang.

В Java был выбран более традиционный подход добавления поддержки многопоточной модели на базе последовательного языка². Вместо того чтобы порождать внешние процессы в многозадачной операционной системе, модель потоков создает новые задачи *в пределах* одного процесса, представленного выполняемой программой. Одним из преимуществ такого решения была прозрачность операционной системы. Например, версии операционной системы Macintosh, предшествовавшие OS X (достаточно важная область для первых версий Java), не поддерживали многозадачность. Без реализации многопоточности в Java параллельные программы Java не переносились бы на Macintosh и другие платформы, нарушая тем самым требование «написано однажды — работает везде»³.

Улучшение структуры кода

Программа, использующая многозадачность на машине с одним процессором, в любой момент времени все равно выполняет только одну операцию, поэтому теоретически должно быть возможно написать ту же программу в однозадачном варианте. Однако параллельность предоставляет важные архитектурные преимущества: она способна заметно упростить структуру программы. Некоторые виды задач — например, моделирование — плохо решаются без поддержки параллельности.

¹ Например, Эрик Рэймонд настоятельно продвигает эту точку зрения в книге «The Art of UNIX Programming» (Addison-Wesley, 2004).

² Иногда высказывается мнение, что попытки «пристегнуть» параллелизм к последовательному языку обречены на неудачу, но вы должны составить собственное мнение.

³ Вообще-то это требование так и не было реализовано в полной мере, поэтому компания Sun уже не так громко продвигает этот лозунг. Парадоксально, но одна из причин, по которой принцип «написано однажды — работает везде» не сработал в полной мере, была связана с проблемами в системе потокового выполнения, которые, возможно, будут исправлены в Java SE5.

Вероятно, каждый сталкивался с моделированием в компьютерных играх или в компьютерных анимациях в фильмах. В моделировании обычно задействовано много взаимодействующих элементов, каждый из которых обладает собственным «интеллектом». И хотя на однопроцессорной машине каждый элемент модели обрабатывается единственным процессором, с точки зрения программирования намного проще считать, что каждый элемент модели выполняется на отдельном процессоре и является независимой задачей.

В полномасштабной модели может быть задействовано очень большое число задач, поскольку все моделируемые элементы могут действовать независимо друг от друга. В многопоточных системах часто устанавливается относительно невысокое ограничение на количество доступных потоков (порядка десятков или сотен). Это ограничение может быть неподконтрольно программе — оно может зависеть от платформы или в случае Java от версии JVM. В Java в общем случае можно считать, что количество потоков недостаточно для выделения отдельного потока каждому элементу масштабной модели.

Типичное решение проблемы заключается в использовании *кооперативной многопоточности*. В Java используется потоковая модель с *вытеснением*, при которой механизм планирования выделяет временные *кванты* (slices) каждому потоку, периодически прерывая его выполнение и осуществляя переключение контекста на другой поток так, чтобы каждому потоку выделялось разумное время на выполнение его задачи. В кооперативной системе каждая задача добровольно уступает управление, для чего программист должен сознательно вставить в каждую задачу соответствующую команду. Кооперативная система обладает двумя важными преимуществами: во-первых, переключение контекста обычно требует существенно меньших затрат, чем в системе с вытеснением, а во-вторых, количество одновременно выполняемых независимых задач теоретически не ограничивается. В моделях с большим количеством элементов такое решение нередко оказывается идеальным. Однако следует помнить, что некоторые кооперативные системы не предусматривают распределения задач между процессорами, и это ограничение может оказаться очень существенным.

С другой стороны, модель параллельного выполнения очень полезна при работе с современными системами передачи сообщений, в которых задействовано множество независимых компьютеров, распределенных в сети. В этом случае все процессы выполняются полностью независимо друг от друга, без возможности совместного доступа к ресурсам. Однако при этом все равно необходимо синхронизировать передачу информации между процессами, чтобы предотвратить возможную потерю данных или их поступление в неправильное время. Даже если вы не планируете часто использовать параллельное программирование в ближайшем будущем, желательно понимать его, чтобы разобраться в архитектурах передачи сообщений, все чаще применяемых при создании распределенных систем.

Параллельное выполнение сопряжено с определенными затратами, включая возрастание сложности, но обычно они компенсируются улучшением архитектуры программы, балансировкой ресурсов и удобством использования. Как правило, потоки позволяют снизить связанность архитектуры; в противном случае частям вашего кода придется заниматься выполнением операций, которые обычно выполняются потоками автоматически.

Основы построения многопоточных программ

Параллельное программирование позволяет разделить программу на несколько независимых частей. Довольно часто бывает необходимо превратить программу в несколько отдельных, самостоятельно выполняющихся подзадач. Каждая из этих самостоятельных подзадач называется *поток* (thread). Поток — это выполняемая параллельно в рамках процесса последовательность команд программы. Таким образом, процесс может одновременно содержать несколько выполняющихся потоков, но вы пишете программу так, как будто каждый поток запускается сам по себе и использует процессор монополично. На самом деле существует некоторый низкоуровневый механизм, который разделяет время процессора, но в основном думать об этом вам не придется.

Модель потоков (и ее поддержка в языке Java) является программным механизмом для упрощения выполнения нескольких операций синхронно в одной и той же программе: процессор вмешивается в происходящее и выделяет каждому потоку некоторый отрезок времени¹. Каждый поток полагает, что он использует процессор монополично, но на самом деле время процессора разделяется между всеми существующими в программе потоками (кроме случаев, в которых программа действительно работает на нескольких процессорах). Однако при использовании потоков вам не нужно задумываться об этих тонкостях, ваш код не зависит от того, на скольких процессорах вам придется работать, и это замечательно. Таким образом, потоки предоставляют способ написания программ с прозрачной масштабируемостью — если ваша программа работает слишком медленно, вы в силах легко ускорить ее, добавив на свой компьютер дополнительные процессоры. Многозадачность и многопоточность являются, похоже, наиболее вескими причинами использования многопроцессорных систем.

Определение задач

Поток выполняет некоторую задачу, поэтому вам нужны средства для описания таких задач. Эти средства предоставляются интерфейсом `Runnable`. Чтобы определить задачу, просто реализуйте `Runnable` и напишите метод `run()`, который выполнит нужные действия.

Например, следующая задача `LiftOff` выводит обратный отсчет перед запуском:

```
//: concurrency/LiftOff.java
// Demonstration of the Runnable interface.

public class LiftOff implements Runnable {
    protected int countDown = 10; // Default
    private static int taskCount = 0;
    private final int id = taskCount++;
    public LiftOff() {}
    public LiftOff(int countDown) {
```

продолжение ↗

¹ Это относится к системам, использующим механизм выделения квантов процессорного времени (например, Windows). В Solaris используется модель FIFO: если в системе не будет активизирован поток с более высоким приоритетом, текущий поток работает до тех пор, пока он не будет заблокирован или завершен. Это означает, что другие потоки с равным приоритетом не будут запущены до тех пор, пока текущий поток не уступит процессор.


```

    this.countDown = countDown;
}
public String status() {
    return "#" + id + "(" +
        (countDown > 0 ? countDown : "Liftoff!") + ")", ";
}
public void run() {
    while(countDown-- > 0) {
        System.out.print(status());
        Thread.yield();
    }
}
} ///:~

```

Идентификатор `id` различает экземпляры задачи. Он объявлен с ключевым словом `final`, потому что значение не должно изменяться после инициализации.

Метод `run()` практически всегда являет собой некоторый цикл, который выполняется, пока поток еще нужен, поэтому вам придется определить условие выхода из такого цикла (можно просто использовать команду `return`, как сделано в рассматриваемой программе). Зачастую метод `run()` реализуется в форме бесконечного цикла; это значит, что завершение потока осуществляется с помощью какого-либо внешнего фактора или он будет выполняться бесконечно (чуть позже в этой главе вы узнаете, как безопасно сигнализировать потоку, чтобы он «остановился»).

Вызов статического метода `Thread.yield()` внутри `run()` является рекомендацией для планировщика потоков (подсистема механизма потоков Java, которая переключает процессор с одного потока на другой). По сути она означает: «Важная часть моего цикла выполнена, и сейчас было бы неплохо переключиться на другую задачу». Этот вызов полностью необязателен, но он используется здесь, потому что с ним результат получается более интересным: вы с большей вероятностью увидите доказательства переключения потоков.

В следующем примере вызов `run()` не управляется отдельным потоком; метод просто напрямую вызывается в `main()` (поток, конечно, при этом используется: тот, который всегда создается для `main()`):

//: concurrency/MainThread.java

```

public class MainThread {
    public static void main(String[] args) {
        LiftOff launch = new LiftOff();
        launch.run();
    }
} /* Output:
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2),
#0(1), #0(Liftoff!),
*///:~

```

Класс, производный от `Runnable`, должен содержать метод `run()`, но ничего особенного в этом методе нет — он не обладает никакими «встроенными» потоковыми способностями. Чтобы добиться потокового выполнения, необходимо явно присоединить задачу к потоку.

Класс Thread

Традиционный способ преобразования объекта `Runnable` в выполняемую задачу основан на передаче его конструктору `Thread`. Следующий пример показывает, как организовать выполнение объекта `LiftOff` с использованием `Thread`:

```
//: concurrency/BasicThreads.java
// Простейшее использование класса Thread.

public class BasicThreads {
    public static void main(String[] args) {
        Thread t = new Thread(new LiftOff());
        t.start();
        System.out.println("Waiting for LiftOff");
    }
} /* Output: (90% match)
Waiting for LiftOff
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2),
#0(1), #0(LiftOff!),
*///:~
```

Конструктору `Thread` необходим только объект `Runnable`. Метод `start()` объекта `Thread` выполняет необходимую инициализацию потока, после чего вызывает метод `run()` объекта `Runnable` для запуска задачи в новом потоке.

На первый взгляд может показаться, что вызов `start()` должен привести к вызову метода с большим временем выполнения, из вывода видно, что `start()` быстро возвращает управление — сообщение «Waiting for LiftOff» появляется до завершения отсчета. Вы вызываете `LiftOff.run()`, выполнение этого метода еще не завершено, но поскольку `LiftOff.run()` выполняется другим потоком, в потоке `main()` можно выполнять другие операции. (Данная возможность не ограничивается потоком `main()` — любой поток может запустить другой поток.)

Итак, в программе выполняются сразу два метода — `main()` и `LiftOff.run()`. Метод `run()` содержит код, выполняемый «одновременно» с другими потоками в программе.

Вы можете легко добавить дополнительные потоки для выполнения других задач. В следующем примере обратный отсчет ведется сразу несколькими задачами¹:

```
//: concurrency/MoreBasicThreads.java
// Добавление новых потоков.

public class MoreBasicThreads {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++)
            new Thread(new LiftOff()).start();
        System.out.println("Waiting for LiftOff");
    }
} /* Output: (Sample)
```

продолжение ➤

¹ В этом случае один поток (`main()`) создает все потоки `LiftOff`. Но если в программе потоки `LiftOff` создаются несколькими потоками, может оказаться, что нескольким объектам `LiftOff` будет присвоено одно значение `id`. Позднее вы узнаете, почему это происходит.

```
Waiting for LiftOff
#0(9), #1(9), #2(9), #3(9), #4(9), #0(8), #1(8), #2(8),
#3(8), #4(8), #0(7), #1(7), #2(7), #3(7), #4(7), #0(6),
#1(6), #2(6), #3(6), #4(6), #0(5), #1(5), #2(5), #3(5),
#4(5), #0(4), #1(4), #2(4), #3(4), #4(4), #0(3), #1(3),
#2(3), #3(3), #4(3), #0(2), #1(2), #2(2), #3(2), #4(2),
#0(1), #1(1), #2(1), #3(1), #4(1), #0(LiftOff!),
#1(LiftOff!), #2(LiftOff!), #3(LiftOff!), #4(LiftOff!),
*///:~
```

Из выходных данных видно, что разные потоки выполняются попеременно, активизируясь и теряя управление. Переключение обеспечивается автоматически планировщиком потоков. Если ваша машина оснащена несколькими процессорами, планировщик распределит выполняемые потоки по разным процессорам¹.

Выходные данные программы будут отличаться в зависимости от запуска, потому что механизм планирования потоков не детерминирован. Более того, результаты могут сильно различаться даже между разными версиями JDK. Например, в ранних JDK квантование осуществлялось с меньшей частотой, поэтому сначала поток 1 мог отработать до конца, затем отработывал поток 2 и т. д. Фактически все выглядело так, словно в программе вызывалась функция, которая полностью проходила весь свой цикл, и так далее, не считая дополнительных затрат на запуск потоков. В последующих версиях JDK механизм квантования был улучшен, так что потоки выполнялись более равномерно. Компания Sun обычно не упоминает о таких изменениях в поведении JDK, поэтому вы не можете рассчитывать на неизменность многопоточного поведения. Лучше всего при написании многопоточного кода строить как можно меньше предположений.

При создании объектов `Thread` метод `main()` не сохраняет ссылки на них. Обычный объект в такой ситуации немедленно стал бы кандидатом на уничтожение уборщиком мусора, но к `Thread` это не относится. Каждый объект `Thread` «регистрируется», так что ссылка на него хранится где-то в системе, и уборщик мусора не тронет его до того момента, когда метод `run()` завершится, а задача прекратит свое существование.

1. (2) Реализуйте интерфейс `Runnable`. В методе `run()` выведите сообщение, после чего вызовите `yield()`. Повторите три раза и верните управление из `run()`. Выведите начальное сообщение в конструкторе, а при завершении выведите конечное сообщение. Создайте несколько задач и организуйте их выполнение с использованием потоков.
2. (2) По образцу `generics/Fibonacci.java` создайте задачу, которая генерирует последовательность из n чисел Фибоначчи (значение n передается в конструкторе задачи). Создайте несколько таких задач и выполните их в потоках.

Использование Executor

Объекты `Executor`, появившиеся в Java SE5 (`java.util.concurrent`), значительно упрощают параллельное программирование и управляют объектами `Thread` за разработчиком. Они образуют дополнительную логическую прослойку между клиентом и исполняемой задачей; вместо того чтобы запускать задачу напрямую, клиент поручает это

¹ Этого не происходило в некоторых ранних версиях Java.

промежуточному объекту. Объект `Executor` позволяет управлять выполнением асинхронных задач без явного управления жизненным циклом потоков; этот механизм рекомендуется применять для выполнения задач в Java SE5/6.

Мы можем воспользоваться `Executor` вместо того, чтобы явно создавать объекты `Thread` в программе `MoreBasicThreads.java`. Объект `LiftOff` «знает», как запустить конкретную задачу; по аналогии с паттерном проектирования «Команда», он предоставляет конкретный метод для ее выполнения. Объект `ExecutorService` (`Executor` с жизненным циклом службы) умеет строить контекст для выполнения объектов `Runnable`. В следующем примере `CachedThreadPool` создает один поток для каждой задачи. Обратите внимание на создание объекта `ExecutorService` статическим методом `Executors`, определяющим тип `Executor`:

```
//: concurrency/CachedThreadPool.java
import java.util.concurrent.*;

public class CachedThreadPool {
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new LiftOff());
        exec.shutdown();
    }
} /* Output: (Sample)
#0(9), #0(8), #1(9), #2(9), #3(9), #4(9), #0(7), #1(8),
#2(8), #3(8), #4(8), #0(6), #1(7), #2(7), #3(7), #4(7),
#0(5), #1(6), #2(6), #3(6), #4(6), #0(4), #1(5), #2(5),
#3(5), #4(5), #0(3), #1(4), #2(4), #3(4), #4(4), #0(2),
#1(3), #2(3), #3(3), #4(3), #0(1), #1(2), #2(2), #3(2),
#4(2), #0(LiftOff!), #1(1), #2(1), #3(1), #4(1),
#1(LiftOff!), #2(LiftOff!), #3(LiftOff!), #4(LiftOff!),
*///:~
```

Очень часто один объект `Executor` может использоваться для создания и управления всеми задачами в вашей системе.

Вызов `shutdown()` предотвращает отправку новых задач объекту `Executor`. Текущий поток (в нашем случае поток, выполняющий `main()`) продолжает выполнять все задачи, отправленные до вызова `shutdown()`. Программа завершается сразу же после завершения всех задач, переданных `Executor`.

`CachedThreadPool` из предыдущего примера легко заменяется другим типом `Executor`. Так, объект `FixedThreadPool` использует ограниченный набор потоков для выполнения переданных задач:

```
//: concurrency/FixedThreadPool.java
import java.util.concurrent.*;

public class FixedThreadPool {
    public static void main(String[] args) {
        // Constructor argument is number of threads:
        ExecutorService exec = Executors.newFixedThreadPool(5);
        for(int i = 0; i < 5; i++)
            exec.execute(new LiftOff());
        exec.shutdown();
    }
}
```

продолжение ➤

```

} /* Output: (Sample)
#0(9), #0(8), #1(9), #2(9), #3(9), #4(9), #0(7), #1(8),
#2(8), #3(8), #4(8), #0(6), #1(7), #2(7), #3(7), #4(7),
#0(5), #1(6), #2(6), #3(6), #4(6), #0(4), #1(5), #2(5),
#3(5), #4(5), #0(3), #1(4), #2(4), #3(4), #4(4), #0(2),
#1(3), #2(3), #3(3), #4(3), #0(1), #1(2), #2(2), #3(2),
#4(2), #0(Liftoff!), #1(1), #2(1), #3(1), #4(1),
#1(Liftoff!), #2(Liftoff!), #3(Liftoff!), #4(Liftoff!),
*///:~

```

При использовании `FixedThreadPool` высокочастотное создание потоков выполняется один раз (в самом начале), поэтому количество потоков ограничивается. Такая схема экономит время, потому что вам не нужно многократно «платить» за создание потока каждой задачи. Кроме того, в событийной системе обработчики событий, требующие применения потоков, будут обслуживаться с нужной быстротой, для чего потоки будут извлекаться из пула. Перерасход ресурсов также исключен, потому что `FixedThreadPool` использует ограниченное количество объектов `Thread`.

Все пулы по возможности повторно используют существующие потоки.

Хотя в книге используются объекты `CachedThreadPool`, рассмотрите возможность использования `FixedThreadPool` в окончательном коде. Объект `CachedThreadPool` обычно создает столько потоков, сколько требуется во время выполнения программы, после чего перестает создавать новые потоки и повторно использует уже существующие. Пожалуй, это первый разумный выбор `Executor`. Только если он создает проблемы, можно подумать о переключении на `FixedThreadPool`.

Объект `SingleThreadExecutor` работает как `FixedThreadPool`, использующий всего один поток¹. Он может пригодиться для любых продолжительных задач, выполняемых в другом потоке, например задачи прослушивания входящих подключений через сокет. Также пулы с одним потоком хорошо подходят для коротких задач, которые должны выполняться отдельно, например мелких задач обновления локального или удаленного журнала или диспетчеризации событий.

Если объект `SingleThreadExecutor` получает более одной задачи, то эти задачи ставятся в очередь. Каждая задача выполняется до завершения, после чего запускается следующая задача; при этом все они используют один поток. В следующем примере видно, что задачи выполняются последовательно в порядке их отправки. Итак, `SingleThreadExecutor` организует последовательное выполнение задач с ведением собственной (скрытой) очереди ожидания задач.

```

//: concurrency/SingleThreadExecutor.java
import java.util.concurrent.*;

```

```

public class SingleThreadExecutor {
    public static void main(String[] args) {
        ExecutorService exec =
            Executors.newSingleThreadExecutor();
        for(int i = 0; i < 5; i++)

```

¹ Он также предоставляет важную гарантию, которая не предоставляется другими видами пулов: невозможность одновременного выполнения двух задач. Это изменяет требования к блокировке для задач (см. далее в этой главе).

```

    exec.execute(new LiftOff());
    exec.shutdown();
}
} /* Output:
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2),
#0(1), #0(Liftoff!), #1(9), #1(8), #1(7), #1(6), #1(5),
#1(4), #1(3), #1(2), #1(1), #1(Liftoff!), #2(9), #2(8),
#2(7), #2(6), #2(5), #2(4), #2(3), #2(2), #2(1),
#2(Liftoff!), #3(9), #3(8), #3(7), #3(6), #3(5), #3(4),
#3(3), #3(2), #3(1), #3(Liftoff!), #4(9), #4(8), #4(7),
#4(6), #4(5), #4(4), #4(3), #4(2), #4(1), #4(Liftoff!),
*///:~

```

Или другой пример: предположим, имеется набор потоков для выполнения задач, использующих файловую систему. Вы можете выполнить эти задачи объектом `SingleThreadExecutor`, чтобы в любой момент времени выполнялась только одна из этих задач. При таком решении вам не нужно обеспечивать синхронизацию общих ресурсов (и вы не повредите файловую систему).

Иногда задача лучше решается синхронизацией по ресурсу (см. далее в этой главе), но `SingleThreadExecutor` позволит избежать лишних хлопот с координацией при построении прототипа.

Последовательное выполнение задач снимает необходимость в синхронизации доступа к объектам.

3. (1) Повторите упражнение 1 с разными типами `Executor`, описанными в этом разделе.
4. (1) Повторите упражнение 2 с разными типами `Executor`, описанными в этом разделе.

Возвращение значений из задач

Объект `Runnable` представляет отдельную задачу, которая выполняет некую работу, но не возвращает значения. Если вы хотите, чтобы при завершении задачи возвращалось значение, реализуйте интерфейс `Callable` вместо интерфейса `Runnable`. Интерфейс `Callable`, появившийся в Java SE5, является обобщением с параметром-типом, представляющим возвращаемое значение метода `call()` (вместо `run()`), а для его активизации должен использоваться метод `ExecutorService submit()`. Простой пример:

```

//: concurrency/CallableDemo.java
import java.util.concurrent.*;
import java.util.*;

class TaskWithResult implements Callable<String> {
    private int id;
    public TaskWithResult(int id) {
        this.id = id;
    }
    public String call() {
        return "result of TaskWithResult " + id;
    }
}

```

продолжение ➤

```

public class CallableDemo {
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        ArrayList<Future<String>> results =
            new ArrayList<Future<String>>();
        for(int i = 0; i < 10; i++)
            results.add(exec.submit(new TaskWithResult(i)));
        for(Future<String> fs : results)
            try {
                // get() blocks until completion:
                System.out.println(fs.get());
            } catch(InterruptedException e) {
                System.out.println(e);
                return;
            } catch(ExecutionException e) {
                System.out.println(e);
            } finally {
                exec.shutdown();
            }
        }
    /* Output:
result of TaskWithResult 0
result of TaskWithResult 1
result of TaskWithResult 2
result of TaskWithResult 3
result of TaskWithResult 4
result of TaskWithResult 5
result of TaskWithResult 6
result of TaskWithResult 7
result of TaskWithResult 8
result of TaskWithResult 9
///:~

```

Метод `submit()` создает объект `Future`, параметризованный по конкретному типу результата, возвращаемого `Callable`. К `Future` можно обратиться с запросом `isDone()`, чтобы проверить, не завершилась ли задача. Когда задача завершится и у нее появится результат, вы можете вызвать метод `get()` для получения этого результата. Метод `get()` можно вызвать и без проверки `isDone()`, но тогда вызов `get()` блокируется до готовности результата. Чтобы избежать блокировки, вызовите `get()` с тайм-аутом или выполните предварительную проверку методом `isDone()`.

Перегруженный метод `Executors.callable()` получает объект `Runnable` и создает `Callable`. Объект `ExecutorService` содержит методы `invoke` для выполнения объектов `Callable`, объединенных в коллекцию.

5. (2) Измените упражнение 2, чтобы задача была представлена объектом `Callable` для суммирования значений всех чисел Фибоначчи. Создайте несколько задач и выведите результаты.

Ожидание

Также для управления потоками можно воспользоваться методом `sleep()`, который переводит поток в состояние ожидания на заданный промежуток времени. Если в классе `ShiftOff` заменить вызов `yield()` на вызов `sleep()`, вы получите следующее:

```

//: concurrency/SleepingTask.java
// Вызов sleep() для приостановки выполнения.
import java.util.concurrent.*;

public class SleepingTask extends LiftOff {
    public void run() {
        try {
            while(countDown-- > 0) {
                System.out.print(status());
                // В старом стиле:
                // Thread.sleep(100);
                // В стиле Java SE5/6:
                TimeUnit.MILLISECONDS.sleep(100);
            }
        } catch (InterruptedException e) {
            System.err.println("Interrupted");
        }
    }
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new SleepingTask());
        exec.shutdown();
    }
} /* Output:
#0(9), #1(9), #2(9), #3(9), #4(9), #0(8), #1(8), #2(8),
#3(8), #4(8), #0(7), #1(7), #2(7), #3(7), #4(7), #0(6),
#1(6), #2(6), #3(6), #4(6), #0(5), #1(5), #2(5), #3(5),
#4(5), #0(4), #1(4), #2(4), #3(4), #4(4), #0(3), #1(3),
#2(3), #3(3), #4(3), #0(2), #1(2), #2(2), #3(2), #4(2),
#0(1), #1(1), #2(1), #3(1), #4(1), #0(Liftoff!),
#1(Liftoff!), #2(Liftoff!), #3(Liftoff!), #4(Liftoff!),
*///:~

```

Вызов `sleep()` может выдать исключение `InterruptedException`, которое, как видно из листинга, перехватывается в `run()`. Так как исключения не передаются между потоками обратно в `main()`, вы должны локально обрабатывать любые исключения, возникающие в задачах.

В Java SE5 появилась новая версия `sleep()`, входящая в класс `TimeUnit` (см. выше). Она делает код более понятным, поскольку позволяет задать единицы времени для задержки `sleep()`. Класс `TimeUnit` также может использоваться для выполнения преобразований, как будет показано позднее.

В зависимости от платформы можно заметить, что задачи выполняются детерминированно — от нуля до четырех, затем снова до нуля. Такое поведение объясняется тем, что после каждой команды вывода каждая задача переходит в ожидание (блокируется), чтобы планировщик мог переключиться на другой поток и активизировать другую задачу. Однако последовательное поведение опирается на нижележащий механизм потоков, который отличается между операционными системами, поэтому рассчитывать на него нельзя. Если вам необходимо управлять порядком выполнения потоков, лучше всего воспользоваться элементами синхронизации (см. далее), или в некоторых случаях — вообще не использовать потоки, а вместо них написать свои собственные взаимодействующие процедуры, которые будут передавать управление друг другу в нужном порядке.

6. (2) Создайте задачу, которая приостанавливается на случайный промежуток времени от 1 до 10 секунд, затем выводит время ожидания и завершается. Создайте и запустите несколько таких задач (количество должно задаваться в командной строке).

Приоритет

Приоритет (*priority*) потока показывает планировщику, насколько важен этот поток. Хотя порядок обращения процессора к существующему набору потоков и не детерминирован, если существует несколько приостановленных потоков, одновременно ожидающих запуска, планировщик сначала запустит поток с большим приоритетом. Впрочем, это не значит, что потоки с младшими приоритетами не выполняются вовсе (то есть отказа в обслуживании из-за приоритетов не возникает). Потоки с более низкими приоритетами просто запускаются чуть реже.

В подавляющем большинстве случаев все потоки должны выполняться с приоритетом по умолчанию. Попытки манипулирования с приоритетами потоков обычно являются ошибкой.

Следующий пример демонстрирует уровни приоритетов. Приоритеты задаются методом `setPriority()` и читаются методом `getPriority()`:

```
//: concurrency/SimplePriorities.java
// Shows the use of thread priorities.
import java.util.concurrent.*;

public class SimplePriorities implements Runnable {
    private int countDown = 5;
    private volatile double d; // Без оптимизации
    private int priority;
    public SimplePriorities(int priority) {
        this.priority = priority;
    }
    public String toString() {
        return Thread.currentThread() + ": " + countDown;
    }
    public void run() {
        Thread.currentThread().setPriority(priority);
        while(true) {
            // Высокотратная прерываемая операция:
            for(int i = 1; i < 100000; i++) {
                d += (Math.PI + Math.E) / (double)i;
                if(i % 1000 == 0)
                    Thread.yield();
            }
            System.out.println(this);
            if(--countDown == 0) return;
        }
    }
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(
                new SimplePriorities(Thread.MIN_PRIORITY));
        exec.execute(
            new SimplePriorities(Thread.MAX_PRIORITY));
    }
}
```



```
    exec.shutdown();
}
} /* Output: (70% match)
Thread[pool-1-thread-6,10,main]: 5
Thread[pool-1-thread-6,10,main]: 4
Thread[pool-1-thread-6,10,main]: 3
Thread[pool-1-thread-6,10,main]: 2
Thread[pool-1-thread-6,10,main]: 1
Thread[pool-1-thread-3,1,main]: 5
Thread[pool-1-thread-2,1,main]: 5
Thread[pool-1-thread-1,1,main]: 5
Thread[pool-1-thread-5,1,main]: 5
Thread[pool-1-thread-4,1,main]: 5
...
*///:~
```

Метод `toString()` переопределяется и использует метод `Thread.toString()`, который выводит имя потока, приоритет и группу, к которой принадлежит поток. Имя потока можно задать в конструкторе, но здесь имена автоматически генерируются в виде `pool-1-thread-1`, `pool-1-thread-2` и т. д. Переопределенный метод `toString()` также выводит значение обратного отсчета. Обратите внимание: вы можете получить ссылку на объект `Thread`, управляющий выполнением задачи, внутри самой задачи, вызовом `Thread.currentThread()`.

Нетрудно заметить, что поток под номером 1 обладает наивысшим приоритетом, а все остальные потоки имеют минимальный приоритет. Приоритет задается в начале `run()`; задавать его в конструкторе бессмысленно, так как объект `Executor` в этот момент еще не начал выполнение задачи.

В метод `run()` были добавлены 100 000 достаточно затратных операций с плавающей запятой, включая суммирование и деление с числом двойной точности `double`. Переменная `d` была отмечена как `volatile`, чтобы компилятор не пытался проводить оптимизацию¹. Без этих вычислений вы не увидите эффекта установки различных приоритетов (попробуйте: закомментируйте цикл `for` с вычислениями). В процессе вычислений вы видите, как планировщик уделяет больше внимания потоку с приоритетом `MAX_PRIORITY` (по крайней мере, таково было поведение программы на машине под управлением Windows XP). И хотя вывод на консоль также является «дорогостоящей» операцией, с ним вы не увидите влияние уровней приоритетов, поскольку вывод на консоль не прерывается (иначе экран был бы заполнен несущими шумом), в то время как математические вычисления, приведенные выше, прерывать допустимо. Вычисления выполняются достаточно долго, соответственно, механизм планирования потоков вмешивается в процесс и чередует потоки, проявляя при этом внимание к более приоритетным, поэтому высокоприоритетные потоки получают предпочтение. Тем не менее, чтобы обеспечить переключение контекста, программа регулярно выполняет команды `yield()`.

¹ Это ключевое слово упоминается вскользь без всякого разъяснения. Его смысл в наложении дополнительных ограничений на объявленные переменные в каждом потоке при выполнении типовых действий: `use`, `assign`, `read`, `load`, `store`, `write`, `lock`, `unlock`. Правила, регламентирующие использование `volatile`, вы найдете в *Java Language Specification, Second Edition*, глава 17, *Threads and Locks*. — Примеч. ред.

В пакете JDK предусмотрено 10 уровней приоритетов, однако это не слишком хорошо согласуется с большинством операционных систем. К примеру, в Windows 2000 имеется 7 классов приоритетов, таким образом, их соответствие неочевидно (хотя в операционной системе Sun Solaris имеется 231 уровень). Единственным переносимым подходом при настройке приоритетов остается использование универсальных констант `MAX_PRIORITY`, `NORM_PRIORITY` и `MIN_PRIORITY`.

Уступки

Если вы знаете, что в текущей итерации цикла в методе `run()` было сделано все, что требовалось, то вы можете подсказать механизму планирования потоков, что процессором теперь может воспользоваться какой-либо другой поток. Эта подсказка (*всего лишь* рекомендация — нет никакой гарантии, что ваша реализация «прислушается» к ней) получает форму вызова метода `yield()`. При вызове `yield()` вы сообщаете, что в системе могут выполняться другие потоки с тем же приоритетом.

В примере `LiftOff.java` вызов `yield()` используется для получения более равномерной обработки разных задач `LiftOff`. Попробуйте закомментировать вызов `Thread.yield()` в `LiftOff.run()` и посмотрите, что получится. В общем и целом метод `yield()` часто используется неправильно, и я не рекомендую полагаться на него как на серьезное средство настройки вашего приложения.

Потоки-демоны

Поток-демон (`daemon`) предоставляет некоторые общие услуги в фоновом режиме, пока выполняется программа, но не является ее неотъемлемой частью. Таким образом, когда все потоки не-демоны заканчивают свою деятельность, программа завершается. И наоборот, если существуют работающие потоки не-демоны, программа продолжает выполнение. Существует, например, поток не-демон, выполняющий метод `main()`.

```
//: concurrency/SimpleDaemons.java
// Потоки-демоны не препятствуют завершению работы программы.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

public class SimpleDaemons implements Runnable {
    public void run() {
        try {
            while(true) {
                TimeUnit.MILLISECONDS.sleep(100);
                print(Thread.currentThread() + " " + this);
            }
        } catch (InterruptedException e) {
            print("sleep() interrupted");
        }
    }
}

public static void main(String[] args) throws Exception {
    for(int i = 0; i < 10; i++) {
        Thread daemon = new Thread(new SimpleDaemons());
        daemon.setDaemon(true); // Необходимо вызвать перед start()
        daemon.start();
    }
}
```

```

    print("All daemons started");
    TimeUnit.MILLISECONDS.sleep(175);
}
} /* Output: (Sample)
All daemons started
Thread[Thread-0,5,main] SimpleDaemons@530daa
Thread[Thread-1,5,main] SimpleDaemons@a62fc3
Thread[Thread-2,5,main] SimpleDaemons@89ae9e
Thread[Thread-3,5,main] SimpleDaemons@1270b73
Thread[Thread-4,5,main] SimpleDaemons@60aeb0
Thread[Thread-5,5,main] SimpleDaemons@16caf43
Thread[Thread-6,5,main] SimpleDaemons@66848c
Thread[Thread-7,5,main] SimpleDaemons@8813f2
Thread[Thread-8,5,main] SimpleDaemons@1d58aae
Thread[Thread-9,5,main] SimpleDaemons@83cc67
...
*///:~

```

Режим демона включается методом `setDaemon()` перед запуском потока.

После того как `main()` выполнит свою работу, ничто не мешает программе незамедлительно завершиться, так как в ней остаются только потоки-демоны. Поэтому, чтобы вы видели результаты запуска всех демонов, поток `main()` ненадолго приостанавливается. Без этого вы увидели бы лишь небольшую часть отчетов потоков-демонов перед выходом из программы (чтобы понаблюдать за этим поведением, поэкспериментируйте с разными значениями задержки `sleep()`).

Программа `SimpleDaemons.java` явно создает объекты `Thread`, чтобы установить для них флаг демона. Для настройки атрибутов потоков, создаваемых объектами `Executor` (демон, приоритет, имя), можно написать собственную реализацию `ThreadFactory`:

```

//: net/mindview/util/DaemonThreadFactory.java
package net.mindview.util;
import java.util.concurrent.*;

public class DaemonThreadFactory implements ThreadFactory {
    public Thread newThread(Runnable r) {
        Thread t = new Thread(r);
        t.setDaemon(true);
        return t;
    }
} ///:~

```

Эта реализация `ThreadFactory` отличается от обычной только установкой флага демона. Новый объект `DemonThreadFactory` можно передать в аргументе `Executors.newCachedThreadPool()`:

```

//: concurrency/DaemonFromFactory.java
// Использование ThreadFactory для создания демонов.
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DaemonFromFactory implements Runnable {
    public void run() {
        try {

```

продолжение ➤

```

        while(true) {
            TimeUnit.MILLISECONDS.sleep(100);
            print(Thread.currentThread() + " " + this);
        }
    } catch (InterruptedException e) {
        print("Interrupted");
    }
}

public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool(
        new DaemonThreadFactory());
    for(int i = 0; i < 10; i++)
        exec.execute(new DaemonFromFactory());
    print("All daemons started");
    TimeUnit.MILLISECONDS.sleep(500); // Небольшая задержка
}
} /* (Execute to see output) */://:~

```

Все статические методы создания `ExecutorService` перегружаются для получения объекта `ThreadFactory`, который будет использоваться для создания новых потоков.

Можно пойти еще дальше и создать вспомогательный класс `DaemonThreadPoolExecutor`:

```

//: net/mindview/util/DaemonThreadPoolExecutor.java
package net.mindview.util;
import java.util.concurrent.*;

public class DaemonThreadPoolExecutor
extends ThreadPoolExecutor {
    public DaemonThreadPoolExecutor() {
        super(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS,
            new SynchronousQueue<Runnable>(),
            new DaemonThreadFactory());
    }
} //://:~

```

Чтобы узнать значения для вызова конструктора базового класса, я просто заглянул в исходный код `Executors.java`.

Чтобы узнать, является ли поток демоном, вызовите его метод `isDaemon()`. Если поток является демоном, то все потоки, которые он создает, также автоматически становятся демонами, что и демонстрируется следующим примером:

```

//: concurrency/Daemons.java
// Потоки-демоны порождают других демонов.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Daemon implements Runnable {
    private Thread[] t = new Thread[10];
    public void run() {
        for(int i = 0; i < t.length; i++) {
            t[i] = new Thread(new DaemonSpawn());
            t[i].start();
            printnb("DaemonSpawn " + i + " started, ");
        }
        for(int i = 0; i < t.length; i++)
            printnb("t[" + i + "].isDaemon() = " +
                t[i].isDaemon() + ", ");
    }
}

```

```

        while(true)
            Thread.yield();
    }
}

class DaemonSpawn implements Runnable {
    public void run() {
        while(true)
            Thread.yield();
    }
}

public class Daemons {
    public static void main(String[] args) throws Exception {
        Thread d = new Thread(new Daemon());
        d.setDaemon(true);
        d.start();
        printlnb("d.isDaemon() = " + d.isDaemon() + ", ");
        // Дать возможность потокам-демонам
        // завершить их стартовые процессы:
        TimeUnit.SECONDS.sleep(1);
    }
} /* Output: (Sample)
d.isDaemon() = true, DaemonSpawn 0 started, DaemonSpawn 1
started, DaemonSpawn 2 started, DaemonSpawn 3 started,
DaemonSpawn 4 started, DaemonSpawn 5 started, DaemonSpawn 6
started, DaemonSpawn 7 started, DaemonSpawn 8 started,
DaemonSpawn 9 started, t[0].isDaemon() = true,
t[1].isDaemon() = true, t[2].isDaemon() = true,
t[3].isDaemon() = true, t[4].isDaemon() = true,
t[5].isDaemon() = true, t[6].isDaemon() = true,
t[7].isDaemon() = true, t[8].isDaemon() = true,
t[9].isDaemon() = true,
*///:~

```

Поток класса `Daemon` переводится в режим демона, а затем порождает связку новых потоков (для которых флаг демона явно не устанавливается), чтобы показать, что они все равно будут демонами. Затем поток переходит в бесконечный цикл, на каждом шаге которого вызывается метод `yield()`, передающий управление другому процессу.

Учтите, что потоки-демоны завершают свои методы `run()` без выполнения блока `finally`:

```

//: concurrency/DaemonsDontRunFinally.java
// Потоки-демоны не выполняют блок finally.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class ADaemon implements Runnable {
    public void run() {
        try {
            print("Starting ADaemon");
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            print("Exiting via InterruptedException");
        } finally {
            print("This should always run?");
        }
    }
}

```

```

public class DaemonsDontRunFinally {
    public static void main(String[] args) throws Exception {
        Thread t = new Thread(new ADaemon());
        t.setDaemon(true);
        t.start();
    }
} /* Output:
Starting ADaemon
*///:~

```

Запустив эту программу, вы увидите, что блок `finally` не выполняется. С другой стороны, если закомментировать вызов `setDaemon()`, то блок `finally` будет выполнен.

Такое поведение верно, даже если оно и покажется вам неожиданным на основании упоминавшихся ранее гарантий для `finally`. Демоны завершаются «моментально» при завершении последнего из не-демонов. Таким образом, сразу же после выхода из `main()` виртуальная машина Java немедленно закроет всех демонов без формальностей, которых вы, возможно, ожидали. Так как «корректное» завершение демонов невозможно, такое решение редко оказывается удачным. Выполнение `Executor` в не-демонах обычно предпочтительно, так как все задачи, управляемые `Executor`, могут быть закрыты одновременно. Как будет показано позже в этой главе, завершение в этом случае проходит упорядоченно.

7. (2) Поэкспериментируйте с разным временем ожидания в `Daemons.java` и посмотрите, что при этом происходит.
8. (1) Измените пример `MoreBasicThreads.java` так, чтобы все потоки выполнялись в режиме демона. Убедитесь в том, что программа завершается сразу же после выхода из `main()`.
9. (3) Измените пример `SimplePriorities.java`, чтобы пользовательская реализация `ThreadFactory` задавала приоритеты потоков.

Разновидности реализации

Во всех предыдущих примерах все классы задач реализуют `Runnable`. В очень простых случаях можно пойти по другому пути и использовать наследование непосредственно от `Thread`:

```

//: concurrency/SimpleThread.java
// Прямое наследование от класса Thread.

public class SimpleThread extends Thread {
    private int countDown = 5;
    private static int threadCount = 0;
    public SimpleThread() {
        // Сохранение имени потока:
        super(Integer.toString(++threadCount));
        start();
    }
    public String toString() {
        return "#" + getName() + "(" + countDown + ")", ";
    }
    public void run() {
        while(true) {

```

```

        System.out.print(this);
        if(--countDown == 0)
            return;
    }
}
public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
        new SimpleThread();
}
} /* Output:
#1(5), #1(4), #1(3), #1(2), #1(1), #2(5), #2(4), #2(3),
#2(2), #2(1), #3(5), #3(4), #3(3), #3(2), #3(1), #4(5),
#4(4), #4(3), #4(2), #4(1), #5(5), #5(4), #5(3), #5(2),
#5(1),
*///:~

```

Имена объектов `Thread` задаются вызовом соответствующего конструктора `Thread`. Для обращения к имени в `toString()` используется имя `getName()`.

Другая идиома, которая тоже встречается на практике, — самоуправляемая реализация `Runnable`:

```

//: concurrency/SelfManaged.java
// Объект Runnable, содержащий свой управляющий поток.

public class SelfManaged implements Runnable {
    private int countDown = 5;
    private Thread t = new Thread(this);
    public SelfManaged() { t.start(); }
    public String toString() {
        return Thread.currentThread().getName() +
            "(" + countDown + ")", ";";
    }
    public void run() {
        while(true) {
            System.out.print(this);
            if(--countDown == 0)
                return;
        }
    }
}
public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
        new SelfManaged();
}
} /* Output:
Thread-0(5), Thread-0(4), Thread-0(3), Thread-0(2), Thread-
0(1), Thread-1(5), Thread-1(4), Thread-1(3), Thread-1(2),
Thread-1(1), Thread-2(5), Thread-2(4), Thread-2(3), Thread-
2(2), Thread-2(1), Thread-3(5), Thread-3(4), Thread-3(3),
Thread-3(2), Thread-3(1), Thread-4(5), Thread-4(4), Thread-
4(3), Thread-4(2), Thread-4(1),
*///:~

```

Происходящее не слишком сильно отличается от наследования от `Thread`, если не считать более громоздкого синтаксиса. Однако реализация интерфейса позволяет использовать наследование от другого класса, тогда как с наследованием от `Thread` такой возможности нет.

Обратите внимание на вызов `start()` в конструкторе. Этот пример достаточно прост и, вероятно, безопасен, однако следует учитывать, что запуск потоков в конструкторе может создать проблемы, потому что другая задача может начать выполнение до завершения конструктора, — а это означает, что другая задача сможет обратиться к объекту в нестабильном состоянии. Это еще одна причина, по которой стоит использовать объекты `Executor` вместо явного создания объектов `Thread`.

Иногда бывает разумно скрыть потоковый код внутри класса, используя для этой цели внутренний класс, как показано в следующем примере:

```
//: concurrency/ThreadVariations.java
// Создание потоков во внутренних классах.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

// Использование именованного внутреннего класса:
class InnerThread1 {
    private int countDown = 5;
    private Inner inner;
    private class Inner extends Thread {
        Inner(String name) {
            super(name);
            start();
        }
        public void run() {
            try {
                while(true) {
                    print(this);
                    if(--countDown == 0) return;
                    sleep(10);
                }
            } catch (InterruptedException e) {
                print("interrupted");
            }
        }
        public String toString() {
            return getName() + ": " + countDown;
        }
    }
    public InnerThread1(String name) {
        inner = new Inner(name);
    }
}
```

```
// Использование анонимного внутреннего класса:
class InnerThread2 {
    private int countDown = 5;
    private Thread t;
    public InnerThread2(String name) {
        t = new Thread(name) {
            public void run() {
                try {
                    while(true) {
                        print(this);
                        if(--countDown == 0) return;
                        sleep(10);
                    }
                }
            }
        };
    }
}
```



```

        } catch(InterruptedException e) {
            print("sleep() interrupted");
        }
    }
    public String toString() {
        return getName() + ": " + countDown;
    }
};
t.start();
}
}

// Использование именованной реализации Runnable:
class InnerRunnable1 {
    private int countDown = 5;
    private Inner inner;
    private class Inner implements Runnable {
        Thread t;
        Inner(String name) {
            t = new Thread(this, name);
            t.start();
        }
        public void run() {
            try {
                while(true) {
                    print(this);
                    if(--countDown == 0) return;
                    TimeUnit.MILLISECONDS.sleep(10);
                }
            } catch(InterruptedException e) {
                print("sleep() interrupted");
            }
        }
        public String toString() {
            return t.getName() + ": " + countDown;
        }
    }
    public InnerRunnable1(String name) {
        inner = new Inner(name);
    }
}
}

```

```

// Использование анонимной реализации Runnable:
class InnerRunnable2 {
    private int countDown = 5;
    private Thread t;
    public InnerRunnable2(String name) {
        t = new Thread(new Runnable() {
            public void run() {
                try {
                    while(true) {
                        print(this);
                        if(--countDown == 0) return;
                        TimeUnit.MILLISECONDS.sleep(10);
                    }
                } catch(InterruptedException e) {
                    print("sleep() interrupted");
                }
            }
        });
    }
}
}

```

```

        public String toString() {
            return Thread.currentThread().getName() +
                ": " + countDown;
        }
    }, name);
    t.start();
}
}

// Отдельный метод для выполнения кода в виде задачи:
class ThreadMethod {
    private int countDown = 5;
    private Thread t;
    private String name;
    public ThreadMethod(String name) { this.name = name; }
    public void runTask() {
        if(t == null) {
            t = new Thread(name) {
                public void run() {
                    try {
                        while(true) {
                            print(this);
                            if(--countDown == 0) return;
                            sleep(10);
                        }
                    } catch(InterruptedException e) {
                        print("sleep() interrupted");
                    }
                }
            };
            public String toString() {
                return getName() + ": " + countDown;
            }
        };
        t.start();
    }
}

public class ThreadVariations {
    public static void main(String[] args) {
        new InnerThread1("InnerThread1");
        new InnerThread2("InnerThread2");
        new InnerRunnable1("InnerRunnable1");
        new InnerRunnable2("InnerRunnable2");
        new ThreadMethod("ThreadMethod").runTask();
    }
} /* (Execute to see output) *///:~

```

`InnerThread1` определяет именованный внутренний класс, который расширяет `Thread` и создает экземпляр этого внутреннего класса в конструкторе. Это имеет смысл, если внутренний класс обладает специальными возможностями (новыми методами), к которым нужно обращаться из других методов. Впрочем, чаще поток создается только для использования возможностей `Thread`, поэтому создавать именованный внутренний класс не нужно. В классе `InnerThread2` представлена альтернатива: анонимный внутренний субкласс `Thread` создается в конструкторе и преобразуется восходящим приведением типа в ссылку на `Thread` с именем `t`. Если другим методам класса потребуется обратиться к `t`, они могут сделать это через интерфейс `Thread`; знать для этого точный тип объекта им не нужно.

Третий и четвертый классы в этом примере повторяют первые два класса, но используют интерфейс `Runnable` вместо класса `Thread`.

Класс `ThreadMethod` демонстрирует создание потока внутри метода. Метод вызывается тогда, когда поток готов к запуску, и возвращает управление после начала работы потока. Если поток выполняет второстепенную операцию и не является жизненно важным для работы класса, вероятно, такое решение полезнее и уместнее запуска потока в конструкторе класса.

- 10.** (4) Измените упражнение 5 по образцу класса `ThreadMethod`, чтобы метод `runTask()` получал аргумент с количеством суммируемых чисел Фибоначчи и при каждом вызове `runTask()` возвращал объект `Future`, полученный при вызове `submit()`.

Терминология

Как показано в предыдущем разделе, у разработчика имеется выбор в отношении реализации параллельных программ на языке Java, и этот выбор может быть непростым. Часто проблемы возникают из-за терминологии, используемой при описании технологий параллельного программирования, — особенно там, где задействованы потоки.

Вероятно, вы уже видите различия между выполняемой задачей и потоком, в котором она выполняется; это различие особенно заметно проявляется в библиотеках Java, потому что вы не можете контролировать класс `Thread` (это разделение становится еще более очевидным для объектов `Executor`, которые создают потоки и управляют ими за вас). Вы создаете задачи и каким-то образом связываете их с потоками, чтобы поток управлял выполнением этой задачи.

В Java класс `Thread` сам по себе ничего не делает. Он только управляет выполнением полученной им задачи. Тем не менее в литературе постоянно используются выражения вида «поток выполняет то или иное действие». Создается впечатление, что поток — это и есть задача. Когда я впервые столкнулся с потоками Java, это впечатление было настолько сильным, что я решил, что класс задачи очевидным образом должен наследовать от `Thread`. Добавьте к этому неудачный выбор имени интерфейса `Runnable`, который, как мне кажется, было бы гораздо уместнее назвать `Task`. Если интерфейс явно не делает ничего, кроме инкапсуляции своих методов, схема «*делает-это-able*» годится, но если интерфейс предназначен для выражения концепции более высокого уровня (как `Task`), то правильнее было бы выбрать концептуальное имя.

Проблема заключается в смешении разных уровней абстракции. На концептуальном уровне мы хотим создать задачу, которая выполняется независимо от других задач, поэтому нужно иметь возможность определить задачу и сказать «Вперед!», не беспокоясь о подробностях. Но физическое создание потоков может обходиться достаточно дорого, поэтому для экономии ресурсов нужно организовать управление потоками. Таким образом, с точки зрения реализации задачи разумно отделять от потоков. Вдобавок потоковая модель Java основана на низкоуровневых *p-потоках* (`pthreads`, `POSIX threads`) из языка C, при работе с которыми необходимо во всех подробностях понимать суть происходящего. Эта низкоуровневая сущность отчасти проникла в реализацию Java, поэтому, чтобы оставаться на более высоком уровне абстракции, необходимо действовать дисциплинированно при написании кода (в этой главе я постараюсь действовать именно так).

Чтобы не создавать неясностей при обсуждении, я постараюсь использовать термин «задача» для описания выполняемой работы и термин «поток» — когда речь идет о конкретном механизме, управляющем выполнением задачи. Таким образом, при обсуждении системы на концептуальном уровне можно просто использовать термин «задача», вообще не упоминая управляющего механизма.

Присоединение к потоку

Каждый поток может вызвать метод `join()`, чтобы дождаться завершения другого потока перед своим продолжением. Если поток вызывает `t.join()` для другого потока `t`, то вызывающий поток приостанавливается до тех пор, пока целевой поток `t` не завершится (когда метод `t.isAlive()` вернет `false`).

Вызвать метод `join()` можно также и с аргументом, определяющим тайм-аут (в миллисекундах или в миллисекундах с наносекундами); если целевой поток не закончит работу за означенный период времени, метод `join()` все равно вернет управление инициатору.

Вызов `join()` может быть прерван вызовом метода `interrupt()` для потока-инициатора, соответственно, понадобится конструкция `try-catch`.

Все эти операции продемонстрированы в следующем примере:

```
//: concurrency/Joining.java
// Как работает join().
import static net.mindview.util.Print.*;

class Sleeper extends Thread {
    private int duration;
    public Sleeper(String name, int sleepTime) {
        super(name);
        duration = sleepTime;
        start();
    }
    public void run() {
        try {
            sleep(duration);
        } catch (InterruptedException e) {
            print(getName() + " was interrupted. " +
                "isInterrupted(): " + isInterrupted());
            return;
        }
        print(getName() + " has awakened");
    }
}

class Joiner extends Thread {
    private Sleeper sleeper;
    public Joiner(String name, Sleeper sleeper) {
        super(name);
        this.sleeper = sleeper;
        start();
    }
    public void run() {
        try {
            sleeper.join();
        }
```

```

    } catch(InterruptedException e) {
        print("Interrupted");
    }
    print(getName() + " join completed");
}
}

public class Joining {
    public static void main(String[] args) {
        Sleeper
            sleepy = new Sleeper("Sleepy", 1500),
            grumpy = new Sleeper("Grumpy", 1500);
        Joiner
            dopey = new Joiner("Dopey", sleepy),
            doc = new Joiner("Doc", grumpy);
        grumpy.interrupt();
    }
} /* Output:
Grumpy was interrupted. isInterrupted(): false
Doc join completed
Sleepy has awakened
Dopey join completed
*///:~

```

Класс `Sleeper` — это тип потока, который приостанавливается на время, указанное в его конструкторе. В методе `run()` вызов метода `sleep()` может закончиться при истечении времени задержки, но может и прерваться. В блоке `catch` выводится информация о прерывании вместе со значением, возвращаемым методом `isInterrupted()`. Когда `interrupt()` для данного потока вызывает другой поток, устанавливается флаг, показывающий, что поток был прерван. Однако этот флаг сбрасывается при обработке исключения, поэтому внутри блока `catch` результатом всегда будет `false`. Флаг используется в других ситуациях, где поток может исследовать свое прерванное состояние в стороне от исключения.

`Joiner` — поток, который ожидает пробуждения потока `Sleeper`, вызывая для последнего метод `join()`. В методе `main()` каждому объекту `Joiner` сопоставляется `Sleeper`, и вы можете видеть в результатах работы программы, что если `Sleeper` был прерван или завершился нормально, `Joiner` прекращает работу вместе с потоком `Sleeper`.

Библиотеки Java SE5 `java.util.concurrent` содержат такие инструменты, как `CyclicBarrier` (см. далее в этой главе); они могут оказаться более эффективными, чем метод `join()` из исходной библиотеки потоков.

Чуткие пользовательские интерфейсы

Как уже было упомянуто, одной из причин использования потоков является необходимость создания чутких пользовательских интерфейсов. И хотя мы доберемся до *графических* пользовательских интерфейсов только в главе 22, вы сможете увидеть простой прототип пользовательского интерфейса, работающего с консолью. Следующий пример представлен в двух версиях: одна «увлекается» вычислениями и никогда не доходит до чтения данных с консоли, а вторая помещает вычисления в задачу и таким образом оказывается способной выполнять вычисления и при этом ожидать консольного ввода.

```

//: concurrency/ResponsiveUI.java
// Чуткость пользовательского интерфейса.
// {RunByHand}

class UnresponsiveUI {
    private volatile double d = 1;
    public UnresponsiveUI() throws Exception {
        while(d > 0)
            d = d + (Math.PI + Math.E) / d;
        System.in.read(); // Never gets here
    }
}

public class ResponsiveUI extends Thread {
    private static volatile double d = 1;
    public ResponsiveUI() {
        setDaemon(true);
        start();
    }
    public void run() {
        while(true) {
            d = d + (Math.PI + Math.E) / d;
        }
    }

    public static void main(String[] args) throws Exception {
        //! new UnresponsiveUI(); // Процесс придется уничтожить
        new ResponsiveUI();
        System.in.read();
        System.out.println(d);    // Вывод информации о ходе выполнения
    }
} ///:~

```

UnresponsiveUI выполняет вычисления в бесконечном цикле `while`. Очевидно, что строка, в которой производится чтение с консоли (условие цикла `while` позволяет «обвести вокруг пальца» компилятор, который полагает, что строка с вводом достижима), никогда не будет достигнута. Если вы уберете комментарий со строки, создающей объект `UnresponsiveUI`, и запустите программу, то для выхода из нее процесс придется уничтожить.

Для того чтобы программа реагировала на действия пользователя, вычисления помещаются в метод `run()`. Когда вы нажмете клавишу **Enter**, то увидите, что вычисления на самом деле выполняются на заднем плане, в то время как программа ожидает ввода пользователя.

Группы потоков

Группа потоков (thread group) хранит совокупность потоков. Подвести итог значению групп потоков можно, приведя цитату из Джошуа Блоша¹, архитектора программного обеспечения из Sun, исправившего и значительно улучшившего библиотеку JDK 1.2:

¹ Книга *Effective Java Programming Language Guide*, Джошуа Блош, издательство Addison-Wesley, 2001.

«Группы потоков лучше всего рассматривать как неудачный эксперимент. Просто не обращайте внимания на их существование».

Если вы потратите время и силы, пытаясь понять смысл групп потоков (как это пытался сделать я), то можете поинтересоваться, почему же Sun не сделала несколько официальных заявлений по данной теме пораньше (тот же вопрос относится и к другим изменениям, имевшим место в Java за последние годы). Лауреат Нобелевской премии, экономист Джозеф Стиглиц придерживается жизненной философии, которая, похоже, применима и здесь¹. Она называется *теорией прогрессирующей ответственности*:

«Часто бывает проще продолжить долгую серию чужих ошибок, чем взять на себя ответственность за ее прекращение».

Перехват исключений

Из-за особой природы потоков вы не можете перехватывать исключения, возбужденные из потока. После того как исключение выходит из метода `run()` задачи, оно передается на консоль, если только вы не примете особые меры по перехвату таких исключений. До выхода Java SE5 для перехвата таких исключений использовались группы потоков, но начиная с Java SE5 проблему можно решить при помощи объектов `Executor`, так что теперь вам не нужно ничего знать о группах потоков (разве что для понимания унаследованного кода; дополнительную информацию о группах потоков можно найти в книге *Thinking in Java, 2nd Edition* на сайте www.MindView.net).

Следующая задача всегда выдает исключение, которое выходит за пределы ее метода `run()`, а метод `main()` демонстрирует, что происходит при ее выполнении:

```
//: concurrency/ExceptionThread.java
// {ThrowsException}
import java.util.concurrent.*;

public class ExceptionThread implements Runnable {
    public void run() {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new ExceptionThread());
    }
} ///:~
```

Результат выглядит так (после некоторого сокращения, чтобы данные помещались в строке):

```
java.lang.RuntimeException
  at ExceptionThread.run(ExceptionThread.java:7)
  at ThreadPoolExecutor$Worker.runTask(Unknown Source)
  at ThreadPoolExecutor$Worker.run(Unknown Source)
  at java.lang.Thread.run(Unknown Source)
```

¹ А также еще в ряде мест, возникающих по мере работы с Java. Впрочем, к чему останавливаться на этом? Я консультировал достаточно проектов, к которым также применима данная философия.

Попытка заключить тело `main()` в блок `try-catch` оказывается безуспешной:

```
//: concurrency/NaiveExceptionHandling.java
// {ThrowsException}
import java.util.concurrent.*;
public class NaiveExceptionHandling {
    public static void main(String[] args) {
        try {
            ExecutorService exec =
                Executors.newCachedThreadPool();
            exec.execute(new ExceptionThread());
        } catch(RuntimeException ue) {
            // Эта команда НЕ БУДЕТ выполняться!
            System.out.println("Exception has been handled!");
        }
    }
} ///:~
```

Результат остается тем же, что и в предыдущем примере: перехваченное исключение.

Чтобы решить проблему, мы изменим способ создания потоков объектом `Executor`. В Java SE5 появился новый интерфейс `Thread.UncaughtExceptionHandler`, который позволяет связать с каждым объектом `Thread` обработчик исключения. Метод `Thread.UncaughtExceptionHandler.uncaughtException()` вызывается автоматически тогда, когда поток собирается прекратить свое существование из-за перехваченного исключения. Чтобы использовать этот интерфейс, мы создадим новый тип `ThreadFactory`, который присоединяет новый объект `Thread.UncaughtExceptionHandler` к каждому создаваемому объекту `Thread`. Созданный класс-фабрика передается методу `Executors`, который создает новый экземпляр `ExecutorService`:

```
//: concurrency/CaptureUncaughtException.java
import java.util.concurrent.*;

class ExceptionThread2 implements Runnable {
    public void run() {
        Thread t = Thread.currentThread();
        System.out.println("run() by " + t);
        System.out.println(
            "eh = " + t.getUncaughtExceptionHandler());
        throw new RuntimeException();
    }
}

class MyUncaughtExceptionHandler implements
Thread.UncaughtExceptionHandler {
    public void uncaughtException(Thread t, Throwable e) {
        System.out.println("caught " + e);
    }
}

class HandlerThreadFactory implements ThreadFactory {
    public Thread newThread(Runnable r) {
        System.out.println(this + " creating new Thread");
        Thread t = new Thread(r);
        System.out.println("created " + t);
        t.setUncaughtExceptionHandler(
```



```

        new MyUncaughtExceptionHandler());
    System.out.println(
        "eh = " + t.getUncaughtExceptionHandler());
    return t;
}
}

public class CaptureUncaughtException {
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool(
            new HandlerThreadFactory());
        exec.execute(new ExceptionThread2());
    }
} /* Output: (90% match)
HandlerThreadFactory@de6ced creating new Thread
created Thread[Thread-0,5,main]
eh = MyUncaughtExceptionHandler@1fb8ee3
run() by Thread[Thread-0,5,main]
eh = MyUncaughtExceptionHandler@1fb8ee3
caught java.lang.RuntimeException
*///:~

```

Дополнительная трассировка подтверждает, что потокам, создаваемым фабрикой, назначается новый экземпляр `UncaughtExceptionHandler`. Как видно, перехваченные исключения теперь перехватываются методом `uncaughtException`.

Предыдущий пример позволяет назначать обработчики для каждого конкретного случая. Если вы знаете, что один обработчик события будет использоваться везде, еще проще установить обработчик перехваченных исключений по умолчанию, который задается в статическом поле класса `Thread`:

```

//: concurrency/SettingDefaultHandler.java
import java.util.concurrent.*;

public class SettingDefaultHandler {
    public static void main(String[] args) {
        Thread.setDefaultUncaughtExceptionHandler(
            new MyUncaughtExceptionHandler());
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new ExceptionThread());
    }
} /* Output:
caught java.lang.RuntimeException
*///:~

```

Этот обработчик вызывается только при отсутствии обработчика перехваченных исключений уровня потока. Система проверяет наличие версии уровня потока, а если она не будет найдена — проверяет, имеется ли у группы потоков специализированный метод `uncaughtException()`; если его нет — вызывает `defaultUncaughtExceptionHandler`.

Совместное использование ресурсов

Однопоточную программу можно представить в виде одинокого путника, передвигающегося по своему пространству задачи и выполняющего по одной операции за

один раз. Раз путник один, вам не приходится принимать во внимание проблему двух сущностей, пытающихся конкурировать за один и тот же ресурс, подобно двум людям, которые хотят одновременно поставить машину в одном месте, вдвоем пройти в одну дверь или даже говорить одновременно.

Когда на «сцену» выходит многозадачность, одиночеству приходит конец. Теперь у вас есть сразу два или три потока, которые стремятся получить доступ к одному и тому же ограниченному ресурсу. Если не предотвратить такие коллизии, два потока могут попытаться получить доступ к одному счету в банке, одновременно распечатать два документа на одном принтере, изменить одно и то же значение и т. п.

Некорректный доступ к ресурсам

Рассмотрим следующий пример, в котором класс генерирует четные числа, а другие задачи эти числа потребляют. Единственная функция потребителей — проверка действительности четных чисел.

Мы начнем с определения класса `EvenChecker`, задачи-потребителя, поскольку он будет использоваться во всех последующих примерах. Чтобы отделить `EvenChecker` от различных типов генераторов, с которыми мы будем экспериментировать, мы создадим абстрактный класс с именем `IntGenerator`, содержащий минимум необходимых методов, о которых должен знать `EvenChecker`: `next()` (генерирование следующего значения) и `cancel()` (отмена). Этот класс не реализует интерфейс `Generator`, поскольку он должен производить `int`, а обобщения не поддерживают параметры примитивных типов.

```
//: concurrency/IntGenerator.java

public abstract class IntGenerator {
    private volatile boolean canceled = false;
    public abstract int next();
    // Для отмены:
    public void cancel() { canceled = true; }
    public boolean isCanceled() { return canceled; }
} ///:~
```

`IntGenerator` содержит метод `cancel()` для изменения состояния флага `canceled`, а также метод `isCanceled()` для проверки отмены. Так как флаг `canceled` относится к типу `boolean`, он является атомарным (то есть простые операции — такие, как присваивание и возвращение значения, — не прерываются при выполнении, и вы не «увидите» поле в промежуточном состоянии в середине этих простых операций. Флаг `canceled` также объявлен с ключевым словом `volatile`, чтобы обеспечить его видимость (атомарность и видимость рассматриваются позже в этой главе).

Для тестирования `IntGenerator` можно воспользоваться следующим классом `EvenChecker`:

```
//: concurrency/EvenChecker.java
import java.util.concurrent.*;

public class EvenChecker implements Runnable {
    private IntGenerator generator;
    private final int id;
```

```
public EvenChecker(IntGenerator g, int ident) {
    generator = g;
    id = ident;
}
public void run() {
    while(!generator.isCanceled()) {
        int val = generator.next();
        if(val % 2 != 0) {
            System.out.println(val + " not even!");
            generator.cancel(); // Отменяет все объекты EvenChecker
        }
    }
}
// Тестирование любого типа IntGenerator:
public static void test(IntGenerator gp, int count) {
    System.out.println("Press Control-C to exit");

    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < count; i++)
        exec.execute(new EvenChecker(gp, i));
    exec.shutdown();
}
// Значение по умолчанию для count:
public static void test(IntGenerator gp) {
    test(gp, 10);
}
} ///:~
```

Следует заметить, что в этом примере класс, который может отменяться, не реализует `Runnable`. Вместо этого все задачи `EvenChecker`, зависящие от объекта `IntGenerator`, проверяют, был ли он отменен (как видно из `run()`). Таким образом, задачи, совместно использующие общий ресурс (`IntGenerator`), следят за ресурсом, который должен получить сигнал завершения. Тем самым устраняется так называемая *ситуация гонки* (*race condition*), при которой две и более задачи торопятся отреагировать на некоторое условие; они вступают в конфликт или по иной причине создают непоследовательные результаты. Вы должны тщательно продумать все возможные сбои в параллельной системе и защититься от них. Например, задача не может зависеть от другой задачи, потому что порядок завершения задач не гарантирован. Зависимость задач от стороннего объекта устраняет потенциальную ситуацию гонки.

Метод `test()` создает и выполняет проверку любого типа `IntGenerator`, запуская набор задач `EvenChecker`, использующих один объект `IntGenerator`. Если `IntGenerator` вызывает сбой, `test()` сообщает о нем и возвращает управление; в противном случае его приходится завершать клавишами `Ctrl-C`.

Задачи `EvenChecker` постоянно читают и проверяют значения из связанного с ними объекта `IntGenerator`. Если при вызове `generator.isCanceled()` получено значение `true`, `run()` возвращает управление; это сообщает `Executor` в `EvenChecker.test()`, что задача завершена. Любая задача `EvenChecker` может вызвать `cancel()` для связанного с ней объекта `IntGenerator`, что заставит все остальные задачи `EvenChecker`, использующие этот объект `IntGenerator`, корректно завершиться. Как будет показано далее, вы увидите, что Java содержит более универсальные механизмы завершения потоков.

Первый объект `IntGenerator`, который мы рассмотрим, в своем методе `next()` создает серию четных значений:

```

//: concurrency/EvenGenerator.java
// Коллизии между потоками.

public class EvenGenerator extends IntGenerator {
    private int currentEvenValue = 0;
    public int next() {
        ++currentEvenValue; // Опасная точка
        ++currentEvenValue;
        return currentEvenValue;
    }
    public static void main(String[] args) {
        EvenChecker.test(new EvenGenerator());
    }
} /* Output: (Sample)
Press Ctrl-C to exit
89476993 not even!
89476993 not even!
*///:~

```

Одна задача может вызвать `next()` после того, как другая задача выполнила первое увеличение `currentEvenValue`, но не второе (в месте, отмеченном комментарием «Опасная точка»). В результате значение оказывается в «некорректном» состоянии. Чтобы доказать, что это возможно, `EvenChecker.test()` создает группу объектов `EvenChecker` для непрерывного чтения вывода `EvenGenerator` и проверки их всех на четность. Если среди значений окажется нечетное, программа сообщает об ошибке и завершается.

Рано или поздно в программе произойдет сбой, потому что задачи `EvenChecker` могут обратиться к информации `EvenGenerator` в тот момент, когда она находится в «некорректном» состоянии. Однако проблема может проявиться только после того, как `EvenGenerator` пройдет много циклов — это зависит от особенностей вашей операционной системы и других подробностей реализации. Чтобы ускорить возникновение сбоя, попробуйте вставить вызов `yield()` между первым и вторым приращением. Это одна из проблем многопоточных программ — на первый взгляд они работают нормально, тогда как на самом деле в них присутствует ошибка, просто ее вероятность очень низка.

Важно, что сама операция приращения состоит из нескольких шагов, а задачи могут приостанавливаться механизмом потоков в середине инкремента — другими словами, инкремент в Java не является атомарной операцией. Таким образом, выполнение даже одного инкремента без защиты задачи небезопасно.

Разрешение спора за разделяемые ресурсы

Этот пример показательно иллюстрирует основную проблему потоков. Вы никогда не знаете, когда поток будет выполняться. Вообразите, что вы сидите за столом с вилкой в руках, собираетесь съесть последний, самый лакомый кусочек, который лежит перед вами здесь же, на блюдечке, и только вы дотрагиваетесь до него вилкой, как он исчезает (так как ваш поток был внезапно приостановлен, и в это время другой поток не постеснялся «стянуть» у вас еду). Вот с такой проблемой нам пришлось столкнуться при написании выполняемых одновременно и использующих общие ресурсы программ. Чтобы многопоточность работала, необходим способ, предотвращающий возможность состязания двух потоков за одну «тарелку», по крайней мере, во время опасных операций.

Предотвратить такое столкновение интересов довольно просто — надо блокировать ресурс для других потоков, пока он находится в ведении одного потока. Первый поток, получивший доступ к ресурсу, вешает на него «замок», и тогда все остальные потоки не смогут получить этот ресурс до тех пор, пока «замок» не будет снят, и только после другой поток овладеет ресурсом и заблокирует его и т. д. Если переднее сиденье машины является для детей ресурсом ограниченного характера, то ребенок, первым крикнувший «Чур, я спереди!», отстоял свое право на «блокировку».

Для решения проблемы соперничества потоков фактически все многопоточные схемы *синхронизируют доступ к разделяемым ресурсам*. Это означает, что доступ к разделяемому ресурсу в один момент времени может получить только одна задача. Чаще всего это выполняется помещением фрагмента кода в предложение блокировки так, что одновременно пройти по этому фрагменту кода может только одна задача. Поскольку такое предложение блокировки дает эффект *взаимного исключения* (mutual exclusion), этот механизм часто называют *мьютексом* (mutex).

Вспомните свою ванную комнату — несколько людей (задачи, находящиеся под управлением потоков) могут захотеть эксклюзивно владеть ею (разделяемым ресурсом). Для доступа в ванную человек стучится в дверь, желая проверить, не занята ли она. Если ванная свободна, он входит в нее и запирает дверь. Любая другая задача, желающая оказаться внутри, «блокирована» в этом действии, поэтому ей приходится ждать у двери, пока ванная не освободится.

Аналогия немного нарушается, когда дверь в ванную комнату снова открывается, и приходит время передать доступ другой задаче. Как люди на самом деле не становятся в очередь, так и здесь мы точно не знаем, кто «зайдет в ванную» следующим, потому что поведение планировщика недетерминировано. Существует гипотетическая группа заблокированных задач, толкущихся у двери, и когда задача, которая занимала «ванную», разблокирует ее и уходит, та задача, что окажется ближе всех к двери, и «войдет» в нее. Как уже было замечено, планировщику можно давать подсказки методами `yield()` и `setPriority()`, но эти подсказки не обязательно будут иметь эффект, в зависимости от вашей платформы и реализации JVM.

В Java есть встроенная поддержка для предотвращения спора за ресурсы в виде ключевого слова `synchronized`. Когда задача желает выполнить фрагмент кода, охраняемый словом `synchronized`, она проверяет, захватывает ее, выполняет код и снимает блокировку.

Совместно используемый ресурс чаще всего является блоком памяти, представляющим объект, но это также может быть файл, порт ввода-вывода и даже что-нибудь вроде принтера. Для управления доступом к совместному ресурсу вы для начала помещаете его внутрь объекта. После этого любой метод, получающий доступ к ресурсу, может быть объявлен как `synchronized`. Это означает, что если задача выполняется внутри одного из объявленных как `synchronized` методов, все остальные задачи не сумеют зайти в свои `synchronized`-методы до тех пор, пока первая задача не вернет управление из своего вызова.

Так как обычно поля класса объявляются закрытыми (`private`), а доступ к их памяти осуществляется только посредством методов, можно предотвратить конфликты, объявив такие методы синхронизированными:

```
synchronized void f() { /* ... */ }
synchronized void g() { /* ... */ }
```

Каждый объект содержит объект простой блокировки (также называемый *монитором*), который автоматически является частью этого объекта (вам не придется писать для нее специального кода). Когда вы вызываете любой синхронизированный (*synchronized*) метод, объект переходит в состояние блокировки, и пока этот метод не закончит свою работу и не снимет блокировку, другие синхронизированные методы для объекта не могут быть вызваны. В только что рассмотренном примере, если одна задача вызывает для объекта метод *f()*, другая задача не сможет вызвать метод *f()* или *g()* до тех пор, пока метод *f()* не завершит свою работу и не сбросит блокировку. Таким образом, существует блокировка, которая совместно используется всеми синхронизированными методами конкретного объекта, и эта блокировка предотвращает запись в память объекта несколькими задачами одновременно.

Учтите, что в многопоточной среде не менее важно объявлять все поля закрытыми (*private*); в противном случае ключевое слово *synchronized* не сможет помешать другой задаче обратиться к полю напрямую, и это приведет к конфликту.

Одна задача может блокировать объект многократно. Это происходит, когда метод вызывает другой метод того же объекта, который, в свою очередь, вызывает еще один метод того же объекта и т. д. Виртуальная машина JVM следит за тем, сколько раз объект был заблокирован. Если объект не был заблокирован, его счетчик равен нулю. Когда задача захватывает объект в первый раз, счетчик становится равным единице. Каждый раз, когда задача снова овладевает объектом блокировки того же объекта, счетчик увеличивается. Естественно, что все это разрешается только задаче, инициировавшей первичную блокировку. При покидании задачей синхронизированного метода счетчик уменьшается на единицу до тех пор, пока не делается равным нулю, после чего объект блокировки данного объекта становится доступен другим задачам.

Также существует отдельная блокировка *уровня класса* (часть объекта *Class*), которая следит за тем, чтобы статические (*static*) синхронизированные (*synchronized*) методы не использовали одновременно общие статические данные класса.

Когда следует применять синхронизацию? Используйте правило синхронизации Брайана¹:

«Если вы записываете данные в переменную, которая может быть затем прочитана другим потоком, или читаете данные из переменной, которая могла быть записана другим потоком, вы должны использовать синхронизацию. Кроме того, и читающий, и записывающий потоки должны синхронизироваться по одной блокировке».

Если класс содержит более одного метода, работающего с критическими данными, вы должны синхронизировать все задействованные методы. Если синхронизировать только один из методов, то другие методы могут игнорировать блокировку. Это важный момент: каждый метод, обращающийся к критическому, совместно используемому ресурсу, должен быть синхронизирован — иначе синхронизация работать не будет.

¹ От Брайана Гетца, соавтора книги «Java Concurrency in Practice» (Брайан Гетц, Тим Пейерс, Джошуа Блош, Джозеф Боубир, Дэвид Холмс и Дуг Ли, издательство Addison-Wesley, 2006).

Синхронизация EvenGenerator

Добавление ключевого слова `synchronized` в `EvenGenerator.java` позволяет предотвратить нежелательный доступ со стороны потоков:

```
//: concurrency/SynchronizedEvenGenerator.java
// Упрощение мьютексов с использованием
// ключевого слова synchronized.
// {RunByHand}

public class
SynchronizedEvenGenerator extends IntGenerator {
    private int currentEvenValue = 0;
    public synchronized int next() {
        ++currentEvenValue;
        Thread.yield(); // Cause failure faster
        ++currentEvenValue;
        return currentEvenValue;
    }
    public static void main(String[] args) {
        EvenChecker.test(new SynchronizedEvenGenerator());
    }
} ///:~
```

Между двумя инкрементами вставляется вызов `Thread.yield()`, повышающий вероятность переключения контекста во время нахождения `currentEvenValue` в некорректном состоянии. Так как мьютекс не позволяет нескольким задачам одновременно находиться в критической секции, это не приведет к ошибке.

Первая задача, входящая в `next()`, устанавливает блокировку, а все остальные задачи, пытающиеся получить блокировку, не смогут этого сделать до тех пор, пока она не будет снята первой задачей. В этой точке механизм планирования выбирает другую задачу, ожидающую получения блокировки. Таким образом, код, защищенный мьютексом, в любой момент времени может выполняться только одной задачей.

- 11. (3)** Создайте класс с двумя полями данных и методом, который работает с этими полями в несколько этапов, чтобы во время выполнения метода поля находились в «некорректном состоянии» (по тому определению, которое вы установите). Добавьте методы чтения полей; создайте несколько потоков, в которых эти методы будут вызываться, и покажите, что данные находятся в некорректном состоянии. Решите проблему при помощи ключевого слова `synchronized`.

Использование объектов Lock

В библиотеку Java SE5 `java.util.concurrent` также включен явный механизм мьютексов, определенных в `java.util.concurrent.locks`. Операции создания, установления и снятия блокировки с объектом `Lock` выполняются явно, и код получается менее элегантным, чем во встроенной форме. С другой стороны, такой подход обладает большей гибкостью при решении некоторых видов задач. Вот как выглядит пример `SynchronizedEvenGenerator.java`, переписанный для явного использования объектов `Lock`:

```
//: concurrency/MutexEvenGenerator.java
// Предотвращение конфликтов между потоками
// с использованием мьютексов.
```

продолжение ➤

```
// {RunByHand}
import java.util.concurrent.locks.*;

public class MutexEvenGenerator extends IntGenerator {
    private int currentEvenValue = 0;
    private Lock lock = new ReentrantLock();
    public int next() {
        lock.lock();
        try {
            ++currentEvenValue;
            Thread.yield(); // Повышение вероятности ошибки
            ++currentEvenValue;
            return currentEvenValue;
        } finally {
            lock.unlock();
        }
    }
    public static void main(String[] args) {
        EvenChecker.test(new MutexEvenGenerator());
    }
} ///:~
```

`MutexEvenGenerator` добавляет мьютекс с именем `lock` и использует методы `lock()` и `unlock()` для создания критической секции в `next()`. При использовании объектов `Lock` важно понять и запомнить эту идиому: сразу же после вызова `lock()` размещается команда `try-finally` с `unlock()` в секции `finally` — это единственный способ гарантировать освобождение блокировки в любом случае. Обратите внимание: в блоке `try` должна находиться команда `return`, которая гарантирует, что команда `unlock()` не будет выполнена слишком рано и данные не станут доступными для другой задачи.

Хотя конструкция `try-finally` требует большего объема кода, чем ключевое слово `synchronized`, она также демонстрирует одно из преимуществ явных объектов `Lock`. Если при использовании ключевого слова `synchronized` произойдет ошибка, программа выдает исключение, но у вас не будет возможности выполнить какие-либо завершающие действия, чтобы система осталась в нормальном состоянии. При явном создании объектов `Lock` можно обеспечить корректность состояния системы при помощи блока `finally`.

В общем случае ключевое слово `synchronized` уменьшает объем кода и снижает вероятность ошибок пользователя, так что объекты `Lock` обычно применяются только в особых ситуациях. Например, с ключевым словом `synchronized` невозможно обработать неудачную попытку получения блокировки или прервать попытки получения блокировки после истечения заданного промежутка времени — в таких случаях необходимо использовать библиотеку `concurrent`:

```
//: concurrency/AttemptLocking.java
// Объекты Lock из библиотеки concurrent позволяют
// отказаться от попыток получения блокировки
// по тайм-ауту.
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class AttemptLocking {
    private ReentrantLock lock = new ReentrantLock();
    public void untimed() {
```



```

    boolean captured = lock.tryLock();
    try {
        System.out.println("tryLock(): " + captured);
    } finally {
        if(captured)
            lock.unlock();
    }
}

public void timed() {
    boolean captured = false;
    try {
        captured = lock.tryLock(2, TimeUnit.SECONDS);
    } catch(InterruptedException e) {
        throw new RuntimeException(e);
    }
    try {
        System.out.println("tryLock(2, TimeUnit.SECONDS): " +
            captured);
    } finally {
        if(captured)
            lock.unlock();
    }
}

public static void main(String[] args) {
    final AttemptLocking al = new AttemptLocking();
    al.untimed(); // True -- блокировка доступна
    al.timed();   // True -- блокировка доступна
    // теперь создаем отдельную задачу для получения блокировки:
    new Thread() {
        { setDaemon(true); }
        public void run() {
            al.lock.lock();
            System.out.println("acquired");
        }
    }.start();
    Thread.yield(); // Предоставляем возможность 2-й задаче
    al.untimed();   // False -- блокировка захвачена задачей
    al.timed();     // False -- блокировка захвачена задачей
}

/* Output:
tryLock(): true
tryLock(2, TimeUnit.SECONDS): true
acquired
tryLock(): false
tryLock(2, TimeUnit.SECONDS): false
*///:~

```

Объект `ReentrantLock` дает возможность попытаться получить блокировку с неудачным исходом, чтобы, если блокировка уже была кем-то захвачена, ваша программа могла заняться чем-то другим вместо ожидания ее освобождения, как в методе `untimed()`. В методе `timed()` программа пытается получить блокировку в течение двух секунд (обратите внимание на использование класса `Java SE5 TimeUnit` для задания единиц времени). В методе `main()` отдельный объект `Thread` создается в виде анонимного класса и получает блокировку, чтобы у методов `untimed()` и `timed()` был ресурс, за который они бы могли конкурировать.

Объекты `Lock` также позволяют управлять установлением и снятием блокировки на более детализированном уровне, чем встроенная блокировка `synchronized`. Эта возможность может пригодиться для реализации специализированных структур синхронизации, например *цепной блокировки* (*hand-over-hand locking*), используемой при переборе узлов связанных списков; код перебора должен установить блокировку следующего узла до того, как снимать блокировку текущего.

Атомарность и видимость изменений

Очень часто в дискуссиях, посвященных механизму потоков в Java, можно услышать такое утверждение: «атомарные операции не требуют синхронизации». *Атомарная операция* — это операция, которую не может прервать планировщик потоков — если она начинается, то продолжается до завершения, без возможности переключения контекста. Атомарность ненадежна и коварна — используйте ее вместо синхронизации только в том случае, если вы являетесь экспертом в области параллельного выполнения или можете воспользоваться помощью эксперта. Если вы считаете, что достаточно умны для подобных «игр с огнем», проверьте себя:

Критерий Гетца¹: Если вы можете написать высокопроизводительную виртуальную машину Java для современного микропроцессора, то вашей квалификации достаточно для того, чтобы хотя бы *подумать* о том, можно ли обойтись без синхронизации².

Об атомарности полезно *знать* — как и о том, что наряду с другими нетривиальными средствами она использовалась для реализации некоторых хитроумных компонентов библиотеки `java.util.concurrent`. Но всеми силами боритесь с искушением использовать ее в своем коде; см. приведенное выше правило синхронизации Брайана.

Атомарность распространяется на «простые операции» с примитивными типами, кроме `long` и `double`. Операции чтения и записи примитивных переменных типов, отличных от `long` и `double`, гарантированно выполняются как неделимые (атомарные). Однако JVM разрешается выполнять чтение и запись 64-разрядных величин (переменных `long` и `double`) за две отдельные 32-разрядные операции, повышая вероятность того, что в середине чтения или записи произойдет переключение контекста, и задачи увидят некорректные результаты (иногда это называется «разрывом слов» (*word tearing*), потому что задача «видит» значение после его частичного изменения). Впрочем, атомарность (простого присваивания и чтения) можно обеспечить, определяя переменную `long` или `double` с ключевым словом `volatile` (учтите, что до выхода Java SE5 ключевое слово `volatile` работало некорректно). Разные реализации JVM могут предоставлять более сильные гарантии, но вы не должны полагаться на платформенно-зависимые аспекты.

Итак, атомарные операции не могут прерываться механизмом потоков. Опытный программист может воспользоваться этим обстоятельством для написания кода без блокировок, не требующего синхронизации. Но даже такая формулировка является

¹ На основании (лишь отчасти) шуточных комментариев Брайана Гетца — эксперта в области параллельного выполнения, который участвовал в написании этой главы.

² И еще один, парный критерий: «Если кто-то утверждает, что многопоточность — не такая уж сложная область, убедитесь в том, что этот человек не отвечает за принятие важных решений в вашем проекте. Если же отвечает — у вас большие неприятности».

упрощенной. Иногда атомарная операция кажется безопасной, но таковой не является. Вероятно, большинство читателей книги не пройдет проверку по критерию Гетца, а следовательно, не обладает достаточной квалификацией для замены синхронизации атомарными операциями. Попытка устранения синхронизации обычно является признаком преждевременной оптимизации, которая только создаст вам лишние проблемы без сколько-нибудь существенного выигрыша.

В многопроцессорных системах (которые теперь реализуются в форме многоядерных процессоров, то есть нескольких процессоров на одной микросхеме) видимость играет существенно более важную роль, чем в однопроцессорных системах. Изменения, вносимые одной задачей, — даже являющиеся атомарными в смысле невозможности прерывания, — могут быть невидимыми для других задач (например, изменения могут временно храниться в локальном кэше процессора), так что разные задачи могут обладать разным представлением о состоянии приложения. Напротив, с механизмом синхронизации изменения, внесенные одной задачей в однопроцессорной системе, будут видимы в границах приложения. Без синхронизации видимость изменений недетерминирована.

Ключевое слово `volatile` также обеспечивает видимость изменений в приложении. Если поле объявляется с ключевым словом `volatile`, это означает, что сразу же после записи в это поле все операции чтения «увидят» изменение. Это утверждение истинно даже при использовании локальных кэшей — `volatile`-поля немедленно записываются в основную память, и чтение выполняется из основной памяти.

Важно понимать, что атомарность и видимость изменений являются разными концепциями. Атомарная операция с полем без ключевого слова `volatile` не обязательно будет актуализирована в основной памяти, поэтому другая задача, читающая значение поле, может не увидеть новое значение. Если сразу несколько задач обращаются к полю, это поле следует объявить с ключевым словом `volatile`; в противном случае обращения к полю должны осуществляться только через механизм синхронизации. Синхронизация также вынуждает запись данных в основную память, и если поле полностью защищено синхронизированными методами или блоками, объявлять его `volatile` не обязательно.

Любая запись, которую выполняет задача, будет видима для этой задачи, так что объявлять с ключевым словом `volatile` поле, видимое только в пределах задачи, не нужно.

Ключевое слово `volatile` не работает в том случае, когда значение поля зависит от его предыдущего значения (например, при увеличении счетчика), а также для полей, значения которых ограничиваются значениями других полей (например, границы `lower` и `upper` класса `Range` должны подчиняться ограничению `lower <= upper`).

Как правило, использование `volatile` вместо `synchronized` безопасно только в том случае, если класс имеет всего одно изменяемое поле. И снова первоочередным вариантом должно быть использование ключевого слова `synchronized` — это самый безопасный вариант, а все остальное сопряжено с риском.

Какие операции выполняются как атомарные? Присваивание и возвращение значения в поле обычно атомарны. Однако в C++ атомарными *могут* быть даже следующие операции:

```
i++; // Операция может быть атомарной в C++
i += 2; // Операция может быть атомарной в C++
```

Впрочем, в C++ все зависит от компилятора и процессора. На C++ невозможно написать кросс-платформенный код, зависящий от атомарности операций, потому что C++, в отличие от Java (SE5), не имеет согласованной *модели памяти*¹.

В Java эти операции определенно *не являются* атомарными, как видно из инструкций JVM, сгенерированных для следующих методов:

```
//: concurrency/Atomicity.java
// {Exec: javap -c Atomicity}

public class Atomicity {
    int i;
    void f1() { i++; }
    void f2() { i += 3; }
} /* Output: (Sample)
...
void f1();
Code:
  0:      aload_0
  1:      dup
  2:      getfield      #2; //Поле i:I
  5:      iconst_1
  6:      iadd
  7:      putfield      #2; //Поле i:I
 10:      return

void f2();
Code:
  0:      aload_0
  1:      dup
  2:      getfield      #2; //Поле i:I
  5:      iconst_3
  6:      iadd
  7:      putfield      #2; //Поле i:I
 10:      return
*///:~
```

Для каждой команды генерируются инструкции `get` и `put` с промежуточными инструкциями. Таким образом, между чтением и записью другая задача может изменить поле, а следовательно, операции атомарными не являются.

Если слепо следовать идее атомарности, вы увидите, что метод `getValue()` в следующей программе соответствует определению:

```
//: concurrency/AtomicityTest.java
import java.util.concurrent.*;

public class AtomicityTest implements Runnable {
    private int i = 0;
    public int getValue() { return i; }
    private synchronized void evenIncrement() { i++; i++; }
    public void run() {
        while(true)

```

¹ Эта проблема должна быть исправлена с выходом запланированного стандарта C++.

```
        evenIncrement();
    }
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        AtomicityTest at = new AtomicityTest();
        exec.execute(at);
        while(true) {
            int val = at.getValue();
            if(val % 2 != 0) {
                System.out.println(val);
                System.exit(0);
            }
        }
    }
}
/* Output: (Sample)
191583767
*///:~
```

Тем не менее программа обнаружит нечетные значения и завершится. Хотя `return i` является атомарной операцией, отсутствие синхронизации позволяет прочесть объект в нестабильном промежуточном состоянии. Вдобавок, поскольку переменная `i` тоже объявлена без `volatile`, могут возникнуть проблемы с видимостью. Оба метода, `getValue()` и `evenIncrement()`, должны быть синхронизированы. Только эксперты в области параллельного выполнения могут пытаться применять оптимизации в подобных ситуациях; еще раз вспомните правило синхронизации Брайана.

В качестве второго примера рассмотрим кое-что еще более простое: класс, производящий серийные номера¹. Каждый раз при вызове метода `nextSerialNumber()` он должен возвращать уникальное значение:

```
//: concurrency/SerialNumberGenerator.java

public class SerialNumberGenerator {
    private static volatile int serialNumber = 0;
    public static int nextSerialNumber() {
        return serialNumber++; // Небезопасно для потоков
    }
}
*///:~
```

Вряд ли можно представить себе класс тривиальнее `SerialNumberGenerator`, и если вы ранее работали с языком C++ или имеете другие низкоуровневые навыки, то, видимо ожидаете, что операция инкремента будет атомарной, так как инкремент обычно реализуется одной инструкцией микропроцессора (при всей ненадежности и отсутствии кросс-платформенности). Но как упоминалось выше, в виртуальной машине Java инкремент *не является* атомарным и состоит из чтения и записи, соответственно, ниша для проблем с потоками найдется даже в такой простой программе. Как вы увидите, проблема здесь связана не с видимостью изменений, а с тем, что `nextSerialNumber()` обращается к совместному, изменяемому значению без синхронизации.

Поле `serialNumber` объявлено как `volatile` потому, что каждый поток может обладать локальным стеком и поддерживать в нем копии некоторых локальных переменных.

¹ Источником вдохновения послужила книга *Effective Java* Джошуа Блоша, изд-во Addison-Wesley, 2001, с. 190.

Если вы объявляете переменную как `volatile`, то тем самым указываете компилятору не проводить оптимизацию, а это приведет к удалению операций чтения и записи, обеспечивающих синхронизацию поля с локальными данными потока. Операции чтения и записи производятся прямо в памяти, без кэширования. Ключевое слово `volatile` также ограничивает изменение порядка обращений компилятором в ходе оптимизации. Тем не менее `volatile` не меняет того факта, что инкремент не является атомарной операцией.

По сути, поле следует объявлять `volatile` тогда, когда к нему могут одновременно обращаться несколько задач и по крайней мере одно из этих обращений является записью. Например, поле, используемое как флаг для остановки задачи, должно быть объявлено `volatile`; в противном случае этот флаг может быть кэширован в регистре, а при изменении флага за пределами задачи кэшированное значение не изменится, и задача не будет знать, что она должна остановиться.

Для тестирования нам понадобится множество, которое не потребует переизбытка памяти в том случае, если обнаружение проблемы отнимет много времени. Показанный здесь класс `CircularSet` многократно использует память, в которой хранятся целые числа (`int`), полагая, что когда запись в множество начинается по новому кругу, вероятность столкновения с перезаписанными значениями минимальна. Методы `add()` и `contains()` объявлены как `synchronized`, чтобы избежать коллизий:

```
//: concurrency/SerialNumberChecker.java
// Операции, которые кажутся безопасными,
// при участии потоков перестают быть таковыми.
// {Args: 4}
import java.util.concurrent.*;

// Многократно использует память во избежание ее исчерпания:
class CircularSet {
    private int[] array;
    private int len;
    private int index = 0;
    public CircularSet(int size) {
        array = new int[size];
        len = size;
        // Инициализируем значением, которое не производится
        // классом SerialNumberGenerator:
        for(int i = 0; i < size; i++)
            array[i] = -1;
    }
    public synchronized void add(int i) {
        array[index] = i;
        // "Перенос" индекса с перезаписью старых элементов:
        index = ++index % len;
    }
    public synchronized boolean contains(int val) {
        for(int i = 0; i < len; i++)
            if(array[i] == val) return true;
        return false;
    }
}

public class SerialNumberChecker {
    private static final int SIZE = 10;
```

```

private static CircularSet serials =
    new CircularSet(1000);
private static ExecutorService exec =
    Executors.newCachedThreadPool();
static class SerialChecker implements Runnable {
    public void run() {
        while(true) {
            int serial =
                SerialNumberGenerator.nextSerialNumber();
            if(serials.contains(serial)) {
                System.out.println("Duplicate: " + serial);
                System.exit(0);
            }
            serials.add(serial);
        }
    }
}

public static void main(String[] args) throws Exception {
    for(int i = 0; i < SIZE; i++)
        exec.execute(new SerialChecker());
    // Остановка через n секунд при наличии аргумента:
    if(args.length > 0) {
        TimeUnit.SECONDS.sleep(new Integer(args[0]));
        System.out.println("No duplicates detected");
        System.exit(0);
    }
}
} /* Output: (Sample)
Duplicate: 8468656
*///:~
} ///:~

```

В классе `SerialNumberChecker` содержится статическое поле `CircularSet`, хранящее все произведенные серийные номера, и вложенный класс `SerialChecker`, который проверяет их уникальность. Создавая несколько потоков, претендующих на серийные номера, вы обнаружите, что при достаточно длительном выполнении какой-нибудь из них довольно быстро получит уже имеющийся номер. Проблема решается добавлением к методу `nextSerialNumber()` ключевого слова `synchronized`.

Предполагается, что безопасными атомарными операциями являются чтение и присвоение примитивов. Однако, как мы увидели в программе `AtomicityTest.java`, все так же просто использовать атомарную операцию для объекта, который находится в нестабильном промежуточном состоянии, так что ожидать, что какие-то предположения оправдаются, дело ненадежное. Самое разумное, что можно сделать в такой ситуации, — последовать правилу синхронизации Брайана.

- 12.** (3) Исправьте ошибку в `AtomicityTest.java`, используя ключевое слово `synchronized`. Сможете ли вы продемонстрировать, что теперь программа работает верно?
- 13.** (1) Исправьте ошибку в `SerialNumberChecker.java`, используя ключевое слово `synchronized`. Сможете ли вы продемонстрировать, что теперь программа работает верно?

Атомарные классы

В Java SE5 появились специальные классы атомарных переменных: `AtomicInteger`, `AtomicLong`, `AtomicReference` и т. д., обеспечивающие выполнение атомарных условных обновлений в форме:

```
boolean compareAndSet(expectedValue, updateValue);
```

Они предназначены для использования оптимизаций атомарности машинного уровня, доступных на некоторых современных процессорах, так что вам не нужно беспокоиться о них. Иногда они могут пригодиться при обычном программировании, но и это бывает только там, где задействована оптимизация производительности. Например, программу `AtomicityTest.java` можно переписать для использования `AtomicInteger`:

```
//: concurrency/AtomicIntegerTest.java
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;

public class AtomicIntegerTest implements Runnable {
    private AtomicInteger i = new AtomicInteger(0);
    public int getValue() { return i.get(); }
    private void evenIncrement() { i.addAndGet(2); }
    public void run() {
        while(true)
            evenIncrement();
    }
    public static void main(String[] args) {
        new Timer().schedule(new TimerTask() {
            public void run() {
                System.err.println("Aborting");
                System.exit(0);
            }
        }, 5000); // Завершение через 5 секунд
        ExecutorService exec = Executors.newCachedThreadPool();
        AtomicIntegerTest ait = new AtomicIntegerTest();
        exec.execute(ait);
        while(true) {
            int val = ait.getValue();
            if(val % 2 != 0) {
                System.out.println(val);
                System.exit(0);
            }
        }
    }
} ///:~
```

Здесь ключевое слово `synchronized` было исключено за счет использования `AtomicInteger`. Так как программа работает без ошибок, в нее добавлен объект `Timer`, который автоматически прерывает выполнение через 5 секунд.

Ниже приведена программа `MutexEvenGenerator.java`, переписанная с использованием `AtomicInteger`:

```
//: concurrency/AtomicEvenGenerator.java
// Атомарные классы иногда могут пригодиться
// в обычном коде.
```



```
// {RunByHand}
import java.util.concurrent.atomic.*;

public class AtomicEvenGenerator extends IntGenerator {
    private AtomicInteger currentEvenValue =
        new AtomicInteger(0);
    public int next() {
        return currentEvenValue.addAndGet(2);
    }
    public static void main(String[] args) {
        EvenChecker.test(new AtomicEvenGenerator());
    }
} ///:~
```

И снова использование `AtomicInteger` позволяет исключить все остальные формы синхронизации.

Следует подчеркнуть, что классы `Atomic` проектировались для построения классов из `java.util.concurrent`, используйте их, только если вы твердо уверены в отсутствии возможных проблем. Обычно безопаснее положиться на блокировки (с ключевым словом `synchronized` или явными объектами `Lock`).

- 14.** (4) Продемонстрируйте масштабируемость кода `java.util.Timer` — создайте программу, которая генерирует множество объектов `Timer`, выполняющих какую-нибудь простую операцию по тайм-ауту.

Критические секции

Иногда бывает нужно предотвратить доступ нескольких потоков только к *части* кода, а не к методу в целом. Фрагмент кода, который изолируется таким способом, называется *критической секцией* (critical section), для его создания также применяется ключевое слово `synchronized`. На этот раз слово `synchronized` указывает на объект, блокировка которого должна использоваться для синхронизации последующего фрагмента кода:

```
synchronized(syncObject) {
    // К этому коду в любой момент времени
    // может обращаться только одна задача
}
```

Такая конструкция иначе называется *синхронизированной блокировкой* (`synchronized block`); чтобы войти в защищенный участок кода, сначала программа должна получить блокировку у `syncObject`. Если эта блокировка уже установлена другой задачей, вход в критическую секцию запрещается до тех пор, пока блокировка не будет снята.

Следующий пример сравнивает два подхода к синхронизации, показывая, насколько увеличивается время, предоставляемое потокам для доступа к объекту при использовании синхронизированной блокировки вместо синхронизированных методов. Вдобавок он демонстрирует, как незащищенный класс может «выжить» в многозадачной среде, если он управляется и защищается другим классом:

```
//: concurrency/CriticalSection.java
// Синхронизация блоков кода вместо целых методов. Также
// демонстрирует защиту класса, небезопасного
// по отношению к потокам, другим - защищенным - классом.
```

продолжение ➤

```

package concurrency;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;

class Pair { // Потокно-небезопасный класс
    private int x, y;
    public Pair(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Pair() { this(0, 0); }
    public int getX() { return x; }
    public int getY() { return y; }
    public void incrementX() { x++; }
    public void incrementY() { y++; }
    public String toString() {
        return "x: " + x + ", y: " + y;
    }
    public class PairValuesNotEqualException
    extends RuntimeException {
        public PairValuesNotEqualException() {
            super("Pair values not equal: " + Pair.this);
        }
    }
    // Произвольный инвариант - переменные должны быть равны:
    public void checkState() {
        if(x != y)
            throw new PairValuesNotEqualException();
    }
}

// Защита Pair в потоково-безопасном классе:
abstract class PairManager {
    AtomicInteger checkCounter = new AtomicInteger(0);
    protected Pair p = new Pair();
    private List<Pair> storage =
        Collections.synchronizedList(new ArrayList<Pair>());
    public synchronized Pair getPair() {
        // Создание копии для защиты оригинала:
        return new Pair(p.getX(), p.getY());
    }
    // Предполагается, что эта операция
    // занимает много времени.
    protected void store(Pair p) {
        storage.add(p);
        try {
            TimeUnit.MILLISECONDS.sleep(50);
        } catch (InterruptedException ignore) {}
    }
    public abstract void increment();
}

// Синхронизация целого метода:
class PairManager1 extends PairManager {
    public synchronized void increment() {
        p.incrementX();
        p.incrementY();
    }
}

```

```
        store(getPair());
    }
}

// Использование критической секции:
class PairManager2 extends PairManager {
    public void increment() {
        Pair temp;
        synchronized(this) {
            p.incrementX();
            p.incrementY();
            temp = getPair();
        }
        store(temp);
    }
}

class PairManipulator implements Runnable {
    private PairManager pm;
    public PairManipulator(PairManager pm) {
        this.pm = pm;
    }
    public void run() {
        while(true)
            pm.increment();
    }
    public String toString() {
        return "Pair: " + pm.getPair() +
            " checkCounter = " + pm.checkCounter.get();
    }
}

class PairChecker implements Runnable {
    private PairManager pm;
    public PairChecker(PairManager pm) {
        this.pm = pm;
    }
    public void run() {
        while(true) {
            pm.checkCounter.incrementAndGet();
            pm.getPair().checkState();
        }
    }
}

public class CriticalSection {
    // Тестирование двух разных подходов:
    static void
    testApproaches(PairManager pman1, PairManager pman2) {
        ExecutorService exec = Executors.newCachedThreadPool();
        PairManipulator
            pm1 = new PairManipulator(pman1),
            pm2 = new PairManipulator(pman2);
        PairChecker
            pcheck1 = new PairChecker(pman1),
            pcheck2 = new PairChecker(pman2);
        exec.execute(pm1);
        exec.execute(pm2);
        exec.execute(pcheck1);
    }
}
```

```

    exec.execute(pcheck2);
    try {
        TimeUnit.MILLISECONDS.sleep(500);
    } catch (InterruptedException e) {
        System.out.println("Sleep interrupted");
    }
    System.out.println("pm1: " + pm1 + "\n" + "pm2: " + pm2);
    System.exit(0);
}
public static void main(String[] args) {
    PairManager
        pman1 = new PairManager1(),
        pman2 = new PairManager2();
    testApproaches(pman1, pman2);
}
} /* Output: (Sample)
pm1: Pair: x: 15, y: 15 checkCounter = 272565
pm2: Pair: x: 16, y: 16 checkCounter = 3956974
*///:~

```

Как было отмечено, класс `Pair` небезопасен по отношению к потокам, поскольку его инвариант (предположительно произвольный) требует равенства обеих переменных. Вдобавок, как мы уже видели в этой главе, операции инкремента небезопасны в отношении к потокам, и так как ни один из методов не был объявлен как `synchronized`, полагать, что объект `Pair` останется неповрежденным в программе с потоками, не стоит.

А теперь представьте, что кто-то передает вам небезопасный класс `Pair` и вы должны использовать его в потоковой среде. Для этого создается класс `PairManager`, который хранит объекты `Pair` и управляет всем доступом к ним. Заметьте, что единственными открытыми (`public`) методами являются `getPair()`, объявленный как `synchronized`, и абстрактный метод `increment()`. Синхронизация этого метода будет осуществлена при его реализации.

Структура класса `PairManager`, в котором часть функциональности базового класса реализуется одним или несколькими абстрактными методами, определенными производными классами, называется на языке паттернов проектирования¹ «шаблонным методом». Паттерны проектирования позволяют инкапсулировать изменения в вашем коде — здесь изменяющаяся часть представлена методом `increment()`. В классе `PairManager1` метод `increment()` полностью синхронизирован, в то время как в классе `PairManager2` синхронизирована только его часть — синхронизируемый блок. Обратите внимание еще раз, что ключевые слова `synchronized` не являются частью сигнатуры метода и могут быть добавлены во время переопределения.

Метод `store()` добавляет объект `Pair` в синхронизированный контейнер `ArrayList`, так что операция безопасна по отношению к потокам. Другими словами, ее не нужно защищать и она размещается вне блока `synchronized` в `PairManager2`.

Класс `PairManipulator` создается для тестирования двух разных типов `PairManager`, для чего он вызывает `increment()` в задаче, тогда как `PairChecker` запускается из другой задачи. Чтобы пользователь получил информацию о том, с какой частотой он может выполнять проверку, `PairChecker` увеличивает `checkCounter` при каждом успешном

¹ См. книгу *Гамма и др.* Приемы объектно-ориентированного проектирования. СПб.: Питер, 2013.

выполнении. Метод `main()` создает два объекта `PairManipulator` и позволяет им поработать в течение какого-то времени, после чего выводятся результаты для каждого объекта `PairManipulator`.

Вероятно, результаты будут сильно отличаться при разных запусках, но обычно вы увидите, что `PairManager1.increment()` предоставляет `PairChecker` существенно меньше доступа, чем метод `PairManager2.increment()`, который содержит синхронизированный блок, а следовательно, предоставляет больше «незаблокированного» времени. Обычно в этом и заключается причина для использования синхронизированных блоков вместо синхронизации целых методов: это делается для того, чтобы повысить доступность кода для других задач (при условии, что это безопасно).

Также для создания критических секций можно использовать объекты `Lock`:

```
//: concurrency/ExplicitCriticalSection.java
// Использование объектов Lock для создания
// критических секций.
package concurrency;
import java.util.concurrent.locks.*;

// Синхронизация всего метода:
class ExplicitPairManager1 extends PairManager {
    private Lock lock = new ReentrantLock();
    public synchronized void increment() {
        lock.lock();
        try {
            p.incrementX();
            p.incrementY();
            store(getPair());
        } finally {
            lock.unlock();
        }
    }
}

// Использование критической секции:
class ExplicitPairManager2 extends PairManager {
    private Lock lock = new ReentrantLock();
    public void increment() {
        Pair temp;
        lock.lock();
        try {
            p.incrementX();
            p.incrementY();
            temp = getPair();
        } finally {
            lock.unlock();
        }
        store(temp);
    }
}

public class ExplicitCriticalSection {
    public static void main(String[] args) throws Exception {
        PairManager
            pman1 = new ExplicitPairManager1(),
```

продолжение ➤

```

        pman2 = new ExplicitPairManager2();
        CriticalSection.testApproaches(pman1, pman2);
    }
} /* Output: (Sample)
pm1: Pair: x: 15, y: 15 checkCounter = 174035
pm2: Pair: x: 16, y: 16 checkCounter = 2608588
*///:~

```

Эта программа, использующая большую часть кода `CriticalSection.java`, создает новые типы `PairManager`, использующие явную блокировку с объектами `Lock`. `ExplicitPairManager2` демонстрирует создание критической секции с использованием объекта `Lock`; вызов `store()` находится вне критической секции.

Синхронизация по другим объектам

Для синхронизированного блока должен быть указан объект, по которому осуществляется синхронизация. Обычно самым разумным выбором оказывается текущий объект, для которого вызывается метод, то есть `synchronized(this)`; этот подход используется в `PairManager2`. В этом случае при получении блокировки для синхронизированного блока другие синхронизированные методы и критические секции этого объекта вызываться не могут. Таким образом, критическая секция при синхронизации по `this` просто сокращает область действия синхронизации.

Иногда синхронизацию приходится выполнять по другому объекту, но в этом случае вы должны убедиться в том, что все задействованные задачи синхронизируют по одному объекту. Следующий пример демонстрирует, что при синхронизации методов объекта по разным блокировкам возможен вход в объект из двух задач.

```

//: concurrency/SyncObject.java
// Synchronizing on another object.
import static net.mindview.util.Print.*;

class DualSynch {
    private Object syncObject = new Object();
    public synchronized void f() {
        for(int i = 0; i < 5; i++) {
            print("f()");
            Thread.yield();
        }
    }
    public void g() {
        synchronized(syncObject) {
            for(int i = 0; i < 5; i++) {
                print("g()");
                Thread.yield();
            }
        }
    }
}

public class SyncObject {
    public static void main(String[] args) {
        final DualSynch ds = new DualSynch();
        new Thread() {

```

```

        public void run() {
            ds.f();
        }
    }.start();
    ds.g();
}
} /* Output: (Sample)
g()
f()
g()
f()
g()
f()
g()
f()
g()
f()
*///:~

```

`DualSync.f()` синхронизируется по `this` (с синхронизацией всего метода), а `g()` содержит синхронизированный блок, который синхронизируется по `syncObject`. Таким образом, эти две синхронизации независимы. Для демонстрации этого факта метод `main()` создает объект `Thread`, вызывающий `f()`. Поток `main()` используется для вызова `g()`. Из результатов видно, что оба метода выполняются одновременно и ни один из них не блокируется синхронизацией другого.

15. (1) Создайте класс с тремя методами, содержащими критические секции, синхронизируемые по одному объекту. Создайте несколько задач, демонстрирующих, что в любой момент времени может выполняться только один из этих методов. Теперь измените методы, чтобы каждый из них синхронизировался по своему объекту, и покажите, что все методы могут выполняться одновременно.
16. (1) Измените упражнение 15 так, чтобы в нем использовались явные блокировки с объектами `Lock`.

Локальная память потоков

Второй способ предотвращения конкуренции задач за совместно используемые ресурсы заключается в устранении самого совместного использования. Локальная память потоков — механизм, автоматически создающий для одной переменной несколько блоков памяти — по одному для каждого потока, использующего объект. Таким образом, если у вас имеются пять потоков, использующих объект с переменной `x`, механизм локальной памяти потоков генерирует пять разных блоков памяти для `x`. По сути это позволяет связать с каждым потоком некоторое состояние.

Созданием и управлением локальной памяти потоков занимается класс `java.lang.ThreadLocal`, как видно из следующего примера:

```

//: concurrency/ThreadLocalVariableHolder.java
// Автоматическое выделение собственной памяти
// каждому потоку.
import java.util.concurrent.*;
import java.util.*;

```

продолжение 

```

class Accessor implements Runnable {
    private final int id;
    public Accessor(int idn) { id = idn; }
    public void run() {
        while(!Thread.currentThread().isInterrupted()) {
            ThreadLocalVariableHolder.increment();
            System.out.println(this);
            Thread.yield();
        }
    }
    public String toString() {
        return "#" + id + ": " +
            ThreadLocalVariableHolder.get();
    }
}

public class ThreadLocalVariableHolder {
    private static ThreadLocal<Integer> value =
        new ThreadLocal<Integer>() {
            private Random rand = new Random(47);
            protected synchronized Integer initialValue() {
                return rand.nextInt(10000);
            }
        };
    public static void increment() {
        value.set(value.get() + 1);
    }
    public static int get() { return value.get(); }
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new Accessor(i));
        TimeUnit.SECONDS.sleep(3); // Небольшая задержка
        exec.shutdownNow();       // Все экземпляры Accessor завершаются
    }
} /* Output: (Sample)
#0: 9259
#1: 556
#2: 6694
#3: 1862
#4: 962
#0: 9260
#1: 557
#2: 6695
#3: 1863
#4: 963
...
*///:~

```

Объекты `ThreadLocal` обычно хранятся в статических полях. При создании объекта `ThreadLocal` вы можете только обратиться к содержимому объекта методами `get()` и `set()`. Метод `get()` возвращает копию объекта, связанную с потоком, а метод `set()` вставляет свой аргумент в объект, хранящийся для потока, и возвращает старый объект, находящийся в хранилище.

Методы `increment()` и `get()` демонстрируют эту возможность в `ThreadLocalVariableHolder`. Обратите внимание: методы `increment()` и `get()` не синхронизированы, потому что `ThreadLocal` гарантирует отсутствие ситуации гонки.

При запуске программы вы увидите доказательства того, что для каждого из потоков выделяется отдельная память, так как каждый поддерживает собственную копию счетчика — при том, что объект `ThreadLocalVariableHolder` только один.

Завершение задач

В некоторых из приведенных примеров методы `cancel()` и `isCanceled()` размещаются в классе, видимом для всех задач. Задачи используют проверку `isCanceled()`, чтобы определить, когда следует завершиться; это разумный подход к решению проблемы. Однако в некоторых ситуациях задачу требуется завершить быстрее. В этом разделе вы узнаете о некоторых аспектах и проблемах такого завершения.

Начнем с рассмотрения примера, который не только демонстрирует проблему завершения, но и дает дополнительный пример совместного доступа к ресурсам.

В этой модели дирекция парка хочет узнать, сколько людей ежедневно заходит в парк через несколько ворот. У каждого ворот имеется вертушка; после увеличения счетчика вертушки увеличивается общий счетчик, представляющий общее количество посетителей парка.

```
//: concurrency/OrnamentalGarden.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Count {
    private int count = 0;
    private Random rand = new Random(47);
    // Удаляем ключевое слово synchronized,
    // чтобы увидеть сбой в системе подсчета:
    public synchronized int increment() {
        int temp = count;
        if(rand.nextBoolean()) // Уступает управление
            Thread.yield();    // в половине случаев
        return (count = ++temp);
    }
    public synchronized int value() { return count; }
}

class Entrance implements Runnable {
    private static Count count = new Count();
    private static List<Entrance> entrances =
        new ArrayList<Entrance>();
    private int number = 0;
    // Для чтения синхронизация не нужна:
    private final int id;
    private static volatile boolean canceled = false;
    // Атомарная операция с volatile-полем:
    public static void cancel() { canceled = true; }
    public Entrance(int id) {
        this.id = id;
        // Задача остается в списке. Также предотвращает
        // уничтожение "мертвых" задач при уборке мусора:
    }
}
```

продолжение ➤

```

    entrances.add(this);
}
public void run() {
    while(!canceled) {
        synchronized(this) {
            ++number;
        }
        print(this + " Total: " + count.increment());
        try {
            TimeUnit.MILLISECONDS.sleep(100);
        } catch(InterruptedException e) {
            print("sleep interrupted");
        }
    }
    print("Stopping " + this);
}
public synchronized int getValue() { return number; }
public String toString() {
    return "Entrance " + id + ": " + getValue();
}
public static int getTotalCount() {
    return count.value();
}

public static int sumEntrances() {
    int sum = 0;
    for(Entrance entrance : entrances)
        sum += entrance.getValue();
    return sum;
}
}

public class OrnamentalGarden {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new Entrance(i));
        // Проработать некоторое время, затем
        // остановиться и собрать данные:
        TimeUnit.SECONDS.sleep(3);
        Entrance.cancel();
        exec.shutdown();
        if(!exec.awaitTermination(250, TimeUnit.MILLISECONDS))
            print("Some tasks were not terminated!");
        print("Total: " + Entrance.getTotalCount());
        print("Sum of Entrances: " + Entrance.sumEntrances());
    }
}

/* Output: (Sample)
Entrance 0: 1 Total: 1
Entrance 2: 1 Total: 3
Entrance 1: 1 Total: 2
Entrance 4: 1 Total: 5
Entrance 3: 1 Total: 4
Entrance 2: 2 Total: 6
Entrance 4: 2 Total: 7
Entrance 0: 2 Total: 8
...
Entrance 3: 29 Total: 143

```

```
Entrance 0: 29 Total: 144
Entrance 4: 29 Total: 145
Entrance 2: 30 Total: 147
Entrance 1: 30 Total: 146
Entrance 0: 30 Total: 149
Entrance 3: 30 Total: 148
Entrance 4: 30 Total: 150
Stopping Entrance 2: 30
Stopping Entrance 1: 30
Stopping Entrance 0: 30
Stopping Entrance 3: 30
Stopping Entrance 4: 30
Total: 150
Sum of Entrances: 150
*///:~
```

Один объект `Count` хранит главный счетчик посетителей парка и хранится в статическом поле класса `Entrance`. Методы `Count.increment()` и `Count.value()` синхронизируются для управления доступом к полю `count`. Метод `increment()` использует объект `Random` для того, чтобы уступать управление вызовом `yield()` примерно в половине случаев, между выборкой `count` в `temp` и увеличением/сохранением `temp` обратно в `count`. Если закомментировать ключевое слово `synchronized` для `increment()`, программа перестает работать, потому что сразу несколько задач будут пытаться обращаться и изменять `count` одновременно (с вызовом `yield()` проблема проявляется быстрее).

Каждая задача `Entrance` поддерживает локальную переменную `number` с количеством посетителей, прошедших через этот конкретный вход. Таким образом обеспечивается дополнительная проверка того, что в объекте `count` было сохранено правильное количество посетителей. `Entrance.run()` просто увеличивает `number` и объект `count`, после чего приостанавливается на 100 миллисекунд.

Так как `Entrance.canceled` — логический флаг, объявленный с ключевым словом `volatile`, с которым выполняются только операции чтения и присваивания (и он никогда не читается в сочетании с другими полями), можно обойтись и без синхронизации доступа к нему. Впрочем, если у вас возникнут сомнения в подобной ситуации, всегда лучше использовать ключевое слово `synchronized`.

Программа прикладывает дополнительные усилия к тому, чтобы все завершилось корректно. Отчасти это сделано для того, чтобы показать, как осторожно следует действовать при завершении многопоточной программы, а отчасти для того, чтобы продемонстрировать полезность `interrupt()` (см. далее).

Через 3 секунды `main()` отправляет `Entrance` статическое сообщение `cancel()`, затем вызывает `shutdown()` для объекта `exes` и вызывает `awaitTermination()` для `exes`. Метод `ExecutorService.awaitTermination()` ожидает завершения каждой задачи, и если все задачи завершатся до истечения тайм-аута — возвращает `true`; в противном случае возвращается значение `false`, показывающее, что не все задачи завершены. И хотя это заставляет каждую задачу выйти из метода `run()`, а следовательно, завершиться, объекты `Entrance` остаются действительными, потому что в конструкторе каждый объект `Entrance` сохраняется в статическом контейнере `List<Entrance>` с именем `entrances`. Таким образом, `sumEntrances()` по-прежнему работает с действительными объектами `Entrance`.

Во время работы программы при прохождении посетителей выводится общий счетчик и счетчики всех входов. Если удалить `synchronized` из объявления `Count.increment()`, вы увидите, что общее количество посетителей отличается от ожидаемого. При синхронизации доступа к `Count` с использованием мьютекса все будет работать правильно. Не забудьте, что `Count.increment()` повышает вероятность ошибок с использованием `temp` и `yield()`. В реальных ситуациях с параллельным выполнением вероятность ошибки статистически может быть очень низкой, и у разработчика может появиться ложное ощущение того, что программа работает правильно. Как и в приведенном выше примере, программа может содержать скрытые проблемы, которые еще никак не проявились, поэтому при рецензировании параллельного кода необходимо действовать особенно внимательно.

17. (2) Создайте модель радиационного счетчика, получающего данные от произвольного количества удаленных датчиков.

Завершение при блокировке

Метод `Entrance.run()` из предыдущего примера вызывает `sleep()` в своем цикле. Мы знаем, что `sleep()` позднее активизируется, а задача перейдет к началу цикла, где ей будет предоставлена возможность прервать цикл проверкой флага `canceled`. Тем не менее `sleep()` — всего лишь одна из ситуаций, в которых выполнение задачи блокируется, а иногда заблокированную задачу приходится прерывать.

Состояния потока

Поток может находиться в одном из четырех возможных состояний.

1. **Переходное (new):** поток находится в этом состоянии очень недолго, только во время создания. Он получает все необходимые системные ресурсы и выполняет инициализацию. На этой стадии он получает право на выделение процессорного времени. Далее поток переводится планировщиком в активное или заблокированное состояние.
2. **Активное (runnable):** в таком состоянии поток будет выполняться тогда, когда у механизма разделения времени процессора появятся для него свободные кванты. Таким образом, поток может как не выполняться, так и выполняться, однако ничто не препятствует последнему при получении потоком процессорного времени; он не «мертв» и не заблокирован.
3. **Блокировки (blocked):** поток может выполняться, однако есть что-то, что мешает ему это сделать. Пока поток имеет данный статус, квантирующий время процессор пропускает его очередь и не выделяет ему циклов на выполнение. До тех пор пока поток не перейдет в рабочее состояние, он не в состоянии произвести ни одной операции.
4. **Завершенное (dead):** в этом состоянии поток не ставится на выполнение и не получает процессорного времени. Его задача завершена, и он не может стать активным. Одним из способов перехода в завершенное состояние является возврат из метода `run()`, но поток задачи также может быть прерван явно, как мы вскоре увидим.

Переход в заблокированное состояние

Поток может перейти в заблокированное состояние по следующим причинам.

- ❑ Вы приказали потоку приостановиться с помощью метода `sleep()`, после этого он будет бездействовать заданное время.
- ❑ Вы приостановили выполнение потока вызовом метода `wait()`. Поток будет простаивать до тех пор, пока не получит сообщение о возобновлении работы: `notify()` или `notifyAll()` (или эквивалентный вызов `signal()` или `signalAll()` для библиотеки Java SE5 `java.util.concurrent`). Мы изучим эти методы в следующем разделе.
- ❑ Поток ожидает завершения некоторого ввода-вывода.
- ❑ Поток пытается вызвать синхронизированный (`synchronized`) метод другого объекта, но его объект блокировки недоступен.

В старом коде можно видеть, как для блокировки и возобновления потоков используются методы `suspend()` и `resume()`, однако они были объявлены устаревшими в современной версии Java (так как они часто приводят к возникновению взаимных блокировок), потому в этой книге они не рассматриваются. Метод `stop()` также считается устаревшим, потому что он не снимает блокировки, полученные потоком, и если логическая целостность объекта будет нарушена, другие задачи могут просматривать его и изменять в этом состоянии. При этом возникают крайне коварные и трудноуловимые ошибки.

Проблема, которую нам нужно решить, заключается в следующем: иногда требуется завершить задачу, находящуюся в заблокированном состоянии. Если вариант с ожиданием достижения точки кода, в которой задача может проверить значение состояния и самостоятельно принять решение о своем завершении, не подходит, необходимо как-то принудительно вывести задачу из заблокированного состояния.

Прерывание

Как нетрудно предположить, выход из метода `Runnable.run()` на середине — дело более хлопотное, чем ожидание до проверки флага отмены (или другой точки, в которой программист готов к выходу из метода). При выходе из заблокированной задачи может возникнуть необходимость в освобождении ресурсов. По этой причине выход из середины метода `run()` задачи больше напоминает исключение, чем что-либо еще, поэтому в потоках Java для такой отмены применяются исключения¹. (Здесь ситуация балансирует на грани ненадлежащего применения исключений, потому что они будут часто использоваться для управления выполнением программы.) Чтобы вернуться в заведомо допустимое состояние при завершении задачи, необходимо тщательно проанализировать пути выполнения кода и написать условие `catch` для освобождения всех ресурсов.

¹ При этом исключения никогда не доставляются асинхронно. Таким образом, не существует опасности прерывания на середине команды/вызова метода. И если вы используете идиому `try-finally` с объектами-мьютексами (вместо ключевого слова `synchronized`), эти мьютексы будут автоматически освобождены при выдаче исключения.

Итак, для завершения заблокированных задач в класс `Thread` был включен метод `interrupt()`, устанавливающий состояние прерывания потока. Поток с установленным состоянием прерывания выдает исключение `InterruptedException`, если он уже заблокирован или пытается выполнить блокирующую операцию. Состояние прерывания сбрасывается при возбуждении исключения или при вызове задач метода `Thread.interrupted()`. Как вы увидите, `Thread.interrupted()` предоставляет второй способ выхода из цикла `run()` без возбуждения исключения.

Чтобы вызвать `interrupt()`, необходимо иметь ссылку на объект `Thread`. Возможно, вы заметили, что новая библиотека `concurrent` старается избегать прямых манипуляций с объектами `Thread` и вместо этого пытается все делать через `Executor`. Если вызвать для объектов `Executor` метод `shutdownNow()`, он отправит вызов `interrupt()` каждому из запущенных им потоков. И это разумно, потому что обычно требуется закрыть все задачи конкретного объекта `Executor` при завершении части проекта или всей программы. Впрочем, встречаются и ситуации, в которых требуется прервать только одну задачу. Если вы используете объект `Executor`, можно получить ссылку на контекст задачи при ее запуске — для этого вместо `execute()` вызывается метод `submit()`. Метод `submit()` возвращает обобщение `Future<?>` с незадаанным параметром, потому что `get()` для него вызываться никогда не будет — такие ссылки на `Future` нужны для того, чтобы вызвать для них `cancel()` и таким образом прервать конкретную задачу. Если передать `true` при вызове `cancel()`, тем самым вы разрешаете вызов `interrupt()` для этого потока с целью его остановки; таким образом, для прерывания конкретных потоков, запущенных `Executor`, следует применять `cancel()`.

Следующий пример демонстрирует основы применения `interrupt()` с объектами `Executor`:

```
//: concurrency/Interrupting.java
// Прерывание заблокированного потока.
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class SleepBlocked implements Runnable {
    public void run() {
        try {
            TimeUnit.SECONDS.sleep(100);
        } catch (InterruptedException e) {
            print("InterruptedException");
        }
        print("Exiting SleepBlocked.run()");
    }
}

class IOBlocked implements Runnable {
    private InputStream in;
    public IOBlocked(InputStream is) { in = is; }
    public void run() {
        try {
            print("Waiting for read():");
            in.read();
        } catch (IOException e) {
            if (Thread.currentThread().isInterrupted()) {
```

```

        print("Interrupted from blocked I/O");
    } else {
        throw new RuntimeException(e);
    }
}
print("Exiting IOBlocked.run()");
}
}

class SynchronizedBlocked implements Runnable {
    public synchronized void f() {
        while(true) // Блокировка никогда не снимается
            Thread.yield();
    }
    public SynchronizedBlocked() {
        new Thread() {
            public void run() {
                f(); // Блокировка устанавливается этим потоком
            }
        }.start();
    }
    public void run() {
        print("Trying to call f()");
        f();
        print("Exiting SynchronizedBlocked.run()");
    }
}

public class Interrupting {
    private static ExecutorService exec =
        Executors.newCachedThreadPool();
    static void test(Runnable r) throws InterruptedException{
        Future<?> f = exec.submit(r);
        TimeUnit.MILLISECONDS.sleep(100);
        print("Interrupting " + r.getClass().getName());
        f.cancel(true); // Прервать, если выполняется
        print("Interrupt sent to " + r.getClass().getName());
    }
    public static void main(String[] args) throws Exception {
        test(new SleepBlocked());
        test(new IOBlocked(System.in));
        test(new SynchronizedBlocked());
        TimeUnit.SECONDS.sleep(3);
        print("Aborting with System.exit(0)");
        System.exit(0); // ... так как два последних прерывания
        // завершились неудачей
    }
} /* Output: (95% match)
Interrupting SleepBlocked
InterruptedException
Exiting SleepBlocked.run()
Interrupt sent to SleepBlocked
Waiting for read():
Interrupting IOBlocked
Interrupt sent to IOBlocked
Trying to call f()
Interrupting SynchronizedBlocked
Interrupt sent to SynchronizedBlocked
Aborting with System.exit(0)
*///:~

```

Каждая задача представляет свою разновидность блокировки. `SleepBlock` — пример прерываемой блокировки, тогда как `IOBlocked` и `SynchronizedBlocked` представляют блокировку непрерываемую¹. Программа доказывает, что ввод-вывод и ожидание по синхронизируемой блокировке не прерываются, но об этом можно догадаться и по исходному коду — для ввода-вывода или попытки вызова синхронизированного метода обработчик `InterruptedException` не требуется.

Первые два класса достаточно тривиальны: метод `run()` вызывает `sleep()` в первом классе и метод `read()` во втором. Однако для демонстрации `SynchronizedBlocked` необходимо сначала получить блокировку. Для этого конструктор создает экземпляр анонимного класса `Thread`, который получает блокировку вызовом `f()` (поток должен отличаться от того, который управляет выполнением `run()` для `SynchronizedBlock`, потому что один поток может многократно устанавливать блокировку по объекту). Так как `f()` никогда не возвращает управление, блокировка никогда не снимается. `SynchronizedBlock.run()` пытается вызвать `f()` и блокируется в ожидании освобождения блокировки.

Из выходных данных видно, что вызов `sleep()` (или любой другой вызов, требующий перехвата `InterruptedException`) можно прервать. Однако вам не удастся прервать задачу, которая пытается установить синхронизированную блокировку или выполнить ввод-вывод. Это обстоятельство немного обескураживает, особенно если вы создаете задачу для выполнения ввода-вывода, поскольку получается, что ввод-вывод может «подвесить» вашу многопоточную программу. Это может стать проблемой, особенно для веб-программ.

Примитивное, но иногда эффективное решение задачи — закрытие ресурса, по которому блокировалась задача:

```
//: concurrency/CloseResource.java
// Прерывание заблокированной задачи посредством
// закрытия ресурса, по которому она блокируется:
// {RunByHand}
import java.net.*;

import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

public class CloseResource {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        ServerSocket server = new ServerSocket(8080);
        InputStream socketInput =
            new Socket("localhost", 8080).getInputStream();
        exec.execute(new IOBlocked(socketInput));
        exec.execute(new IOBlocked(System.in));
        TimeUnit.MILLISECONDS.sleep(100);
        print("Shutting down all threads");
        exec.shutdownNow();
    }
}
```

¹ В некоторых версиях JDK также предоставляется поддержка `InterruptedIOException`. Тем не менее она была реализована лишь частично и только на некоторых платформах. При возбуждении этого исключения объекты ввода-вывода становятся непригодными к использованию. Вероятно, в будущих версиях поддержка этого исключения будет устранена.


```

    TimeUnit.SECONDS.sleep(1);
    print("Closing " + socketInput.getClass().getName());
    socketInput.close(); // Освобождение заблокированного потока
    TimeUnit.SECONDS.sleep(1);
    print("Closing " + System.in.getClass().getName());
    System.in.close(); // Освобождение заблокированного потока
}
} /* Output: (85% match)
Waiting for read():
Waiting for read():
Shutting down all threads
Closing java.net.SocketInputStream
Interrupted from blocked I/O
Exiting IOBlocked.run()
Closing java.io.BufferedInputStream
Exiting IOBlocked.run()
*///:~

```

После вызова `shutdownNow()` задержки перед вызовом `close()` для двух входных потоков более наглядно показывают, что задача разблокируется при закрытии используемого ресурса. Интересно, что `interrupt()` появляется при закрытии `Socket`, но не при закрытии `System.in`.

К счастью, классы `nio`, представленные в главе 18, предоставляют более цивилизованный механизм прерывания ввода-вывода. Блокированные `nio`-каналы автоматически реагируют на прерывания:

```

//: concurrency/NIOInterruption.java
// Прерывание заблокированного канала NIO.
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class NIOBlocked implements Runnable {
    private final SocketChannel sc;
    public NIOBlocked(SocketChannel sc) { this.sc = sc; }
    public void run() {
        try {
            print("Waiting for read() in " + this);
            sc.read(ByteBuffer.allocate(1));
        } catch (ClosedByInterruptException e) {
            print("ClosedByInterruptException");
        } catch (AsynchronousCloseException e) {
            print("AsynchronousCloseException");
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        print("Exiting NIOBlocked.run() " + this);
    }
}

public class NIOInterruption {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
    }
}

```

продолжение ➤

```

ServerSocket server = new ServerSocket(8080);
InetSocketAddress isa =
    new InetSocketAddress("localhost", 8080);
SocketChannel sc1 = SocketChannel.open(isa);
SocketChannel sc2 = SocketChannel.open(isa);
Future<?> f = exec.submit(new NIOBlocked(sc1));
exec.execute(new NIOBlocked(sc2));
exec.shutdown();
TimeUnit.SECONDS.sleep(1);
// Прерывание через вызов cancel:
f.cancel(true);
TimeUnit.SECONDS.sleep(1);
// Разблокирование посредством закрытия канала:
sc2.close();
}
} /* Output: (Sample)
Waiting for read() in NIOBlocked@7a84e4
Waiting for read() in NIOBlocked@15c7850
ClosedByInterruptException
Exiting NIOBlocked.run() NIOBlocked@15c7850
AsynchronousCloseException
Exiting NIOBlocked.run() NIOBlocked@7a84e4
*///:~

```

Как видно из приведенного примера, для разблокирования также можно закрыть канал, хотя такая необходимость возникает очень редко. Учтите, что использование `execute()` для запуска обеих задач с вызовом `e.shutdownNow()` позволяет легко завершить все; сохранение `Future` в приведенном примере было необходимо лишь для того, чтобы прерывание было передано только одному из двух потоков¹.

18. (2) Создайте класс, не являющийся задачей, с методом, который вызывает `sleep()` на долгий промежуток времени. Создайте задачу, которая вызывает метод первого класса. В `main()` запустите задачу и прервите ее методом `interrupt()`. Убедитесь в том, что задача завершилась корректно.
19. (4) Измените пример `OrnamentalGarden.java` так, чтобы в нем использовался метод `interrupt()`.
20. (1) Измените пример `CachedThreadPool.java` так, чтобы все задачи получали `interrupt()` перед завершением.

Блокирование по мьютексу

Как было показано в примере `Interrupting.java`, при попытке вызова синхронизированного метода для объекта, для которого уже была установлена блокировка, вызывающая задача приостанавливается до получения блокировки. Следующий пример показывает, как один мьютекс может многократно захватываться одной задачей:

```

//: concurrency/MultiLock.java
// Поток может многократно захватывать
// одну блокировку.
import static net.mindview.util.Print.*;

```

¹ Эрик Варга помог мне с подготовкой материалов этого раздела.

```

public class MultiLock {
    public synchronized void f1(int count) {
        if(count-- > 0) {
            print("f1() calling f2() with count " + count);
            f2(count);
        }
    }
    public synchronized void f2(int count) {
        if(count-- > 0) {
            print("f2() calling f1() with count " + count);
            f1(count);
        }
    }
    public static void main(String[] args) throws Exception {
        final MultiLock multiLock = new MultiLock();
        new Thread() {
            public void run() {
                multiLock.f1(10);
            }
        }.start();
    }
} /* Output:
f1() calling f2() with count 9
f2() calling f1() with count 8
f1() calling f2() with count 7
f2() calling f1() with count 6
f1() calling f2() with count 5
f2() calling f1() with count 4
f1() calling f2() with count 3
f2() calling f1() with count 2
f1() calling f2() with count 1
f2() calling f1() with count 0
*///:~

```

В методе `main()` создается объект `Thread` для вызова `f1()`, после чего `f1()` и `f2()` вызывают друг друга до тех пор, пока счетчик не станет равен нулю. Так как задача уже установила блокировку объекта `multiLock` внутри первого вызова `f1()`, эта же задача повторно устанавливает ее в вызове `f2()`, и т. д. Все это вполне разумно, потому что одна задача должна быть способна вызывать другие синхронизированные методы в том же объекте; эта задача уже удерживает блокировку.

Как упоминалось ранее при обсуждении непрерываемого ввода-вывода, любая ситуация, в которой задача может быть заблокирована без возможности прерывания, может парализовать работу программы. Одним из новшеств библиотеки `Java SE5 concurrency` стала возможность прерывания задач, заблокированных по `ReentrantLock`, — в отличие от задач, заблокированных по синхронизированным методам или критическим секциям:

```

//: concurrency/Interrupting2.java
// Прерывание задачи, заблокированной по ReentrantLock.
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import static net.mindview.util.Print.*;

class BlockedMutex {
    private Lock lock = new ReentrantLock();
    public BlockedMutex() {

```

продолжение ➤

```

    // Немедленное получение блокировки для демонстрации
    // прерывания задач, заблокированных по ReentrantLock:
    lock.lock();
}
public void f() {
    try {
        // Никогда не будет доступен для второй задачи
        lock.lockInterruptibly(); // Специальный вызов
        print("lock acquired in f()");
    } catch (InterruptedException e) {
        print("Interrupted from lock acquisition in f()");
    }
}
}

class Blocked2 implements Runnable {
    BlockedMutex blocked = new BlockedMutex();
    public void run() {
        print("Waiting for f() in BlockedMutex");
        blocked.f();
        print("Broken out of blocked call");
    }
}

public class Interrupting2 {
    public static void main(String[] args) throws Exception {
        Thread t = new Thread(new Blocked2());
        t.start();
        TimeUnit.SECONDS.sleep(1);
        System.out.println("Issuing t.interrupt()");
        t.interrupt();
    }
}
/* Output:
Waiting for f() in BlockedMutex
Issuing t.interrupt()
Interrupted from lock acquisition in f()
Broken out of blocked call
*///:~

```

Класс `BlockedMutex` имеет конструктор, который получает собственную блокировку `lock` текущего объекта и не освобождает ее. По этой причине при попытке вызова `f()` из второй задачи (не той, которая создала `BlockedMutex`) выполнение всегда будет блокироваться из-за невозможности получения `Mutex`. В классе `Blocked2` метод `run()` будет останавливаться при вызове `blocked.f()`. При запуске программы вы увидите, что в отличие от вызова ввода-вывода, `interrupt()` может прервать вызов, заблокированный по мьютексу¹.

Проверка прерывания

Учтите, что при вызове `interrupt()` для потока прерывание может случиться только при входе задачи в блокирующую операцию (или если задача уже находится в ней) — кроме непрерываемого ввода-вывода или заблокированных синхронизированных

¹ Учтите, что вызов `t.interrupt()` может случиться до вызова `blocked.f()` (хотя это и маловероятно).

методов, когда вы ничего не можете сделать. Но что, если вы написали код, который может выдавать или не выдавать такой блокирующий вызов в зависимости от условий выполнения? Если выход может осуществляться только с возбуждением исключения к блокирующему вызову, вы не всегда сможете покинуть цикл `run()`. Таким образом, если вы вызываете `interrupt()` для остановки задачи, вашей задаче необходим второй способ выхода на случай, если цикл `run()` не совершает никаких блокирующих вызовов.

Такая возможность обеспечивается *состоянием прерывания*, которое устанавливается вызовом `interrupt()`. Состояние прерывания проверяется вызовом `interrupted()`. Оно не только сообщает, был ли вызван метод `interrupt()`, но и сбрасывает состояние прерывания. Сброс состояния прерывания предотвращает повторные оповещения о прерывании задачи: оповещение поступает либо через одиночное исключение `InterruptedException`, либо через одну успешную проверку `Thread.interrupted()`. Если вы захотите снова проверить, возникло ли прерывание, сохраните результат вызова `Thread.interrupted()`.

Следующий пример демонстрирует типичную идиому, которую следует использовать в методе `run()` для обработки как блокированных, так и неблокированных возможностей при установленном состоянии прерывания:

```
//: concurrency/InterruptingIdiom.java
// Общая идиома прерывания задачи.
// {Args: 1100}
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class NeedsCleanup {
    private final int id;
    public NeedsCleanup(int ident) {
        id = ident;
        print("NeedsCleanup " + id);
    }
    public void cleanup() {
        print("Cleaning up " + id);
    }
}

class Blocked3 implements Runnable {
    private volatile double d = 0.0;
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // точка1
                NeedsCleanup n1 = new NeedsCleanup(1);
                // try-finally начинается сразу же за определением
                // n1, чтобы гарантировать освобождение n1:
                try {
                    print("Sleeping");
                    TimeUnit.SECONDS.sleep(1);
                }
                // точка2
                NeedsCleanup n2 = new NeedsCleanup(2);
                // Гарантирует правильное освобождение n2:
                try {
                    print("Calculating");
                }
            }
        }
    }
}
```

продолжение ➤

```

        // Продолжительная неблокирующая операция:
        for(int i = 1; i < 2500000; i++)
            d = d + (Math.PI + Math.E) / d;
        print("Finished time-consuming operation");
    } finally {
        n2.cleanup();
    }
} finally {
    n1.cleanup();
}
}
print("Exiting via while() test");
} catch(InterruptedException e) {
    print("Exiting via InterruptedException");
}
}
}

public class InterruptingIdiom {
    public static void main(String[] args) throws Exception {
        if(args.length != 1) {
            print("usage: java InterruptingIdiom delay-in-mS");
            System.exit(1);
        }
        Thread t = new Thread(new Blocked3());
        t.start();
        TimeUnit.MILLISECONDS.sleep(new Integer(args[0]));
        t.interrupt();
    }
} /* Output: (Sample)
NeedsCleanup 1
Sleeping
NeedsCleanup 2
Calculating
Finished time-consuming operation
Cleaning up 2
Cleaning up 1
NeedsCleanup 1
Sleeping
Cleaning up 1
Exiting via InterruptedException
*///:~

```

Класс `NeedsCleanup` демонстрирует необходимость правильного освобождения ресурсов при выходе из цикла через исключение. Обратите внимание: за всеми ресурсами `NeedsCleanup`, созданными в `Blocked3.run()`, должны немедленно следовать блоки `try-finally`, которые гарантируют, что метод `cleanup()` будет вызван в любом случае.

Программе необходимо передать аргумент командной строки — время задержки перед вызовом `interrupt()` (в миллисекундах). Используя разные значения задержки, можно выходить из `Blocked3.run()` в разных точках цикла: в блокирующем вызове `sleep()` и в неблокирующих математических вычислениях. Вы увидите, что при вызове `interrupt()` после комментария «точка2» (во время неблокирующей операции) сначала завершается цикл, затем уничтожаются все локальные объекты, и наконец, происходит выход из цикла через начальную команду `while`. Но если `interrupt()` вызывается между комментариями «точка1» и «точка2» (после команды `while`, но перед

блокирующей операцией `sleep()` или во время нее), то выход из задачи происходит через `InterruptedException` при первой попытке выполнения блокирующей операции. В этом случае освобождаются только объекты `NeedsCleanup`, которые были созданы до точки выдачи исключения, и вы можете выполнить в блоке `catch` любые другие завершающие действия.

Класс, который должен реагировать на `interrupt()`, должен установить политику, гарантирующую сохранение корректного состояния. Обычно это означает, что за созданием всех объектов, требующих завершающих действий, должны следовать конструкции `try-finally`, чтобы завершение происходило независимо от способа выхода из цикла `run()`. Такой код может работать, но, к сожалению, из-за отсутствия автоматических вызовов деструкторов в Java он зависит от того, насколько аккуратно программист на стороне клиента пишет необходимые блоки `try-finally`.

Взаимодействие между задачами

Как вы уже видели, при использовании потоков для одновременного выполнения нескольких задач можно предотвратить некорректное использование ресурсов других задач; для этого поведение двух задач синхронизируется при помощи объекта блокировки (мьютекса). Таким образом, если две задачи конкурируют друг с другом за общий ресурс (обычно память), вы используете мьютекс, чтобы задачи обращались к этому ресурсу по очереди.

После решения этой проблемы следующим шагом становится организация взаимодействия между задачами, чтобы разные задачи могли совместно работать над выполнением одной операции. Теперь думать приходится уже не о разделении, а о согласовании взаимодействий, так как некоторые части задачи могут быть решены раньше других. Происходящее напоминает планирование проекта: строительство дома начинается с закладки фундамента, но прокладка арматуры может производиться одновременно с возведением железобетонного каркаса, причем обе задачи должны быть завершены до заливки фундамента. Санитарно-техническое оборудование должно быть готово к моменту заливки бетонной плиты, бетонная плита должна быть на месте до начала монтажа несущих профилей и т. д. Некоторые из этих задач могут выполняться параллельно, но некоторые шаги требуют, чтобы для продолжения были завершены все задачи.

Ключевым аспектом взаимодействия задач является *согласование* (handshaking) — обмен сигналами для установления связи между задачами. Для этого мы используем ту же основу: мьютекс, который в данном случае гарантирует, что только одна задача может ответить на сигнал. Таким образом устраняются любые возможные ситуации гонки. Кроме мьютекса, добавляется механизм приостановки задачи до изменения некоторого внешнего состояния, которое сообщает, что задача может двигаться вперед. В этом разделе рассматриваются аспекты согласования между задачами, которое безопасно реализуется с использованием методов класса `Object` `wait()` и `notifyAll()`. Библиотека `Java SE5 concurrency` также предоставляет объекты `Condition` с методами `await()` и `signal()`. Мы рассмотрим возникающие проблемы и их возможные решения.

wait() и notifyAll()

Метод `wait()` позволяет дождаться изменения некоторого условия, неподконтрольного текущему методу. Часто это условие изменяется другой задачей. Программа не должна «крутиться» в цикле, проверяя условие внутри задачи; такое поведение называется активным ожиданием и обычно приводит к неэффективному расходованию ресурсов процессора. По этой причине `wait()` приостанавливает задачу, ожидая изменения некоторого условия, и только при вызове `notify()` или `notifyAll()` (который сообщает, что произошло нечто интересное) задача активизируется и проверяет изменения. Таким образом, `wait()` предоставляет механизм синхронизации действий между задачами.

Важно понять, что метод `sleep()` *не освобождает* объект блокировки — как и метод `yield()`. С другой стороны, когда задача входит в `wait()` внутри метода, выполнение этого потока приостанавливается, а блокировка снимается. Это означает, что блокировка может быть получена другой задачей, а остальные потоки могут вызывать другие синхронизированные методы объекта во время выполнения `wait()`. Это очень важно, потому что изменения, ради которых может активизироваться приостановленная задача, обычно порождаются другими методами. Таким образом, при вызове `wait()` вы фактически говорите: «Я сделал все, что мог на данный момент, и сейчас перехожу к ожиданию, но при этом я хочу разрешить выполнение других синхронизированных операций».

Существует две формы метода `wait()`. Для первой формы значение аргумента совпадает со значением аргумента метода `sleep()`: это продолжительность приостановки в миллисекундах. Разница между методами состоит в следующем.

1. При выполнении метода `wait()` блокируемый объект освобождается.
2. Выйти из состояния ожидания, установленного `wait()`, можно двумя способами: с помощью уведомления `notify()` или `notifyAll()` либо по истечении срока ожидания.

Вторая форма — без аргументов — применяется более широко. Эта версия метода `wait()` заставит поток простаивать, пока не придет уведомление `notify()` или `notifyAll()`.

Пожалуй, самое интересное в методах `wait()`, `notify()` и `notifyAll()` — их принадлежность к базовому классу `Object`, а не к классу потоков `Thread`. Хотя это и кажется немного нелогичным — нечто, относящееся исключительно к механизму потоков, помещается в общий базовый класс, — на самом деле это решение совершенно оправданно, поскольку означенные методы манипулируют блокировками, которые неявно присутствуют в каждом объекте. В результате вы можете использовать ожидание (`wait()`) в любом синхронизированном методе или блоке, не обращая внимания на то, наследует ли этот класс от `Thread` или реализует ли он `Runnable`. Вообще говоря, *единственное* место, где допустимо вызывать метод `wait()`, — это синхронизированный метод или блок (метод `sleep()` можно вызывать в любом месте, так как он не манипулирует блокировкой). Если вы вызовете метод `wait()` или `notify()` в несинхронизированном методе, программа скомпилируется, однако при ее выполнении возникнет исключение `IllegalMonitorStateException` с несколько туманным сообщением: «текущий поток не является владельцем» («current thread not owner»). Это сообщение означает, что поток, востребовавший методы `wait()`, `notify()` или `notifyAll()`, должен быть «хозяином» блокируемого объекта (овладеть объектом блокировки) перед вызовом любого из данных методов.

Возможно «попросить» объект провести операции с помощью его собственного объекта блокировки. Чтобы сделать это, для начала необходимо захватить объект блокировки данного объекта. Например, если вы хотите отправить вызов `notifyAll()` объекту `x`, то должны сделать это в синхронизируемом блоке, захватывающем объект блокировки объекта `x`:

```
synchronized(x) {  
    x.notifyAll();  
}
```

Рассмотрим простой пример. Автомат `WaxOMatic.java` содержит два процесса: один наносит воск на кузов машины (`Car`), а другой полирует его. Задача полировки не может быть выполнена, пока не будет завершена задача нанесения, а задача нанесения должна дожидаться завершения полировки, прежде чем нанести другой слой воска. И `WaxOn` и `WaxOff` используют объект `Car`, который использует методы `wait()` и `notifyAll()` для приостановки и активизации задач, ожидающих изменения внешнего условия:

```
//: concurrency/waxomatic/WaxOMatic.java  
// Простейшее взаимодействие между задачами.  
package concurrency.waxomatic;  
import java.util.concurrent.*;  
import static net.mindview.util.Print.*;  
  
class Car {  
    private boolean waxOn = false;  
    public synchronized void waxed() {  
        waxOn = true; // Готово к полировке  
        notifyAll();  
    }  
    public synchronized void buffed() {  
        waxOn = false; // Готово к нанесению слоя воска  
        notifyAll();  
    }  
    public synchronized void waitForWaxing()  
    throws InterruptedException {  
        while(waxOn == false)  
            wait();  
    }  
    public synchronized void waitForBuffing()  
    throws InterruptedException {  
        while(waxOn == true)  
            wait();  
    }  
}  
  
class WaxOn implements Runnable {  
    private Car car;  
    public WaxOn(Car c) { car = c; }  
    public void run() {  
        try {  
            while(!Thread.interrupted()) {  
                printnb("Wax On! ");  
                TimeUnit.MILLISECONDS.sleep(200);  
                car.waxed();  
                car.waitForBuffing();  
            }  
        }  
    }  
}
```

```

    } catch(InterruptedException e) {
        print("Exiting via interrupt");
    }
    print("Ending Wax On task");
}
}

class WaxOff implements Runnable {
    private Car car;
    public WaxOff(Car c) { car = c; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                car.waitForWaxing();
                printnb("Wax Off! ");
                TimeUnit.MILLISECONDS.sleep(200);
                car.buffed();
            }
        } catch(InterruptedException e) {
            print("Exiting via interrupt");
        }
        print("Ending Wax Off task");
    }
}

public class WaxOMatic {
    public static void main(String[] args) throws Exception {
        Car car = new Car();
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new WaxOff(car));
        exec.execute(new WaxOn(car));
        TimeUnit.SECONDS.sleep(5); // Проработать некоторое время...
        exec.shutdownNow();        // Прервать все задачи
    }
} /* Output: (95% match)
Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On!
Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off!
Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On!
Wax Off! Wax On! Wax Off! Wax On! Exiting via interrupt
Ending Wax On task
Exiting via interrupt
Ending Wax Off task
*///:~

```

Здесь объект `Car` содержит логическую переменную `waxOn`, обозначающую состояние процесса нанесения воска/полировки.

В методе `waitForWaxing()` проверяется флаг `waxOn`, и если он равен `false`, вызывающая задача приостанавливается вызовом `wait()`. Важно, что это происходит в синхронизированном методе, где задача захватила блокировку. При вызове `wait()` поток приостанавливается, а блокировка освобождается. Освобождение блокировки очень важно, потому что для безопасного изменения состояния объекта (например, для изменения `waxOn` в `true` — что должно произойти, чтобы приостановленная задача продолжилась) блокировка должна быть доступна, чтобы ее могла захватить другая задача. В этом примере, когда другая задача вызывает `waxed()`, показывая, что пришло время что-то сделать, для изменения `waxOn` в состояние `true` должна быть захвачена блокировка.

Впоследствии `waxed()` вызывает `notifyAll()`, что приводит к «пробуждению» задачи, приостановленной вызовом `wait()`. Чтобы задача активизировалась из `wait()`, она должна сначала заново захватить блокировку, освобожденную при входе в `wait()`. Задача не активизируется, пока блокировка не станет доступной¹.

Метод `waxOn.run()` представляет первый шаг процесса восковой полировки машины, и он выполняет свою операцию: вызов `sleep()` моделирует время, необходимое для нанесения воска. Затем он сообщает «машине», что операция завершена, и вызывает метод `waitForBuffing()`, который приостанавливает задачу методом `wait()`, пока задача `waxOff` не вызовет `buffed()` для машины, изменяя состояние и вызывая `notifyAll()`. С другой стороны, `waxOff.run()` немедленно переходит в `waitForWaxing()`, а следовательно, приостанавливается до того момента, когда `waxOn` нанесет воск и будет вызван метод `waxed()`. При выполнении этой программы можно увидеть, как этот двухшаговый процесс повторяется с передачей управления между двумя задачами. Через 5 секунд `interrupt()` останавливает оба потока; при вызове `shutdownNow()` для объекта `ExecutorService` вызывается метод `interrupt()` для всех задач, находящихся под его управлением.

Предыдущий пример наглядно показывает, что вызов `wait()` должен быть заключен в цикл `while`, проверяющий интересующее вас условие(-я). Это важно, потому что:

- ❑ несколько задач могут ожидать одного объекта блокировки по одной причине, и первая активизировавшаяся задача может изменить ситуацию (даже если вы этого не делаете, кто-то другой может использовать наследование от вашего класса). В такой ситуации задача должна приостанавливаться снова до тех пор, пока не изменится условие;
- ❑ к тому моменту, когда задача активизируется из `wait()`, может случиться так, что другая задача изменит ситуацию, и текущая задача окажется неспособной (или незаинтересованной) к выполнению операции. Задача должна быть снова приостановлена повторным вызовом `wait()`;
- ❑ также может оказаться, что задачи ожидают блокировки вашего объекта по другим причинам (в данной ситуации необходимо использовать `notifyAll()`). В этом случае необходимо проверить, произошла ли активизация по правильной причине, и если нет — снова вызвать `wait()`.

Следовательно, крайне необходимо, чтобы ваша программа проверяла интересующее вас условие и возвращалась к `wait()`, если условие не выполняется. Идиоматически эта схема формулируется с использованием `while`.

- 21. (2)** Создайте два объекта `Runnable`; один содержит метод `run()`, который запускает и вызывает `wait()`. Второй класс должен сохранять ссылку на первый объект

¹ На некоторых платформах существует третий способ выхода из `wait()`: так называемая *спонтанная активизация*. По сути это означает, что поток может преждевременно выйти из заблокированного состояния (в процессе ожидания по условной переменной или семафору) без «подсказок» в виде `notify()` или `notifyAll()` (или их эквивалентов для новых объектов `Condition`). Поток просто активизируется «сам собой». Спонтанная активизация существует из-за того, что реализация потоков POSIX или их эквивалентов на некоторых платформах не так проста, как хотелось бы. Разрешение спонтанной активизации упрощает построение библиотек (таких, как `pthreads`) на таких платформах.

`Runnable`. Его метод `run()` должен вызывать `notifyAll()` для первой задачи после истечения некоторого количества секунд, чтобы первая задача могла вывести сообщение. Протестируйте свои классы с использованием `Executor`.

- 22.** (4) Реализуйте пример активного ожидания. Одна задача приостанавливается на некоторое время, после чего устанавливает флаг. Вторая задача следит за флагом в цикле `while` (активное ожидание), и когда флаг принимает значение `true`, сбрасывает его в состояние `false` и выводит информацию об изменении на консоль. Обратите внимание, сколько времени программа проводит в активном ожидании, и создайте вторую версию программы, использующую `wait()` вместо активного ожидания.

Пропущенные сигналы

При координации двух потоков с использованием `notify()/wait()` или `notifyAll()/wait()` возможна потеря сигналов. Допустим, поток `T1` оповещает `T2`, а два потока реализуются по следующей (ошибочной) схеме:

```
T1:
synchronized(sharedMonitor) {
    <настройка условия для T2>
    sharedMonitor.notify();
}

T2:
while(someCondition) {
    // Точка 1
    synchronized(sharedMonitor) {
        sharedMonitor.wait();
    }
}
```

<настройка условия для T2> — действие, которое предотвращает вызов `wait()` потоком `T2` (если это не было сделано ранее).

Предположим, `T2` проверяет условие `someCondition` и обнаруживает, что оно истинно. В точке 1 планировщик потоков может переключиться на `T1`. Поток `T1` выполняет настройку, после чего вызывает `notify()`. Когда `T2` продолжает выполнение, он уже не успевает осознать, что условие изменилось, и слепо входит в `wait()`. Вызов `notify()` будет пропущен, и `T2` будет неопределенно долго ожидать сигнала, который уже был отправлен, что приведет к взаимной блокировке.

Проблема решается предотвращением ситуации гонки по переменной `someCondition`. Вот как выглядит правильный подход для `T2`:

```
synchronized(sharedMonitor) {
    while(someCondition)
        sharedMonitor.wait();
}
```

Если теперь `T1` выполняется в первую очередь, то при возвращении управления `T2` определит, что условие изменилось, и не будет входить в `wait()`. И наоборот, если сначала выполняется `T2`, произойдет вход в `wait()` с последующей активизацией со стороны `T1`. Таким образом, возможность пропуска сигнала исключается.

notify() и notifyAll()

Так как формально для одного объекта `Car` более одной задачи может находиться в ожидании `wait()`, вместо простого `notify()` будет безопаснее вызвать `notifyAll()`. Однако структура приведенной выше программы такова, что в `wait()` будет находиться только одна задача, поэтому вы можете использовать `notify()` вместо `notifyAll()`.

Использование `notify()` вместо `notifyAll()` является оптимизацией. Только одна задача из многих кандидатов, ожидающих по объекту блокировки, будет активизирована вызовом `notify()`, так что при попытке использования `notify()` вы должны быть уверены в том, что активизируется именно та задача, которая вам нужна. Кроме того, для использования `notify()` все задачи должны ожидать по одному условию. При использовании `notify()` изменение условия должно быть актуально только для одной задачи. Наконец, эти ограничения всегда должны быть истинны для всех возможных субклассов. Если какие-либо из этих условий не выполняются, используйте `notifyAll()` вместо `notify()`.

В дискуссиях по поводу потоков Java часто встречается невразумительное утверждение, будто вызов `notifyAll()` активизирует «все ожидающие задачи». Означает ли это, что любая задача, находящаяся в `wait()` в каком угодно месте программы, будет активизироваться вызовом `notifyAll()`? В следующем примере код, связанный с `Task2`, показывает, что это не так — только задачи, ожидающие конкретной блокировки, будут активизированы при вызове `notifyAll()` для этой блокировки:

```
//: concurrency/NotifyVsNotifyAll.java
import java.util.concurrent.*;
import java.util.*;

class Blocker {
    synchronized void waitingCall() {
        try {
            while(!Thread.interrupted()) {
                wait();
                System.out.print(Thread.currentThread() + " ");
            }
        } catch (InterruptedException e) {
            // Допустимый способ выхода
        }
    }
    synchronized void prod() { notify(); }
    synchronized void prodAll() { notifyAll(); }
}

class Task implements Runnable {
    static Blocker blocker = new Blocker();
    public void run() { blocker.waitingCall(); }
}

class Task2 implements Runnable {
    // Отдельный объект Blocker:
    static Blocker blocker = new Blocker();
    public void run() { blocker.waitingCall(); }
}
```

```

public class NotifyVsNotifyAll {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new Task());
        exec.execute(new Task2());
        Timer timer = new Timer();
        timer.scheduleAtFixedRate(new TimerTask() {
            boolean prod = true;
            public void run() {
                if(prod) {
                    System.out.print("\nnotify() ");
                    Task.blocker.prod();
                    prod = false;
                } else {
                    System.out.print("\nnotifyAll() ");
                    Task.blocker.prodAll();
                    prod = true;
                }
            }
        }, 400, 400); // Выполнять каждые .4 секунды
        TimeUnit.SECONDS.sleep(5); // Проработать некоторое время...
        timer.cancel();
        System.out.println("\nTimer canceled");
        TimeUnit.MILLISECONDS.sleep(500);
        System.out.print("Task2.blocker.prodAll() ");
        Task2.blocker.prodAll();
        TimeUnit.MILLISECONDS.sleep(500);
        System.out.println("\nShutting down");
        exec.shutdownNow(); // Прервать все задачи
    }
}

/* Output: (Sample)
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-5,5,main] Thread[pool-1-thread-4,5,main]
Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-
2,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-2,5,main] Thread[pool-1-thread-3,5,main]
Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-
5,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-5,5,main] Thread[pool-1-thread-4,5,main]
Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-
2,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-2,5,main] Thread[pool-1-thread-3,5,main]
Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-
5,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-5,5,main] Thread[pool-1-thread-4,5,main]
Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-
2,5,main]
notify() Thread[pool-1-thread-1,5,main]

```

```

notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-2,5,main] Thread[pool-1-thread-3,5,main]
Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-
5,5,main]
Timer canceled
Task2.blocker.prodAll() Thread[pool-1-thread-6,5,main]
Shutting down
*///:~

```

Каждая из задач `Task` и `Task2` имеет собственный объект `Blocker`, так что все объекты `Task` блокируются по `Task.blocker`, а все объекты `Task2` блокируются по `Task2.blocker`. В методе `main()` объект `java.util.Timer` настраивается на выполнение метода `run()` каждые 4/10 секунды, и этот метод `run()` переключается между вызовами `notify()` и `notifyAll()` для `Task.blocker` через методы `prod`.

Из выходных данных видно, что хотя объект `Task2` существует и блокируется по `Task2.blocker`, вызовы `notify()` или `notifyAll()` для `Task.blocker` не приводят к активизации `Task2`. Аналогичным образом в конце `main()` метод `cancel()` вызывается для `timer`, и несмотря на отмену таймера, первые пять задач продолжают работать и блокируются в своих вызовах `Task.blocker.waitingCall()`. Результат вызова `Task2.blocker.prodAll()` не включает никаких задач, ожидающих по блокировке в `Task.blocker`.

Происходящее также становится более понятным, если присмотреться к методам `prod()` и `prodAll()` в `Blocker`. Эти методы синхронизированы; это означает, что они получают собственную блокировку, поэтому вполне логично, что вызовы `notify()` или `notifyAll()` относятся только к этой блокировке, а следовательно, активизируют только те задачи, которые ожидают по этой конкретной блокировке.

Метод `blocker.waitingCall()` достаточно прост, чтобы вы могли использовать запись `for(;;)` вместо `while(!Thread.interrupted())` с тем же эффектом, потому что в данном примере нет различий между выходом из цикла с исключением и выходом из него с проверкой флага `interrupted()` — в обоих случаях выполняется один код. Впрочем, формально в этом примере проверяется `interrupted()`, потому что существуют два разных пути выхода. Если позднее вы решите добавить в цикл дополнительный код, появляется риск внесения ошибки, если не будут предусмотрены оба пути выхода из цикла.

- 23.** (7) Продемонстрируйте, что пример `WaxOMatic.java` успешно работает при использовании `notify()` вместо `notifyAll()`.

Производители и потребители

В качестве примера заглянем в ресторан, в котором на всех один шеф-повар и один официант. Официант должен ждать, пока шеф-повар приготовит кушанье. Как только шеф-повар заканчивает стряпню, он подает официанту сигнал, и последний доставляет блюдо до места и возвращается к ожиданию. Это великолепный пример кооперации задач: шеф-повар представляет из себя *производителя*, а официант является *потребителем*. Задачи должны согласовать свои действия друг с другом при приготовлении и потреблении еды, а система должна корректно завершить свою работу. Вот как выглядит ситуация, смоделированная в коде:

```

//: concurrency/Restaurant.java
// Взаимодействие потоков в модели "производитель-потребитель"
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Meal {
    private final int orderNum;
    public Meal(int orderNum) { this.orderNum = orderNum; }
    public String toString() { return "Meal " + orderNum; }
}

class WaitPerson implements Runnable {
    private Restaurant restaurant;
    public WaitPerson(Restaurant r) { restaurant = r; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                synchronized(this) {
                    while(restaurant.meal == null)
                        wait(); // ... Пока повар приготовит блюдо
                }
                print("Waitperson got " + restaurant.meal);
                synchronized(restaurant.chef) {
                    restaurant.meal = null;
                    restaurant.chef.notifyAll(); // Готово для следующего блюда
                }
            }
        } catch(InterruptedException e) {
            print("WaitPerson interrupted");
        }
    }
}

class Chef implements Runnable {
    private Restaurant restaurant;
    private int count = 0;
    public Chef(Restaurant r) { restaurant = r; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                synchronized(this) {
                    while(restaurant.meal != null)
                        wait(); // ... Когда заберут блюдо
                }
                if(++count == 10) {
                    print("Out of food, closing");
                    restaurant.exec.shutdownNow();
                }
                printnb("Order up! ");
                synchronized(restaurant.waitPerson) {
                    restaurant.meal = new Meal(count);
                    restaurant.waitPerson.notifyAll();
                }
                TimeUnit.MILLISECONDS.sleep(100);
            }
        } catch(InterruptedException e) {
            print("Chef interrupted");
        }
    }
}

```



```

public class Restaurant {
    Meal meal;
    ExecutorService exec = Executors.newCachedThreadPool();
    WaitPerson waitPerson = new WaitPerson(this);
    Chef chef = new Chef(this);
    public Restaurant() {
        exec.execute(chef);
        exec.execute(waitPerson);
    }
    public static void main(String[] args) {
        new Restaurant();
    }
} /* Output:
Order up! Waitperson got Meal 1
Order up! Waitperson got Meal 2
Order up! Waitperson got Meal 3
Order up! Waitperson got Meal 4
Order up! Waitperson got Meal 5
Order up! Waitperson got Meal 6
Order up! Waitperson got Meal 7
Order up! Waitperson got Meal 8
Order up! Waitperson got Meal 9
Out of food, closing
WaitPerson interrupted
Order up! Chef interrupted
*///:~

```

Объект `Restaurant` является «координационным центром» как для `WaitPerson`, так и для `Chef`. Оба объекта должны знать, в каком ресторане `Restaurant` они работают, так как заказы они получают из окошка этого ресторана, `restaurant.meal`. В методе `run()` объект `WaitPerson` переходит в состояние ожидания `wait()`, приостанавливая поток, пока не получит оповещения `notifyAll()` от `Chef`. Так как это очень простая программа, мы знаем, что освобождения объекта `WaitPerson` будет ждать только одна задача — сама задача `WaitPerson`. По этой причине теоретически можно использовать `notify()` вместо `notifyAll()`. В более сложных ситуациях освобождения определенного объекта могут ожидать несколько потоков, соответственно, вам не дано знать, какой поток следует призывать к работе. Решением является использование метода `notifyAll()`, который пробуждает все потоки, ожидающие снятия блокировки. Каждый поток должен после этого определить, было ли поступившее извещение для него актуальным.

После того как `Chef` выдаст `Meal` и оповестит `WaitPerson`, `Chef` ожидает, пока `WaitPerson` не заберет блюдо и не оповестит задачу `Chef`, которая может взяться за «приготовление» следующего объекта `Meal`.

Заметьте, что вызов `wait()` заключен в цикл `while`, который проверяет то же условие, которого и ожидает поток. Поначалу это кажется странным — раз уж вы ожидаете заказа, то как только вы проснетесь, заказ должен быть готов, ведь так? Проблема здесь в том, что в многозадачном приложении некий другой поток может вмешаться в процесс и увести заказ из-под носа, пока `WaitPerson` протирает глаза. Единственно безопасный подход — это *всегда* использовать с `wait()` следующую конструкцию (естественно, с правильно организованной синхронизацией и программированием возможной потери сигналов):

```

while(conditionIsNotMet)
    wait();

```

Она гарантирует, что прежде чем вы выберетесь из цикла ожидания, условие будет выполнено, а в том случае, если вы были оповещены о чем-то не имеющем отношения к условию (что может случиться в варианте метода `notifyAll()`) или условие изменилось еще до того, как вы полностью покинули цикл ожидания, гарантирован возврат к ожиданию.

Заметьте, что вызов `notifyAll()` должен сначала захватить объект блокировки объекта `waitPerson`. Вызов метода `wait()` в методе `waitPerson.run()` автоматически освобождает блокируемый объект, значит, это возможно. Благодаря тому, что для вызова метода `notifyAll()` требуется владеть объектом блокировки, гарантируется, что два потока, пытающиеся вызвать `notifyAll()` по одному объекту, не «перебегут друг другу дорогу».

Оба метода `run()` спроектированы с расчетом на корректное завершение, для чего весь вызов `run()` заключается в блок `try()`. Блок `catch` закрывается сразу же после закрывающей фигурной скобки метода `run()`, так что если задача получает `InterruptedException`, она немедленно завершается после перехвата исключения.

В коде `Chef` обратите внимание на то, что после вызова `shutdownNow()` *можно* просто вернуть управление из `run()`; собственно, обычно именно так и следует поступать. Тем не менее наше решение немного интереснее. Помните, что `shutdownNow()` отправляет `interrupt()` всем задачам, запущенным `ExecutorService`. Но в случае `Chef` задача не завершается немедленно после получения `interrupt()`, потому что прерывание возбуждает `InterruptedException` только тогда, когда задача пытается войти в (прерываемую) блокирующую операцию. Таким образом, сначала выводится сообщение «Order up!», а затем возбуждается исключение `InterruptedException`, когда `Chef` пытается вызвать `sleep()`. Если убрать вызов `sleep()`, задача доберется до начала цикла `run()` и завершится из-за проверки `Thread.interrupted()` без возбуждения исключения.

В предыдущем примере используется только одна точка, в которой одна задача сохраняет объект, чтобы другая задача могла позднее этим объектом воспользоваться. Однако в типичной реализации «производитель-потребитель» для хранения производимых и потребляемых объектов используется очередь FIFO. Такие очереди более подробно рассматриваются позже в этой главе.

- 24.** (1) Решите задачу сотрудничества с одним производителем и одним потребителем, используя методы `wait()` и `notifyAll()`. Производитель не должен переполнить буфер потребителя-получателя, что может произойти в том случае, если первый работает быстрее второго. Если же потребитель действует оперативнее, он не должен считывать одни и те же данные по несколько раз. Не стройте никаких предположений в отношении относительной скорости их работы.
- 25.** (1) В классе `Chef` из примера `Restaurant.java` верните управление (`return`) из `run()` после вызова `shutdownNow()`. Обратите внимание на различия в поведении программы.
- 26.** (8) Добавьте в пример `Restaurant.java` класс `BusBoy` («уборщик»). После того как блюдо будет доставлено, класс `waitPerson` должен оповестить `BusBoy` о необходимости убрать.

Явное использование объектов Lock и Condition

В библиотеке Java SE5 `java.util.concurrent` существуют дополнительные средства, которые могут использоваться для переработки `WaxOMatic.java`. `Condition` — несложный класс, использующий мьютекс и обеспечивающий возможность приостановки задачи вызовом `await()` для `Condition`. Когда происходит изменение внешнего состояния, которое может означать, что задача может продолжить выполнение, вы оповещаете задачу вызовом `signal()` (для активизации одной задачи) или `signalAll()` (для активизации всех задач, которые приостановились по этому объекту `Condition`) (как и в случае с `notifyAll()`, вызов `signalAll()` является более надежным решением).

Ниже приведен пример `WaxOMatic.java`, переписанный с включением объекта `Condition`, который приостанавливает задачу внутри `waitForWaxing()` или `waitForBuffing()`:

```
//: concurrency/waxomatic2/WaxOMatic2.java
// Использование объектов Lock и Condition.
package concurrency.waxomatic2;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import static net.mindview.util.Print.*;

class Car {
    private Lock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();
    private boolean waxOn = false;
    public void waxed() {
        lock.lock();
        try {
            waxOn = true; // Готово к полировке
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
    public void buffed() {
        lock.lock();
        try {
            waxOn = false; // Готово к нанесению слоя воска
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
    public void waitForWaxing() throws InterruptedException {
        lock.lock();
        try {
            while(waxOn == false)
                condition.await();
        } finally {
            lock.unlock();
        }
    }
    public void waitForBuffing() throws InterruptedException{
        lock.lock();
        try {
            while(waxOn == true)
```

продолжение ➤

```

        condition.await();
    } finally {
        lock.unlock();
    }
}
}

class WaxOn implements Runnable {
    private Car car;
    public WaxOn(Car c) { car = c; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                printnb("Wax On! ");
                TimeUnit.MILLISECONDS.sleep(200);
                car.waxed();
                car.waitForBuffing();
            }
        } catch(InterruptedException e) {
            print("Exiting via interrupt");
        }
        print("Ending Wax On task");
    }
}

class WaxOff implements Runnable {
    private Car car;
    public WaxOff(Car c) { car = c; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                car.waitForWaxing();
                printnb("Wax Off! ");
                TimeUnit.MILLISECONDS.sleep(200);
                car.buffed();
            }
        } catch(InterruptedException e) {
            print("Exiting via interrupt");
        }
        print("Ending Wax Off task");
    }
}

public class WaxOMatic2 {
    public static void main(String[] args) throws Exception {
        Car car = new Car();
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new WaxOff(car));
        exec.execute(new WaxOn(car));
        TimeUnit.SECONDS.sleep(5);
        exec.shutdownNow();
    }
}
/* Output: (90% match)
Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On!
Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off!
Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On!
Wax Off! Wax On! Wax Off! Wax On! Exiting via interrupt
Ending Wax Off task

```

```
Exiting via interrupt
Ending Wax On task
*///:~
```

В конструкторе `Car` один объект `Lock` производит объект `Condition`, который используется для управления взаимодействиями между задачами. Однако объект `Condition` не содержит информации о состоянии процесса, поэтому нам понадобится дополнительная логическая переменная `waxOn`.

Сразу же после каждого вызова `lock()` должна следовать конструкция `try-finally`, которая гарантирует, что блокировка будет снята в любом случае. Как и со встроенными версиями, задача должна захватить блокировку, прежде чем вызывать `await()`, `signal()` или `signalAll()`.

Следует заметить, что это решение сложнее предыдущего, и сложность в данном случае ничего не дает. Объекты `Lock` и `Condition` необходимы только для более сложных потоковых задач.

- 27. (2)** Измените пример `Restaurant.java` так, чтобы в нем явно использовались объекты `Lock` и `Condition`.

Производители-потребители и очереди

Методы `wait()` и `notifyAll()` решают проблему взаимодействия задач на довольно низком уровне, с согласованием каждого взаимодействия. Во многих случаях можно подняться на более высокий уровень абстракции и решить проблему кооперации задач при помощи синхронизированной очереди, которая разрешает выполнять вставку и удаление только одной задаче. Эта возможность предоставляется интерфейсом `java.util.concurrent.BlockingQueue`, который имеет ряд стандартных реализаций. Чаще всего используется очередь неограниченного размера `LinkedBlockingQueue`; `ArrayBlockingQueue` имеет фиксированный размер, поэтому количество элементов, которые можно поместить в очередь до того, как она будет заблокирована, ограничено.

Эти очереди также приостанавливают задачу-потребителя, если эта задача попытается получить объект из пустой очереди, и возобновляют выполнение при появлении элементов. Блокирующие очереди способны решить многие задачи куда более простым и надежным способом, чем `wait()` и `notifyAll()`.

Ниже приведен простой пример с организацией последовательного выполнения объектов `LiftOff`. Потребителем является объект `LiftOffRunner`, который извлекает каждый объект `LiftOff` из очереди `BlockingQueue` и выполняет его. (То есть он использует собственный поток, явно вызывая `run()`, вместо того чтобы порождать новый поток для каждой задачи.)

```
//: concurrency/TestBlockingQueues.java
// {RunByHand}
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class LiftOffRunner implements Runnable {
    private BlockingQueue<LiftOff> rockets;
```

продолжение ➤

```

public LiftOffRunner(BlockingQueue<LiftOff> queue) {
    rockets = queue;
}
public void add(LiftOff lo) {
    try {
        rockets.put(lo);
    } catch (InterruptedException e) {
        print("Interrupted during put()");
    }
}
public void run() {
    try {
        while(!Thread.interrupted()) {
            LiftOff rocket = rockets.take();
            rocket.run(); // Использовать этот поток
        }
    } catch (InterruptedException e) {
        print("Waking from take()");
    }
    print("Exiting LiftOffRunner");
}
}

public class TestBlockingQueues {
    static void getkey() {
        try {
            // Компенсация различий Windows/Linux в длине результата,
            // производимого клавишей Enter:
            new BufferedReader(
                new InputStreamReader(System.in)).readLine();
        } catch (java.io.IOException e) {
            throw new RuntimeException(e);
        }
    }
    static void getkey(String message) {
        print(message);
        getkey();
    }

    static void
    test(String msg, BlockingQueue<LiftOff> queue) {
        print(msg);
        LiftOffRunner runner = new LiftOffRunner(queue);
        Thread t = new Thread(runner);
        t.start();
        for(int i = 0; i < 5; i++)
            runner.add(new LiftOff(5));
        getkey("Press 'Enter' (" + msg + ")");
        t.interrupt();
        print("Finished " + msg + " test");
    }
    public static void main(String[] args) {
        test("LinkedBlockingQueue", // Неограниченный размер
            new LinkedBlockingQueue<LiftOff>());
        test("ArrayBlockingQueue", // Фиксированный размер
            new ArrayBlockingQueue<LiftOff>(3));
        test("SynchronousQueue", // Размер 1
            new SynchronousQueue<LiftOff>());
    }
}

```

```

    }
} ///:~

```

Задачи помещаются в очередь `BlockingQueue` в `main()` и извлекаются из `BlockingQueue` в `LiftOffRunner`. Заметьте, что `LiftOffRunner` может игнорировать проблемы синхронизации, потому что эти проблемы решаются в `BlockingQueue`.

- 28.** (3) Измените пример `TestBlockingQueues.java` и добавьте новую задачу, которая помещает `LiftOff` в `BlockingQueue` (вместо того, чтобы делать это в `main()`).

Очередь `BlockingQueue` с элементами `toast`

Рассмотрим пример использования `BlockingQueue` — автомат с тремя операциями: одна готовит тост, другая мажет его маслом, а третья наносит джем на тост с маслом. Объекты `Toast` можно передавать между процессами через `BlockingQueue`:

```

//: concurrency/ToastOMatic.java
// Тостер с использованием очередей.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Toast {
    public enum Status { DRY, BUTTERED, JAMMED }
    private Status status = Status.DRY;
    private final int id;
    public Toast(int idn) { id = idn; }
    public void butter() { status = Status.BUTTERED; }
    public void jam() { status = Status.JAMMED; }
    public Status getStatus() { return status; }
    public int getId() { return id; }
    public String toString() {
        return "Toast " + id + ": " + status;
    }
}

class ToastQueue extends LinkedBlockingQueue<Toast> {}

class Toaster implements Runnable {
    private ToastQueue toastQueue;
    private int count = 0;
    private Random rand = new Random(47);
    public Toaster(ToastQueue tq) { toastQueue = tq; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                TimeUnit.MILLISECONDS.sleep(
                    100 + rand.nextInt(500));
                // Приготовление тоста
                Toast t = new Toast(count++);
                print(t);
                // Вставка в очередь
                toastQueue.put(t);
            }
        } catch(InterruptedException e) {
            print("Toaster interrupted");
        }
    }
}

```

```

        print("Toaster off");
    }
}

```

// Нанесение масла:

```

class Butterer implements Runnable {
    private ToastQueue dryQueue, butteredQueue;
    public Butterer(ToastQueue dry, ToastQueue buttered) {
        dryQueue = dry;
        butteredQueue = buttered;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Блокирует до готовности следующего тоста:
                Toast t = dryQueue.take();
                t.butter();
                print(t);
                butteredQueue.put(t);
            }
        } catch(InterruptedException e) {
            print("Butterer interrupted");
        }
        print("Butterer off");
    }
}

```

// Нанесение джема на тост с маслом:

```

class Jammer implements Runnable {
    private ToastQueue butteredQueue, finishedQueue;
    public Jammer(ToastQueue buttered, ToastQueue finished) {
        butteredQueue = buttered;
        finishedQueue = finished;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Блокирует до готовности следующего тоста:
                Toast t = butteredQueue.take();
                t.jam();
                print(t);
                finishedQueue.put(t);
            }
        } catch(InterruptedException e) {
            print("Jammer interrupted");
        }
        print("Jammer off");
    }
}

```

// Потребление тоста:

```

class Eater implements Runnable {
    private ToastQueue finishedQueue;
    private int counter = 0;
    public Eater(ToastQueue finished) {
        finishedQueue = finished;
    }
    public void run() {

```



```

try {
    while(!Thread.interrupted()) {
        // Блокирует до готовности следующего тоста:
        Toast t = finishedQueue.take();
        // Проверить, что тосты следуют по порядку,
        // а все тосты намазаны джемом:
        if(t.getId() != counter++ ||
            t.getStatus() != Toast.Status.JAMMED) {
            print(">>>> Error: " + t);
            System.exit(1);
        } else
            print("Chomp! " + t);
    }
} catch(InterruptedException e) {
    print("Eater interrupted");
}
print("Eater off");
}
}

public class ToastOMatic {
    public static void main(String[] args) throws Exception {
        ToastQueue dryQueue = new ToastQueue(),
            butteredQueue = new ToastQueue(),
            finishedQueue = new ToastQueue();
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new Toaster(dryQueue));
        exec.execute(new Butterer(dryQueue, butteredQueue));
        exec.execute(new Jammer(butteredQueue, finishedQueue));
        exec.execute(new Eater(finishedQueue));
        TimeUnit.SECONDS.sleep(5);
        exec.shutdownNow();
    }
} /* (Execute to see output) *///:~

```

Toast отлично демонстрирует полезность перечислений. Обратите внимание на отсутствие явной синхронизации (с использованием объектов Lock или ключевого слова synchronized) — синхронизация неявно обеспечивается очередями (которые синхронизируются во внутренней реализации) и строением системы — с каждым объектом Toast в любой момент времени работает только одна задача. Так как очереди блокируются, процессы приостанавливаются и возобновляются автоматически. Как видите, упрощение от применения BlockingQueue получается весьма серьезным. Связность классов, возникающая при явном использовании команд wait() и notifyAll(), устраняется, потому что каждый класс взаимодействует только со своими очередями BlockingQueue.

- 29.** (8) Измените пример ToastOMatic.java так, чтобы в нем создавались сэндвичи с арахисовым маслом и желе, с использованием двух разных «конвейеров» (один для арахисового масла, другой для желе), которые затем сливаются воедино.

Использование каналов для ввода-вывода между потоками

Часто бывает полезно организовать взаимодействие потоков посредством механизмов ввода-вывода. Библиотеки потоков могут предлагать поддержку ввода-вывода между потоками в форме *каналов* (pipes). Последние существуют в стандартной библиотеке

ввода-вывода Java в виде классов `PipedWriter` (позволяет потоку записывать в канал) и `PipedReader` (предоставляет возможность другому потоку считывать из того же канала). Происходящее напоминает вариант задачи «производитель-потребитель», для которой канал является типовым решением. Канал по сути представляет собой блокирующую очередь, существовавшую в Java до появления `BlockingQueue`.

В следующем простом примере две задачи используют канал для взаимодействия.

```
//: concurrency/PipedIO.java
// Использование каналов для организации ввода-вывода между задачами
import java.util.concurrent.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Sender implements Runnable {
    private Random rand = new Random(47);
    private PipedWriter out = new PipedWriter();
    public PipedWriter getPipedWriter() { return out; }
    public void run() {
        try {
            while(true)
                for(char c = 'A'; c <= 'z'; c++) {
                    out.write(c);
                    TimeUnit.MILLISECONDS.sleep(rand.nextInt(500));
                }
        } catch(IOException e) {
            print(e + " Sender write exception");
        } catch(InterruptedException e) {
            print(e + " Sender sleep interrupted");
        }
    }
}

class Receiver implements Runnable {
    private PipedReader in;
    public Receiver(Sender sender) throws IOException {
        in = new PipedReader(sender.getPipedWriter());
    }
    public void run() {
        try {
            while(true) {
                // Блокируется до появления символов:
                printnb("Read: " + (char)in.read() + ", ");
            }
        } catch(IOException e) {
            print(e + " Receiver read exception");
        }
    }
}

public class PipedIO {
    public static void main(String[] args) throws Exception {
        Sender sender = new Sender();
        Receiver receiver = new Receiver(sender);
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(sender);
    }
}
```

```
    exec.execute(receiver);
    TimeUnit.SECONDS.sleep(4);
    exec.shutdownNow();
}
} /* Output: (65% match)
Read: A, Read: B, Read: C, Read: D, Read: E, Read: F, Read:
G, Read: H, Read: I, Read: J, Read: K, Read: L, Read: M,
java.lang.InterruptedException: sleep interrupted Sender
sleep interrupted
java.io.InterruptedIOException Receiver read exception
*///:~
```

Классы `Sender` и `Receiver` представляют задачи, которым требуется взаимодействовать друг с другом. В классе `Sender` создается канал `PipedWriter`, представляющий собой автономный объект, однако при создании канала `PipedReader` в классе `Receiver` необходимо указать в конструкторе ссылку на `PipedWriter`. `Sender` записывает данные в канал `Writer` и бездействует в течение случайно выбранного промежутка времени. Однако в классе `Receiver` вы не найдете следов присутствия `sleep()` или `wait()`. Когда он проводит чтение методом `read()`, то автоматически блокируется при отсутствии данных.

Заметьте, что потоки `sender` и `receiver` запускаются из `main()` *после* того, как объекты были полностью сконструированы. Если запускать не полностью сконструированные объекты, каналы на различных платформах могут демонстрировать несогласованное поведение. (Учтите, что решение с `BlockingQueue` более надежно и просто в использовании.)

Важное различие между `PipedReader` и обычным вводом-выводом проявляется при вызове `shutdownNow()` — объект `PipedReader` является прерываемым, тогда как, например, если заменить вызов `in.read()` на `System.in.read()`, вызов `interrupt()` не приведет к выходу из вызова `read()`.

- 30.** (1) Измените пример `PipedIO.java`, чтобы в нем вместо канала использовалась очередь `BlockingQueue`.

Взаимная блокировка

Итак, потоки способны перейти в заблокированное состояние, а объекты могут обладать синхронизированными методами, которые запрещают использование объекта до тех пор, пока не будет снята блокировка. Вероятна ситуация, в которой один поток ожидает другой поток, тот, в свою очередь, ждет освобождения еще одного потока и так далее, пока эта цепочка не замыкается на поток, который ожидает освобождения первого потока. Получается замкнутый круг потоков, которые дожидаются освобождения друг друга и никто не может двинуться первым. Такая ситуация называется *тупиком*, или *взаимной блокировкой* (deadlock).

Если вы запускаете программу и она незамедлительно оказывается в положении взаимной блокировки, вы сразу понимаете, что у вас проблемы и их следует найти. По-настоящему неприятна ситуация, когда ваша программа по всем признакам работает прекрасно, но тем не менее потенциально способна войти во взаимную блокировку. В таких ситуациях ничто не указывает на возможность возникновения взаимоблокировки, и такая возможность «тихо» присутствует в программе, пока неожиданно-негаданно

не проявится у заказчика (и скорее всего, легко воспроизвести эту ситуацию вам не удастся). Таким образом, тщательное проектирование программы с целью предотвращения взаимных блокировок — важнейшая часть разработки параллельных программ.

Рассмотрим классический пример взаимной блокировки, предложенный Дейкстрой: *задачу «обедающие философы»*. В стандартной формулировке участвуют пять философов (но в показанном дальше примере допустимо любое количество). Часть времени философы проводят размышляя, часть времени проводят за едой. Во время размышлений философы не нуждаются в «общих ресурсах», но во время обеда они сидят за круглым столом с ограниченным количеством столовых приборов. В описании оригинальной задачи столовыми приборами являются вилки, и для того чтобы взять спагетти из миски в центре стола, необходимо две вилки, хотя логичнее будет заменить вилки на палочки для еды — очевидно, что каждому философу понадобятся две палочки.

К сожалению, у философов очень мало денег, поэтому они смогли позволить себе приобрести лишь пять палочек. Последние разложены кругом по столу, между философами. Когда философу захочется поест, ему придется взять палочку слева и справа. Если с какой-либо стороны желаемая палочка уже в руке другого философа, только что оторвавшемуся от размышлений придется подождать, пока она освободится.

```
//: concurrency/Chopstick.java
// Обедающие философы.
```

```
public class Chopstick {
    private boolean taken = false;
    public synchronized
    void take() throws InterruptedException {
        while(taken)
            wait();
        taken = true;
    }
    public synchronized void drop() {
        taken = false;
        notifyAll();
    }
} ///:~
```

Никакие два философа (*Philosopher*) не могут успешно взять (*take()*) одну и ту же палочку (*Chopstick*) одновременно. Вдобавок, если палочка уже захвачена одним философом, другой философ может ожидать (*wait()*), пока палочка освободится (ее текущий владелец вызовет *drop()*).

Когда задача *Philosopher* вызывает *take()*, *Philosopher* ожидает до тех пор, пока флаг *taken* не перейдет в состояние *false* (то есть пока палочка не будет освобождена философом, который удерживает ее в данный момент). Затем задача устанавливает флаг *taken* в *true*, показывая тем самым, что *Chopstick* теперь удерживается новым объектом *Philosopher*. Завершив использование *Chopstick*, *Philosopher* вызывает *drop()*, чтобы изменить состояние флага, и вызывает *notifyAll()* для всех остальных объектов *Philosopher*, которые могут ожидать *Chopstick* в вызове *wait()*.

```
//: concurrency/Philosopher.java
// Обедающий философ
import java.util.concurrent.*;
```

```

import java.util.*;
import static net.mindview.util.Print.*;

public class Philosopher implements Runnable {
    private Chopstick left;
    private Chopstick right;
    private final int id;
    private final int ponderFactor;
    private Random rand = new Random(47);
    private void pause() throws InterruptedException {
        if(ponderFactor == 0) return;
        TimeUnit.MILLISECONDS.sleep(
            rand.nextInt(ponderFactor * 250));
    }
    public Philosopher(Chopstick left, Chopstick right,
        int ident, int ponder) {
        this.left = left;
        this.right = right;
        id = ident;
        ponderFactor = ponder;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                print(this + " " + "thinking");
                pause();
                // Философ проголодался
                print(this + " " + "grabbing right");
                right.take();
                print(this + " " + "grabbing left");
                left.take();
                print(this + " " + "eating");
                pause();
                right.drop();
                left.drop();
            }
        } catch(InterruptedException e) {
            print(this + " " + "exiting via interrupt");
        }
    }
    public String toString() { return "Philosopher " + id; }
} ///:~

```

В методе `Philosopher.run()` каждый философ непрерывно «размышляет» и «ест». Метод `pause()` вызывает `sleep()` на случайный период, если значение `ponderFactor` отлично от нуля. Таким образом, философ размышляет некоторое случайное время, затем пытается захватить левую и правую палочки, ест в течение случайного времени, после чего все повторяется заново.

Теперь мы можем создать версию этой программы с взаимной блокировкой:

```

//: concurrency/DeadlockingDiningPhilosophers.java
// Демонстрация потенциальной взаимной блокировки.
// {Args: 0 5 timeout}
import java.util.concurrent.*;

public class DeadlockingDiningPhilosophers {
    public static void main(String[] args) throws Exception {

```

продолжение ➤

```

int ponder = 5;
if(args.length > 0)
    ponder = Integer.parseInt(args[0]);
int size = 5;
if(args.length > 1)
    size = Integer.parseInt(args[1]);
ExecutorService exec = Executors.newCachedThreadPool();
Chopstick[] sticks = new Chopstick[size];
for(int i = 0; i < size; i++)
    sticks[i] = new Chopstick();
for(int i = 0; i < size; i++)
    exec.execute(new Philosopher(
        sticks[i], sticks[(i+1) % size], i, ponder));
if(args.length == 3 && args[2].equals("timeout"))
    TimeUnit.SECONDS.sleep(5);
else {
    System.out.println("Press 'Enter' to quit");
    System.in.read();
}
exec.shutdownNow();
}
} /* (Execute to see output) *///:~

```

Если философы проводят за размышлениями существенно меньше времени, чем за едой, они будут конкурировать за палочки, и взаимная блокировка проявится намного быстрее.

Первый аргумент командной строки изменяет коэффициент `ponder`, влияя на время, проводимое философами за размышлениями. Если у вас много философов или они проводят много времени за размышлениями, взаимная блокировка может возникать крайне редко, хотя теоретическая возможность остается. Если задать аргумент командной строки равным 0, взаимная блокировка возникнет достаточно быстро.

Объектам `Chopstick` не нужны внутренние идентификаторы; они идентифицируются по своим позициям в массиве `sticks`. Конструктор `Philosopher` получает ссылки на левый и правый объекты `Chopstick`. Каждый объект `Philosopher`, за исключением последнего, инициализируется в позиции между следующей парой объектов `Chopstick`. Последнему объекту `Philosopher` в качестве правого задается нулевой объект `Chopstick`, так что круг замыкается: последний философ сидит справа от первого, поэтому они оба пользуются «нулевой» палочкой `Chopstick`. Теперь может оказаться, что все философы попытаются одновременно взяться за еду, ожидая, пока сидящий справа философ не положит свою палочку. Это приведет к взаимной блокировке программы.

Если философы тратят на размышления больше времени, чем на еду, вероятность обращения к совместным ресурсам (`Chopstick`) снижается и создается впечатление, что программа избавлена от блокировок (при ненулевом значении `ponder` или большом количестве объектов `Philosopher`), хотя это и не так. Этот пример интересен именно тем, что он показывает: программа может на первый взгляд работать абсолютно корректно, хотя на самом деле в ней может возникнуть ситуация взаимной блокировки.

Для решения проблемы необходимо осознавать, что взаимная блокировка возникает при стечении следующих четырех обстоятельств.

1. Взаимное исключение: по крайней мере один ресурс, используемый потоками, не должен быть разделяемым. В нашем случае одной палочкой для еды не могут одновременно есть два философа.

2. По крайней мере один процесс должен удерживать ресурс и ожидать выделения ресурса, в настоящее время удерживаемого другим процессом. То есть для возникновения взаимной блокировки философ должен держать при себе одну палочку и ожидать другую.
3. Ресурс нельзя насильно отбирать у процесса. Все процессы должны освобождать ресурсы естественным путем. Наши философы вежливы и не станут выхватывать палочки друг у друга.
4. Должно произойти круговое ожидание, когда процесс ожидает ресурс, занятый другим процессом, который, в свою очередь, ждет ресурс, удерживаемый еще одним процессом, и так далее, пока один из процессов не будет ожидать ресурса, занятого первым процессом, что и приведет к порочному кругу. В примере `DeadlockingDiningPhilosophers.java` круговое ожидание происходит потому, что каждый философ пытается сначала получить правую палочку, а потом левую.

Так как для возникновения тупиковой ситуации необходимо наличие *всех* перечисленных условий, для упреждения взаимной блокировки достаточно нарушить всего лишь одно из них. В нашей программе взаимную блокировку проще всего предотвратить, нарушая четвертое условие: оно выполняется, поскольку каждый философ старается брать палочки в определенном порядке — сначала правую, потом левую. Из-за этого легко попасть в ситуацию, когда каждый из них держит свою правую палочку и ждет освобождения левой, что и приводит к круговому ожиданию. Но если последний философ будет сначала брать левую палочку, а потом правую, он никогда не помешает философу, сидящему справа от него, взять его палочку. Это всего лишь одно решение проблемы, но ее также можно предотвратить, нарушая одно из оставшихся условий (за подробностями обращайтесь к книгам, полностью посвященным потокам)¹.

```
//: concurrency/FixedDiningPhilosophers.java
// Обедающие философы без взаимной блокировки.
// {Args: 5 5 timeout}
import java.util.concurrent.*;
```

продолжение ➤

¹ Другое решение предлагается в руководстве *Java Tutorial* от Sun. Оно заключается в нумерации палочек от 1 до 5 и в требовании, чтобы философы брали первой палочку с наименьшим номером. Не имеет значения, кто возьмет палочку 1 первым, так как он оставляет доступными другие палочки. В итоге тупик исключается. Демонстрирует этот подход (нерабочий) графический апплет `Java tutorial\essential\threads\example-swing\DiningPhilosophers.java`. Чтобы привести апплет в рабочее состояние, нужно совсем немного: (1) импортировать дополнительный пакет (`import com.bruceeckel.swing.*;`), (2) использовать инструмент Console — описать метод (см. главу 14) `main`:

```
public static void main(String[] args) {
    Console.run(new DiningPhilosophers(), 250, 350); }
```

и (3): объявления `new ImageIcon(getURL(".gif"))`; в начале метода `init` сократить до `new ImageIcon(".gif");`.

Очень подробно задача «обедающих философов» рассматривается в книге «Сетевые спецификации, моделирование и проектирование вычислительных комплексов, систем и сетей» (Беляков М. А., Костюхин Р. А.), частично доступной по адресу <http://alice.stup.ac.ru/~dvn/complex>. Учитывая, что все-таки философы тоже люди и способны на заговор и козни друг против друга, в конечном счете предлагается просто принудительно ограничить число одновременно находящихся за столом мудрецов до четырех. — *Примеч. ред.*

```

public class FixedDiningPhilosophers {
    public static void main(String[] args) throws Exception {
        int ponder = 5;
        if(args.length > 0)
            ponder = Integer.parseInt(args[0]);
        int size = 5;
        if(args.length > 1)
            size = Integer.parseInt(args[1]);
        ExecutorService exec = Executors.newCachedThreadPool();
        Chopstick[] sticks = new Chopstick[size];
        for(int i = 0; i < size; i++)
            sticks[i] = new Chopstick();
        for(int i = 0; i < size; i++)
            if(i < (size-1))
                exec.execute(new Philosopher(
                    sticks[i], sticks[i+1], i, ponder));
            else
                exec.execute(new Philosopher(
                    sticks[0], sticks[i], i, ponder));
        if(args.length == 3 && args[2].equals("timeout"))
            TimeUnit.SECONDS.sleep(5);
        else {
            System.out.println("Press 'Enter' to quit");
            System.in.read();
        }
        exec.shutdownNow();
    }
} /* (Execute to see output) *///:~

```

Требуя, чтобы последний философ брал и клал левую палочку до правой, мы устраняем взаимную блокировку, и программа будет работать без проблем.

В языке Java *нет инструментов для предупреждения тупиков*; вам придется положиться на себя и программировать аккуратнее. Впрочем, вряд ли эти слова утешат того, кому придется отлаживать программу, оказавшуюся в тупике.

- 31.** (8) Измените пример `DeadlockingDiningPhilosophers.java` так, чтобы когда философ завершает пользоваться палочками, он бросает их в корзину. Когда философ собирается поесть, он берет две палочки из корзины. Устраняет ли это возможность взаимной блокировки? Можно ли ввести риск взаимной блокировки простым сокращением количества палочек?

Новые библиотечные компоненты

В библиотеке Java SE `java.util.concurrent library` появилось несколько новых классов, предназначенных для решения проблем параллельного выполнения. Научившись пользоваться ими, вы сможете строить более простые и мощные параллельные программы.

В этом разделе приведена представительная подборка примеров различных компонентов, хотя некоторые компоненты (те, которые реже используются в повседневной работе) здесь не рассматриваются.

Поскольку эти компоненты предназначены для решения разных задач, классифицировать их довольно трудно, поэтому я постараюсь начать с простых примеров и постепенно переходить к примерам возрастающей сложности.

CountDownLatch

Класс предназначен для синхронизации одной или нескольких задач, ожидающих завершения набора операций, выполняемых другими задачами.

Объект `CountDownLatch` получает исходное значение счетчика, и любая задача, вызывающая `await()` для этого объекта, блокируется до тех пор, пока счетчик не уменьшится до нуля. Другие задачи могут вызывать `countDown()` для объекта, чтобы уменьшить счетчик, — предположительно до момента, когда задача завершит свою работу. Объект `CountDownLatch` спроектирован для «одноразового» использования, то есть его счетчик невозможно сбросить. Если вам нужна версия с возможностью сброса, используйте `CyclicBarrier`.

Задачи, вызывающие `countDown()`, не блокируются при вызове. Только вызов `await()` блокируется до того момента, когда счетчик уменьшится до нуля.

Типичный пример — разбиение задачи на n независимо решаемых подзадач, для которых создается объект `CountDownLatch` со значением n . При завершении каждая подзадача вызывает `countDown()` для объекта-счетчика. Подзадачи, ожидающие решения задачи, вызывают `await()` для `CountDownLatch`, чтобы приостановить свое выполнение до его завершения.

Следующий шаблон кода демонстрирует этот прием:

```
//: concurrency/CountDownLatchDemo.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Выполнение части общей задачи:
class TaskPortion implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private static Random rand = new Random(47);
    private final CountDownLatch latch;
    TaskPortion(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            doWork();
            latch.countDown();
        } catch (InterruptedException ex) {
            // Допустимый способ выхода
        }
    }
    public void doWork() throws InterruptedException {
        TimeUnit.MILLISECONDS.sleep(rand.nextInt(2000));
        print(this + "completed");
    }
    public String toString() {
        return String.format("%1$-3d ", id);
    }
}
```

```
// Ожидание по CountdownLatch:
class WaitingTask implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private final CountdownLatch latch;
    WaitingTask(CountdownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            latch.await();
            print("Latch barrier passed for " + this);
        } catch (InterruptedException ex) {
            print(this + " interrupted");
        }
    }
    public String toString() {
        return String.format("WaitingTask %1$-3d ", id);
    }
}

public class CountdownLatchDemo {
    static final int SIZE = 100;
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        // Все должны совместно использовать один объект CountdownLatch:
        CountdownLatch latch = new CountdownLatch(SIZE);
        for(int i = 0; i < 10; i++)
            exec.execute(new WaitingTask(latch));
        for(int i = 0; i < SIZE; i++)
            exec.execute(new TaskPortion(latch));
        print("Launched all tasks");
        exec.shutdown(); // Выход при завершении всех задач
    }
} /* (Execute to see output) *///:~
```

Объект `TaskPortion` ожидает в течение случайного времени, имитируя завершение части задачи, а объект `WaitingTask` моделирует часть системы, которая должна дождаться завершения исходной части задачи. Все задачи работают с одним объектом `CountdownLatch`, который определяется в `main()`.

- 32. (7)** Используйте `CountdownLatch` для решения задачи корреляции результатов отдельных входов в `OrnamentalGarden.java`. Удалите лишний код из новой версии примера.

Потоковая безопасность библиотеки

Класс `TaskPortion` содержит статический объект `Random`, а это означает, что несколько задач могут вызвать `Random.nextInt()` одновременно. Безопасен ли такой подход?

В данном случае возможные проблемы решаются передачей `TaskPortion` собственного объекта `Random`, то есть удалением спецификатора `static`. Но для методов стандартной библиотеки Java в целом остается тот же вопрос: какие методы являются потоково-безопасными, а какие нет?

К сожалению, документация JDK на этот счет хранит молчание. Как выясняется, вызов `Random.nextInt()` безопасен по отношению к потокам, но, к сожалению, безопасность

приходится выяснять для каждого конкретного случая посредством веб-поиска или просмотра библиотечного кода Java. Вряд ли это можно назвать хорошим решением для языка программирования, который (по крайней мере теоретически) создавался для поддержки параллельного программирования.

CyclicBarrier

Класс `CyclicBarrier` используется в тех ситуациях, когда вы хотите создать группу параллельно работающих задач, а затем дождаться их завершения, прежде чем переходить к следующему шагу (отдаленное подобие `join()`). Все задачи останавливаются у «барьера», чтобы затем дружно двинуться вперед. Класс `CyclicBarrier` очень похож на `CountDownLatch`, но класс `CountDownLatch` предназначен для «одноразового» использования, а объекты `CyclicBarrier` могут использоваться снова и снова.

В начале своего знакомства с компьютерами я увлекался компьютерным моделированием, и параллельное выполнение было ключевым фактором, который делал их возможным. Самая первая программа, написанная мной¹, моделировала лошадиные скачки, была написана на BASIC и называлась (из-за ограничений на имена файлов) `HOSRAC.BAS`. Ниже приведена объектно-ориентированная, потоковая версия этой программы с использованием `CyclicBarrier`.

```
//: concurrency/HorseRace.java
// Использование CyclicBarrier.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Horse implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private int strides = 0;
    private static Random rand = new Random(47);
    private static CyclicBarrier barrier;
    public Horse(CyclicBarrier b) { barrier = b; }
    public synchronized int getStrides() { return strides; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                synchronized(this) {
                    strides += rand.nextInt(3); // Выдает 0, 1 или 2
                }
                barrier.await();
            }
        } catch (InterruptedException e) {
            // Допустимый способ выхода
        } catch (BrokenBarrierException e) {
            // Об этом исключении нужно получить информацию
            throw new RuntimeException(e);
        }
    }
}
```

продолжение ➤

¹ Тогда я учился в средней школе; в нашем классе был установлен телетайп ASR-33 с 110-бодовым акустическим модемом для связи с HP-1000.

```

public String toString() { return "Horse " + id + " "; }
public String tracks() {
    StringBuilder s = new StringBuilder();
    for(int i = 0; i < getStrides(); i++)
        s.append("*");
    s.append(id);
    return s.toString();
}
}

public class HorseRace {
    static final int FINISH_LINE = 75;
    private List<Horse> horses = new ArrayList<Horse>();
    private ExecutorService exec =
        Executors.newCachedThreadPool();
    private CyclicBarrier barrier;
    public HorseRace(int nHorses, final int pause) {
        barrier = new CyclicBarrier(nHorses, new Runnable() {
            public void run() {
                StringBuilder s = new StringBuilder();
                for(int i = 0; i < FINISH_LINE; i++)
                    s.append("="); // Препятствие на треке
                print(s);
                for(Horse horse : horses)
                    print(horse.tracks());
                for(Horse horse : horses)
                    if(horse.getStrides() >= FINISH_LINE) {
                        print(horse + "won!");
                        exec.shutdownNow();
                        return;
                    }
            }
            try {
                TimeUnit.MILLISECONDS.sleep(pause);
            } catch(InterruptedException e) {
                print("barrier-action sleep interrupted");
            }
        }
    });
    for(int i = 0; i < nHorses; i++) {
        Horse horse = new Horse(barrier);
        horses.add(horse);
        exec.execute(horse);
    }
}

public static void main(String[] args) {
    int nHorses = 7;
    int pause = 200;
    if(args.length > 0) { // Необязательный аргумент
        int n = new Integer(args[0]);
        nHorses = n > 0 ? n : nHorses;
    }
    if(args.length > 1) { // Необязательный аргумент
        int p = new Integer(args[1]);
        pause = p > -1 ? p : pause;
    }
    new HorseRace(nHorses, pause);
}
} /* (Execute to see output) *///:~

```

`CyclicBarrier` можно назначить «барьерное действие» — объект `Runnable`, который автоматически выполняется при обнулении счетчика; это еще одно различие между `CyclicBarrier` и `CountdownLatch`. В данном примере барьерное действие создается как анонимный класс, который передается конструктору `CyclicBarrier`.

Я пытался заставить каждую «лошадь» выводить информацию о себе, но порядок вывода зависел от менеджера задач. `CyclicBarrier` позволяет каждой «лошади» сделать все, что ей необходимо для перемещения вперед, после чего «лошадь» должна ожидать у барьера, пока не «подтянутся» все остальные. После этого `CyclicBarrier` автоматически вызывает свое барьерное действие, чтобы по порядку вывести информацию обо всех «лошадях».

Когда все задачи пересекают барьер, он автоматически готов к следующему кругу.

Чтобы воссоздать эффект очень простой анимации, уменьшите размер консольного окна, чтобы в нем были видны только «лошади».

DelayQueue

Неограниченная очередь `BlockingQueue` объектов, реализующих интерфейс `Delayed`. Объект может быть извлечен из очереди только по истечении назначенной задержки. Очередь сортируется таким образом, что в ее начале размещается объект, с момента истечения задержки которого прошло наибольшее время. Если ни у одного элемента задержка не прошла, начальный элемент отсутствует, и вызов `poll()` вернет `null` (по этой причине в очередь нельзя помещать элементы `null`).

В следующем примере объекты `Delayed` сами являются задачами, а `DelayedTaskConsumer` извлекает из очереди самую «неотложную» задачу (с наибольшим сроком истечения) и запускает ее. Таким образом, `DelayQueue` представляет собой разновидность приоритетной очереди.

```
//: concurrency/DelayQueueDemo.java
import java.util.concurrent.*;
import java.util.*;
import static java.util.concurrent.TimeUnit.*;
import static net.mindview.util.Print.*;

class DelayedTask implements Runnable, Delayed {
    private static int counter = 0;
    private final int id = counter++;
    private final int delta;
    private final long trigger;
    protected static List<DelayedTask> sequence =
        new ArrayList<DelayedTask>();
    public DelayedTask(int delayInMilliseconds) {
        delta = delayInMilliseconds;
        trigger = System.nanoTime() +
            NANOSECONDS.convert(delta, MILLISECONDS);
        sequence.add(this);
    }
    public long getDelay(TimeUnit unit) {
        return unit.convert(
            trigger - System.nanoTime(), NANOSECONDS);
    }
}
```

продолжение ➤

```

public int compareTo(Delayed arg) {
    DelayedTask that = (DelayedTask)arg;
    if(trigger < that.trigger) return -1;
    if(trigger > that.trigger) return 1;
    return 0;
}
public void run() { printlnb(this + " "); }
public String toString() {
    return String.format("[%1$-4d]", delta) +
        " Task " + id;
}
public String summary() {
    return "(" + id + ":" + delta + ")";
}
public static class EndSentinel extends DelayedTask {
    private ExecutorService exec;
    public EndSentinel(int delay, ExecutorService e) {
        super(delay);
        exec = e;
    }
    public void run() {
        for(DelayedTask pt : sequence) {
            printlnb(pt.summary() + " ");
        }
        print();
        print(this + " Calling shutdownNow()");
        exec.shutdownNow();
    }
}
}

class DelayedTaskConsumer implements Runnable {
    private DelayQueue<DelayedTask> q;
    public DelayedTaskConsumer(DelayQueue<DelayedTask> q) {
        this.q = q;
    }
    public void run() {
        try {
            while(!Thread.interrupted())
                q.take().run(); // Запуск задачи из текущего потока
        } catch (InterruptedException e) {
            // Допустимый способ выхода
        }
        print("Finished DelayedTaskConsumer");
    }
}

public class DelayQueueDemo {
    public static void main(String[] args) {
        Random rand = new Random(47);
        ExecutorService exec = Executors.newCachedThreadPool();
        DelayQueue<DelayedTask> queue =
            new DelayQueue<DelayedTask>();
        // Заполнение задачами со случайными задержками:
        for(int i = 0; i < 20; i++)
            queue.put(new DelayedTask(rand.nextInt(5000)));
        // Назначение точки останова
        queue.add(new DelayedTask.EndSentinel(5000, exec));
        exec.execute(new DelayedTaskConsumer(queue));
    }
}

```

```

}
} /* Output:
[128 ] Task 11 [200 ] Task 7 [429 ] Task 5 [520 ] Task 18
[555 ] Task 1 [961 ] Task 4 [998 ] Task 16 [1207] Task 9
[1693] Task 2 [1809] Task 14 [1861] Task 3 [2278] Task 15
[3288] Task 10 [3551] Task 12 [4258] Task 0 [4258] Task 19
[4522] Task 8 [4589] Task 13 [4861] Task 17 [4868] Task 6
(0:4258) (1:555) (2:1693) (3:1861) (4:961) (5:429) (6:4868)
(7:200) (8:4522) (9:1207) (10:3288) (11:128) (12:3551)
(13:4589) (14:1809) (15:2278) (16:998) (17:4861) (18:520)
(19:4258) (20:5000)
[5000] Task 20 Calling shutdownNow()
Finished DelayedTaskConsumer
*///:~

```

`DelayedTask` содержит контейнер `List<DelayedTask>` с именем `sequence`, который сохраняет порядок создания задач, чтобы мы могли убедиться, что сортировка действительно выполняется.

Интерфейс `Delayed` содержит единственный метод `getDelay()`, который сообщает, сколько времени осталось до истечения времени задержки или сколько времени прошло с ее истечения. Этот метод заставляет нас использовать класс `TimeUnit`, потому что аргумент относится к этому типу. Впрочем, этот класс очень удобен, потому что он позволяет легко выполнять преобразования единиц без лишних вычислений. Например, значение `delta` хранится в миллисекундах, но метод `Java SE5 System.nanoTime()` возвращает время в наносекундах. Чтобы преобразовать значение `delta`, достаточно указать, в каких единицах оно задано и в каких единицах вы бы хотели его получить:

```
NANOSECONDS.convert(delta, MILLISECONDS);
```

В `getDelay()` предпочтительные единицы передаются в аргументе `unit`. Мы можем использовать их для преобразования времени к единицам, запрашиваемым вызывающей стороной, даже не зная, что это за единицы (это простой пример паттерна «Стратегия»: часть алгоритма передается в аргументе).

Для выполнения сортировки интерфейс `Delayed` также наследует интерфейс `Comparable`, поэтому нужно реализовать метод `compareTo()` для получения разумной сортировки. Методы `toString()` и `summary()` обеспечивают форматирование вывода, а вложенный класс `EndSentinel` предоставляет возможность завершить работу, для чего объект `EndSentinel` помещается в последний элемент очереди.

Поскольку объект `DelayedTaskConsumer` сам является задачей, он имеет собственный объект `Thread`, который может использоваться для запуска каждой задачи, извлеченной из очереди. Поскольку задачи выполняются в порядке приоритетной очереди, в этом примере нет необходимости порождать отдельные потоки для выполнения `DelayedTask`.

Из выходных данных видно, что порядок создания задач не влияет на порядок выполнения — задачи выполняются в порядке задержки, как и ожидалось.

PriorityBlockingQueue

По сути это приоритетная очередь с блокирующими операциями выборки. В следующем примере объекты приоритетной очереди представляют собой задачи, которые

извлекаются из очереди в порядке приоритетов. Для установления этого порядка `PrioritizedTask` передается числовой приоритет:

```

//: concurrency/PriorityBlockingQueueDemo.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class PrioritizedTask implements
Runnable, Comparable<PrioritizedTask> {
    private Random rand = new Random(47);
    private static int counter = 0;
    private final int id = counter++;
    private final int priority;
    protected static List<PrioritizedTask> sequence =
        new ArrayList<PrioritizedTask>();
    public PrioritizedTask(int priority) {
        this.priority = priority;
        sequence.add(this);
    }
    public int compareTo(PrioritizedTask arg) {
        return priority < arg.priority ? 1 :
            (priority > arg.priority ? -1 : 0);
    }
    public void run() {
        try {
            TimeUnit.MILLISECONDS.sleep(rand.nextInt(250));
        } catch (InterruptedException e) {
            // Допустимый способ выхода
        }
        print(this);
    }
    public String toString() {
        return String.format("[%1$-3d]", priority) +
            " Task " + id;
    }
    public String summary() {
        return "(" + id + ":" + priority + ")";
    }
    public static class EndSentinel extends PrioritizedTask {
        private ExecutorService exec;
        public EndSentinel(ExecutorService e) {
            super(-1); // Самый низкий приоритет в программе
            exec = e;
        }
        public void run() {
            int count = 0;
            for(PrioritizedTask pt : sequence) {
                printnb(pt.summary());
                if(++count % 5 == 0)
                    print();
            }
            print();
            print(this + " Calling shutdownNow()");
            exec.shutdownNow();
        }
    }
}

```



```

class PrioritizedTaskProducer implements Runnable {
    private Random rand = new Random(47);
    private Queue<Runnable> queue;
    private ExecutorService exec;
    public PrioritizedTaskProducer(
        Queue<Runnable> q, ExecutorService e) {
        queue = q;
        exec = e; // Используется для EndSentinel
    }
    public void run() {
        // Неограниченная очередь; блокировка никогда не происходит.
        // Заполняем случайными приоритетами:
        for(int i = 0; i < 20; i++) {
            queue.add(new PrioritizedTask(rand.nextInt(10)));
            Thread.yield();
        }
        // Добавляем высокоприоритетные задания:
        try {
            for(int i = 0; i < 10; i++) {
                TimeUnit.MILLISECONDS.sleep(250);
                queue.add(new PrioritizedTask(10));
            }
            // Добавляем задания начиная с низких приоритетов:
            for(int i = 0; i < 10; i++)
                queue.add(new PrioritizedTask(i));
            // Стоп, останавливающий все задачи:
            queue.add(new PrioritizedTask.EndSentinel(exec));
        } catch(InterruptedException e) {
            // Допустимый способ выхода
        }
        print("Finished PrioritizedTaskProducer");
    }
}

```

```

class PrioritizedTaskConsumer implements Runnable {
    private PriorityBlockingQueue<Runnable> q;
    public PrioritizedTaskConsumer(
        PriorityBlockingQueue<Runnable> q) {
        this.q = q;
    }
    public void run() {
        try {
            while(!Thread.interrupted())
                // Для запуска задачи используется текущий поток:
                q.take().run();
        } catch(InterruptedException e) {
            // Допустимый способ выхода
        }
        print("Finished PrioritizedTaskConsumer");
    }
}

```

```

public class PriorityBlockingQueueDemo {
    public static void main(String[] args) throws Exception {
        Random rand = new Random(47);
        ExecutorService exec = Executors.newCachedThreadPool();
        PriorityBlockingQueue<Runnable> queue =
            new PriorityBlockingQueue<Runnable>();
    }
}

```

```

        exec.execute(new PrioritizedTaskProducer(queue, exec));
        exec.execute(new PrioritizedTaskConsumer(queue));
    }
} /* (Execute to see output) *///:~

```

Как и в предыдущем примере, последовательность создания объектов `PrioritizedTask` запоминается в последовательности `List` для сравнения с фактическим порядком выполнения. Метод `run()` ожидает непродолжительное время, после чего выводит информацию об объекте, а `EndSentinel` предоставляет ту же функциональность, что и прежде, с гарантией того, что это последний объект в очереди.

`PrioritizedTaskProducer` и `PrioritizedTaskConsumer` связываются друг с другом через `PriorityBlockingQueue`. Поскольку блокирующая реализация очереди обеспечивает всю необходимую синхронизацию, явная синхронизация не нужна — при чтении из очереди вам не обязательно думать, есть ли в ней элементы, потому что при отсутствии элементов очередь просто блокирует читающую сторону.

Управление оранжереей на базе `ScheduledExecutor`

В главе 10 был представлен пример управляющей системы для оранжереи. Эта система включала/выключала различные компоненты и производила прочие настройки. Процесс управления можно рассматривать как параллельную задачу: каждое событие в оранжерее представлено задачей, которая запускается в заранее определенное время. Класс `ScheduledThreadPoolExecutor` предоставляет только сервис, необходимый для решения задачи. Используя либо `schedule()` (для однократного запуска задачи), либо `scheduleAtFixedRate()` (для повторения задачи с регулярными интервалами), вы настраиваете объекты `Runnable` для выполнения в некоторый момент будущего. Сравните следующее решение с подходом, использованным в главе 10, и обратите внимание, насколько упрощается решение при использовании готовых инструментов — таких, как `ScheduledThreadPoolExecutor`.

```

//: concurrency/GreenhouseScheduler.java
// Переработка примера innerclasses/GreenhouseController.java
// с использованием ScheduledThreadPoolExecutor.
// {Args: 5000}
import java.util.concurrent.*;
import java.util.*;

public class GreenhouseScheduler {
    private volatile boolean light = false;
    private volatile boolean water = false;
    private String thermostat = "Day";
    public synchronized String getThermostat() {
        return thermostat;
    }
    public synchronized void setThermostat(String value) {
        thermostat = value;
    }
    ScheduledThreadPoolExecutor scheduler =
        new ScheduledThreadPoolExecutor(10);
    public void schedule(Runnable event, long delay) {
        scheduler.schedule(event, delay, TimeUnit.MILLISECONDS);
    }
}

```

```
public void
repeat(Runnable event, long initialDelay, long period) {
    scheduler.scheduleAtFixedRate(
        event, initialDelay, period, TimeUnit.MILLISECONDS);
}
class LightOn implements Runnable {
    public void run() {
        // Поместите сюда код управления оборудованием,
        // выполняющий непосредственное включение света.
        System.out.println("Turning on lights");
        light = true;
    }
}
class LightOff implements Runnable {
    public void run() {
        // Поместите сюда код управления оборудованием,
        // выполняющий выключение света.
        System.out.println("Turning off lights");
        light = false;
    }
}
class WaterOn implements Runnable {
    public void run() {
        // Здесь размещается код управления оборудованием.
        System.out.println("Turning greenhouse water on");
        water = true;
    }
}
class WaterOff implements Runnable {
    public void run() {
        // Здесь размещается код управления оборудованием.
        System.out.println("Turning greenhouse water off");
        water = false;
    }
}
class ThermostatNight implements Runnable {
    public void run() {
        // Здесь размещается код управления оборудованием.
        System.out.println("Thermostat to night setting");
        setThermostat("Night");
    }
}
class ThermostatDay implements Runnable {
    public void run() {
        // Здесь размещается код управления оборудованием.
        System.out.println("Thermostat to day setting");
        setThermostat("Day");
    }
}
class Bell implements Runnable {
    public void run() { System.out.println("Bing!"); }
}
class Terminate implements Runnable {
    public void run() {
        System.out.println("Terminating");
        scheduler.shutdownNow();
        // Для этого задания необходимо запустить отдельную задачу,
        // так как планировщик завершен:
    }
}
```

```

    new Thread() {
        public void run() {
            for(DataPoint d : data)
                System.out.println(d);
        }
    }.start();
}
}
// Новая возможность: сбор данных
static class DataPoint {
    final Calendar time;
    final float temperature;
    final float humidity;
    public DataPoint(Calendar d, float temp, float hum) {
        time = d;
        temperature = temp;
        humidity = hum;
    }
    public String toString() {
        return time.getTime() +
            String.format(
                " temperature: %1$.1f humidity: %2$.2f",
                temperature, humidity);
    }
}
private Calendar lastTime = Calendar.getInstance();
{ // Регулировка даты до получаса
    lastTime.set(Calendar.MINUTE, 30);
    lastTime.set(Calendar.SECOND, 00);
}
private float lastTemp = 65.0f;
private int tempDirection = +1;
private float lastHumidity = 50.0f;
private int humidityDirection = +1;
private Random rand = new Random(47);
List<DataPoint> data = Collections.synchronizedList(
    new ArrayList<DataPoint>());
class CollectData implements Runnable {
    public void run() {
        System.out.println("Collecting data");
        synchronized(GreenhouseScheduler.this) {
            // Имитировать более длинный интервал:
            lastTime.set(Calendar.MINUTE,
                lastTime.get(Calendar.MINUTE) + 30);
            // Направление меняется в 1 из 5 случаев:
            if(rand.nextInt(5) == 4)
                tempDirection = -tempDirection;
            // Сохранение предыдущего значения:
            lastTemp = lastTemp +
                tempDirection * (1.0f + rand.nextFloat());
            if(rand.nextInt(5) == 4)
                humidityDirection = -humidityDirection;
            lastHumidity = lastHumidity +
                humidityDirection * rand.nextFloat();
            // Calendar необходимо клонировать, в противном случае все
            // объекты DataPoint будут содержать ссылки на то же
            // значение lastTime. Для простейших объектов - таких,
            // как Calendar, - достаточно вызова clone().

```

```

        data.add(new DataPoint((Calendar)lastTime.clone(),
                                lastTemp, lastHumidity));
    }
}
}
public static void main(String[] args) {
    GreenhouseScheduler gh = new GreenhouseScheduler();
    gh.schedule(gh.new Terminate(), 5000);
    // Former "Restart" class not necessary:
    gh.repeat(gh.new Bell(), 0, 1000);
    gh.repeat(gh.new ThermostatNight(), 0, 2000);
    gh.repeat(gh.new LightOn(), 0, 200);
    gh.repeat(gh.new LightOff(), 0, 400);
    gh.repeat(gh.new WaterOn(), 0, 600);
    gh.repeat(gh.new WaterOff(), 0, 800);
    gh.repeat(gh.new ThermostatDay(), 0, 1400);
    gh.repeat(gh.new CollectData(), 500, 500);
}
} /* (Execute to see output) *///:~

```

Эта версия изменяет структуру кода и добавляет новую возможность: сбор данных температуры и влажности. Объект `DataPoint` содержит и выводит один элемент данных, а планируемая задача `CollectData` генерирует моделируемые данные и добавляет их в `List<DataPoint>` в `Greenhouse` при каждом запуске.

Обратите внимание на ключевые слова `volatile` и `synchronized`; они предотвращают нежелательное взаимодействие задач. Все методы `List`, хранящие `DataPoint`, синхронизируются с использованием метода `synchronizedList()` из библиотеки `java.util.Collections` при создании `List`.

- 33. (7)** Измените пример `GreenhouseScheduler.java` так, чтобы в нем использовался класс `DelayQueue` вместо `ScheduledExecutor`.

Semaphore

Обычная блокировка (из `concurrent.locks` или встроенная блокировка `synchronized`) разрешает обращаться к ресурсу в любой момент времени только одной задаче. *Семафор со счетчиком* разрешает *n* задачам обращаться к ресурсу одновременно. Также можно представить, что семафор выдает «разрешения» на использование ресурсов, хотя на самом деле объекты разрешений не используются.

В качестве примера рассмотрим концепцию пула объектов, который управляет ограниченным набором объектов и позволяет захватить объект для использования, а затем вернуть обратно после завершения работы с ним. Такая функциональность может быть инкапсулирована в обобщенном классе.

```

//: concurrency/Pool.java
// Использование семафора с пулом для ограничения
// количества задач, которые могут использовать ресурс.
import java.util.concurrent.*;
import java.util.*;

public class Pool<T> {
    private int size;
    private List<T> items = new ArrayList<T>();

```

продолжение ➤

```

private volatile boolean[] checkedOut;
private Semaphore available;
public Pool(Class<T> classObject, int size) {
    this.size = size;
    checkedOut = new boolean[size];
    available = new Semaphore(size, true);
    // Заполнение пула объектами:
    for(int i = 0; i < size; ++i)
        try {
            // Предполагается конструктор по умолчанию:
            items.add(classObject.newInstance());
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
}
public T checkOut() throws InterruptedException {
    available.acquire();
    return getItem();
}
public void checkIn(T x) {
    if(releaseItem(x))
        available.release();
}
private synchronized T getItem() {
    for(int i = 0; i < size; ++i)
        if(!checkedOut[i]) {
            checkedOut[i] = true;
            return items.get(i);
        }
    return null; // Семафор не позволяет перейти сюда
}
private synchronized boolean releaseItem(T item) {
    int index = items.indexOf(item);
    if(index == -1) return false; // Нет в списке
    if(checkedOut[index]) {
        checkedOut[index] = false;
        return true;
    }
    return false; // Объект не был выдан
}
} ///:~

```

В этой упрощенной форме конструктор использует `newInstance()` для заполнения пула объектами. Когда пользователю потребуется новый объект, он вызывает `checkOut()`, а когда работа с объектом будет завершена, он передается `checkIn()`.

В массиве `checkedOut` отслеживаются выданные объекты, для управления которыми используются методы `getItem()` и `releaseItem()`. В свою очередь, эти методы защищаются семафором `available`, так что в `checkOut()` `available` блокирует вызов при отсутствии семафорных разрешений (это означает, что в пуле не осталось объектов). В `checkIn()`, если возвращаемый объект действителен, разрешение возвращается семафору.

Для создания примера мы воспользуемся `Fat` — типом объекта, создание которого сопряжено с большими затратами, а конструктор выполняется достаточно долго:

```

//: concurrency/Fat.java
// Объект, создание которого обходится дорого.

```

```

public class Fat {
    private volatile double d; // Предотвращение оптимизации
    private static int counter = 0;
    private final int id = counter++;
    public Fat() {
        // Затратная, прерываемая операция:
        for(int i = 1; i < 10000; i++) {
            d += (Math.PI + Math.E) / (double)i;
        }
    }
    public void operation() { System.out.println(this); }
    public String toString() { return "Fat id: " + id; }
} ///:~

```

Чтобы снизить затраты на создание, мы создадим пул таких объектов. Для тестирования класса `Pool` будет создана задача, которая получает объекты `Fat`, какое-то время удерживает их, а потом возвращает обратно:

```

//: concurrency/SemaphoreDemo.java
// Тестирование класса Pool
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Задача для получения ресурса из пула:
class CheckoutTask<T> implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private Pool<T> pool;
    public CheckoutTask(Pool<T> pool) {
        this.pool = pool;
    }
    public void run() {
        try {
            T item = pool.checkOut();
            print(this + "checked out " + item);
            TimeUnit.SECONDS.sleep(1);
            print(this + "checking in " + item);
            pool.checkIn(item);
        } catch (InterruptedException e) {
            // Допустимый способ завершения
        }
    }
    public String toString() {
        return "CheckoutTask " + id + " ";
    }
}

public class SemaphoreDemo {
    final static int SIZE = 25;
    public static void main(String[] args) throws Exception {
        final Pool<Fat> pool =
            new Pool<Fat>(Fat.class, SIZE);
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < SIZE; i++)
            exec.execute(new CheckoutTask<Fat>(pool));
        print("All CheckoutTasks created");
        List<Fat> list = new ArrayList<Fat>();
    }
}

```

продолжение ➤

```

for(int i = 0; i < SIZE; i++) {
    Fat f = pool.checkOut();
    printlnb(i + ": main() thread checked out ");
    f.operation();
    list.add(f);
}
Future<?> blocked = exec.submit(new Runnable() {
    public void run() {
        try {
            // Семафор предотвращает дополнительную выдачу,
            // поэтому вызов блокируется:
            pool.checkOut();
        } catch(InterruptedException e) {
            print("checkOut() Interrupted");
        }
    }
});
TimeUnit.SECONDS.sleep(2);
blocked.cancel(true); // Break out of blocked call
print("Checking in objects in " + list);
for(Fat f : list)
    pool.checkIn(f);
for(Fat f : list)
    pool.checkIn(f); // Второй вызов checkIn игнорируется
exec.shutdown();
}
} /* (Execute to see output) *///:~

```

В методе `main()` создается объект `Pool` для объектов `Fat`, и группа задач `CheckoutTask` начинает работать с `Pool`. Затем поток `main()` начинает получать объекты `Fat`, не возвращая их в пул. После того как в пуле кончатся объекты, `Semaphore` запретит получение новых объектов. Метод `run()` объекта `blocked` блокируется, и через две секунды следует вызов `cancel()`. `Pool` игнорирует лишние возвраты объектов.

Этот пример полагается на то, что клиент `Pool` будет тщательно соблюдать правила и добровольно возвращать использованные объекты; самое простое решение... когда оно работает. Тем не менее если у вас нет полной уверенности, в книге «Thinking in Patterns» (www.MindView.net) исследуются дополнительные возможности управления объектами, выделенными из пулов.

Exchanger

`Exchanger` — барьер, который меняет местами объекты двух задач. Когда задача входит в барьер, она использует один объект, а при выходе его место занимает объект, ранее удерживавшийся другой задачей. Как правило, объекты `Exchanger` используются тогда, когда одна задача выполняет затратные операции создания объектов, а другая задача эти объекты потребляет; таким образом, создание новых объектов может происходить параллельно с их потреблением.

Чтобы использовать класс `Exchanger`, мы создадим задачи производителя и потребителя, которые посредством обобщений и генераторов будут работать с любыми видами объектов, и применим их к классу `Fat`. `ExchangerProducer` и `ExchangerConsumer` используют `List<T>` в качестве заменяемого объекта; каждая задача содержит объект `Exchanger`

для `List<T>`. При вызове `Exchanger.exchange()` задача блокируется до тех пор, пока задача-партнер не вызовет свой метод `exchange()`; когда оба метода `exchange()` будут завершены, объекты `List<T>` меняются местами:

```
//: concurrency/ExchangerDemo.java
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

class ExchangerProducer<T> implements Runnable {
    private Generator<T> generator;
    private Exchanger<List<T>> exchanger;
    private List<T> holder;
    ExchangerProducer(Exchanger<List<T>> exchg,
        Generator<T> gen, List<T> holder) {
        exchanger = exchg;
        generator = gen;
        this.holder = holder;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                for(int i = 0; i < ExchangerDemo.size; i++)
                    holder.add(generator.next());
                // Полный список меняется с пустым:
                holder = exchanger.exchange(holder);
            }
        } catch(InterruptedException e) {
            // Допустимый способ завершения
        }
    }
}

class ExchangerConsumer<T> implements Runnable {
    private Exchanger<List<T>> exchanger;
    private List<T> holder;
    private volatile T value;
    ExchangerConsumer(Exchanger<List<T>> ex, List<T> holder){
        exchanger = ex;
        this.holder = holder;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                holder = exchanger.exchange(holder);
                for(T x : holder) {
                    value = x; // Выборка значения
                    holder.remove(x); // Возможно для CopyOnWriteArrayList
                }
            }
        } catch(InterruptedException e) {
            // Допустимый способ завершения
        }
        System.out.println("Final value: " + value);
    }
}
```

продолжение ➤

```

public class ExchangerDemo {
    static int size = 10;
    static int delay = 5; // Секунды
    public static void main(String[] args) throws Exception {
        if(args.length > 0)
            size = new Integer(args[0]);
        if(args.length > 1)
            delay = new Integer(args[1]);
        ExecutorService exec = Executors.newCachedThreadPool();
        Exchanger<List<Fat>> xc = new Exchanger<List<Fat>>();
        List<Fat>
            producerList = new CopyOnWriteArrayList<Fat>(),
            consumerList = new CopyOnWriteArrayList<Fat>();
        exec.execute(new ExchangerProducer<Fat>(xc,
            BasicGenerator.create(Fat.class), producerList));
        exec.execute(
            new ExchangerConsumer<Fat>(xc, consumerList));
        TimeUnit.SECONDS.sleep(delay);
        exec.shutdownNow();
    }
} /* Output: (Sample)
Final value: Fat id: 29999
*///:~

```

В `main()` создается один объект `Exchanger`, используемый обеими задачами, и создаются два контейнера `CopyOnWriteArrayList`, которые будут меняться местами. Эта конкретная разновидность `List` допускает вызов метода `remove()` во время перебора списка без выдачи исключения `ConcurrentModificationException`. Объект `ExchangerProducer` заполняет `List`, а затем меняет полный список с пустым, полученным от `ExchangerConsumer`. Из-за `Exchanger` заполнение одного списка может происходить одновременно с использованием другого списка.

- 34.** (1) Измените пример `ExchangerDemo.java` так, чтобы вместо `Fat` использовался ваш собственный класс.

Моделирование

Одна из самых интересных и творческих областей параллельного программирования — компьютерное моделирование. В параллельной модели каждый компонент может быть представлен отдельной задачей, что существенно упрощает программирование. Многие видеоигры и компьютерные анимации реализуются на базе моделей, а приведенные ранее примеры `HorseRace.java` и `GreenhouseScheduler.java` также могут рассматриваться как модели.

Модель кассира

Эта классическая модель может использоваться для представления любых ситуаций, в которых объекты появляются случайным образом и обслуживаются ограниченным количеством серверов за случайное время. Построенная модель позволяет определить идеальное количество серверов.

В следующем примере каждый клиент банка обслуживается за некоторое время, которое определяется случайным образом для каждого клиента. При этом заранее не

известно, сколько клиентов появится в каждый интервал; эта величина также определяется случайным образом.

```
//: concurrency/BankTellerSimulation.java
// Использование очередей и многопоточной модели.
// {Args: 5}
import java.util.concurrent.*;
import java.util.*;

// Объекты, доступные только для чтения,
// не требуют синхронизации:
class Customer {
    private final int serviceTime;
    public Customer(int tm) { serviceTime = tm; }
    public int getServiceTime() { return serviceTime; }
    public String toString() {
        return "[" + serviceTime + "]";
    }
}

// Очередь клиентов умеет выводить свое состояние:
class CustomerLine extends ArrayBlockingQueue<Customer> {
    public CustomerLine(int maxLineSize) {
        super(maxLineSize);
    }
    public String toString() {
        if(this.size() == 0)
            return "[Empty]";
        StringBuilder result = new StringBuilder();
        for(Customer customer : this)
            result.append(customer);
        return result.toString();
    }
}

// Случайное добавление клиентов в очередь:
class CustomerGenerator implements Runnable {
    private CustomerLine customers;
    private static Random rand = new Random(47);
    public CustomerGenerator(CustomerLine cq) {
        customers = cq;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                TimeUnit.MILLISECONDS.sleep(rand.nextInt(300));
                customers.put(new Customer(rand.nextInt(1000)));
            }
        } catch (InterruptedException e) {
            System.out.println("CustomerGenerator interrupted");
        }
        System.out.println("CustomerGenerator terminating");
    }
}

class Teller implements Runnable, Comparable<Teller> {
    private static int counter = 0;
```

продолжение ⌘

```

private final int id = counter++;
// Клиенты, обслуженные за смену:
private int customersServed = 0;
private CustomerLine customers;
private boolean servingCustomerLine = true;
public Teller(CustomerLine cq) { customers = cq; }
public void run() {
    try {
        while(!Thread.interrupted()) {
            Customer customer = customers.take();
            TimeUnit.MILLISECONDS.sleep(
                customer.getServiceTime());
            synchronized(this) {
                customersServed++;
                while(!servingCustomerLine)
                    wait();
            }
        }
    } catch(InterruptedException e) {
        System.out.println(this + "interrupted");
    }
    System.out.println(this + "terminating");
}
public synchronized void doSomethingElse() {
    customersServed = 0;
    servingCustomerLine = false;
}
public synchronized void serveCustomerLine() {
    assert !servingCustomerLine:"already serving: " + this;
    servingCustomerLine = true;
    notifyAll();
}
public String toString() { return "Teller " + id + " "; }
public String shortString() { return "T" + id; }
// Используется приоритетной очередью:
public synchronized int compareTo(Teller other) {
    return customersServed < other.customersServed ? -1 :
        (customersServed == other.customersServed ? 0 : 1);
}
}

class TellerManager implements Runnable {
    private ExecutorService exec;
    private CustomerLine customers;
    private PriorityQueue<Teller> workingTellers =
        new PriorityQueue<Teller>();
    private Queue<Teller> tellersDoingOtherThings =
        new LinkedList<Teller>();
    private int adjustmentPeriod;
    private static Random rand = new Random(47);
    public TellerManager(ExecutorService e,
        CustomerLine customers, int adjustmentPeriod) {
        exec = e;
        this.customers = customers;
        this.adjustmentPeriod = adjustmentPeriod;
        // Начать с одного кассира:
        Teller teller = new Teller(customers);
        exec.execute(teller);
    }
}

```

```
        workingTellers.add(teller);
    }
    public void adjustTellerNumber() {
        // Система управления - изменяя числа, можно выявить
        // проблемы стабильности в управляющем механизме.
        // Если очередь слишком длинная, добавить кассира:
        if(customers.size() / workingTellers.size() > 2) {
            // Если кассиры отдыхают или занимаются другим делом,
            // вернуть одного:
            if(tellersDoingOtherThings.size() > 0) {
                Teller teller = tellersDoingOtherThings.remove();
                teller.serveCustomerLine();
                workingTellers.offer(teller);
                return;
            }
            // Иначе создать (принять на работу) нового кассира
            Teller teller = new Teller(customers);
            exec.execute(teller);
            workingTellers.add(teller);
            return;
        }
        // Если очередь достаточно короткая, убрать кассира:
        if(workingTellers.size() > 1 &&
            customers.size() / workingTellers.size() < 2)
            reassignOneTeller();
        // Если очереди нет, достаточно одного кассира:
        if(customers.size() == 0)
            while(workingTellers.size() > 1)
                reassignOneTeller();
    }
    // Отправить кассира на другую работу или отдых:
    private void reassignOneTeller() {
        Teller teller = workingTellers.poll();
        teller.doSomethingElse();
        tellersDoingOtherThings.offer(teller);
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                TimeUnit.MILLISECONDS.sleep(adjustmentPeriod);
                adjustTellerNumber();
                System.out.print(customers + " { ");
                for(Teller teller : workingTellers)
                    System.out.print(teller.shortString() + " ");
                System.out.println("}");
            }
        } catch(InterruptedException e) {
            System.out.println(this + "interrupted");
        }
        System.out.println(this + "terminating");
    }
    public String toString() { return "TellerManager "; }
}

public class BankTellerSimulation {
    static final int MAX_LINE_SIZE = 50;
    static final int ADJUSTMENT_PERIOD = 1000;
    public static void main(String[] args) throws Exception {
```

```

ExecutorService exec = Executors.newCachedThreadPool();
// Если очередь слишком длинная, клиенты уходят:
CustomerLine customers =
    new CustomerLine(MAX_LINE_SIZE);
exec.execute(new CustomerGenerator(customers));
// Менеджер добавляет и удаляет кассиров по необходимости:
exec.execute(new TellerManager(
    exec, customers, ADJUSTMENT_PERIOD));
if(args.length > 0) // Необязательный аргумент
    TimeUnit.SECONDS.sleep(new Integer(args[0]));
else {
    System.out.println("Press 'Enter' to quit");
    System.in.read();
}
exec.shutdownNow();
}
} /* Output: (Sample)
[429][200][207] { T0 T1 }
[861][258][140][322] { T0 T1 }
[575][342][804][826][896][984] { T0 T1 T2 }
[984][810][141][12][689][992][976][368][395][354] { T0 T1
T2 T3 }
Teller 2 interrupted
Teller 2 terminating
Teller 1 interrupted
Teller 1 terminating
TellerManager interrupted
TellerManager terminating
Teller 3 interrupted
Teller 3 terminating
Teller 0 interrupted
Teller 0 terminating
CustomerGenerator interrupted
CustomerGenerator terminating
*///:~

```

Объекты `Customer` очень просты — они содержат всего одно поле `final int`. Так как эти объекты никогда не изменяются, они доступны только для чтения, а следовательно, не требуют синхронизации или применения `volatile`. Кроме того, каждая задача `Teller` удаляет из входной очереди по одному объекту `Customer` и работает с ним до завершения, так что к `Customer` в любой момент времени будет обращаться только одна задача.

Объект `CustomerLine` представляет одну очередь, в которой клиенты ожидают обслуживания объектом `Teller`. Он представляет собой контейнер `ArrayBlockingQueue` с методом `toString()`, который выводит результат в нужном формате.

Объект `CustomerGenerator` присоединяется к `CustomerLine` и помещает объекты `Customer` в очередь со случайными интервалами.

`Teller` извлекает объекты `Customer` из `CustomerLine` и обрабатывает их по одному, сохраняя количество объектов `Customer`, обслуженных за эту конкретную смену. При недостатке клиентов его можно переключить на другую работу методом `doSomethingElse()`, а при избытке вызывается метод `serveCustomerLine()`. Чтобы выбрать следующего кассира, возвращаемого на работу, метод `compareTo()` проверяет количество обслуженных клиентов, чтобы объект `PriorityQueue` мог автоматически вернуть кассира с наименьшей наработкой.

Объект `TellerManager` занимает центральное место в управлении. Он наблюдает за всеми кассирами и событиями, происходящими с клиентами. Одна из интересных особенностей нашей модели заключается в том, что она пытается подобрать оптимальное количество кассиров для конкретного потока покупателей. В этом можно убедиться в методе `adjustTellerNumber()`, предназначенном для стабильного добавления или удаления кассиров. Во всех системах управления существуют проблемы со стабильностью; если они реагируют на изменения слишком быстро, то становятся нестабильными, а если слишком медленно — система переходит в одно из крайних состояний.

- 35.** (8) Измените пример `BankTellerSimulation.java`, чтобы он моделировал веб-клиентов, обращающихся с запросами к фиксированному количеству серверов. Программа должна определять нагрузку, с которой может справиться данная группа серверов.

Моделирование ресторана

Эта модель расширяет простой пример `Restaurant.java`, приведенный ранее в этой главе. Он добавляет новые компоненты модели — такие, как заказы (`Order`) и тарелки (`Plate`), — и повторно использует классы меню из главы 19.

Также в ней продемонстрировано использование класса `Java SE5 SynchronousQueue` — блокирующей очереди, не имеющей внутренней емкости, так что каждый вызов `put()` должен ожидать `take()`, и наоборот. Все выглядит так, словно вы передаете кому-то объект «из рук в руки» — передача работает только в том случае, если получатель протянул руку и готов к получению объекта. В нашем примере `SynchronousQueue` представляет столовый прибор перед посетителем и подчеркивает то, что блюда подаются по одному.

Остальные классы и функциональные аспекты этого примера либо повторяют структуру `Restaurant.java`, либо представляют операции реально существующего ресторана:

```
//: concurrency/restaurant2/RestaurantWithQueues.java
// {Args: 5}
package concurrency.restaurant2;
import enumerated.menu.*;
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Заказ передается официанту,
// который передает его повару:
class Order { // (Объект передачи данных)
    private static int counter = 0;
    private final int id = counter++;
    private final Customer customer;
    private final WaitPerson waitPerson;
    private final Food food;
    public Order(Customer cust, WaitPerson wp, Food f) {
        customer = cust;
        waitPerson = wp;
        food = f;
    }
    public Food item() { return food; }
    public Customer getCustomer() { return customer; }
    public WaitPerson getWaitPerson() { return waitPerson; }
```

продолжение ➤

```

    public String toString() {
        return "Order: " + id + " item: " + food +
            " for: " + customer +
            " served by: " + waitPerson;
    }
}

// Повар возвращает готовое блюдо:
class Plate {
    private final Order order;
    private final Food food;
    public Plate(Order ord, Food f) {
        order = ord;
        food = f;
    }
    public Order getOrder() { return order; }
    public Food getFood() { return food; }
    public String toString() { return food.toString(); }
}

class Customer implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private final WaitPerson waitPerson;
    // Посетитель может получать только одно блюдо за раз:
    private SynchronousQueue<Plate> placeSetting =
        new SynchronousQueue<Plate>();
    public Customer(WaitPerson w) { waitPerson = w; }
    public void
    deliver(Plate p) throws InterruptedException {
        // Блокируется только в том случае, если
        // посетитель еще не закончил с предыдущим блюдом:
        placeSetting.put(p);
    }
    public void run() {
        for(Course course : Course.values()) {
            Food food = course.randomSelection();
            try {
                waitPerson.placeOrder(this, food);
                // Блокируется до получения блюда:
                print(this + "eating " + placeSetting.take());
            } catch(InterruptedException e) {
                print(this + "waiting for " +
                    course + " interrupted");
                break;
            }
        }
        print(this + "finished meal, leaving");
    }
    public String toString() {
        return "Customer " + id + " ";
    }
}

class WaitPerson implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private final Restaurant restaurant;

```



```

BlockingQueue<Plate> filledOrders =
    new LinkedBlockingQueue<Plate>();
public WaitPerson(Restaurant rest) { restaurant = rest; }
public void placeOrder(Customer cust, Food food) {
    try {
        // Вообще говоря, этот вызов не должен блокироваться,
        // потому что LinkedBlockingQueue не имеет
        // ограничений по размеру:
        restaurant.orders.put(new Order(cust, this, food));
    } catch(InterruptedException e) {
        print(this + " placeOrder interrupted");
    }
}
public void run() {
    try {
        while(!Thread.interrupted()) {
            // Блокируется, пока блюдо не будет готово
            Plate plate = filledOrders.take();
            print(this + "received " + plate +
                " delivering to " +
                plate.getOrder().getCustomer());
            plate.getOrder().getCustomer().deliver(plate);
        }
    } catch(InterruptedException e) {
        print(this + " interrupted");
    }
    print(this + " off duty");
}
public String toString() {
    return "WaitPerson " + id + " ";
}
}

class Chef implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private final Restaurant restaurant;
    private static Random rand = new Random(47);
    public Chef(Restaurant rest) { restaurant = rest; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Блокируется до появления заказа:
                Order order = restaurant.orders.take();
                Food requestedItem = order.item();
                // Время на подготовку заказа:
                TimeUnit.MILLISECONDS.sleep(rand.nextInt(500));
                Plate plate = new Plate(order, requestedItem);
                order.getWaitPerson().filledOrders.put(plate);
            }
        } catch(InterruptedException e) {
            print(this + " interrupted");
        }
        print(this + " off duty");
    }
    public String toString() { return "Chef " + id + " "; }
}

class Restaurant implements Runnable {

```

```

private List<WaitPerson> waitPersons =
    new ArrayList<WaitPerson>();
private List<Chef> chefs = new ArrayList<Chef>();
private ExecutorService exec;
private static Random rand = new Random(47);
BlockingQueue<Order>
    orders = new LinkedBlockingQueue<Order>();
public Restaurant(ExecutorService e, int nWaitPersons,
    int nChefs) {
    exec = e;
    for(int i = 0; i < nWaitPersons; i++) {
        WaitPerson waitPerson = new WaitPerson(this);
        waitPersons.add(waitPerson);
        exec.execute(waitPerson);
    }
    for(int i = 0; i < nChefs; i++) {
        Chef chef = new Chef(this);
        chefs.add(chef);
        exec.execute(chef);
    }
}
public void run() {
    try {
        while(!Thread.interrupted()) {
            // Приходит новый посетитель;
            // ему назначается официант:
            WaitPerson wp = waitPersons.get(
                rand.nextInt(waitPersons.size()));
            Customer c = new Customer(wp);
            exec.execute(c);
            TimeUnit.MILLISECONDS.sleep(100);
        }
    } catch(InterruptedException e) {
        print("Restaurant interrupted");
    }
    print("Restaurant closing");
}
}

public class RestaurantWithQueues {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        Restaurant restaurant = new Restaurant(exec, 5, 2);
        exec.execute(restaurant);
        if(args.length > 0) // Необязательный аргумент
            TimeUnit.SECONDS.sleep(new Integer(args[0]));
        else {
            print("Press 'Enter' to quit");
            System.in.read();
        }
        exec.shutdownNow();
    }
}
/* Output: (Sample)
WaitPerson 0 received SPRING_ROLLS delivering to Customer 1
Customer 1 eating SPRING_ROLLS
WaitPerson 3 received SPRING_ROLLS delivering to Customer 0
Customer 0 eating SPRING_ROLLS
WaitPerson 0 received BURRITO delivering to Customer 1
Customer 1 eating BURRITO

```

```

WaitPerson 3 received SPRING_ROLLS delivering to Customer 2
Customer 2 eating SPRING_ROLLS
WaitPerson 1 received SOUP delivering to Customer 3
Customer 3 eating SOUP
WaitPerson 3 received VINDALOO delivering to Customer 0
Customer 0 eating VINDALOO
WaitPerson 0 received FRUIT delivering to Customer 1
...
*///:~

```

В этом примере следует обратить внимание на управление сложностью с применением очередей для взаимодействия между задачами. Этот прием сильно упрощает параллельное программирование за счет инверсии управления: задачи не взаимодействуют друг с другом напрямую. Вместо этого они отправляют объекты друг другу через очереди. Задача-получатель обрабатывает объект, рассматривая его как сообщение (вместо того, чтобы возиться с сообщением, «наложенным» на объект). Если вы будете следовать этой схеме там, где это возможно, вероятность того, что построенная вами параллельная система будет надежной, существенно увеличится.

- 36.** (10) Измените пример `RestaurantWithQueues.java` так, чтобы на каждый стол создавался один объект бланка заказа `OrderTicket`. Замените `order` на `orderTicket` и добавьте класс `Table`, представляющий стол с несколькими посетителями.

Распределение работы

Следующая модель связывает воедино многие концепции этой главы. Рассмотрим гипотетическую роботизированную линию для сборки машин. Каждый объект `Car` создается за несколько этапов: сначала строится рама, затем устанавливается двигатель, приводной механизм и колеса.

```

//: concurrency/CarBuilder.java
// Сложный пример взаимодействия задач.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Car {
    private final int id;
    private boolean
        engine = false, driveTrain = false, wheels = false;
    public Car(int idn) { id = idn; }
    // Пустой объект Car:
    public Car() { id = -1; }
    public synchronized int getId() { return id; }
    public synchronized void addEngine() { engine = true; }
    public synchronized void addDriveTrain() {
        driveTrain = true;
    }
    public synchronized void addWheels() { wheels = true; }
    public synchronized String toString() {
        return "Car " + id + " [" + " engine: " + engine
            + " driveTrain: " + driveTrain
            + " wheels: " + wheels + " ]";
    }
}

class CarQueue extends LinkedBlockingQueue<Car> {}

```

продолжение ➤

```

class ChassisBuilder implements Runnable {
    private CarQueue carQueue;
    private int counter = 0;
    public ChassisBuilder(CarQueue cq) { carQueue = cq; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                TimeUnit.MILLISECONDS.sleep(500);
                // Создание рамы:
                Car c = new Car(counter++);
                print("ChassisBuilder created " + c);
                // Помещение в очередь
                carQueue.put(c);
            }
        } catch(InterruptedException e) {
            print("Interrupted: ChassisBuilder");
        }
        print("ChassisBuilder off");
    }
}

class Assembler implements Runnable {
    private CarQueue chassisQueue, finishingQueue;
    private Car car;
    private CyclicBarrier barrier = new CyclicBarrier(4);
    private RobotPool robotPool;
    public Assembler(CarQueue cq, CarQueue fq, RobotPool rp){
        chassisQueue = cq;
        finishingQueue = fq;
        robotPool = rp;
    }
    public Car car() { return car; }
    public CyclicBarrier barrier() { return barrier; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Блокируется, пока рама не будет доступна:
                car = chassisQueue.take();
                // Привлечение роботов для выполнения работы:
                robotPool.hire(EngineRobot.class, this);
                robotPool.hire(DriveTrainRobot.class, this);
                robotPool.hire(WheelRobot.class, this);
                barrier.await(); // Пока роботы не закончат работу
                // Машина помещается в очередь finishingQueue
                // для дальнейшей работы
                finishingQueue.put(car);
            }
        } catch(InterruptedException e) {
            print("Exiting Assembler via interrupt");
        } catch(BrokenBarrierException e) {
            // Исключение, о котором нужно знать
            throw new RuntimeException(e);
        }
        print("Assembler off");
    }
}

class Reporter implements Runnable {

```

```
private CarQueue carQueue;
public Reporter(CarQueue cq) { carQueue = cq; }
public void run() {
    try {
        while(!Thread.interrupted()) {
            print(carQueue.take());
        }
    } catch(InterruptedException e) {
        print("Exiting Reporter via interrupt");
    }
    print("Reporter off");
}
}

abstract class Robot implements Runnable {
    private RobotPool pool;
    public Robot(RobotPool p) { pool = p; }
    protected Assembler assembler;
    public Robot assignAssembler(Assembler assembler) {
        this.assembler = assembler;
        return this;
    }
    private boolean engage = false;
    public synchronized void engage() {
        engage = true;
        notifyAll();
    }
    // Часть run(), отличная для каждого робота:
    abstract protected void performService();
    public void run() {
        try {
            powerDown(); // Ожидать, пока не понадобится
            while(!Thread.interrupted()) {
                performService();
                assembler.barrier().await(); // Синхронизация
                // Задание выполнено...
                powerDown();
            }
        } catch(InterruptedException e) {
            print("Exiting " + this + " via interrupt");
        } catch(BrokenBarrierException e) {
            // Исключение, о котором нужно знать
            throw new RuntimeException(e);
        }
        print(this + " off");
    }
    private synchronized void
    powerDown() throws InterruptedException {
        engage = false;
        assembler = null;          // Отключение от Assembler
                                   // Возвращение в пул:
        pool.release(this);
        while(engage == false) // Выключение питания
            wait();
    }
    public String toString() { return getClass().getName(); }
}

class EngineRobot extends Robot {
```

продолжение ➤

```

    public EngineRobot(RobotPool pool) { super(pool); }
    protected void performService() {
        print(this + " installing engine");
        assembler.car().addEngine();
    }
}

class DriveTrainRobot extends Robot {
    public DriveTrainRobot(RobotPool pool) { super(pool); }
    protected void performService() {
        print(this + " installing DriveTrain");
        assembler.car().addDriveTrain();
    }
}

class WheelRobot extends Robot {
    public WheelRobot(RobotPool pool) { super(pool); }
    protected void performService() {
        print(this + " installing Wheels");
        assembler.car().addWheels();
    }
}

class RobotPool {
    // Незаметно предотвращает использование идентичных элементов:
    private Set<Robot> pool = new HashSet<Robot>();
    public synchronized void add(Robot r) {
        pool.add(r);
        notifyAll();
    }
    public synchronized void
    hire(Class<? extends Robot> robotType, Assembler d)
    throws InterruptedException {
        for(Robot r : pool)
            if(r.getClass().equals(robotType)) {
                pool.remove(r);
                r.assignAssembler(d);
                r.engage();    // Включение для выполнения задания
                return;
            }
        wait();                // Нет доступных кандидатов
        hire(robotType, d);    // Повторная попытка с рекурсией
    }
    public synchronized void release(Robot r) { add(r); }
}

public class CarBuilder {
    public static void main(String[] args) throws Exception {
        CarQueue chassisQueue = new CarQueue(),
            finishingQueue = new CarQueue();
        ExecutorService exec = Executors.newCachedThreadPool();
        RobotPool robotPool = new RobotPool();
        exec.execute(new EngineRobot(robotPool));
        exec.execute(new DriveTrainRobot(robotPool));
        exec.execute(new WheelRobot(robotPool));
        exec.execute(new Assembler(
            chassisQueue, finishingQueue, robotPool));
        exec.execute(new Reporter(finishingQueue));
        // Создание рам приводит конвейер в движение:
    }
}

```

```
    exec.execute(new ChassisBuilder(chassisQueue));  
    TimeUnit.SECONDS.sleep(7);  
    exec.shutdownNow();  
}  
} /* (Execute to see output) *///:~
```

Объекты Car перемещаются из одного места в другое через очередь CarQueue — разновидность `LinkedBlockingQueue`. Объект `ChassisBuilder` создает «минимальную» версию Car и помещает ее в CarQueue. Объект `Assembler` получает Car из CarQueue и привлекает объекты Robot для работы над ним. Использование `CyclicBarrier` позволяет `Assembler` дожидаться, пока все объекты Robot завершат свою работу; в этот момент Car помещается в выходную очередь CarQueue для перемещения к следующей операции. Потребителем последнего объекта CarQueue является объект `Reporter`, который просто выводит Car, чтобы продемонстрировать, что все задачи были выполнены.

Объекты Robot объединяются в пул, и когда возникает необходимость в выполнении операции, из пула извлекается соответствующий объект. После завершения операции Robot возвращается в пул.

В методе `main()` создаются все необходимые объекты и инициализируются задачи; `ChassisBuilder` начинается в последнюю очередь для запуска процесса (впрочем, из-за поведения `LinkedBlockingQueue` эта задача с таким же успехом может быть запущена первой). Обратите внимание: в этой программе выполняются все рекомендации по сроку жизни объектов и задач, приведенные в этой главе, поэтому процесс завершения безопасен.

Все методы объекта Car синхронизированы. В данном примере эта синхронизация избыточна, поскольку на фабрике объекты Car перемещаются в очередях, и с каждым объектом в любой момент времени может работать только одна задача. По сути, очереди обеспечивают последовательный доступ к Car. Вы рискуете попасть в типичную ловушку — «Давайте попробуем оптимизировать программу и уберем синхронизацию класса Car, потому что здесь она вроде бы не нужна». Но позднее эта система может быть подключена к другой, в которой Car нуждается в синхронизации, и тогда она перестает работать.

Брайан Гетц замечает по этому поводу:

Намного проще сказать: «Объект Car может использоваться из нескольких потоков, поэтому его стоит сделать потоково-безопасным наиболее очевидным способом». Мне это напоминает таблички с надписью: „На перила не опираться“, которые ставят в парках перед обрывами. Конечно, настоящая цель этих табличек — не помешать вам опираться на перила, а не дать вам упасть с обрыва. Однако выполнить правило „На перила не опираться“ намного проще, чем выполнить правило „С обрыва не падать“».

- 37.** (2) Измените пример `CarBuilder.java` и добавьте в сборку машины новый этап с установкой выхлопной системы, кузова и щитков. Как и на втором этапе, следует считать, что эти операции могут выполняться одновременно роботами.
- 38.** (3) Смоделируйте сценарий строительства дома, описанный в этой главе, по образцу кода `CarBuilder.java`.

Оптимизация

В библиотеку Java SE5 `java.util.concurrent` включены классы, предназначенные для повышения производительности. При просмотре библиотеки `concurrent` трудно сразу понять, какие классы предназначены для повседневного использования (как, например, `BlockingQueue`), а какие существуют только для улучшения производительности. В этом разделе рассматриваются некоторые проблемы быстродействия и классы, предназначенные для оптимизации.

Сравнение технологий мьютексов

Теперь, когда Java поддерживает старое ключевое слово `synchronized` наряду с новыми классами Java SE5 `Lock` и `Atomic`, интересно сравнить разные варианты, чтобы вы лучше поняли, чем полезен каждый из них и в каких ситуациях его стоит применять.

Было бы наивно делать выводы на основании простого тестирования, как в следующем примере:

```
//: concurrency/SimpleMicroBenchmark.java
// Опасность микрохронометража.
import java.util.concurrent.locks.*;

abstract class Incrementable {
    protected long counter = 0;
    public abstract void increment();
}

class SynchronizingTest extends Incrementable {
    public synchronized void increment() { ++counter; }
}

class LockingTest extends Incrementable {
    private Lock lock = new ReentrantLock();
    public void increment() {
        lock.lock();
        try {
            ++counter;
        } finally {
            lock.unlock();
        }
    }
}

public class SimpleMicroBenchmark {
    static long test(Incrementable incr) {
        long start = System.nanoTime();
        for(long i = 0; i < 10000000L; i++)
            incr.increment();
        return System.nanoTime() - start;
    }
    public static void main(String[] args) {
        long synchTime = test(new SynchronizingTest());
        long lockTime = test(new LockingTest());
        System.out.printf("synchronized: %1$10d\n", synchTime);
        System.out.printf("Lock:           %1$10d\n", lockTime);
        System.out.printf("Lock/synchronized = %1$.3f",
```



```
        (double)lockTime/((double)synchTime);
    }
} /* Output: (75% match)
synchronized: 244919117
Lock:         939098964
Lock/synchronized = 3.834
*///:~
```

Из результатов видно, что вызовы синхронизированного метода выполняются быстрее, чем вызовы с использованием `ReentrantLock`. Что происходит?

Этот пример демонстрирует опасность так называемого «микрохронометража»¹. Этим термином обычно обозначается изолированное тестирование отдельной функции вне контекста. Конечно, для проверки утверждений типа «`Lock` работает намного быстрее `synchronized`» все равно придется писать тесты. Однако при написании таких тестов необходимо знать, что именно происходит во время компиляции и выполнения.

В этом примере присутствует ряд проблем. Во-первых, настоящие различия в производительности проявляются только при конкуренции за мьютексы, поэтому в программе должны использоваться множественные задачи, которые пытаются обратиться к секциям кода, защищенным мьютексами. В предыдущем примере каждый мьютекс проверяется изолированно одним потоком `main()`.

Во-вторых, встречая ключевое слово `synchronized`, компилятор может применять специальные оптимизации — и может быть, даже заметить, что программа является однопоточной. Компилятор может даже определить, что счетчик просто увеличивается фиксированное число раз, и просто заранее вычислить результат. Компиляторы и исполнительные системы работают по разным правилам, поэтому трудно предсказать заранее, что именно произойдет, но мы должны предотвратить возможные попытки прогнозирования результата компилятором.

Чтобы создать состоятельный тест, придется усложнить программу. Прежде всего, нам понадобится несколько задач — и не только задач, изменяющих внутренние значения, но и задач, читающих эти значения (иначе оптимизатор может обнаружить, что значения не используются). Кроме того, вычисления должны быть достаточно сложными и непредсказуемыми, чтобы компилятор не мог применить агрессивные оптимизации. Для этого мы будем использовать предварительную загрузку большого массива случайных чисел `int` (предварительная загрузка снижает влияние вызовов `Random.nextInt()` в основных циклах) и использование этих значений при суммировании:

```
//: concurrency/SynchronizationComparisons.java
// Сравнение производительности объектов Lock и Atomic
// с ключевым словом synchronized.
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.concurrent.locks.*;
import java.util.*;
import static net.mindview.util.Print.*;
```

продолжение ➤

¹ Брайан Гетц оказал большую помощь, объясняя мне эти вопросы. Дополнительная информация об измерении производительности приведена по адресу www.ibm.com/developerworks/library/j-jtp12214.

```

abstract class Accumulator {
    public static long cycles = 50000L;
    // Количество объектов Modifier и Reader в каждом тесте:
    private static final int N = 4;
    public static ExecutorService exec =
        Executors.newFixedThreadPool(N*2);
    private static CyclicBarrier barrier =
        new CyclicBarrier(N*2 + 1);
    protected volatile int index = 0;
    protected volatile long value = 0;
    protected long duration = 0;
    protected String id = "error";
    protected final static int SIZE = 100000;
    protected static int[] preLoaded = new int[SIZE];
    static {
        // Загрузка массива случайных чисел:
        Random rand = new Random(47);
        for(int i = 0; i < SIZE; i++)
            preLoaded[i] = rand.nextInt();
    }
    public abstract void accumulate();
    public abstract long read();
    private class Modifier implements Runnable {
        public void run() {
            for(long i = 0; i < cycles; i++)
                accumulate();
            try {
                barrier.await();
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
    private class Reader implements Runnable {
        private volatile long value;
        public void run() {
            for(long i = 0; i < cycles; i++)
                value = read();
            try {
                barrier.await();
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
    public void timedTest() {
        long start = System.nanoTime();
        for(int i = 0; i < N; i++) {
            exec.execute(new Modifier());
            exec.execute(new Reader());
        }
        try {
            barrier.await();
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
        duration = System.nanoTime() - start;
        printf("%-13s: %13d\n", id, duration);
    }
}

```

```
public static void
report(Accumulator acc1, Accumulator acc2) {
    printf("%-22s: %.2f\n", acc1.id + "/" + acc2.id,
        (double)acc1.duration/(double)acc2.duration);
}
}

class Baseline extends Accumulator {
    { id = "Baseline"; }
    public void accumulate() {
        value += preLoaded[index++];
        if(index >= SIZE) index = 0;
    }
    public long read() { return value; }
}

class SynchronizedTest extends Accumulator {
    { id = "synchronized"; }
    public synchronized void accumulate() {
        value += preLoaded[index++];
        if(index >= SIZE) index = 0;
    }
    public synchronized long read() {
        return value;
    }
}

class LockTest extends Accumulator {
    { id = "Lock"; }
    private Lock lock = new ReentrantLock();
    public void accumulate() {
        lock.lock();
        try {
            value += preLoaded[index++];
            if(index >= SIZE) index = 0;
        } finally {
            lock.unlock();
        }
    }
    public long read() {
        lock.lock();
        try {
            return value;
        } finally {
            lock.unlock();
        }
    }
}

class AtomicTest extends Accumulator {
    { id = "Atomic"; }
    private AtomicInteger index = new AtomicInteger(0);
    private AtomicLong value = new AtomicLong(0);
    public void accumulate() {
        // Решение не может использовать сразу несколько
        // объектов Atomic. Тем не менее оно дает
        // представление о производительности:
        int i = index.getAndIncrement();
```

продолжение ↗

```

        value.getAndAdd(preLoaded[i]);
        if(++i >= SIZE)
            index.set(0);
    }
    public long read() { return value.get(); }
}

public class SynchronizationComparisons {
    static BaseLine baseline = new BaseLine();
    static SynchronizedTest synch = new SynchronizedTest();
    static LockTest lock = new LockTest();
    static AtomicTest atomic = new AtomicTest();
    static void test() {
        print("=====");
        printf("%-12s : %13d\n", "Cycles", Accumulator.cycles);
        baseline.timedTest();
        synch.timedTest();
        lock.timedTest();
        atomic.timedTest();
        Accumulator.report(synch, baseline);
        Accumulator.report(lock, baseline);
        Accumulator.report(atomic, baseline);
        Accumulator.report(synch, lock);
        Accumulator.report(synch, atomic);
        Accumulator.report(lock, atomic);
    }
    public static void main(String[] args) {
        int iterations = 5; // По умолчанию
        if(args.length > 0) // Возможность изменения iterations
            iterations = new Integer(args[0]);
        // При первом проходе заполняется пул потоков:
        print("Warmup");
        baseline.timedTest();
        // Теперь исходный тест не включает затраты
        // на первый запуск потоков.
        // Создание множественных элементов данных:
        for(int i = 0; i < iterations; i++) {
            test();
            Accumulator.cycles *= 2;
        }
        Accumulator.exec.shutdown();
    }
}

/* Output: (Sample)
Warmup
BaseLine      :      34237033
=====
Cycles        :      50000
BaseLine      :      20966632
synchronized :      24326555
Lock          :      53669950
Atomic        :      30552487
synchronized/BaseLine : 1.16
Lock/BaseLine : 2.56
Atomic/BaseLine : 1.46
synchronized/Lock : 0.45
synchronized/Atomic : 0.79
Lock/Atomic    : 1.76
=====
Cycles        :      100000

```

```
BaseLine      :      41512818
synchronized  :      43843003
Lock          :      87430386
Atomic        :      51892350
synchronized/BaseLine : 1.06
Lock/BaseLine : 2.11
Atomic/BaseLine : 1.25
synchronized/Lock : 0.50
synchronized/Atomic : 0.84
Lock/Atomic    : 1.68
=====
Cycles        :      200000
BaseLine      :      80176670
synchronized  :      5455046661
Lock          :      177686829
Atomic        :      101789194
synchronized/BaseLine : 68.04
Lock/BaseLine : 2.22
Atomic/BaseLine : 1.27
synchronized/Lock : 30.70
synchronized/Atomic : 53.59
Lock/Atomic    : 1.75
=====
Cycles        :      400000
BaseLine      :      160383513
synchronized  :      780052493
Lock          :      362187652
Atomic        :      202030984
synchronized/BaseLine : 4.86
Lock/BaseLine : 2.26
Atomic/BaseLine : 1.26
synchronized/Lock : 2.15
synchronized/Atomic : 3.86
Lock/Atomic    : 1.79
=====
Cycles        :      800000
BaseLine      :      322064955
synchronized  :      336155014
Lock          :      704615531
Atomic        :      393231542
synchronized/BaseLine : 1.04
Lock/BaseLine : 2.19
Atomic/BaseLine : 1.22
synchronized/Lock : 0.47
synchronized/Atomic : 0.85
Lock/Atomic    : 1.79
=====
Cycles        :      1600000
BaseLine      :      650004120
synchronized  :      52235762925
Lock          :      1419602771
Atomic        :      796950171
synchronized/BaseLine : 80.36
Lock/BaseLine : 2.18
Atomic/BaseLine : 1.23
synchronized/Lock : 36.80
synchronized/Atomic : 65.54
Lock/Atomic    : 1.78
```

продолжение ↗

```

=====
Cycles      :      3200000
BaseLine    :      1285664519
synchronized : 96336767661
Lock        :      2846988654
Atomic      :      1590545726
synchronized/BaseLine : 74.93
Lock/BaseLine : 2.21
Atomic/BaseLine : 1.24
synchronized/Lock : 33.84
synchronized/Atomic : 60.57
Lock/Atomic   : 1.79
*///:~

```

Программа использует паттерн проектирования «Шаблонный метод» для выделения всего общего кода в базовый класс и изоляции всего переменного кода в реализациях `accumulate()` и `read()` производного класса. В каждом из производных классов `SynchronizedTest`, `LockTest` и `AtomicTest` мы видим, как `accumulate()` и `read()` выражают разные способы реализации взаимного исключения.

В этой программе задачи выполняются с использованием пула `FixedThreadPool`, чтобы потоки создавались в самом начале работы, а затраты на создание были исключены из тестов. Просто для надежности исходный тест дублируется, а первый результат отбрасывается, потому что он включает создание исходного потока.

Объект `CyclicBarrier` необходим, потому что, прежде чем объявлять завершенным каждый тест, мы должны быть уверены в том, что все задачи были завершены.

Условие `static` используется для предварительной загрузки массива случайных чисел до начала каких-либо тестов. Если генерирование случайных чисел сопряжено с какими-то дополнительными затратами, они не отразятся на результатах тестов.

При каждом вызове `accumulate()` метод переходит к следующей позиции массива `preLoaded` (с циклическим возвратом к началу массива) и прибавляет к `value` другое случайно сгенерированное число. Множественные задачи `Modifier` и `Reader` обеспечивают конкуренцию за объект `Accumulator`.

Обратите внимание: в `AtomicTest` я замечаю, что ситуация слишком сложна для использования объектов `Atomic` — по сути, при участии нескольких объектов `Atomic` вам, вероятно, придется отступить и воспользоваться более традиционными мьютексами (в документации JDK указано, что использование объектов `Atomic` работает только в том случае, если критические обновления объекта ограничены одной переменной). Однако этот тест оставлен в программе, чтобы вы могли получить представление о преимуществах объектов `Atomic` в области производительности.

В методе `main()` тест выполняется многократно; возможно, вы захотите запросить более 5 повторений (по умолчанию). Для каждого повторения количество тестовых циклов удваивается, и вы можете понаблюдать за поведением разных мьютексов при выполнении в течение все большего времени. Как видно из выходных данных, результаты выглядят несколько неожиданно. Для первых четырех итераций ключевое слово `synchronized` работает эффективнее `lock` или `Atomic`. Но внезапно достигается какой-то критический порог, и `synchronized` теряет свою эффективность, тогда как `lock` и `Atomic` приблизительно поддерживают пропорцию по отношению

к тесту `BaseLine`, а следовательно, начинают существенно превосходить `synchronized` по эффективности.

Помните, что эта программа всего лишь дает представление о различиях между разными реализациями мьютексов, а приведенные результаты описывают эти различия на моем конкретном компьютере в конкретных обстоятельствах. Эксперименты показывают, что при разном количестве потоков и изменении времени выполнения могут происходить значительные изменения в поведении. Некоторые оптимизации времени выполнения активизируются только после того, как программа проработает несколько минут — или несколько часов для серверных программ.

С учетом сказанного достаточно очевидно, что решения с `Lock` обычно существенно превосходят по эффективности решения с `synchronized`, а дополнительные затраты при использовании `synchronized` изменяются в широких пределах, тогда как в решениях с `Lock` они относительно постоянны.

Означает ли это, что ключевое слово `synchronized` не стоит использовать? Необходимо учесть два фактора: во-первых, в `SynchronizationComparisons.java` тела защищаемых методов чрезвычайно малы. В общем случае это считается желательным — защита мьютексом применяется только к тем секциям, для которых это абсолютно необходимо. Однако на практике защищаемые секции могут быть больше секций в нашем примере, поэтому доля времени, проводимого в теле метода, может значительно превышать затраты на вход и выход из мьютекса и может компенсировать любые преимущества от ускорения мьютекса. Конечно, узнать это можно только одним способом (в процессе оптимизации, не раньше) — опробовать разные решения и посмотреть, как они отражаются на результатах.

Во-вторых, из чтения кода этой главы становится ясно, что ключевое слово `synchronized` порождает намного более понятный код, чем идиома «установка блокировки-try/finally-снятие блокировки», необходимая для `Lock`. По этой причине в данной главе в основном используется ключевое слово `synchronized`. Как я упоминал ранее в этой книге, код читается намного чаще, чем пишется, — при программировании общение с другими людьми важнее общения с компьютером, так что удобочитаемость кода играет очень важную роль. Итак, разумно начать с ключевого слова `synchronized` и переходить на объекты `Lock` только тогда, когда вы выполняете оптимизацию для повышения производительности.

Контейнеры без блокировок

Как упоминалось в главе 11, контейнеры являются одним из фундаментальных инструментов во всех областях программирования, в том числе и в параллельном программировании. По этой причине ранние версии контейнеров — такие, как `Vector` и `Hashtable`, — содержали много синхронизированных методов, что создавало неприемлемо высокие дополнительные затраты в многопоточных приложениях. В Java 1.2 новая библиотека контейнеров была рассинхронизирована, а класс `Collections` получил разные статические синхронизированные методы для разных типов контейнеров. И хотя такой подход позволял разработчику решить, нужно ли применять синхронизацию при работе с контейнером, дополнительные затраты по-прежнему базировались на

`synchronized`. В Java SE5 появились новые контейнеры, предназначенные специально для повышения производительности потоково-безопасных операций, использующие хорошо продуманную реализацию для устранения блокировок.

Общая стратегия, заложенная в основу этих контейнеров, выглядит так: изменения в контейнерах могут происходить одновременно с чтением при условии, что «читатели» видят только результаты завершенных изменений. Изменение выполняется с отдельной копией части структуры данных (а иногда с копией всей структуры), причем эта копия остается невидимой в процессе изменения. Только тогда, когда изменения будут завершены, измененная структура заменяет «основную» структуру данных в атомарной операции, после чего изменения становятся видны при чтении.

В `CopyOnWriteArrayList` запись приводит к созданию копии всего используемого массива. Исходный массив остается на месте, так что во время изменения копии возможно безопасное чтение. Когда изменение будет завершено, атомарная операция меняет массивы, и новое состояние будет отражено в последующих операциях чтения. Одно из преимуществ `CopyOnWriteArrayList` заключается в том, что этот контейнер не выдает исключение `ConcurrentModificationException` при использовании нескольких итераторов для перебора и модификации списка, так что вам не придется писать специальный код для защиты от таких исключений (как приходилось в прошлом). `CopyOnWriteArraySet` использует `CopyOnWriteArrayList` для реализации своего поведения.

`ConcurrentHashMap` и `ConcurrentLinkedQueue` используют сходные средства, чтобы обеспечить возможность параллельного чтения и записи, но копируются и изменяются только части контейнера (вместо всего контейнера). Операции чтения не «увидят» изменения, пока те не будут закреплены в контейнере. `ConcurrentHashMap` не возбуждает исключения `ConcurrentModificationExceptions`.

Вопросы производительности

Если вы в основном ограничиваетесь чтением из контейнера без блокировок, операции с ним будут выполняться намного быстрее, чем с его `synchronized`-аналогом, из-за дополнительных затрат на установление и снятие блокировок. Этот принцип справедлив и при небольшом количестве операций записи в контейнер без блокировок, но здесь будет интересно выяснить, что же в данном контексте можно считать «небольшим». В этом разделе я постараюсь дать некоторое представление о различиях в производительности между контейнерами в разных условиях. Начнем с общей тестовой среды для тестирования любых типов контейнеров, включая `Map`. Параметр `C` представляет тип контейнера:

```
//: concurrency/Tester.java
// Программа для тестирования контейнеров
// с параллельным доступом.
import java.util.concurrent.*;
import net.mindview.util.*;

public abstract class Tester<C> {
    static int testReps = 10;
    static int testCycles = 1000;
    static int containerSize = 1000;
    abstract C containerInitializer();
```



```
abstract void startReadersAndWriters();
C testContainer;
String testId;
int nReaders;
int nWriters;
volatile long readResult = 0;
volatile long readTime = 0;
volatile long writeTime = 0;
CountDownLatch endLatch;
static ExecutorService exec =
    Executors.newCachedThreadPool();
Integer[] writeData;
Tester(String testId, int nReaders, int nWriters) {
    this.testId = testId + " " +
        nReaders + "r " + nWriters + "w";
    this.nReaders = nReaders;
    this.nWriters = nWriters;
    writeData = Generated.array(Integer.class,
        new RandomGenerator.Integer(), containerSize);
    for(int i = 0; i < testReps; i++) {
        runTest();
        readTime = 0;
        writeTime = 0;
    }
}
void runTest() {
    endLatch = new CountDownLatch(nReaders + nWriters);
    testContainer = containerInitializer();
    startReadersAndWriters();
    try {
        endLatch.await();
    } catch(InterruptedException ex) {
        System.out.println("endLatch interrupted");
    }
    System.out.printf("%-27s %14d %14d\n",
        testId, readTime, writeTime);
    if(readTime != 0 && writeTime != 0)
        System.out.printf("%-27s %14d\n",
            "readTime + writeTime =", readTime + writeTime);
}
abstract class TestTask implements Runnable {
    abstract void test();
    abstract void putResults();
    long duration;
    public void run() {
        long startTime = System.nanoTime();
        test();
        duration = System.nanoTime() - startTime;
        synchronized(Tester.this) {
            putResults();
        }
        endLatch.countDown();
    }
}
public static void initMain(String[] args) {
    if(args.length > 0)
        testReps = new Integer(args[0]);
    if(args.length > 1)
```

```

        testCycles = new Integer(args[1]);
        if(args.length > 2)
            containerSize = new Integer(args[2]);
        System.out.printf("%-27s %14s %14s\n",
            "Type", "Read time", "Write time");
    }
} ///:~

```

Абстрактный метод `containerInitializer()` возвращает инициализированный контейнер, который необходимо протестировать; он сохраняется в поле `testContainer`. Другой абстрактный метод `startReadersAndWriters()` запускает задачи чтения и записи, которые читают и изменяют тестируемый контейнер. Разные тесты проводятся с разными количествами задач, чтобы увидеть эффект конкуренции за блокировку (для синхронизированных контейнеров) и записи (для контейнеров без блокировок).

Конструктор получает разную информацию о тесте (идентификаторы аргументов должны быть понятны без комментариев), после чего `repetitions` вызывает метод `runTest()` раз. `runTest()` создает объект `CountDownLatch` (чтобы тест мог определить, когда все задачи завершены), инициализирует контейнер, вызывает `startReadersAndWriters()` и ожидает, когда все задачи завершатся.

Каждый класс `Reader` или `Writer` строится на базе класса `TestTask`, который измеряет продолжительность абстрактного метода `test()`, после чего вызывает `putResults()` внутри синхронизированного блока для сохранения результатов.

Чтобы использовать эту программную среду (в которой вы узнаете паттерн проектирования «Шаблонный метод»), необходимо использовать наследование от `Tester` для конкретного типа контейнера, который требуется протестировать, и предоставить подходящие классы `Reader` и `Writer`:

```

//: concurrency/ListComparisons.java
// {Args: 1 10 10} (Fast verification check during build)
// Приблизительное сравнение производительности
// потоково-безопасных списков.
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

abstract class ListTest extends Tester<List<Integer>> {
    ListTest(String testId, int nReaders, int nWriters) {
        super(testId, nReaders, nWriters);
    }

    class Reader extends TestTask {
        long result = 0;
        void test() {
            for(long i = 0; i < testCycles; i++)
                for(int index = 0; index < containerSize; index++)
                    result += testContainer.get(index);
        }
        void putResults() {
            readResult += result;
            readTime += duration;
        }
    }
}

```

```

class Writer extends TestTask {
    void test() {
        for(long i = 0; i < testCycles; i++)
            for(int index = 0; index < containerSize; index++)
                testContainer.set(index, writeData[index]);
    }
    void putResults() {
        writeTime += duration;
    }
}

void startReadersAndWriters() {
    for(int i = 0; i < nReaders; i++)
        exec.execute(new Reader());
    for(int i = 0; i < nWriters; i++)
        exec.execute(new Writer());
}

}

class SynchronizedArrayListTest extends ListTest {
    List<Integer> containerInitializer() {
        return Collections.synchronizedList(
            new ArrayList<Integer>(
                new CountingIntegerList(containerSize)));
    }
    SynchronizedArrayListTest(int nReaders, int nWriters) {
        super("Synched ArrayList", nReaders, nWriters);
    }
}

class CopyOnWriteArrayListTest extends ListTest {
    List<Integer> containerInitializer() {
        return new CopyOnWriteArrayList<Integer>(
            new CountingIntegerList(containerSize));
    }
    CopyOnWriteArrayListTest(int nReaders, int nWriters) {
        super("CopyOnWriteArrayList", nReaders, nWriters);
    }
}

public class ListComparisons {
    public static void main(String[] args) {
        Tester.initMain(args);
        new SynchronizedArrayListTest(10, 0);
        new SynchronizedArrayListTest(9, 1);
        new SynchronizedArrayListTest(5, 5);
        new CopyOnWriteArrayListTest(10, 0);
        new CopyOnWriteArrayListTest(9, 1);
        new CopyOnWriteArrayListTest(5, 5);
        Tester.exec.shutdown();
    }
}

/* Output: (Sample)
Type                                Read time      Write time
Synched ArrayList 10r 0w            232158294700      0
Synched ArrayList 9r 1w             198947618203    24918613399
readTime + writeTime =              223866231602
Synched ArrayList 5r 5w             117367305062    132176613508
readTime + writeTime =              249543918570
CopyOnWriteArrayList 10r 0w          758386889        0

```

продолжение ➤

```
CopyOnWriteArrayList 9r 1w      741305671      136145237
readTime + writeTime =          877450908
CopyOnWriteArrayList 5r 5w      212763075      67967464300
readTime + writeTime =          68180227375
*///:~
```

В `ListTest` классы `Reader` и `Writer` выполняют конкретные действия для `List<Integer>`. В `Reader.putResults()` сохраняется значение `duration`, а также `result`, чтобы предотвратить исключение вычислений в результате оптимизации. Затем метод `startReadersAndWriters()` определяется для создания и выполнения конкретных задач `Reader` и `Writer`.

Класс `ListTest` может использоваться для дальнейшего наследования с переопределением `containerInitializer()` для создания и инициализации конкретных тестовых контейнеров.

В методе `main()` представлены разновидности тестов с разным количеством задач чтения и записи. Вы можете изменить тестовые переменные через аргументы командной строки благодаря вызову `Tester.initMain(args)`.

По умолчанию каждый тест выполняется 10 раз; это помогает стабилизировать результаты, которые могут изменяться из-за таких операций JVM, как адаптивная оптимизация и уборка мусора¹. Результаты были отредактированы, чтобы в них отображалась только последняя итерация из каждого теста. Из выходных данных видно, что синхронизированный контейнер `ArrayList` обеспечивает примерно постоянную производительность независимо от количества задач чтения и записи — задачи чтения конкурируют с другими задачами чтения за блокировки так же, как это делают задачи записи. С другой стороны, `CopyOnWriteArrayList` работает намного быстрее при отсутствии задач записи, но даже при пяти задачах записи наблюдается значительный рост производительности. Похоже, что с применением `CopyOnWriteArrayList` можно не сдерживаться; кажется, влияние записи в список достаточно долго перевешивается последствиями синхронизации всего списка. Конечно, чтобы понять, какой из двух подходов лучше, необходимо опробовать оба подхода в своем приложении.

И снова подчеркну, что абсолютные показатели в этом тесте не значат ничего, потому что в вашем случае числа наверняка будут другими. Тест нужен только для того, чтобы дать представления о различиях в поведении двух типов контейнеров. Так как контейнер `CopyOnWriteArraySet` использует `CopyOnWriteArrayList`, он ведет себя аналогично и отдельный тест для него не нужен.

Сравнение реализаций Map

Та же тестовая программа позволяет приблизительно сравнить производительность синхронизированной карты `HashMap` с `ConcurrentHashMap`:

```
//: concurrency/MapComparisons.java
// {Args: 1 10 10} (Fast verification check during build)
// Приближенное сравнение производительности
// потоково-безопасных реализаций Map.
import java.util.concurrent.*;
```

¹ За вводной информацией о проведении хронометража в условиях динамической компиляции Java обращайтесь к статье www-128.ibm.com/developerworks/library/j-jtp12214.

```
import java.util.*;
import net.mindview.util.*;

abstract class MapTest
extends Tester<Map<Integer,Integer>> {
    MapTest(String testId, int nReaders, int nWriters) {
        super(testId, nReaders, nWriters);
    }
    class Reader extends TestTask {
        long result = 0;
        void test() {
            for(long i = 0; i < testCycles; i++)
                for(int index = 0; index < containerSize; index++)
                    result += testContainer.get(index);
        }
        void putResults() {
            readResult += result;
            readTime += duration;
        }
    }
    class Writer extends TestTask {
        void test() {
            for(long i = 0; i < testCycles; i++)
                for(int index = 0; index < containerSize; index++)
                    testContainer.put(index, writeData[index]);
        }
        void putResults() {
            writeTime += duration;
        }
    }
    void startReadersAndWriters() {
        for(int i = 0; i < nReaders; i++)
            exec.execute(new Reader());
        for(int i = 0; i < nWriters; i++)
            exec.execute(new Writer());
    }
}

class SynchronizedHashMapTest extends MapTest {
    Map<Integer,Integer> containerInitializer() {
        return Collections.synchronizedMap(
            new HashMap<Integer,Integer>(
                MapData.map(
                    new CountingGenerator.Integer(),
                    new CountingGenerator.Integer(),
                    containerSize)));
    }
    SynchronizedHashMapTest(int nReaders, int nWriters) {
        super("Synched HashMap", nReaders, nWriters);
    }
}

class ConcurrentHashMapTest extends MapTest {
    Map<Integer,Integer> containerInitializer() {
        return new ConcurrentHashMap<Integer,Integer>(
            MapData.map(
                new CountingGenerator.Integer(),
                new CountingGenerator.Integer(), containerSize));
    }
}
```

```

ConcurrentHashMapTest(int nReaders, int nWriters) {
    super("ConcurrentHashMap", nReaders, nWriters);
}
}

public class MapComparisons {
    public static void main(String[] args) {
        Tester.initMain(args);
        new SynchronizedHashMapTest(10, 0);
        new SynchronizedHashMapTest(9, 1);
        new SynchronizedHashMapTest(5, 5);
        new ConcurrentHashMapTest(10, 0);
        new ConcurrentHashMapTest(9, 1);
        new ConcurrentHashMapTest(5, 5);
        Tester.exec.shutdown();
    }
} /* Output: (Sample)
Type                Read time      Write time
Synched HashMap 10r 0w    306052025049          0
Synched HashMap 9r 1w    428319156207    47697347568
readTime + writeTime =    476016503775
Synched HashMap 5r 5w    243956877760    244012003202
readTime + writeTime =    487968880962
ConcurrentHashMap 10r 0w    23352654318          0
ConcurrentHashMap 9r 1w    18833089400    1541853224
readTime + writeTime =    20374942624
ConcurrentHashMap 5r 5w    12037625732    11850489099
readTime + writeTime =    23888114831
*///:~

```

Последствия от добавления задач записи с `ConcurrentHashMap` еще менее очевидны, чем для `CopyOnWriteArrayList`, но `ConcurrentHashMap` использует другую реализацию, которая сводит к минимуму влияние операций записи.

Оптимистическая блокировка

Хотя объекты `Atomic` выполняют атомарные операции (такие, как `decrementAndGet()`), некоторые классы `Atomic` также позволяют реализовать так называемую «оптимистическую блокировку». Это означает, что мьютекс при вычислениях реально не используется, но после завершения вычислений и готовности к обновлению объекта `Atomic` используется метод с именем `compareAndSet()`. Методу передается старое и новое значения, и если старое значение не согласуется со значением в объекте `Atomic`, операция завершается неудачей — это означает, что другая задача успела изменить объект. В общем случае мы могли бы воспользоваться мьютексом (`synchronized` или `Lock`) для предотвращения одновременной модификации объекта несколькими задачами, но в данной ситуации проявляем «оптимизм»: оставляем данные незаблокированными и надеемся, что ни одна задача не вмешается и не изменит их. И снова все это делается ради производительности — применение `Atomic` вместо `synchronized` или `Lock` обеспечивает существенный выигрыш в производительности.

Что произойдет, если операция `compareAndSet()` будет выполнена неудачно? Здесь начинаются сложности, из-за которых оптимистическая блокировка применяется только для задач, которые можно адаптировать для этих требований. Вы должны

решить, что происходит при неудаче; это очень важный момент, потому что если вы не сможете каким-то образом восстановить работу программы, то вместо оптимистичной блокировки следует применять традиционные мьютексы. Возможно, операцию можно повторить, и со второго раза все получится. А может быть, ошибку можно просто проигнорировать — в некоторых моделях потеря одного элемента данных теряется на фоне общей картины (конечно, для принятия такого решения необходимо достаточно хорошо понимать вашу модель).

Возьмем фиктивную модель из 100 000 «генов» длиной 30 (допустим, начало некоей разновидности генетических алгоритмов). При каждой «эволюции» генетического алгоритма выполняются очень серьезные вычисления, поэтому вы решаете использовать многопроцессорную машину для распределения задач и повышения производительности. Кроме того, вы используете объекты Atomic вместо Lock, чтобы избежать лишних затрат, связанных с применением мьютексов. (Естественно, это решение строится уже после того, как вы сначала записали свой код в простейшем варианте с ключевым словом synchronized. Программа заработала, вы убедились, что она выполняется слишком медленно, и начали применять средства повышения производительности!) Из-за природы модели, если во время вычисления произойдет коллизия, обнаружившая эту коллизию задача может просто проигнорировать ее и не обновлять значение. Вот как это выглядит:

```
//: concurrency/FastSimulation.java
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class FastSimulation {
    static final int N_ELEMENTS = 100000;
    static final int N_GENES = 30;
    static final int N_EVOLVERS = 50;
    static final AtomicInteger[][] GRID =
        new AtomicInteger[N_ELEMENTS][N_GENES];
    static Random rand = new Random(47);
    static class Evolver implements Runnable {
        public void run() {
            while(!Thread.interrupted()) {
                // Случайный выбор элемента:
                int element = rand.nextInt(N_ELEMENTS);
                for(int i = 0; i < N_GENES; i++) {
                    int previous = element - 1;
                    if(previous < 0) previous = N_ELEMENTS - 1;
                    int next = element + 1;
                    if(next >= N_ELEMENTS) next = 0;
                    int oldvalue = GRID[element][i].get();
                    // Модель вычислений:
                    int newvalue = oldvalue +
                        GRID[previous][i].get() + GRID[next][i].get();
                    newvalue /= 3; // Вычисление среднего по трем значениями
                    if(!GRID[element][i]
                        .compareAndSet(oldvalue, newvalue)) {
                        // Наша задача - решить, как обработать ошибку. Здесь
                        // мы сообщаем о ней и игнорируем; наша модель
                        // справится с этой проблемой.
                    }
                }
            }
        }
    }
}
```

продолжение ➤

```
        print("Old value changed from " + oldvalue);
    }
}
}
}
}
public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < N_ELEMENTS; i++)
        for(int j = 0; j < N_GENES; j++)
            GRID[i][j] = new AtomicInteger(rand.nextInt(1000));
    for(int i = 0; i < N_EVOLVERS; i++)
        exec.execute(new Evolver());
    TimeUnit.SECONDS.sleep(5);
    exec.shutdownNow();
}
} /* (Execute to see output) */~
```

Все элементы размещаются в массиве с предположением, что это улучшит производительность (это предположение будет проверено в упражнении). Каждый объект `Evolver` вычисляет среднее значение элемента с предыдущим и следующим элементом, и если при переходе к обновлению происходит сбой, программа просто выводит значение и продолжает работу. Обратите внимание на отсутствие мьютексов в программе.

- 39.** (6) Насколько оправданны предположения `FastSimulation.java`? Попробуйте переключиться на массив обычных чисел `int` вместо `AtomicInteger` и используйте мьютексы `Lock`. Сравните производительность двух версий программы.

ReadWriteLock

Класс `ReadWriteLocks` оптимизирует ситуацию с относительно редкой записью и частыми чтениями из структуры данных. Несколько задач могут читать данные одновременно при условии, что ни одна задача не пытается их записывать. Если установлена блокировка записи, то чтение становится невозможным до ее освобождения.

Совершенно невозможно предсказать, улучшит ли `ReadWriteLock` производительность вашей программы. Это зависит от таких аспектов, как относительная частота чтения данных (по сравнению с частотой их записи), время операций чтения и записи (блокировка более сложна, поэтому при коротких операциях преимущества не проявляются), интенсивность конкуренции потоков, а также от того, выполняется ли программа на многопроцессорной машине. В конечном итоге узнать, принесет ли пользу `ReadWriteLock` на вашей машине или нет, можно только одним способом — попробовать на практике.

Следующий пример демонстрирует простейшее использование `ReadWriteLock`:

```

//: concurrency/ReaderWriterList.java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ReaderWriterList<T> {
    private ArrayList<T> lockedList;
    // Make the ordering fair:

```



```
private ReentrantReadWriteLock lock =
    new ReentrantReadWriteLock(true);
public ReaderWriterList(int size, T initialValue) {
    lockedList = new ArrayList<T>(
        Collections.nCopies(size, initialValue));
}
public T set(int index, T element) {
    Lock wlock = lock.writeLock();
    wlock.lock();
    try {
        return lockedList.set(index, element);
    } finally {
        wlock.unlock();
    }
}
public T get(int index) {
    Lock rlock = lock.readLock();
    rlock.lock();
    try {
        // Показать, что блокировка для чтения может быть
        // получена несколькими задачами чтения:
        if(lock.getReadLockCount() > 1)
            print(lock.getReadLockCount());
        return lockedList.get(index);
    } finally {
        rlock.unlock();
    }
}
public static void main(String[] args) throws Exception {
    new ReaderWriterListTest(30, 1);
}

class ReaderWriterListTest {
    ExecutorService exec = Executors.newCachedThreadPool();
    private final static int SIZE = 100;
    private static Random rand = new Random(47);
    private ReaderWriterList<Integer> list =
        new ReaderWriterList<Integer>(SIZE, 0);
    private class Writer implements Runnable {
        public void run() {
            try {
                for(int i = 0; i < 20; i++) { // 2-секундный тест
                    list.set(i, rand.nextInt());
                    TimeUnit.MILLISECONDS.sleep(100);
                }
            } catch(InterruptedException e) {
                // Допустимый способ выхода
            }
            print("Writer finished, shutting down");
            exec.shutdownNow();
        }
    }
    private class Reader implements Runnable {
        public void run() {
            try {
                while(!Thread.interrupted()) {
                    for(int i = 0; i < SIZE; i++) {
```

продолжение ➤

```

        list.get(i);
        TimeUnit.MILLISECONDS.sleep(1);
    }
}
} catch (InterruptedException e) {
    // Допустимый способ выхода
}
}
}
}
public ReaderWriterListTest(int readers, int writers) {
    for(int i = 0; i < readers; i++)
        exec.execute(new Reader());
    for(int i = 0; i < writers; i++)
        exec.execute(new Writer());
}
} /* (Execute to see output) *///:~

```

Контейнер `ReaderWriterList` может содержать фиксированное число элементов любого типа. Конструктору передается нужный размер списка и исходный объект для заполнения списка. Метод `set()` получает блокировку записи, чтобы вызвать метод `ArrayList.set()`, а метод `get()` получает блокировку чтения для вызова `ArrayList.get()`. Кроме того, метод `get()` проверяет, была ли получена блокировка чтения более чем одной задачей чтения, и если проверка дает положительный результат — выводит их количество для демонстрации того, что блокировка чтения может быть получена сразу несколькими задачами.

Чтобы протестировать `ReaderWriterList`, класс `ReaderWriterListTest` создает задачи чтения и записи для `ReaderWriterList<Integer>`. Обратите внимание: задач записи намного меньше, чем задач чтения.

Заглянув в документацию JDK для `ReentrantReadWriteLock`, вы увидите, что класс содержит и другие методы, а при работе с ним приходится учитывать факторы «равнодоступности» и «выбора политики». Это достаточно сложный инструмент, и применять его следует только в том случае, когда вы активно ищете пути к повышению производительности. В первой версии программы следует использовать обычную синхронизацию и применять `ReadWriteLock` только при необходимости.

- 40.** (6) По образцу `ReaderWriterList.java` создайте реализацию `ReaderWriterMap` на базе `HashMap`. Проанализируйте ее производительность при помощи измененной программы `MapComparisons.java`. Какие результаты она показывает по сравнению с синхронизированной реализацией `HashMap` и `ConcurrentHashMap`?

Активные объекты

Вероятно, к этой странице вы уже поняли, что параллельное программирование в Java — очень сложная тема и ее очень трудно правильно использовать. Вдобавок на первый взгляд происходящее выглядит противоестественно — задачи работают параллельно, а разработчик должен тратить огромные силы на то, чтобы они не мешали работать друг другу.

Если вы когда-либо писали на ассемблере, многопоточное программирование вызывает похожие ощущения: важна каждая мелочь, вы сами отвечаете за все, нет никакой «страховки» в виде проверки компилятора.

Может, что-то не так с самой потоковой моделью? В конце концов, она пришла почти неизменной из мира процедурного программирования. Возможно, существует более удобная модель параллельности, которая лучше подходит для объектно-ориентированного программирования.

Одно из альтернативных решений связано с использованием *активных объектов*. Эти объекты называются «активными»¹, потому что каждый объект поддерживает собственный рабочий поток и очередь сообщений, а все запросы к этому объекту ставятся в очередь для последовательного выполнения. Таким образом, при использовании активных объектов организуется последовательная обработка *сообщений, а не методов*; это означает, что нам не нужно защищаться от проблем, которые могут возникнуть при прерывании задачи на середине ее цикла.

Когда вы отправляете сообщение активному объекту, это сообщение преобразуется в задачу, которая помещается в очередь объекта для выполнения в будущем. Класс Java SE5 `Future` хорошо подходит для реализации этой схемы. Рассмотрим простой пример с двумя методами и организацией очереди вызовов:

```
//: concurrency/ActiveObjectDemo.java
// В аргументах асинхронных методов могут передаваться
// только константы, неизменные значения,
// "отсоединенные объекты" или другие активные объекты.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;
```

```
public class ActiveObjectDemo {
    private ExecutorService ex =
        Executors.newSingleThreadExecutor();
    private Random rand = new Random(47);
    // Вставка случайной задержки для моделирования
    // времени вычисления:
    private void pause(int factor) {
        try {
            TimeUnit.MILLISECONDS.sleep(
                100 + rand.nextInt(factor));
        } catch (InterruptedException e) {
            print("sleep() interrupted");
        }
    }
    public Future<Integer>
    calculateInt(final int x, final int y) {
        return ex.submit(new Callable<Integer>() {
            public Integer call() {
                print("starting " + x + " + " + y);
                pause(500);
                return x + y;
            }
        });
    }
    public Future<Float>
    calculateFloat(final float x, final float y) {
        return ex.submit(new Callable<Float>() {
```

продолжение ➤

¹ Спасибо Аллену Голубу, который не пожалел своего времени и объяснил мне эту тему.

```

    public Float call() {
        print("starting " + x + " + " + y);
        pause(2000);
        return x + y;
    }
});
}
public void shutdown() { ex.shutdown(); }
public static void main(String[] args) {
    ActiveObjectDemo d1 = new ActiveObjectDemo();
    // Предотвращает ConcurrentModificationException:
    List<Future<?>> results =
        new CopyOnWriteArrayList<Future<?>>();
    for(float f = 0.0f; f < 1.0f; f += 0.2f)
        results.add(d1.calculateFloat(f, f));
    for(int i = 0; i < 5; i++)
        results.add(d1.calculateInt(i, i));
    print("All asynch calls made");
    while(results.size() > 0) {
        for(Future<?> f : results)
            if(f.isDone()) {
                try {
                    print(f.get());
                } catch(Exception e) {
                    throw new RuntimeException(e);
                }
                results.remove(f);
            }
        }
    d1.shutdown();
}
} /* Output: (85% match)
All asynch calls made
starting 0.0 + 0.0
starting 0.2 + 0.2
0.0
starting 0.4 + 0.4
0.4
starting 0.6 + 0.6
0.8
starting 0.8 + 0.8
1.2
starting 0 + 0
1.6
starting 1 + 1
0
starting 2 + 2
2
starting 3 + 3
4
starting 4 + 4
6
8
*///:~

```

«Однопоточный исполнитель», полученный вызовом `Executors.newSingleThreadExecutor()`, поддерживает собственную неограниченную блокирующую очередь и имеет только один поток, который извлекает задачи из очереди и выполняет их до

завершения. Все, что нам нужно сделать в `calculateInt()` и `calculateFloat()`, — это отправить вызовом `submit()` новый объект `Callable` в ответ на вызов метода; таким образом вызовы метода преобразуются в сообщения. Тело метода содержится в методе `call()` в анонимном внутреннем классе. Обратите внимание: возвращаемое значение каждого активного объекта представляет собой `Future` с обобщенным параметром, определяющим фактический возвращаемый тип метода. Таким образом, вызов метода возвращает управление почти немедленно, а вызывающая сторона использует `Future` для обнаружения момента завершения задачи и получения фактического возвращаемого значения. Эта схема описывает самый сложный случай, но если вызов не имеет возвращаемого значения, то процесс упрощается.

В `main()` создается контейнер `List<Future<?>>` для хранения объектов `Future`, возвращаемых сообщениями `calculateFloat()` и `calculateInt()`, отправленными активному объекту. Список проверяется вызовом `isDone()` для каждого объекта `Future`, который удаляется из `List` при завершении, после чего происходит обработка его результатов. Стоит заметить, что использование `CopyOnWriteArrayList` устраняет необходимость в копировании `List` для предотвращения исключений `ConcurrentModificationException`.

Чтобы предотвратить формирование логических привязок между потоками, передаваемые при вызове методов активного объекта аргументы должны быть доступными только для чтения, быть другими активными объектами или *отсоединенными* (*disconnected*) объектами (мой термин), то есть объектами, не имеющими связей ни с одной другой задачей (это требование трудно проконтролировать, потому что языковая поддержка для него отсутствует).

Активные объекты обладают следующими характеристиками.

1. Каждый объект имеет собственный рабочий поток.
2. Каждый объект полностью контролирует доступ к своим полям (контроль более жесткий, чем у обычных классов, у которых защита полей не является обязательной).
3. Все взаимодействие между активными объектами происходит в форме сообщений между этими объектами.
4. Все сообщения между активными объектами помещаются в очередь.

Результаты выглядят довольно привлекательно. Поскольку сообщение от одного активного объекта к другому может быть заблокировано только задержкой с его постановкой в очередь — очень короткой и не зависящей от других объектов, отправка сообщения фактически является неблокируемой (худшее, что может произойти, — небольшая задержка). Так как все взаимодействия в системе активных объектов осуществляются только через сообщения, два объекта не могут быть заблокированы при конкуренции за вызов метода другого объекта; таким образом, взаимные блокировки невозможны, что является большим шагом вперед. Так как рабочий поток в активном объекте выполняет только одно сообщение за раз, конкуренция за ресурсы отсутствует, и вам не нужно беспокоиться о синхронизации методов. Синхронизация все равно осуществляется, но на уровне сообщений (при формировании очереди вызовов методов, чтобы они выполнялись по одному).

К сожалению, без прямой поддержки со стороны компилятора это решение получается слишком громоздким. Однако в области активных объектов наблюдается прогресс — как и в области, называемой агентно-ориентированным программированием. *Агенты* (agents) фактически являются активными объектами, но агентно-ориентированные системы также обеспечивают прозрачность между сетями и машинами. Меня не удивит, если агентно-ориентированное программирование со временем станет преемником объектно-ориентированного программирования, потому что оно сочетает объекты с относительно простым параллельным решением.

Дополнительную информацию об активных объектах и агентах можно найти в Интернете. В частности, некоторые концепции активных объектов описаны у Чарльза Энтони Ричарда Хоара в теории взаимодействующих последовательных процессов (CSP, Communicating Sequential Processes).

41. (6) Добавьте в пример `ActiveObjectDemo.java` обработчик сообщения, не имеющий возвращаемого значения, и вызовите его в `main()`.
42. (7) Измените пример `WaxOMatic.java` так, чтобы в нем использовались активные объекты.

Проект. Используйте аннотации и `Javassist` для создания аннотации класса `@Active`, преобразующей целевой класс в активный объект.

Резюме

В этой главе я постарался изложить основы параллельного программирования с применением потоков Java. Из этого материала вы должны были понять следующее.

1. В программе могут выполняться несколько независимых задач.
2. Разработчик должен проанализировать все возможные проблемы при завершении этих задач.
3. Задачи могут взаимодействовать друг с другом через общие ресурсы. Для предотвращения конфликтов используются мьютексы (блокировки).
4. В плохо спроектированной программе могут возникнуть взаимные блокировки.

Очень важно понимать, когда рационально использовать параллельное выполнение, а когда этого делать не стоит. Основные причины его поддержки:

- ☐ возможность управления несколькими подзадачами, одновременное выполнение которых позволяет эффективнее распоряжаться ресурсами компьютера (включая возможность незаметного распределения этих задач по нескольким процессорам);
- ☐ улучшение структуры кода;
- ☐ удобство для конечного пользователя.

Классический пример сбалансированного расходования ресурсов — использование процессора во время ожидания завершения операций ввода-вывода. Улучшение структуры кода обычно проявляется в компьютерном моделировании. Классический пример чуткого пользовательского интерфейса — отслеживание кнопки «Остановить» во время продолжительного процесса загрузки.

Дополнительным преимуществом потоков является то, что они заменяют «тяжелое» переключение контекста процессов (тысячи и более инструкций) «легким» переключением контекста выполнения (около 100 инструкций). Так как все потоки процесса разделяют одно и то же пространство памяти, легкое переключение затрагивает только выполнение программы и локальные переменные. С другой стороны, чередование процессов — тяжелое переключение контекста — требует обновления всего пространства памяти.

Основные недостатки многозадачности:

1. Замедление программы, связанное с ожиданием освобождения заблокированных ресурсов.
2. Дополнительная нагрузка на процессор для управления потоками.
3. Совершенно ненужная сложность, являющаяся следствием нелучших решений при проектировании программы.
4. Нестандартные ситуации, такие как «голодание»¹, динамические тупики, состязания за ресурсы и взаимные блокировки.
5. Непоследовательное поведение на различных платформах. Например, при разработке некоторых примеров для данной книги я обнаружил возникновение ситуаций гонки, быстро проявлявшихся на некоторых компьютерах, но незаметных на других. При разработке на последних вы можете быть неприятно удивлены после распространения программы.

Одно из самых больших затруднений с потоками возникает, когда несколько потоков одновременно пытаются использовать один и тот же ресурс — например, память объекта. Вы должны сделать так, чтобы этого ни в коем случае не произошло. Для этого нужно с умом подойти к возможностям ключевого слова `synchronized`, полезного инструмента языка, требующего полного понимания всех механизмов программы, в противном случае неизбежно возникновение взаимной блокировки.

Вдобавок параллельное программирование сродни искусству. Язык Java существует для того, чтобы вы могли свободно создавать столько объектов, сколько вам нужно для решения вашей задачи — по крайней мере, в теории это так. (Например, создание миллионов объектов для проведения проекционно-разностного анализа вряд ли будет иметь смысл в Java.) Однако оказывается, что количество потоков упирается в определенный «потолок», так как после превышения этой границы потоки становятся неподатливыми. Это критическое число трудно определить, зачастую оно зависит от операционной системы и виртуальной машины Java, значение может находиться где-то в районе сотни, а может исчисляться тысячами. Если для решения своей задачи вам

¹ «Голодание» (иначе отеснение, «подвисание») имеет место, когда один или более потоков процесса блокируются в получении доступа к ресурсу и не могут вследствие этого двигаться дальше. Тупик — это крайняя форма голодания, когда два или более потоков блокируются по условию, которое не может быть удовлетворено. В случае динамического тупика (livelock) два или более процесса руководствуются изменениями в других, уступая свое право на ресурс, из-за чего они «гонятся по кругу» и ни один из них не может выполнять никакой полезной работы. Принципиальное отличие от просто тупика (deadlock) — отсутствие блокировки потоков. — *Примеч. ред.*

требуется «горстка» потоков, это ограничение к вам не относится, но при разработке больших программ оно становится сдерживающим фактором.

Каким бы простым ни казалось параллельное программирование с применением конкретного языка или библиотеки, считайте его своего рода «черной магией»: всегда найдется какая-то неожиданность, которая уколёт вас тогда, когда вы меньше всего этого ожидаете. Этим и интересна задача «обедающих философов»: ее можно настроить так, чтобы взаимная блокировка возникала очень редко, создавая иллюзию нормально работающей программы.

В общем случае параллельное программирование следует применять осторожно и осмотрительно. В ситуации, требующей больших и сложных систем с взаимодействием потоков, лучше воспользоваться специализированным языком вроде *Erlang*. Это один из нескольких функциональных языков, специализированный для параллельного программирования. Возможно, вам удастся использовать такой язык для написания отдельных частей программы, требующих взаимодействий между потоками (если такие взаимодействия достаточно интенсивны и при этом достаточно сложны, чтобы оправдать такое решение).

Дополнительная литература

К сожалению, в области параллельного программирования много ошибочной информации — это лишь подчеркивает, насколько сложна эта тема и как просто решить, будто вы разбираетесь в ней. Я это знаю по собственному опыту; в прошлом я уже неоднократно полагал, что понимаю тонкости многопоточного программирования, и обнаруживал, что я ошибаюсь — и наверняка это еще случится в будущем. Когда вы беретесь за новый документ о параллельном программировании, всегда приходится заниматься самостоятельными изысканиями и выяснять, в какой степени автор понимает (или не понимает) тему. Ниже перечислены некоторые книги, которые я могу с полной уверенностью назвать надежными источниками.

Java Concurrency in Practice, авторы: Брайан Гетц, Тим Пейерлс, Джошуа Блош, Джозеф Боубир, Дэвид Холмс и Дуг Ли (Addison-Wesley, 2006). По сути, это самая авторитетная работа в мире параллельного программирования на Java.

Concurrent Programming in Java, второе издание, автор: Дуг Ли (Addison-Wesley, 2000). Книга была выпущена задолго до появления Java SE5, но большая часть работы Дуга была воплощена в новых библиотеках `java.util.concurrent`, поэтому книга исключительно важна для полноценного понимания проблем параллельности. Автор выходит за рамки параллельности Java и обсуждает современный менталитет параллельности для разных языков и технологий. Местами материал оказывается слишком сложным, и все же книга заслуживает того, чтобы прочитать ее несколько раз (желательно с перерывом в несколько месяцев для усвоения информации). Дуг — один из немногих людей в этом мире, действительно разбирающийся в параллельности, так что вы не пожалеете о потраченном времени.

The Java Language Specification, третье издание (глава 17), авторы: Гослинг, Джой, Стили и Брача (Addison-Wesley, 2005). Техническая спецификация, доступная в виде электронного документа по адресу <http://java.sun.com/docs/books/jls>.

22

Графический интерфейс

Основная рекомендация при проектировании программы — «делайте простое еще легче, а трудное возможным»¹.

В первой версии Java (1.0) библиотека графического пользовательского интерфейса (GUI) создавалась так, чтобы полученный с ее помощью пользовательский интерфейс выглядел хорошо на всех платформах. Однако поставленная цель не была достигнута. Вместо этого результат, получаемый с помощью *пакета разработки абстрактного оконного интерфейса* (Abstract Window Toolkit, AWT) Java 1.0, на всех платформах выглядел одинаково посредственно. Вдобавок его возможности были крайне ограничены: можно было использовать только четыре шрифта, отсутствовал доступ к расширенным возможностям пользовательского интерфейса вашей системы. К тому же модель программирования AWT была запутанной, громоздкой и совсем не объектно-ориентированной. Один из студентов, посещавший мои семинары (он работал в фирме Sun как раз в то время, когда там создавался язык Java), объяснил, в чем тут дело: начальный вариант пакета AWT был задуман, спланирован и реализован фактически за один месяц. Такую «скорострельность» иначе как чудесной не назовешь, но в результате мы еще раз убедились, как важна в программировании фаза анализа и планирования.

Ситуация значительно улучшилась с появлением новой модели событий AWT в выпуске Java версии 1.1. Эта модель использует куда более понятный объектно-ориентированный подход для обработки событий, с ее появлением стало возможно возникновение технологии JavaBeans, которая является основой модели компонентного программирования и ориентирована на визуальные системы быстрой разработки программ. Выпуск Java версии 2 (JDK 1.2) заканчивает трансформацию неудачной библиотеки AWT, фактически замещая ее библиотекой *Java Foundation Classes* (JFC), ее раздел по работе с GUI называется «Swing». Эта библиотека состоит из обширной коллекции компонентов JavaBean, понятных и простых в использовании; они хорошо работают и для самостоятельного построения графического интерфейса, и в средах

¹ Есть еще одна разновидность этого выражения — это так называемый «принцип не изумлять понапрасну» или, что то же самое, «не преподносите пользователю сюрпризов».

визуальной разработки. Правило «3-й ревизии», известное по индустрии производства программного обеспечения (хороший результат появляется только после 3-й ревизии продукта), судя по всему, работает и для языков программирования.

В этой главе описывается только современная библиотека графического пользовательского интерфейса Swing. Предполагается, и не без оснований, что она станет окончательной и основной библиотекой Java для создания таких интерфейсов¹. Если по каким-либо причинам вам потребуется использовать устаревшую библиотеку AWT (при поддержке старых программ или из-за ограничений вашего браузера), начальные сведения о ней вы можете найти в первом издании этой книги (ее можно загрузить с сайта *www.MindView.net*). Стоит отметить, что некоторые компоненты AWT все еще присутствуют в Java, и иногда они бывают нужны.

Обратите внимание на то, что данная глава не является справочником по всем компонентам библиотеки Swing и тем более не содержит списка методов всех описанных классов. Мы ограничимся лишь простым вводным курсом. Библиотека Swing весьма обширна, и в этой главе вы только познакомитесь с ее основными концепциями. Если вам понадобится что-то большее, Swing сможет удовлетворить ваши потребности, если у вас хватит терпения на поиски нужной информации.

Я полагаю, что вы уже загрузили и установили документацию JDK (с сайта *http://java.sun.com*). Тогда вы сможете просмотреть документацию для классов пакета `javax.swing`, чтобы познакомиться со всеми подробностями и методами библиотеки Swing. Вы также можете поискать информацию в Интернете; лучше всего начать с учебника Swing, созданного компанией Sun, по адресу *http://java.sun.com/docs/books/tutorial/uiswing*.

Также существует довольно большое количество книг (еще более толстых), посвященных целиком и полностью библиотеке Swing. Обратитесь к ним, если у вас возникнут трудности или вы захотите изменить стандартное поведение этой библиотеки.

По мере изучения Swing вы узнаете, что:

- ❑ эта библиотека обладает великолепной моделью программирования, лучшей, чем в каких-либо других языках или средах разработки программ. Компоненты JavaBean (которые будут описаны ближе к концу этой главы) образуют «каркас» этой библиотеки;
- ❑ «построители GUI» (среды визуального программирования) учитывают все «требуемые этикетом» аспекты полноценной среды разработки Java-программ. Компоненты JavaBean и модель библиотеки Swing позволяют построителю GUI генерировать код по мере добавления на форму новых компонентов. Это не только значительно увеличивает скорость разработки программ, основанных на графическом интерфейсе, но к тому же позволяет программисту быстро экспериментировать, опробовать различные варианты одной и той же программы и выбирать наилучшее решение;
- ❑ простота и продуманность библиотеки Swing приводят к тому, что даже при использовании автоматизированных средств разработки интерфейса вместо «ручного»

¹ Стоит отметить, что корпорация IBM разработала новую библиотеку пользовательского интерфейса с открытыми исходными текстами для своего текстового редактора Eclipse (*www.Eclipse.org*), альтернативного Swing.

кодирования получается вполне разумный и удобочитаемый код — и это решает проблемы старых автоматизированных визуальных программных сред, которые за просто генерировали совершенно невразумительные результаты.

Библиотека Swing содержит все компоненты, необходимые для создания современного стильного интерфейса, — начиная от кнопок с картинками и заканчивая деревьями и таблицами. Это большая библиотека, однако она разработана с учетом сложности решаемой задачи: если задача проста, простым будет и решение и много кода писать не придется; если же решается сложная задача, то соответственно возрастает объем кода и его сложность.

Вероятно, более всего вам понравится в библиотеке Swing ее «ортогональность». Это значит, что как только вы «ухватили» ее основные идеи и понятия, для вас не будет трудностью применить их в любом месте. При написании примеров данной главы я обычно угадывал имена методов с первого раза, никуда не заглядывая (вот что значит удачно выбранное соглашение об именах!). Несомненно, это признак качественного проектирования библиотеки. Вдобавок вы всегда вправе вставлять компоненты в другие компоненты, и все будет работать правильно.

Управление с клавиатуры встроено во все компоненты: вы можете запустить и использовать приложение Swing без мыши, и это не потребует дополнительного программирования. Очень просто добавить прокрутку — вы просто встраиваете компонент в компонент `JScrollPane` при добавлении на форму. Такие возможности, как всплывающие подсказки, реализуются одной строкой кода.

Для лучшей переносимости библиотека Swing полностью написана на Java.

Библиотека Swing также поддерживает относительно новую возможность — «подключаемый GUI», который позволяет динамически изменять внешний вид и поведение пользовательского интерфейса для пользователей, работающих на различных платформах и операционных системах. В принципе, возможно (хотя и сложно) придумать и реализовать свой вариант такого «внешнего вида и поведения» (look and feel). Некоторые примеры можно найти в Интернете¹.

Несмотря на все свои положительные аспекты, библиотека Swing подойдет не всем; кроме того, она не решила все проблемы пользовательского интерфейса, запланированные ее разработчиками. В конце этой главы мы рассмотрим два альтернативных решения: SWT — финансируемую IBM и разработанную для редактора Eclipse, но свободно распространяемую в виде автономной библиотеки с открытым кодом, и Flex — технологию Macromedia для разработки клиентской части веб-приложений на базе Flash.

Апплет

При первом появлении Java большая часть шумихи по поводу языка была связана с *апплетами* — программами, которые могли пересылаться по Интернету для выполнения в браузере (внутри так называемой *песочницы* по соображениям безопасности).

¹ Мой любимый пример — стиль оформления от Кена Арнольда, при котором окна выглядят так, словно они нарисованы на салфетке (см. <http://napkinlaf.sourceforge.net>).

Предполагалось, что апплеты Java станут следующей стадией эволюции Интернета, а в большинстве первых книг по Java считалось, что читатели интересуются языком прежде всего потому, что собираются писать апплеты.

По разным причинам эта революция так и не наступила. Проблемы были в значительной мере связаны с тем, что на многих машинах не было программного обеспечения Java, необходимого для запуска апплетов, а пользователей вовсе не радовала перспектива загрузки и установки 10-мегабайтного пакета для запуска программы, случайно обнаруженной в Интернете, — а многих она даже пугала. Апплеты Java как механизм доставки приложений на сторону клиента так и не набрали критической массы, и хотя они еще встречаются в отдельных местах, в целом апплеты считаются пережитком прошлого.

Впрочем, это не означает, что апплеты не являются интересной и ценной технологией. Если на машинах пользователей заведомо установлена среда JRE (например, в корпоративной среде), апплеты (или механизм JNLP/Java Web Start, о котором будет рассказано далее в этой главе) могут стать идеальным средством распространения клиентских программ и их автоматического обновления без типичных затрат на распространение и установку нового программного обеспечения.

Введение в технологию апплетов приведено в приложении к книге на сайте *www.MindView.net*.

Основы Swing

Большинство приложений Swing строится на основе простейшего объекта `JFrame`, который создает окно в используемой вами операционной системе. Заголовок окна задается в конструкторе `JFrame`:

```
//: gui/HelloSwing.java
import javax.swing.*;

public class HelloSwing {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Hello Swing");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 100);
        frame.setVisible(true);
    }
} ///:~
```

Метод `setDefaultCloseOperation()` сообщает `JFrame`, что делать, когда пользователь пытается закрыть программу. Константа `EXIT_ON_CLOSE` означает, что программа должна завершиться. Без этого вызова по умолчанию не происходит ничего, так что приложение не закроется.

Метод `setSize()` задает размер окна в пикселах.

Обратите внимание на последнюю строку:

```
frame.setVisible(true);
```

Без нее на экране ничего не появится.

Чтобы приложение стало чуть более интересным, мы добавим в `JFrame` объект `JLabel`:

```
///  
gui/HelloLabel.java  
import javax.swing.*;  
import java.util.concurrent.*;  
  
public class HelloLabel {  
    public static void main(String[] args) throws Exception {  
        JFrame frame = new JFrame("Hello Swing");  
        JLabel label = new JLabel("A Label");  
        frame.add(label);  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        frame.setSize(300, 100);  
        frame.setVisible(true);  
        TimeUnit.SECONDS.sleep(1);  
        label.setText("Hey! This is Different!");  
    }  
} ///  
~
```

Через одну секунду текст надписи `JLabel` изменится. В такой тривиальной программе это забавно и безопасно, но вообще прямые взаимодействия с компонентами GUI из метода `main()` нежелательны. Swing создает отдельный программный поток для получения событий пользовательского интерфейса и обновления экрана. Если вы начнете изменять состояние экрана из других потоков, это может привести к конфликтам и взаимным блокировкам, описанным в главе 21.

Вместо этого другие потоки (такие, как `main()` в нашем примере) должны передавать выполняемые задачи потоку диспетчеризации событий Swing¹. Для этого задача передается методу `SwingUtilities.invokeLater()`, который помещает ее в очередь событий для выполнения (со временем) потоком диспетчеризации событий. В нашем примере это выглядело бы так:

```
///  
gui/SubmitLabelManipulationTask.java  
import javax.swing.*;  
import java.util.concurrent.*;  
  
public class SubmitLabelManipulationTask {  
    public static void main(String[] args) throws Exception {  
        JFrame frame = new JFrame("Hello Swing");  
        final JLabel label = new JLabel("A Label");  
        frame.add(label);  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        frame.setSize(300, 100);  
        frame.setVisible(true);  
        TimeUnit.SECONDS.sleep(1);  
        SwingUtilities.invokeLater(new Runnable() {  
            public void run() {  
                label.setText("Hey! This is Different!");  
            }  
        });  
    }  
} ///  
~
```

Здесь мы уже не работаем с `JLabel` напрямую, а ставим в очередь объект `Runnable`. Поток диспетчеризации событий выполнит все необходимые действия, когда доберется до

¹ Формально поток диспетчеризации событий принадлежит библиотеке AWT.

этой задачи в очереди. А во время выполнения `Runnable` он не делает ничего другого, что исключает возможные конфликты — при условии, что весь код вашей программы использует постановку операций в очередь через `SwingUtilities.invokeLater()`. Это относится и к методу `main()` — вместо вызова методов `Swing`, как это делается в приведенном примере, он должен поставить задачу в очередь событий¹. Итак, правильно написанная программа должна выглядеть примерно так:

```
//: gui/SubmitSwingProgram.java
import javax.swing.*;
import java.util.concurrent.*;

public class SubmitSwingProgram extends JFrame {
    JLabel label;
    public SubmitSwingProgram() {
        super("Hello Swing");
        label = new JLabel("A Label");
        add(label);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 100);
        setVisible(true);
    }
    static SubmitSwingProgram ssp;
    public static void main(String[] args) throws Exception {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() { ssp = new SubmitSwingProgram(); }
        });
        TimeUnit.SECONDS.sleep(1);
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                ssp.label.setText("Hey! This is Different!");
            }
        });
    }
} ///:~
```

Обратите внимание: вызов `sleep()` не включен в конструктор. Если поместить его туда, исходный текст `JLabel` вообще не появится, потому что конструктор не завершится до отработки `sleep()` и вставки новой надписи. Но если `sleep()` находится в конструкторе или любой операции пользовательского интерфейса, это означает, что вы останавливаете поток диспетчеризации событий во время `sleep()`, чего делать не рекомендуется.

1. (1) Измените пример `HelloSwing.java` и продемонстрируйте, что приложение не закроеется без вызова `setDefaultCloseOperation()`.
2. (2) Измените пример `HelloLabel.java` и продемонстрируйте, что добавление надписей осуществляется динамически, добавляя случайное количество надписей.

Вспомогательный класс

Чтобы избавиться от лишнего кода, мы воспользуемся изложенными выше концепциями и создадим вспомогательный класс, который будет использоваться в примерах `Swing` этой главы:

¹ Эта практика была введена в `Java SE5`, поэтому во многих старых программах она не применяется. Это вовсе не указывает на невежество авторов — рекомендуемые практики находятся в постоянном развитии.

```
//: net/mindview/util/SwingConsole.java
// Вспомогательная библиотека для запуска с консоли
// примеров Swing (как апплетов, так и JFrame).
package net.mindview.util;
import javax.swing.*.*;

public class SwingConsole {
    public static void
    run(final JFrame f, final int width, final int height) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                f.setTitle(f.getClass().getSimpleName());
                f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                f.setSize(width, height);
                f.setVisible(true);
            }
        });
    }
} ///:~
```

Возможно, вы сами захотите воспользоваться этим классом, поэтому он включен в библиотеку `net.mindview.util`. Чтобы использовать его, ваше приложение должно выполняться в `JFrame` (как и все примеры в этой главе). Статический метод `run()` назначает заголовком окна простое имя класса `JFrame`.

3. (3) Измените пример `SubmitSwingProgram.java` так, чтобы в нем использовался класс `SwingConsole`.

Создание кнопки

Создать кнопку очень просто: надо вызвать конструктор класса `JButton` с текстом, который должен выводиться на кнопке. Позже вы увидите, что можно проделывать и более интересные вещи (например, разместить изображение на кнопке).

Обычно при создании кнопки в классе определяется новое поле типа `JButton`, чтобы позднее к ней можно было обратиться из программы.

Кнопка `JButton` является графическим компонентом — маленьким окном, — которое автоматически перерисовывается при обновлении главного окна приложения. Это значит, что вам не нужно программно прорисовывать кнопку или любой другой элемент графического интерфейса; вы просто размещаете их на форме, а об остальном они позаботятся сами. Обычно кнопка размещается на форме в конструкторе:

```
//: gui/Button1.java
// Putting buttons on a Swing application.
import javax.swing.*.*;
import java.awt.*.*;
import static net.mindview.util.SwingConsole.*;

public class Button1 extends JFrame {
    private JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    public Button1() {
```

```

        setLayout(new FlowLayout());
        add(b1);
        add(b2);
    }
    public static void main(String[] args) {
        run(new Button1(), 200, 100);
    }
} ///:~

```

Здесь появилось кое-что новое: перед размещением элементов на панели `JFrame` ее размечают с помощью «менеджера расположения» (layout manager) типа `FlowLayout`. С помощью менеджера расположения панель самостоятельно определяет, где будут находиться компоненты на форме. По умолчанию апплеты используют менеджер `BorderLayout`, но в данном случае он нам не подходит, так как по умолчанию после добавления нового компонента он скрывает уже добавленные компоненты. Менеджер расположения `FlowLayout` размещает компоненты на форме равномерно, в направлении слева направо и сверху вниз.

4. (1) Покажите, что без вызова `setLayout()` в `Button1.java` в программе появится только одна кнопка.

Перехват событий

Вы, конечно, заметили, что после компиляции и запуска программы, когда вы нажимаете на кнопки, ровным счетом ничего не происходит. Чтобы в программе выполнялись какие-то действия, необходимо написать дополнительный код, который будет определять, что делать при нажатии кнопки. Основа управляемого событиями программирования, к которому большей частью относится программирование графического интерфейса (GUI), состоит в привязке события к коду, который это событие обрабатывает.

Библиотека `Swing` позволяет четко отделить интерфейс (графические компоненты) от реализации (код, который должен выполняться, когда компонент сообщает о некотором событии). Каждый компонент библиотеки `Swing` способен сообщить обо всех событиях, которые могут с ним произойти, и каждое событие можно обработать индивидуально. Таким образом, если вам не интересно, например, прохождение над вашей кнопкой курсора мыши, вы просто не регистрируете свою заинтересованность в этом событии. Это очень понятный и элегантный способ программирования событий. Как только вы поймете его основы, вы сможете легко использовать компоненты `Swing`, которых до этого даже в глаза не видели, — вообще такая модель событий относится к технологии визуальных компонентов `JavaBean` в целом (эту технологию мы обсудим чуть позже).

Для начала мы сконцентрируемся на главном событии, каковое есть у каждого компонента. В случае с кнопкой `JButton` таким «основным» событием является нажатие этой кнопки. Чтобы зарегистрировать свою заинтересованность в событии нажатия кнопки, вызовите метод класса `JButton` с именем `addActionListener()`. Этот метод ожидает получить аргумент, который должен представлять из себя объект, реализующий интерфейс `ActionListener`. Интерфейс `ActionListener` состоит из одного метода с именем `actionPerformed()`. Значит, для присоединения кода обработки главного события кнопки `JButton` нужно реализовать интерфейс `ActionListener` в своем классе

и зарегистрировать объект этого класса с `JButton` методом `addActionListener()`. Метод `actionPerformed()` данного объекта будет вызван при нажатии на кнопку (этот механизм часто называют *обратным вызовом* (callback)).

Но что должно происходить при нажатии на кнопку? Мы бы хотели увидеть на экране какие-либо изменения, поэтому рассмотрим новый компонент библиотеки Swing — текстовое поле `JTextField`. Это область, где можно ввести текст вручную или в нашем случае вставить его программно. Хотя существует несколько различных способов создания текстового поля `JTextField`, мы выберем тот, который говорит конструктору, какого размера должно быть новое текстовое поле. Как только новое поле размещается на форме, можно изменять его содержимое с помощью метода `setText()` (класс `JTextField` содержит очень много полезных методов, их перечень вы найдете в документации JDK на сайте <http://java.sun.com>). Вот как это будет выглядеть:

```
//: gui/Button2.java
// Реакция на нажатия кнопки.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Button2 extends JFrame {
    private JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    private JTextField txt = new JTextField(10);
    class ButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String name = ((JButton)e.getSource()).getText();
            txt.setText(name);
        }
    }
    private ButtonListener bl = new ButtonListener();
    public Button2() {
        b1.addActionListener(bl);
        b2.addActionListener(bl);
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
        add(txt);
    }
    public static void main(String[] args) {
        run(new Button2(), 200, 150);
    }
} ///:~
```

Процесс создания текстового поля `JTextField` и размещения его на форме схож с процессом создания и размещения кнопок `JButton` и практически одинаков для всех компонентов Swing. Нововведение рассмотренной программы — появление упомянутого интерфейса `ActionListener`, который реализуется классом `ButtonListener`. Аргументом метода `actionPerformed()` этого интерфейса является объект `ActionEvent`, который содержит исчерпывающую информацию о событии и источнике его возникновения. В данном случае я хотел описать нажатую кнопку: метод `getSource()` возвращает объект, где произошло событие, и я предположил (используя преобразование типа), что

этот объект — кнопка `JButton`. Метод `getText()` возвращает текст на кнопке, который помещается в текстовое поле `JTextField`, чтобы показать, что при нажатии кнопки действительно выполняется код обработки события.

В конструкторе также вызывается метод `addActionListener()`, чтобы зарегистрировать объект `ButtonListener` для обеих кнопок программы.

Часто бывает удобнее реагировать на события с помощью анонимного внутреннего класса, который реализует интерфейс `ActionListener`, особенно если учесть, что обычно используется только один экземпляр этого класса для каждого класса слушателя. Программу `Button2.java` можно изменить так, чтобы она использовала анонимный внутренний класс:

```
//: gui/Button2b.java
// Using anonymous inner classes.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Button2b extends JFrame {
    private JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    private JTextField txt = new JTextField(10);
    private ActionListener bl = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String name = ((JButton)e.getSource()).getText();
            txt.setText(name);
        }
    };
    public Button2b() {
        b1.addActionListener(bl);
        b2.addActionListener(bl);
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
        add(txt);
    }
    public static void main(String[] args) {
        run(new Button2b(), 200, 150);
    }
} ///:~
```

В примерах этой книги при обработке событий предпочтение отдается анонимным внутренним классам (где это возможно).

5. (4) Создайте приложение с использованием класса `SwingConsole`. Разместите одно текстовое поле и три кнопки. При нажатии каждой кнопки в текстовом поле должен появляться соответствующий текст.

Текстовые области

Текстовая область `JTextArea` очень похожа на однострочное поле `JTextField`, но она позволяет редактировать несколько строк текста и имеет расширенные возможности.

Особенно полезен метод `append()`; с его помощью стандартный вывод с легкостью переводится в текстовое поле, а с программами становится удобнее работать, поскольку выведенные сообщения не исчезают как раньше — их можно просмотреть и прокрутить назад. В качестве примера рассмотрим программу, заполняющую текстовое поле `JTextArea` данными, которые производит генератор `geography` из главы 17:

```
//: gui/TextArea.java
// Использование элемента управления JTextArea.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

public class TextArea extends JFrame {
    private JButton
        b = new JButton("Add Data"),
        c = new JButton("Clear Data");
    private JTextArea t = new JTextArea(20, 40);
    private Map<String,String> m =
        new HashMap<String,String>();
    public TextArea() {
        // Использовать все данные:
        m.putAll(Countries.capitals());
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                for(Map.Entry me : m.entrySet())
                    t.append(me.getKey() + ": " + me.getValue()+"\n");
            }
        });
        c.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                t.setText("");
            }
        });
        setLayout(new FlowLayout());
        add(new JScrollPane(t));
        add(b);
        add(c);
    }
    public static void main(String[] args) {
        run(new TextArea(), 475, 425);
    }
} ///:~
```

В конструкторе карта `Map` заполняется названиями стран и их столиц. Заметьте, что для обеих кнопок слушатели `ActionListener` создаются и добавляются без определения промежуточных переменных, ведь нам не понадобится снова обращаться к ним во время работы программы. Кнопка `Add Data` форматирует и вставляет в текстовое поле все данные, а кнопка `Clear Data` вызывает метод `setText()`, чтобы удалить весь текст из того же поля.

При добавлении текстового поля `JTextArea` в `JFrame` оно упаковывается в компонент `JScrollPane`, который будет обеспечивать прокрутку текста, когда он перестанет

помещаться на рабочий экран. Это все, что нужно сделать для получения полнофункциональной прокрутки. Разбираясь с тем, как организуется такая же прокрутка в других программных средах, я еще больше оценил простоту и мощност таких компонентов, как панель `JScrollPane`.

6. (7) Преобразуйте пример `strings/TestRegularExpression.java` в интерактивную программу Swing, в которой входная строка вводится в первой области `JTextArea`, а регулярное выражение — в поле `JTextField`. Результат должен выводиться во второй области `JTextArea`.
7. (5) Создайте приложение с использованием класса `SwingConsole` и добавьте все компоненты Swing, содержащие метод `addActionListener()`. (Нужную информацию можно найти в документации JDK на сайте <http://java.sun.com>. Подсказка: найдите имя метода `addActionListener()` в индексе.) Организуйте перехват их событий и выведите для каждого события соответствующее сообщение в текстовом поле.
8. (6) Почти все компоненты Swing являются производными от класса `Component`, содержащего метод `setCursor()`. Найдите этот метод в документации JDK. Создайте приложение и замените указатель мыши одним из стандартных указателей класса `Cursor`.

Управление расположением компонентов

Способ размещения компонентов на форме в Java отличается от способов других систем построения графического интерфейса. Во-первых, все делается прямо в программе; нет никаких «ресурсов», которые управляют расположением компонентов. Во-вторых, расположение компонентов на форме определяется не их абсолютной позицией в пикселах, а «менеджером расположения», который решает, где будут находиться компоненты в зависимости от порядка их добавления на форму вызовом `add()`. Размеры, внешний вид и позиции одних и тех же компонентов будут значительно отличаться при использовании различных менеджеров расположения. Вдобавок менеджеры расположения следят за размером апплета или окна приложения и приспособливают к нему компоненты: при изменении размеров окна размеры, вид и позиции компонентов, содержащихся в этом окне, также изменяются.

Классы `JApplet`, `JFrame`, `JWindow`, `JDialog`, `JPanel` и т. д. могут содержать и отображать графические компоненты `Component`. В классе `Container` имеется метод с именем `setLayout()`, позволяющий выбрать произвольный менеджер компоновки. В этом разделе мы изучим разнообразные менеджеры расположения, помещая в них кнопки (самый простой эксперимент). Следить за нажатиями кнопок мы не будем, в данной ситуации нас интересует лишь показ различных раскладок кнопок в окне.

BorderLayout

По умолчанию `JFrame` использует менеджер расположения `BorderLayout`. Без дополнительных указаний он берет компонент, добавленный методом `add()`, и помещает его в центр панели, растягивая границы компонента к сторонам панели.

Концепция `BorderLayout` выделяет четыре пограничные области и центральную область. Добавляя что-то на панель с типом размещения `BorderLayout`, вы можете использовать перегруженный метод `add()`, которому в первом аргументе передается константа. Это значение может быть одним из следующих.

<code>BorderLayout.NORTH</code>	Вдоль верхней границы
<code>BorderLayout.SOUTH</code>	Вдоль нижней границы
<code>BorderLayout.EAST</code>	По правой стороне
<code>BorderLayout.WEST</code>	По левой стороне
<code>BorderLayout.CENTER</code>	Заполняет центр панели, растягивая компонент к ее границам или до ближайших компонентов

Если не указывать местоположение компонента явно, по умолчанию он вставляется в центр панели.

В следующем примере используется менеджер расположения `BorderLayout`, используемый по умолчанию для `JFrame`:

```
//: gui/BorderLayout1.java
// Демонстрация BorderLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class BorderLayout1 extends JFrame {
    public BorderLayout1() {
        add(BorderLayout.NORTH, new JButton("North"));
        add(BorderLayout.SOUTH, new JButton("South"));
        add(BorderLayout.EAST, new JButton("East"));
        add(BorderLayout.WEST, new JButton("West"));
        add(BorderLayout.CENTER, new JButton("Center"));
    }
    public static void main(String[] args) {
        run(new BorderLayout1(), 300, 250);
    }
} ///:~
```

Для всех позиций, за исключением центральной (`CENTER`), размещаемый на них компонент сжимается в одном измерении и растягивается в другом измерении до максимально возможных значений. В случае размещения по центру компонент растягивается в обоих направлениях, в попытке полностью занять контейнер.

FlowLayout

Этот менеджер расположения просто последовательно «выкладывает» компоненты на форму один за другим, слева направо, пока не заполняется все доступное пространство на одной строке, затем он переходит вниз на следующую строку компонентов и продолжает в том же порядке.

В следующем примере функции конструктора формы вручаются `FlowLayout`, затем на форме размещаются кнопки. Заметьте, что укладчик `FlowLayout` использует

«естественный» размер компонентов. Например, кнопки `JButton` будут подогнаны под прямоугольники их надписей.

```
//: gui/FlowLayout1.java
// Демонстрация последовательного расположения FlowLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class FlowLayout1 extends JFrame {
    public FlowLayout1() {
        setLayout(new FlowLayout());
        for(int i = 0; i < 20; i++)
            add(new JButton("Button " + i));
    }
    public static void main(String[] args) {
        run(new FlowLayout1(), 300, 300);
    }
} ///:~
```

При применении `FlowLayout` все компоненты сжимаются до наименьшего из возможных размеров, и это иногда приводит к неожиданному их поведению. Например, размеры надписи `JLabel` подбираются по размерам текста, и попытка выровнять этот текст, например по правому краю, ни к чему не приведет.

GridLayout

Менеджер расположения `GridLayout` позволяет вам построить таблицу компонентов, и каждый новый компонент помещается в таблицу в направлении слева направо и сверху вниз относительно предыдущего. В конструкторе этого «компонентоукладчика» надо указать необходимое вам количество строк и столбцов, они будут иметь одинаковые размеры:

```
//: gui/GridLayout1.java
// Демонстрация табличного расположения GridLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class GridLayout1 extends JFrame {
    public GridLayout1() {
        setLayout(new GridLayout(7,3));
        for(int i = 0; i < 20; i++)
            add(new JButton("Button " + i));
    }
    public static void main(String[] args) {
        run(new GridLayout1(), 300, 300);
    }
} ///:~
```

Здесь создается таблица для 21 компонента, но добавляется в нее только 20 кнопок. Последняя ячейка таблицы остается незаполненной, поскольку `GridLayout` не пытается «оптимизировать» расположение.

GridBagLayout

Менеджер расположения `GridBagLayout` предоставляет выдающиеся возможности для расположения элементов; он позволяет подробно указать, как должны располагаться области окна, как они должны менять размер при изменении границ окна. Но за все надо платить — `GridBagLayout` является самым сложным менеджером расположения, и понять его нелегко. Основное его предназначение — автоматическое генерирование кода в построителе GUI (построители обычно используют `GridBagLayout`, а не абсолютные позиции в пикселах). Если ваш пользовательский интерфейс настолько изощрен, что прочие укладчики компонентов вас не устраивают, тогда для создания этого интерфейса лучше привлечь автоматизированное средство. Если вам все же хочется погрузиться в хитросплетения макетов `GridBagLayout`, могу порекомендовать вам любую книгу, полностью посвященную библиотеке `Swing`.

Также можно воспользоваться менеджером `TableLayout`, который не входит в библиотеку `Swing`, но может загружаться с сайта <http://java.sun.com>. Этот компонент работает на базе `GridBagLayout` и скрывает большую часть сложностей, поэтому он может серьезно упростить этот способ.

Абсолютное позиционирование

Также возможно указать абсолютное расположение графических компонентов, то есть их позиции в пикселах. Для этого следует:

1. Для используемого вами контейнера (`Container`) вместо менеджера расположения передать методу `setLayout()` ссылку `null`: `setLayout(null)`.
2. Для каждого компонента вызвать метод `setBounds()` или `reshape()` (в зависимости от версии языка), передавая ограничивающий прямоугольник компонента в пикселах. Это также можно сделать в конструкторе или методе `paint()`, все зависит от того, какого результата вы добиваетесь.

Некоторые GUI-построители широко используют описанный подход, но, вообще говоря, это не лучший способ создания графического интерфейса.

BoxLayout

Так как у людей возникало очень много проблем в понимании и применении `GridBagLayout`, в библиотеку `Swing` включили дополнительный менеджер расположения `BoxLayout`, который перенял многие преимущества `GridBagLayout` без его сложностей. Благодаря этому вы можете постоянно использовать его при «ручном» создании графического интерфейса (повторюсь еще раз, если ваш интерфейс становится слишком хитер, пусть автоматизированный построитель GUI позаботится обо всех деталях без вашего участия). Менеджер `BoxLayout` дает вам возможность размещать компоненты и вертикально, и горизонтально, а также позволяет управлять пространством между компонентами. Примеры использования `BoxLayout` приведены в приложении к книге на сайте www.MindView.net.

Лучший вариант?

Библиотека Swing способна на многое; с ее помощью можно создать полнофункциональные программы, написав несколько строк кода. Примеры в этой книге относительно просты, и для лучшего усвоения материала в них стоит создавать пользовательский интерфейс «вручную». Просто совмещая несколько простых режимов расположения, можно получить весьма неплохой интерфейс. Однако в некоторый момент построение графического интерфейса вручную теряет смысл — оно становится слишком сложным, и время, потраченное на эту работу, расходуется неэффективно. Создатели Java и библиотеки Swing ориентировали язык и библиотеки на поддержку автоматизированных средств построения GUI специально ради того, чтобы ускорить процесс разработки программ и сэкономить ваше время. Если вы понимаете, как действуют менеджеры расположения и что лежит в основе системы обработки событий (описанной в следующем разделе), умение размещать компоненты путем кодирования для вас не так важно — пусть это сделает подходящий визуальный инструмент (в конце концов, язык Java разрабатывался для увеличения продуктивности программистов).

Модель событий библиотеки Swing

В модели событий библиотеки Swing любой графический компонент может *инициировать* (initiate) событие («запускать» (fire) событие). Каждое отдельное событие представлено определенным классом. Инициированное событие принимается одним или несколькими «слушателями» (listeners), которые его обрабатывают. Таким образом, источник и место обработки события могут быть разделены. Компоненты Swing обычно используются без изменений, как есть, однако код для обработки их событий каждый раз пишется новый. Это превосходный пример разделения интерфейса и реализации.

Слушатель каждого события — это объект класса, реализующего определенный интерфейс, характерный для всех слушателей этого события. От программиста потребуется лишь создать объект-слушатель и зарегистрировать его в компоненте, который порождает интересующее его событие. Регистрация проводится соответствующим методом `addXXXListener()`, принадлежащим компоненту, иницилирующему событие, где `XXX` — тип обрабатываемого слушателем события. Узнать, какие события можно обработать в компоненте, легко — для этого нужно посмотреть имена его методов `addXXXListener`, и если вы попытаетесь прослушивать недопустимое событие, то получите сообщение об ошибке во время компиляции программы. Чуть позже в этой главе мы увидим, что JavaBeans также используют имена `addXXXListener` для определения того, какие события поддерживает компонент `JavaBean`.

Таким образом, вся логика обработки события будет сосредоточена в классе слушателя. При создании класса слушателя единственным необходимым условием является реализация подходящего интерфейса. Вы можете создавать классы глобальных слушателей, однако в такой ситуации чрезвычайно удобны внутренние классы, и не только потому, что они позволяют сгруппировать слушатели в определенном месте программы и не смешивать вычисления и поддержку пользовательского интерфейса, но и потому, что они обладают ссылкой на внешние классы, что позволяет им легко преодолевать границы классов и подсистем программы.

Все примеры, разобранные нами в данной главе, уже использовали общие принципы модели событий библиотеки Swing, а конец текущего раздела прояснит до конца все частности этой модели.

Типы событий и слушателей

Во всех компонентах библиотеки Swing содержатся методы `addXXXListener()` и `removeXXXListener()`, позволяющие добавлять и удалять слушателей определенных событий (`XXX` обычно представляет аргумент метода: например, `addMouseListener(MyListener m)`). В табл. 22.1 перечислены все основные события, их слушатели, используемые методы, а также компоненты, которые поддерживают эти события (в них имеются соответствующие методы `addXXXListener()` и `removeXXXListener()`). Следует помнить, что данная модель событий проектировалась с расчетом на расширение, поэтому вам могут попасться новые события и их слушатели, не описанные в таблице.

Таблица 22.1. Типы событий и слушателей

Событие, интерфейс слушателя и методы регистрации и удаления слушателей	Компоненты, поддерживающие данное событие
ActionEvent ActionListener addActionListener() removeActionListener()	JButton, JList, JTextField, JMenuItem и производные от них компоненты, включая JCheckBoxMenuItem, JMenu и JPopupMenu
AdjustmentEvent AdjustmentListener addAdjustmentListener() removeAdjustmentListener()	JScrollbar, а также любой произвольный объект, реализующий интерфейс Adjustable
ComponentEvent ComponentListener addComponentListener() removeComponentListener()	*Базовый компонент Component и производные от него, включая JButton, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog и JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea и JTextField
ContainerEvent ContainerListener addContainerListener() removeContainerListener()	Базовый контейнер Container и производные от него классы, включая JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog и JFrame
FocusEvent FocusListener addFocusListener() removeFocusListener()	Базовый компонент Component и его производные*
KeyEvent KeyListener addKeyListener() removeKeyListener()	Базовый компонент Component и его производные*

продолжение ⇨

Таблица 22.1 (продолжение)

Событие, интерфейс слушателя и методы регистрации и удаления слушателей	Компоненты, поддерживающие данное событие
MouseEvent MouseListener addMouseListener() removeMouseListener()	Базовый компонент Component и его производные*
MouseEvent ¹ MouseMotionListener addMouseMotionListener() removeMouseMotionListener()	Базовый компонент Component и его производные*
WindowEvent WindowListener addWindowListener() removeWindowListener()	Окно Window и его производные, включая JDialog, JFileDialog и JFrame
ItemEvent ItemListener addItemListener() removeItemListener()	JCheckBox, JCheckBoxMenuItem, JComboBox, JList и все, что реализует интерфейс ItemSelectable
TextEvent TextListener addTextListener() removeTextListener()	Все производные от базового компонента JTextComponent, включая текстовые поля JTextArea и JTextField

Как видите, каждый компонент поддерживает определенный набор событий. Искать в документации все поддерживаемые компонентом события довольно утомительно. Проще модифицировать программу ShowMethods.java, созданную нами в главе 14, чтобы она показывала все виды слушателей, которые поддерживаются некоторым компонентом.

В главе 14 мы познакомились с *отражением* и использовали этот механизм для получения списка методов определенного класса — полного списка методов или его подмножества, содержащего только те методы, имена которых включают в себя указанные ключевые слова. Отражение автоматически показывает *все* методы класса, не требуя отдельного анализа всех базовых классов данного класса. Таким образом, оно является неплохим подспорьем при программировании: так как имена большинства методов Java-классов являются достаточно подробными и самодокументируемыми, вы можете провести поиск методов, которые содержат интересующие вас слова. Когда вы найдете то, что вас интересует, обращайтесь к документации JDK.

¹ События с именем MouseMotionEvent не существует, хотя кажется, что оно должно быть. И нажатия на кнопки мыши, и перемещение мыши встроены в событие MouseEvent, так что второе его появление в таблице не случайно и не является ошибкой.

Ниже приведена более полезная GUI-версия, приспособленная специально для поиска методов `addXXXListener` в компонентах Swing:

```
//: gui/ShowAddListeners.java
// Вывод списка доступных методов "addXXXListener"
// для любого класса библиотеки Swing.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.regex.*;
import static net.mindview.util.SwingConsole.*;

public class ShowAddListeners extends JFrame {
    private JTextField name = new JTextField(25);
    private JTextArea results = new JTextArea(40, 65);
    private static Pattern addListener =
        Pattern.compile("(add\\w+?Listener\\(\\(.*?\\)\\)");
    private static Pattern qualifier =
        Pattern.compile("\\w+\\.");
    class NameL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String nm = name.getText().trim();
            if(nm.length() == 0) {
                results.setText("No match");
                return;
            }
            Class<?> kind;
            try {
                kind = Class.forName("javax.swing." + nm);
            } catch(ClassNotFoundException ex) {
                results.setText("No match");
                return;
            }
            Method[] methods = kind.getMethods();
            results.setText("");
            for(Method m : methods) {
                Matcher matcher =
                    addListener.matcher(m.toString());
                if(matcher.find())
                    results.append(qualifier.matcher(
                        matcher.group(1)).replaceAll("") + "\n");
            }
        }
    }
    public ShowAddListeners() {
        NameL nameListener = new NameL();
        name.addActionListener(nameListener);
        JPanel top = new JPanel();
        top.add(new JLabel("Swing class name (press Enter):"));
        top.add(name);
        add(BorderLayout.NORTH, top);
        add(new JScrollPane(results));
        // Исходные данные и проверка:
        name.setText("JTextArea");
        nameListener.actionPerformed(
            new ActionEvent("", 0, ""));
    }
}
```

```

public static void main(String[] args) {
    run(new ShowAddListeners(), 500, 400);
}
} ///:~

```

Интересующее вас имя класса вводится в текстовом поле `name`. Результаты извлекаются с помощью регулярных выражений и выводятся в многострочном поле `JTextArea`.

Также стоит заметить, что в графическом интерфейсе нет кнопок и других компонентов, явно запускающих поиск. В них просто нет необходимости, поскольку за содержимым текстового поля `JTextField` следит слушатель `ActionListener`. Когда вы изменяете содержимое текстового поля и нажимаете клавишу `Enter`, список методов немедленно обновляется. Если текстовое поле не пустое, это служит сигналом программе выполнить поиск класса с помощью метода `Class.forName()`. Если имя класса задано неверно, то вызов метода `Class.forName()` заканчивается неудачно, в результате чего возбуждается исключение. Оно перехватывается, а в текстовое поле вместо списка методов выводится сообщение «Нет совпадений». Но если вы правильно ввели в текстовое поле имя класса библиотеки `Swing` (с учетом регистра букв), вызов метода `Class.forName()` закончится удачно, и метод `getMethods()` вернет массив объектов `Method`, представляющих методы.

В примере используются два регулярных выражения. Первое, `addListener`, производит поиск слова `add`, за которым следует произвольное количество символов, с последующим словом `Listener` и списком аргументов в скобках. Заметьте, что это регулярное выражение целиком заключено в скобки, что позволяет при совпадении использовать его как «группу» регулярных выражений. В методе `Name1.actionPerformed()` создается объект `Matcher`, для этого каждый объект `Method` передается в метод `Pattern.matcher()`. Когда для полученного объекта `Matcher` вызывается метод `find()`, значение `true` возвращается только при успешном поиске, и в этом случае вы сможете выбрать первую подходящую по критерию и заключенную в скобки группу, вызвав метод `group(1)`. Полученная строка все еще содержит префиксы полных имен классов, которые удаляются объектом `qualifier` типа `Pattern` так же, как это делалось в примере `ShowMethods.java`.

В конце конструктора в `name` помещается начальное значение и запускается событие `ActionEvent`, чтобы провести тест на конкретных данных.

Эта программа предоставляет удобные средства для изучения возможностей компонентов библиотеки `Swing`. Когда вы узнаете, какие события поддерживает определенный компонент, у вас будет достаточно сведений для обработки этих событий. Вам остается следующее.

1. Взять имя класса события и убрать из него слово `Event`. К остатку добавьте слово `Listener`. Вы получили имя интерфейса, который необходимо реализовать вашему (внутреннему) классу.
2. Реализуйте этот интерфейс и напишите методы для тех событий, которые вы хотели бы обработать. Например, если требуется проследить за перемещениями мыши, напишите код метода `mouseMoved()` интерфейса `MouseMotionListener`. (Вам придется реализовать все остальные методы интерфейса, а их зачастую немало, но есть удобный способ минимизировать работу, и мы его скоро изучим.)

3. Создайте объект класса слушателя, созданного вами во втором пункте. Зарегистрируйте его в подходящем компоненте с помощью метода, имя которого можно узнать следующим образом: добавьте префикс `add` к имени класса слушателя. Например, `addMouseListener()`.

Вот некоторые из интерфейсов слушателей (табл. 22.2).

Таблица 22.2. Некоторые интерфейсы слушателей

Интерфейс слушателя (с адаптером)	Методы интерфейса
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
ComponentListener ComponentAdapter	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent)
ContainerListener ContainerAdapter	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener FocusAdapter	focusGained(FocusEvent) focusLost(FocusEvent)
KeyListener KeyAdapter	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener MouseAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener MouseMotionAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener WindowAdapter	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)

Этот список далеко не полон — отчасти из-за того, что модель событий позволяет легко создавать дополнительные типы событий и ассоциированных слушателей. Поэтому вы будете регулярно сталкиваться с библиотеками, имеющими собственные события; впрочем, пугаться не стоит — знания, полученные в этом разделе, позволят вам быстро в них разобраться.

Адаптеры слушателей упрощают задачу

Глядя на таблицу, вы можете заметить, что некоторые интерфейсы содержат всего один метод. Эти интерфейсы реализуются легко, один метод это все-таки не несколько. Однако интерфейсы слушателей с несколькими методами не настолько просты. Так, для обработки щелчка кнопки мыши (если он уже не обработан за вас, например, кнопкой) вам потребуется написать метод `mouseClicked()`. Но так как `MouseListener` является интерфейсом, вы должны реализовать все его методы, даже если они ничего не делают. Это неудобно и попросту раздражает.

Для решения этой проблемы некоторые (но не все) интерфейсы слушателей, в которых имеется более одного метода, снабжены *адаптерами* (`adapter`), имена которых также приведены в предыдущей таблице. Адаптер реализует все методы интерфейса, оставляя их пустыми (они ничего не делают). При наследовании от адаптера переопределяются только те методы, которые вы намерены изменить. Например, типичный слушатель событий от мыши `MouseListener` выглядит следующим образом:

```
class MyMouseListener extends MouseAdapter {
    public void mouseClicked(MouseEvent e) {
        // Действие по щелчку кнопки мыши...
    }
}
```

Предназначение адаптеров в том и состоит — они значительно облегчают создание слушателей.

Впрочем, при использовании адаптеров следует помнить о возможной ловушке. Предположим, вы пишете код `MouseAdapter`, который вроде бы ничем не отличается от предыдущего:

```
class MyMouseListener extends MouseAdapter {
    public void MouseClicked(MouseEvent e) {
        // Действие по щелчку кнопки мыши...
    }
}
```

Но хотя этот код выглядит нормально, работать он не будет, и поиски причины ошибки могут основательно затянуться. Все компилируется и прекрасно работает, за исключением одного, самого важного — при щелчке код вызываться не будет. Вы нашли ошибку? Она кроется в написании имени метода: `MouseClicked()` вместо `mouseClicked()`. Изменение регистра одной буквы повлекло за собой появление совершенно нового метода. Однако при щелчке кнопки мыши вызывается вовсе не новый метод, а другой, определенный в интерфейсе, поэтому желаемых действий не происходит. Несмотря на отдельные неудобства, интерфейсы гарантируют то, что будут реализованы те методы, какие нужны.

Чтобы гарантировать, что метод действительно переопределяется, стоит воспользоваться встроенной аннотацией `@Override` в приведенном выше коде.

9. (5) Взяв за отправную точку код `ShowAddListeners.java`, создайте программу с полной функциональностью `TypeInfo.ShowMethods.java`.

Отслеживание нескольких событий

Чтобы доказать вам, что события на самом деле запускаются компонентом по мере их возникновения, проведем эксперимент — проследим за несколькими событиями кнопки `JButton` (а не просто за ее нажатиями). Эксперимент также показывает, как создать собственный объект кнопки наследованием от класса `JButton`¹.

Класс `MyButton` определен как внутренний в классе `TrackEvent`, соответственно, он имеет доступ к родительскому окну и вправе изменить содержимое имеющихся в нем текстовых полей, чтобы записать информацию о своем состоянии. Конечно, это решение крайне ограничено, поскольку класс `MyButton` может использоваться только в сочетании с `TrackEvent`. Такого рода код часто называют «сильно связанным»:

```
//: gui/TrackEvent.java
// Вывод информации о событиях по мере их возникновения.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

public class TrackEvent extends JFrame {
    private HashMap<String, JTextField> h =
        new HashMap<String, JTextField>();
    private String[] event = {
        "focusGained", "focusLost", "keyPressed",
        "keyReleased", "keyTyped", "mouseClicked",
        "mouseEntered", "mouseExited", "mousePressed",
        "mouseReleased", "mouseDragged", "mouseMoved"
    };
    private MyButton

        b1 = new MyButton(Color.BLUE, "test1"),
        b2 = new MyButton(Color.RED, "test2");
    class MyButton extends JButton {
        void report(String field, String msg) {
            h.get(field).setText(msg);
        }
        FocusListener fl = new FocusListener() {
            public void focusGained(FocusEvent e) {
                report("focusGained", e.paramString());
            }
            public void focusLost(FocusEvent e) {
                report("focusLost", e.paramString());
            }
        };
        KeyListener kl = new KeyListener() {
            public void keyPressed(KeyEvent e) {
                report("keyPressed", e.paramString());
            }
            public void keyReleased(KeyEvent e) {
                report("keyReleased", e.paramString());
            }
        }
    }
}
```

продолжение ➤

¹ В языке Java версий 1.0/1.1 нельзя было даже с пользой создать класс, производный от класса кнопки. Это был лишь один из многих просчетов в библиотеке AWT.

```

    public void keyTyped(KeyEvent e) {
        report("keyTyped", e paramString());
    }
};
MouseListener ml = new MouseListener() {
    public void mouseClicked(MouseEvent e) {
        report("mouseClicked", e paramString());
    }
    public void mouseEntered(MouseEvent e) {
        report("mouseEntered", e paramString());
    }
    public void mouseExited(MouseEvent e) {
        report("mouseExited", e paramString());
    }
    public void mousePressed(MouseEvent e) {
        report("mousePressed", e paramString());
    }
    public void mouseReleased(MouseEvent e) {
        report("mouseReleased", e paramString());
    }
};
MouseMotionListener mml = new MouseMotionListener() {
    public void mouseDragged(MouseEvent e) {
        report("mouseDragged", e paramString());
    }
    public void mouseMoved(MouseEvent e) {
        report("mouseMoved", e paramString());
    }
};
public MyButton(Color color, String label) {
    super(label);
    setBackground(color);
    addFocusListener(fl);
    addKeyListener(kl);
    addMouseListener(ml);
    addMouseMotionListener(mml);
}
}
public TrackEvent() {
    setLayout(new GridLayout(event.length + 1, 2));
    for(String evt : event) {
        JTextField t = new JTextField();
        t.setEditable(false);
        add(new JLabel(evt, JLabel.RIGHT));
        add(t);
        h.put(evt, t);
    }
    add(b1);
    add(b2);
}
public static void main(String[] args) {
    run(new TrackEvent(), 700, 500);
}
} ///:~

```

В конструкторе класса `MyButton` цвет фона кнопки устанавливается вызовом `setBackground()`. Слушатели всех событий регистрируются с помощью простых вызовов методов.

Класс `TrackEvent` содержит таблицу `HashMap`, строки которой описывают события, и текстовые поля `TextField`, в которые вставляется информация о событии. Конечно, можно было бы не помещать их в таблицу, а создать все по отдельности, в качестве членов класса, но я думаю, вы согласитесь с тем, что в данном случае «табличный» подход упрощает код и делает его элегантнее. Особенно просто становится расширять программу новыми событиями — вы добавляете еще одну строчку в массив `event`, а все остальное происходит автоматически.

Методу `report()` при вызове передаются два аргумента: имя события и строка с параметрами этого события. Он использует таблицу `HashMap` из внешнего класса для поиска содержимого `TextField`, ассоциированного с названием данного события, и отображает строку с параметрами события.

С этим примером забавно поэкспериментировать, поскольку вы мгновенно видите, какие события запускаются в вашей программе и что с ними происходит.

10. (6) Создайте приложение на базе `SwingConsole`, с кнопкой (`JButton`) и текстовым полем (`TextField`). Напишите и присоедините подходящего слушателя событий, так чтобы при получении кнопкой фокуса все вводимые в это время с клавиатуры символы появлялись в текстовом поле.
11. (4) Унаследуйте новый тип кнопки от класса `JButton`. Каждый раз при нажатии кнопки она должна менять свой цвет на другой, который выбирается случайным образом. Чтобы узнать, как генерировать случайный цвет, найдите пример `ColorBoxes.java` (см. далее в этой главе).
12. (4) Организуйте отслеживание еще одного события в программе `TrackEvent.java`, для чего напишите код, обрабатывающий это событие. Самостоятельно выберите тип отслеживаемого события.

Компоненты Swing

Теперь, когда вы знаете, что такое менеджеры расположения и модель событий, пришло время разобраться с конкретными компонентами библиотеки `Swing`. Этот раздел кратко описывает основные компоненты библиотеки и их важнейшие свойства, которые вы будете использовать чаще всего. Все примеры сделаны достаточно небольшими, чтобы вы могли легко скопировать из них код и воспользоваться им в вашей собственной программе.

Помните, что:

1. Вы сможете рассмотреть исходный текст этих примеров прямо во время их запуска в браузере, так как он включен в HTML-страницу вместе с апплетом (эти страницы можно найти на www.MindView.net).
2. Документация `JDK` на сайте <http://java.sun.com> содержит исчерпывающую информацию обо всех классах библиотеки `Swing` и их методах (здесь же рассказано только о некоторых из них).
3. Для компонентов применена удобная система именования, поэтому можно просто угадать, как написать имя и зарегистрировать слушателя определенного события. В изучении отдельных компонентов вам поможет программа `ShowAddListeners.java`.

Когда ваш графический интерфейс станет сложным, используйте автоматизированный построитель GUI.

КНОПКИ

Библиотека Swing включает в себя несколько различных видов кнопок. Все кнопки, флажки, переключатели, даже пункты меню унаследованы от базового класса `AbstractButton` (который, раз уж от него наследуются даже пункты меню, лучше было бы назвать *AbstractSelector* или еще более обобщенно и абстрактно). Но до меню дело дойдет несколько позже, а пока следующий пример демонстрирует различные виды кнопок:

```

/: gui/Buttons.java
/ Различные кнопки Swing.
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.basic.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Buttons extends JFrame {
    private JButton jb = new JButton("JButton");
    private BasicArrowButton
        up = new BasicArrowButton(BasicArrowButton.NORTH),
        down = new BasicArrowButton(BasicArrowButton.SOUTH),
        right = new BasicArrowButton(BasicArrowButton.EAST),
        left = new BasicArrowButton(BasicArrowButton.WEST);
    public Buttons() {
        setLayout(new FlowLayout());
        add(jb);
        add(new JToggleButton("JToggleButton"));
        add(new JCheckBox("JCheckBox"));
        add(new JRadioButton("JRadioButton"));
        JPanel jp = new JPanel();
        jp.setBorder(new TitledBorder("Directions"));
        jp.add(up);
        jp.add(down);
        jp.add(left);
        jp.add(right);
        add(jp);
    }
    public static void main(String[] args) {
        run(new Buttons(), 350, 200);
    }
}
///:~

```

Программа начинает с кнопки со стрелками `BasicArrowButton` из пакета `javax.swing.plaf.basic`, затем переходит к остальным кнопкам. Запустив этот пример, вы увидите, что кнопка-переключатель `JToggleButton` сохраняет свое состояние, «включено» или «выключено». Флажки `JCheckBox` и переключатели `JRadioButton` похожи друг на друга, на их можно щелкнуть и задать требуемый статус (они унаследованы от кнопки-переключателя `JToggleButton`).

ГРУППЫ КНОПОК

Если вы хотите, чтобы переключатели использовались для определения взаимоисключающих состояний, вам нужно создать соответствующую «группу кнопок» `ButtonGroup`.

Но, как показывает следующий пример, в такую группу можно добавить любую кнопку, унаследованную от `AbstractButton`.

Чтобы избежать повторения одного и того же кода, следующий пример использует отражение для формирования групп, состоящих из различных типов кнопок. Делается это в методе `makeBPanel()`, который создает группу кнопок на панели `JPanel`. Вторым аргументом метода `makeBPanel()` — массив строк (`String`). Для каждой строки создается кнопка класса, указанного в первом аргументе, после чего она добавляется на панель:

```
//: gui/ButtonGroups.java
// Использование отражения для создания групп кнопок,
// производных от базовой кнопки AbstractButton.
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.lang.reflect.*;
import static net.mindview.util.SwingConsole.*;

public class ButtonGroups extends JFrame {
    private static String[] ids = {
        "June", "Ward", "Beaver", "Wally", "Eddie", "Lumpy"
    };
    static JPanel makeBPanel(
        Class<? extends AbstractButton> kind, String[] ids) {
        ButtonGroup bg = new ButtonGroup();
        JPanel jp = new JPanel();
        String title = kind.getName();
        title = title.substring(title.lastIndexOf('.') + 1);
        jp.setBorder(new TitledBorder(title));
        for(String id : ids) {
            AbstractButton ab = new JButton("failed");
            try {
                // Получение динамического конструктора
                // с аргументом типа String:
                Constructor ctor =
                    kind.getConstructor(String.class);
                // Создание нового объекта:
                ab = (AbstractButton)ctor.newInstance(id);
            } catch(Exception ex) {
                System.err.println("can't create " + kind);
            }
            bg.add(ab);
            jp.add(ab);
        }
        return jp;
    }
    public ButtonGroups() {
        setLayout(new FlowLayout());
        add(makeBPanel(JButton.class, ids));
        add(makeBPanel(JToggleButton.class, ids));
        add(makeBPanel(JCheckBox.class, ids));
        add(makeBPanel(JRadioButton.class, ids));
    }
    public static void main(String[] args) {
        run(new ButtonGroups(), 500, 350);
    }
} //::~~
```

Заголовок рамки панели `JPanel` образуется из имени класса, из которого убирается информация о пути и пакетах. Каждая новая кнопка `AbstractButton` сначала инициализируется как обычная кнопка `JButton` с надписью «Failed»; таким образом, при возникновении исключения, даже если вы не увидите сообщение об ошибке, надпись на кнопке подскажет вам, что случилось. Метод `getConstructor()` возвращает объект `Constructor`, представляющий собой конструктор с аргументами, типы которых указаны в массиве объектов `Class`, переданных в метод `getConstructor()`. После этого остается вызвать метод `newInstance()` и передать ему массив объектов с аргументами конструктора — в нашем случае это строка из массива `ids`.

Для получения набора кнопок, работающих по принципу взаимоисключающих состояний, вы формируете группу кнопок `ButtonGroup` и добавляете в нее все кнопки, которые должны иметь соответствующее поведение. Запустив программу, вы увидите, что взаимоисключающее поведение проявляют любые кнопки, кроме обычных `JButton`.

Значки

Вы можете использовать значок `Icon` внутри `JLabel` или в компонентах, унаследованных от базовой кнопки `AbstractButton` (включая `JButton`, `JCheckBox`, `JRadioButton`, а также все виды меню (`JMenuItem`)). Как будет показано в одном из будущих примеров, это делается достаточно тривиально. Следующий пример исследует всевозможные способы применения значков `Icon` в кнопках и их производных.

Для значков годятся любые файлы в формате GIF; картинки в данном примере являются частью комплектации этой книги, их можно найти на сайте *www.MindView.net*. Чтобы открыть файл с изображением и загрузить его, нужно просто создать объект `ImageIcon`, передав его конструктору имя файла с изображением. После этого полученный значок (`Icon`) можно использовать в программе.

```
//: gui/Faces.java
// Поведение значков в кнопках JButton.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Faces extends JFrame {
    private static Icon[] faces;
    private JButton jb, jb2 = new JButton("Disable");
    private boolean mad = false;
    public Faces() {
        faces = new Icon[]{
            new ImageIcon(getClass().getResource("Face0.gif")),
            new ImageIcon(getClass().getResource("Face1.gif")),
            new ImageIcon(getClass().getResource("Face2.gif")),
            new ImageIcon(getClass().getResource("Face3.gif")),
            new ImageIcon(getClass().getResource("Face4.gif")),
        };
        jb = new JButton("JButton", faces[3]);
        setLayout(new FlowLayout());
        jb.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if(mad) {
```

```

        jb.setIcon(faces[3]);
        mad = false;
    } else {
        jb.setIcon(faces[0]);
        mad = true;
    }
    jb.setVerticalAlignment(JButton.TOP);
    jb.setHorizontalAlignment(JButton.LEFT);
}
});
jb.setRolloverEnabled(true);
jb.setRolloverIcon(faces[1]);
jb.setPressedIcon(faces[2]);
jb.setDisabledIcon(faces[4]);
jb.setToolTipText("Yow!");
add(jb);
jb2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if(jb.isEnabled()) {
            jb.setEnabled(false);
            jb2.setText("Enable");
        } else {
            jb.setEnabled(true);
            jb2.setText("Disable");
        }
    }
});
add(jb2);
}
public static void main(String[] args) {
    run(new Faces(), 250, 125);
}
} ///:~

```

Значки (Icon) можно использовать во многих конструкторах различных компонентов библиотеки Swing, однако не запрещается установить его и после создания кнопки — для этого предназначен метод `setIcon()`. В этом примере также показано, как на кнопку `JButton` (как и для любой кнопки `AbstractButton`) помещаются различные значки, которые появляются при нажатии кнопки, ее отключении, проведении над кнопкой курсором мыши. Вы увидите, что подобная анимация весьма оживляет кнопку.

Подсказки

В предыдущем примере к кнопке была добавлена «подсказка» (tool tip). Практически все классы, которые послужат основой вашего графического интерфейса, унаследованы от базового компонента библиотеки Swing `JComponent`, в котором имеется метод с названием `setToolTipText(String)`. Поэтому практически для всего, что вы помещаете на форму, можно вызвать этот метод, чтобы подключить соответствующую подсказку (`jc` — ссылка на компонент класса, производного от `JComponent`):

```
jc.setToolTipText("Это подсказка");
```

Здесь, когда пользователь задержит курсор мыши над компонентом `JComponent` на определенный период времени, около курсора мыши всплывет небольшой прямоугольник с текстом подсказки.

Текстовые поля

Этот пример показывает дополнительные возможности текстовых полей `TextField`:

```

//: gui/TextFields.java
// Text fields and Java events.
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class TextFields extends JFrame {
    private JButton
        b1 = new JButton("Get Text"),
        b2 = new JButton("Set Text");
    private JTextField
        t1 = new JTextField(30),
        t2 = new JTextField(30),
        t3 = new JTextField(30);
    private String s = "";
    private UpperCaseDocument ucd = new UpperCaseDocument();
    public TextFields() {
        t1.setDocument(ucd);
        ucd.addDocumentListener(new T1());
        b1.addActionListener(new B1());
        b2.addActionListener(new B2());
        t1.addActionListener(new T1A());
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
        add(t1);
        add(t2);
        add(t3);
    }
    class T1 implements DocumentListener {
        public void changedUpdate(DocumentEvent e) {}
        public void insertUpdate(DocumentEvent e) {
            t2.setText(t1.getText());
            t3.setText("Text: " + t1.getText());
        }
        public void removeUpdate(DocumentEvent e) {
            t2.setText(t1.getText());
        }
    }
    class T1A implements ActionListener {
        private int count = 0;
        public void actionPerformed(ActionEvent e) {
            t3.setText("t1 Action Event " + count++);
        }
    }
    class B1 implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            if(t1.getSelectedText() == null)
                s = t1.getText();
            else
                s = t1.getSelectedText();
        }
    }
}

```

```

        t1.setEditable(true);
    }
}
class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        ucd.setUpperCase(false);
        t1.setText("Inserted by Button 2: " + s);
        ucd.setUpperCase(true);
        t1.setEditable(false);
    }
}
public static void main(String[] args) {
    run(new TextFields(), 375, 200);
}
}

class UpperCaseDocument extends PlainDocument {
    private boolean upperCase = true;
    public void setUpperCase(boolean flag) {
        upperCase = flag;
    }
    public void
    insertString(int offset, String str, AttributeSet attSet)
    throws BadLocationException {
        if(upperCase) str = str.toUpperCase();
        super.insertString(offset, str, attSet);
    }
}
} ///:~

```

Текстовое поле `t3` служит для отчета о действиях в текстовом поле `t1`. Вы увидите, что «слушатель действий» поля `t1` срабатывает только при нажатии в этом поле клавиши `Enter`.

К текстовому полю `t1` присоединены несколько слушателей. Слушатель `T1` реализует интерфейс `DocumentListener` и откликается на любые изменения в «документе» (в нашем случае это содержимое текстового поля `JTextField`). Он автоматически копирует текст из поля `t1` в поле `t2`. Вдобавок поле `t1` использует в качестве документа класс, производный от базового класса документа `PlainDocument`, с именем `UpperCaseDocument`, который при вводе преобразует все символы в верхний регистр. Он автоматически обнаруживает нажатия клавиши `BackSpace` и производит удаление символов, изменяет позицию курсора и вообще делает все так, как и ожидает пользователь.

- 13.** (3) Измените программу `TextFields.java` так, чтобы символы в текстовом поле `t2` оставались в том регистре, в котором они были напечатаны, а не преобразовывались автоматически к верхнему регистру.

Рамки

Класс `JComponent` содержит метод с именем `setBorder()`, который позволяет вам разместить на компонентах различные, порой весьма любопытные окантовки. Следующий пример демонстрирует несколько различных рамок, предоставляемых библиотекой `Swing`, используя для этого метод `showBorders()`, который создает панель `JPanel` и отображает на ней рамку. Пример также использует `RTTI` для определения имени класса

рамки (из которого исключается информация пути), которое затем размещается в середине каждой панели JPanel:

```
//: gui/Borders.java
// Разные варианты рамок в Swing.
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Borders extends JFrame {
    static JPanel showBorder(Border b) {
        JPanel jp = new JPanel();
        jp.setLayout(new BorderLayout());
        String nm = b.getClass().toString();
        nm = nm.substring(nm.lastIndexOf('.') + 1);
        jp.add(new JLabel(nm, JLabel.CENTER),
            BorderLayout.CENTER);
        jp.setBorder(b);
        return jp;
    }
    public Borders() {
        setLayout(new GridLayout(2,4));
        add(showBorder(new TitledBorder("Title")));
        add(showBorder(new EtchedBorder()));
        add(showBorder(new LineBorder(Color.BLUE)));
        add(showBorder(
            new MatteBorder(5,5,30,30,Color.GREEN)));
        add(showBorder(
            new BevelBorder(BevelBorder.RAISED)));
        add(showBorder(
            new SoftBevelBorder(BevelBorder.LOWERED)));
        add(showBorder(new CompoundBorder(
            new EtchedBorder(),
            new LineBorder(Color.RED))));
    }
    public static void main(String[] args) {
        run(new Borders(), 500, 300);
    }
} ///:~
```

Вы также можете создать свои собственные рамки и использовать их в кнопках, надписях и т. д. — в любых компонентах, производных от JComponent.

Мини-редактор

Управляющий элемент JTextPane предоставляет обширные возможности по редактированию текстов, не требуя от программиста больших усилий. Следующий пример демонстрирует простейший вариант использования этого компонента, в котором не используется большая часть его возможностей:

```
//: gui/TextPane.java
// JTextPane - миниатюрный редактор текста.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```



```
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

public class TextPane extends JFrame {
    private JButton b = new JButton("Add Text");
    private JTextPane tp = new JTextPane();
    private static Generator sg =
        new RandomGenerator.String(7);
    public TextPane() {
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                for(int i = 1; i < 10; i++)
                    tp.setText(tp.getText() + sg.next() + "\n");
            }
        });
        add(new JScrollPane(tp));
        add(BorderLayout.SOUTH, b);
    }
    public static void main(String[] args) {
        run(new TextPane(), 475, 425);
    }
} ///:~
```

Единственная кнопка просто добавляет в текст случайно выбранные символы. Предназначение компонента `JTextPane` — редактирование текста «на месте», поэтому в нем нет метода для добавления текста `append()`. В нашем примере (который не использует возможности компонента `JTextPane` даже в малой степени) программа получает уже существующий текст, изменяет его и помещает результат обратно в окно редактирования с помощью метода `setText()`.

Элементы добавляются в `JFrame` в режиме по умолчанию `BorderLayout`. Компонент `JTextPane` добавляется (внутри `JScrollPane`) без указания области, поэтому он просто будет помещен в центр контейнера и растянут к его границам. Компонент `JButton` добавляется в «южную» область (`SOUTH`), то есть в нижней части контейнера.

Обратите внимание на такие встроенные возможности редактора `JTextPane`, как автоматический перевод текста на новую строку. Полный перечень свойств этого компонента можно найти в документации JDK.

- 14. (2)** Измените пример `TextPane.java` так, чтобы вместо `JTextPane` в нем использовался компонент `JTextArea`.

Флажки

Флажок позволяет произвести выбор из двух возможных состояний («включено/выключено»), он состоит из небольшого прямоугольника и сопроводительного текста. В прямоугольник обычно помещается маленький крестик «х» (или какой-либо другой индикатор статуса), или он остается незаполненным, все зависит от состояния флажка.

Флажок `JCheckBox`, как правило, создается конструктором, которому передается текст надписи. Вы можете узнать состояние флажка и установить его, а также считать или изменить текст надписи уже после создания флажка.

Когда пользователь изменяет состояние флажка `JCheckBox`, происходит событие, которое можно обработать точно так же, как и в случае с обычной кнопкой `JButton`, то есть

с помощью слушателя `ActionListener`. Следующий пример использует многострочное текстовое поле `JTextArea` для вывода состояния всех флажков:

```
//: gui/CheckBoxes.java
// Использование флажков JCheckBox.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class CheckBoxes extends JFrame {
    private JTextArea t = new JTextArea(6, 15);
    private JCheckBox
        cb1 = new JCheckBox("Check Box 1"),
        cb2 = new JCheckBox("Check Box 2"),
        cb3 = new JCheckBox("Check Box 3");
    public CheckBoxes() {
        cb1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                trace("1", cb1);
            }
        });
        cb2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                trace("2", cb2);
            }
        });
        cb3.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                trace("3", cb3);
            }
        });
        setLayout(new FlowLayout());
        add(new JScrollPane(t));
        add(cb1);
        add(cb2);
        add(cb3);
    }
    private void trace(String b, JCheckBox cb) {
        if(cb.isSelected())
            t.append("Box " + b + " Set\n");
        else
            t.append("Box " + b + " Cleared\n");
    }
    public static void main(String[] args) {
        run(new CheckBoxes(), 200, 300);
    }
} ///:~
```

Метод `trace()` отправляет имя выбранного флажка `JCheckBox` и его текущее состояние в текстовое поле `JTextArea`, вызывая метод `append()`. Таким образом строится список всех флажков с информацией об их состоянии.

- 15.** (5) Добавьте флажок к приложению, созданному в упражнении 5, обработайте его событие и поместите в текстовое поле другой текст.

Переключатели

Концепция «радиокнопок» (переключателей) пришла в программирование GUI из старых, еще не электронных радиоприемников, где при нажатии одной кнопки остальные «выскакивали». Интерфейсные переключатели тоже позволяют выбрать один вариант из многих.

Чтобы создать связанную группу кнопок переключателя `JRadioButton`, надо объединить их в группу (`ButtonGroup`) (на форме может существовать произвольное количество таких групп). Одна из кнопок изначально может быть «включена» (передачей `true` во втором аргументе конструктора). Если вы попытаетесь включить несколько кнопок сразу, то только та кнопка, что была указана последней, сохранит это состояние. Все остальные автоматически отключатся.

Вот простой пример использования переключателей, в котором для перехвата событий используются объекты `ActionListener`:

```
//: gui/RadioButton.java
// Использование переключателей JRadioButton.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class RadioButtons extends JFrame {
    private JTextField t = new JTextField(15);
    private ButtonGroup g = new ButtonGroup();
    private JRadioButton
        rb1 = new JRadioButton("one", false),
        rb2 = new JRadioButton("two", false),
        rb3 = new JRadioButton("three", false);
    private ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            t.setText("Radio button " +
                ((JRadioButton)e.getSource()).getText());
        }
    };
    public RadioButtons() {
        rb1.addActionListener(al);
        rb2.addActionListener(al);
        rb3.addActionListener(al);
        g.add(rb1); g.add(rb2); g.add(rb3);
        t.setEditable(false);
        setLayout(new FlowLayout());
        add(t);
        add(rb1);
        add(rb2);
        add(rb3);
    }
    public static void main(String[] args) {
        run(new RadioButtons(), 200, 125);
    }
} ///:~
```

Для вывода информации о состоянии переключателей используется текстовое поле. Редактирование этого поля запрещено, поскольку в данном случае оно используется для вывода информации, а не для ввода. Это одна из альтернатив для надписи `JLabel`.

Раскрывающиеся списки

Как и переключатели, раскрывающийся список предназначен для того, чтобы пользователь смог выбрать один элемент из группы возможных значений. Однако данный способ более «компактен», к тому же здесь легче осуществить динамическое изменение элементов списка, не пугая пользователя. (Переключатели тоже можно изменять динамически, однако такие метаморфозы раздражают пользователя.)

По умолчанию `JComboBox` отличается от аналогичных списков ОС Windows, которые позволяют вам выбирать из списка *или* набирать свое собственное значение. Чтобы получить список такого типа, необходимо вызвать метод `setEditable()`. В остальном список `JComboBox` работает так же, и он тоже позволяет выбрать один и только один элемент списка. В следующем примере созданный раскрывающийся список заполняется несколькими пунктами, а затем при нажатии кнопки к ним присоединяются новые пункты.

```

//: gui/ComboBoxes.java
// Использование раскрывающихся списков.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class ComboBoxes extends JFrame {
    private String[] description = {
        "Ebullient", "Obtuse", "Recalcitrant", "Brilliant",
        "Somnescent", "Timorous", "Florid", "Putrescent"
    };
    private JTextField t = new JTextField(15);
    private JComboBox c = new JComboBox();
    private JButton b = new JButton("Add items");
    private int count = 0;
    public ComboBoxes() {
        for(int i = 0; i < 4; i++)
            c.addItem(description[count++]);
        t.setEditable(false);
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if(count < description.length)
                    c.addItem(description[count++]);
            }
        });
        c.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                t.setText("index: " + c.getSelectedIndex() + " " +
                    ((JComboBox)e.getSource()).getSelectedItem());
            }
        });
        setLayout(new FlowLayout());
        add(t);
        add(c);
        add(b);
    }
    public static void main(String[] args) {
        run(new ComboBoxes(), 200, 175);
    }
} ///:~

```

В текстовом поле `JTextField` выводится индекс выделения (порядковый номер текущего выделенного элемента) и его текст.

Списки

Обычные списки довольно существенно отличаются от только что рассмотренных раскрывающихся списков, и не только внешне. В то время как раскрывающийся список (`JComboBox`) становится виден целиком только при выборе его элемента, просто список `JList` постоянно занимает на экране определенное количество строк и не изменяется. Если вам понадобятся в работе результаты выбора, вызовите метод `getSelectedValues()`, который возвращает массив объектов `String` с текстами выбранных элементов списка.

Список `JList` позволяет выбирать несколько элементов одновременно: если вы щелкнете кнопкой мыши на нужных вам элементах, удерживая клавишу `Ctrl`, то выделены будут они все. Если выделить один элемент, а затем, удерживая клавишу `Shift`, щелкнуть на другом элементе, выделяются все элементы между этими двумя. Для исключения элемента из выделенной группы щелкните на нем, удерживая клавишу `Ctrl`.

```
//: gui/List.java
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class List extends JFrame {
    private String[] flavors = {
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
        "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };
    private DefaultListModel lItems = new DefaultListModel();
    private JList lst = new JList(lItems);
    private JTextArea t =
        new JTextArea(flavors.length, 20);
    private JButton b = new JButton("Add Item");
    private ActionListener bl = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            if(count < flavors.length) {
                lItems.add(0, flavors[count++]);
            } else {
                // Отключить, в список List добавлены
                // все необходимые элементы
                b.setEnabled(false);
            }
        }
    };
    private ListSelectionListener ll =
        new ListSelectionListener() {
            public void valueChanged(ListSelectionEvent e) {
                if(e.getValueIsAdjusting()) return;
                t.setText("");
                for(Object item : lst.getSelectedValues())
```

продолжение ➤

```

        t.append(item + "\n");
    }
};
private int count = 0;
public List() {
    t.setEditable(false);
    setLayout(new FlowLayout());
    // Создание рамки для компонентов:
    Border brd = BorderFactory.createMatteBorder(
        1, 1, 2, 2, Color.BLACK);
    lst.setBorder(brd);
    t.setBorder(brd);
    // Добавляем в список первые четыре элемента
    for(int i = 0; i < 4; i++)
        lItems.addElement(flavors[count++]);
    add(t);
    add(lst);
    add(b);
    // Регистрация слушателей событий
    lst.addListSelectionListener(l1);
    b.addActionListener(bl);
}
public static void main(String[] args) {
    run(new List(), 250, 375);
}
} ///:~

```

Список также был заключен в рамку.

Если вам нужно просто разместить массив строк (`String`) в списке `JList`, все можно сделать намного проще: передайте массив конструктору класса `JList`, и список элементов будет создан автоматически. Единственная причина присутствия в этом примере «модели списка» — необходимость изменения содержимого списка во время выполнения программы.

В списках `JList` отсутствует автоматическая прямая поддержка прокрутки. Конечно, проблема решается очень просто: упакуйте список `JList` в панель прокрутки `JScrollPane`, а все остальное будет сделано автоматически.

- 16.** (5) Упростите пример `List.java`: передайте массив конструктору и устраните динамическое включение элементов в список.

Панель вкладок

Панель вкладок `JTabbedPane` позволяет создать диалоговое окно с набором вкладок, у которого к одной из сторон прижат набор «корешков» вкладок. Чтобы переключиться к содержимому другого диалогового окна, достаточно щелкнуть на корешке нужной вкладки.

```

//: gui/TabbedPane1.java
// Демонстрация панели вкладок.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

```

```

public class TabbedPane1 extends JFrame {
    private String[] flavors = {
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
        "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };
    private JTabbedPane tabs = new JTabbedPane();
    private JTextField txt = new JTextField(20);
    public TabbedPane1() {
        int i = 0;
        for(String flavor : flavors)
            tabs.addTab(flavors[i],
                new JButton("Tabbed pane " + i++));
        tabs.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                txt.setText("Tab selected: " +
                    tabs.getSelectedIndex());
            }
        });
        add(BorderLayout.SOUTH, txt);
        add(tabs);
    }
    public static void main(String[] args) {
        run(new TabbedPane1(), 400, 250);
    }
} ///:~

```

Запустив программу, вы увидите, что панель вкладок `JTabbedPane` автоматически выстраивает вкладки в несколько рядов, если для них не хватает места в одном ряду. Чтобы убедиться в этом, измените размеры окна при запуске программы из командной строки.

Окна сообщений

Оконные среды часто содержат стандартный набор окон сообщений, который позволяет вам быстро информировать пользователя о чем-то или получать от него информацию. В библиотеке Swing эти окна сообщений создаются классом `JOptionPane`. С его помощью можно получить большое количество разнообразных информационных окон (некоторые из них весьма впечатляют), но чаще всего используются окна с сообщениями программы и окна с подтверждениями выбора, которые вызываются посредством статических (`static`) методов `JOptionPane.showMessageDialog()` и `JOptionPane.showConfirmDialog()`. Следующий пример демонстрирует некоторые из окон сообщений, предоставляемых классом `JOptionPane`:

```

//: gui/MessageBoxes.java
// Демонстрация возможностей JOptionPane.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class MessageBoxes extends JFrame {
    private JButton[] b = {
        new JButton("Alert"), new JButton("Yes/No"),
        new JButton("Color"), new JButton("Input"),
        new JButton("3 Vals")
    };
}

```

```

private JTextField txt = new JTextField(15);
private ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String id = ((JButton)e.getSource()).getText();
        if(id.equals("Alert"))
            JOptionPane.showMessageDialog(null,
                "There's a bug on you!", "Hey!",
                JOptionPane.ERROR_MESSAGE);
        else if(id.equals("Yes/No"))
            JOptionPane.showConfirmDialog(null,
                "or no", "choose yes",
                JOptionPane.YES_NO_OPTION);
        else if(id.equals("Color")) {
            Object[] options = { "Red", "Green" };
            int sel = JOptionPane.showOptionDialog(
                null, "Choose a Color!", "Warning",
                JOptionPane.DEFAULT_OPTION,
                JOptionPane.WARNING_MESSAGE, null,
                options, options[0]);
            if(sel != JOptionPane.CLOSED_OPTION)
                txt.setText("Color Selected: " + options[sel]);
        } else if(id.equals("Input")) {
            String val = JOptionPane.showInputDialog(
                "How many fingers do you see?");
            txt.setText(val);
        } else if(id.equals("3 Vals")) {
            Object[] selections = {"First", "Second", "Third"};
            Object val = JOptionPane.showInputDialog(
                null, "Choose one", "Input",
                JOptionPane.INFORMATION_MESSAGE,
                null, selections, selections[0]);
            if(val != null)
                txt.setText(val.toString());
        }
    }
};

public MessageBoxes() {
    setLayout(new FlowLayout());
    for(int i = 0; i < b.length; i++) {
        b[i].addActionListener(al);
        add(b[i]);
    }
    add(txt);
}

public static void main(String[] args) {
    run(new MessageBoxes(), 200, 200);
}
} ///:~

```

Чтобы написать всего один `ActionListener` для всех кнопок, на свой страх и риск я выбрал довольно рискованное решение с проверкой текста кнопок для определения нужного поведения. Риск состоит в том, что при написании текста легко немного ошибиться, пропустить одну букву или набрать ее в другом регистре, и обнаружить подобную ошибку будет нелегко.

Заметьте, что методы `showOptionDialog()` и `showInputDialog()` возвращают объекты, которые содержат значение, введенное пользователем.

- 17.** (5) Создайте приложение с возможностью запуска инструментом `SwingConsole`. В документации JDK, которая имеется на сайте <http://java.sun.com>, найдите описание текстового поля с паролем `JPasswordField` и добавьте этот компонент на форму. Если пользователь набирает правильный пароль, выведите окно с подтверждающим это сообщением, используя возможности `JOptionPane`.
- 18.** (4) Модифицируйте программу `MessageBoxes.java` так, чтобы она присоединяла к каждой кнопке отдельный слушатель `ActionListener` (вместо сравнения текста на кнопках).

Меню

Все компоненты, способные отображать меню, — `JApplet`, `JFrame`, `JDialog` и их потомки — содержат метод `setJMenuBar()`, который принимает в качестве аргумента линейку меню `JMenuBar` (для компонента допускается только одна линейка меню). К компоненту `JMenuBar` можно добавлять меню `JMenu`, к которым, в свою очередь, уже позволено добавлять пункты меню `JMenuItem`¹. С любым пунктом меню `JMenuItem` можно добавлять компонент `ActionListener`, который вызывается при выборе этого меню.

В Java и Swing меню формируется прямо в коде программы. Пример создания очень простого меню:

```
//: gui/SimpleMenus.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class SimpleMenus extends JFrame {
    private JTextField t = new JTextField(15);
    private ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            t.setText(((JMenuItem)e.getSource()).getText());
        }
    };
    private JMenu[] menus = {
        new JMenu("Winken"), new JMenu("Blinken"),
        new JMenu("Nod")
    };
    private JMenuItem[] items = {
        new JMenuItem("Fee"), new JMenuItem("Fi"),
        new JMenuItem("Fo"), new JMenuItem("Zip"),
        new JMenuItem("Zap"), new JMenuItem("Zot"),
        new JMenuItem("Olly"), new JMenuItem("Oxen"),
        new JMenuItem("Free")
    };
    public SimpleMenus() {
        for(int i = 0; i < items.length; i++) {
            items[i].addActionListener(al);
            menus[i % 3].add(items[i]);
        }
    }
}
```

продолжение ↗

¹ Обычные пункты меню также можно присоединять непосредственно к линейке меню тем же методом `add`. — *Примеч. ред.*

```

JMenuBar mb = new JMenuBar();
for(JMenu jm : menus)
    mb.add(jm);
setJMenuBar(mb);
setLayout(new FlowLayout());
add(t);
}
public static void main(String[] args) {
    run(new SimpleMenus(), 200, 150);
}
} ///:~

```

Использование целочисленного деления с остатком (`i%3`) помогает распределить пункты меню между тремя составными меню `JMenu`. С каждым пунктом меню `JMenuItem` должен быть связан компонент `ActionListener`; в нашем случае использован один слушатель для всех пунктов меню, но чаще для каждого объекта `JMenuItem` используется отдельный слушатель.

Класс пункта меню `JMenuItem` является производным от класса базовой кнопки `AbstractButton`, поэтому он обладает некоторыми «кнопочными» характеристиками. Сам по себе он представляет пункт меню, который можно включить в раскрывающиеся меню программы. От этого класса также унаследовано еще три типа: составное меню `JMenu` для хранения пунктов меню (для формирования каскадных меню), меню `JCheckBoxMenuItem`, позволяющее помечать флажками пункты меню, и меню `JRadioButtonMenuItem`, которое содержит переключатель.

В качестве более интересного примера рассмотрим меню на базе уже знакомого примера. В нем продемонстрированы каскадные меню, мнемоники (комбинации клавиш), меню с флажками `JCheckBoxMenuItem` и динамическое изменение меню:

```

//: gui/Menus.java
// Подменю, флажки, переключение меню,
// клавиатурные мнемоники и команды.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Menus extends JFrame {
    private String[] flavors = {
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
        "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };
    private JTextField t = new JTextField("No flavor", 30);
    private JMenuBar mb1 = new JMenuBar();
    private JMenu
        f = new JMenu("File"),
        m = new JMenu("Flavors"),
        s = new JMenu("Safety");
    // Альтернативное решение:
    private JCheckBoxMenuItem[] safety = {
        new JCheckBoxMenuItem("Guard"),
        new JCheckBoxMenuItem("Hide")
    };
    private JMenuItem[] file = { new JMenuItem("Open") };
    // A second menu bar to swap to:

```

```

private JMenuBar mb2 = new JMenuBar();
private JMenu fooBar = new JMenu("fooBar");
private JMenuItem[] other = {
    // Добавить мнемонику очень просто, однако
    // в конструкторе это можно сделать только для JMenuItem:
    new JMenuItem("Foo", KeyEvent.VK_F),
    new JMenuItem("Bar", KeyEvent.VK_A),
    // Без мнемоник:
    new JMenuItem("Baz"),
};
private JButton b = new JButton("Swap Menus");
class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JMenuBar m = getJMenuBar();
        setJMenuBar(m == mb1 ? mb2 : mb1);
        validate(); // Обновляем окно
    }
}
class ML implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JMenuItem target = (JMenuItem)e.getSource();
        String actionCommand = target.getActionCommand();
        if(actionCommand.equals("Open")) {
            String s = t.getText();
            boolean chosen = false;
            for(String flavor : flavors)
                if(s.equals(flavor))
                    chosen = true;
            if(!chosen)
                t.setText("Choose a flavor first!");
            else
                t.setText("Opening " + s + ". Mmm, mm!");
        }
    }
}
class FL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JMenuItem target = (JMenuItem)e.getSource();
        t.setText(target.getText());
    }
}
// Также можно создать разные классы для разных
// пунктов MenuItem. Тогда не придется определять,
// какой именно пункт меню был выбран:
class Fool implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText("Foo selected");
    }
}
class BarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText("Bar selected");
    }
}
class BazL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText("Baz selected");
    }
}

```

```

class CMIL implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
        JCheckBoxMenuItem target =
            (JCheckBoxMenuItem)e.getSource();
        String actionCommand = target.getActionCommand();
        if(actionCommand.equals("Guard"))
            t.setText("Guard the Ice Cream! " +
                "Guarding is " + target.getState());
        else if(actionCommand.equals("Hide"))
            t.setText("Hide the Ice Cream! " +
                "Is it hidden? " + target.getState());
    }
}

public Menu() {
    ML ml = new ML();
    CMIL cmil = new CMIL();
    safety[0].setActionCommand("Guard");
    safety[0].setMnemonic(KeyEvent.VK_G);
    safety[0].addItemListener(cmil);
    safety[1].setActionCommand("Hide");
    safety[1].setMnemonic(KeyEvent.VK_H);
    safety[1].addItemListener(cmil);
    other[0].addActionListener(new Fool());
    other[1].addActionListener(new BarL());
    other[2].addActionListener(new BazL());
    FL fl = new FL();
    int n = 0;
    for(String flavor : flavors) {
        JMenuItem mi = new JMenuItem(flavor);
        mi.addActionListener(fl);
        m.add(mi);
        // Добавление разделителей с интервалами:
        if((n++ + 1) % 3 == 0)
            m.addSeparator();
    }
    for(JCheckBoxMenuItem sfty : safety)
        s.add(sfty);
    s.setMnemonic(KeyEvent.VK_A);
    f.add(s);
    f.setMnemonic(KeyEvent.VK_F);
    for(int i = 0; i < file.length; i++) {
        file[i].addActionListener(fl);
        f.add(file[i]);
    }
    mb1.add(f);
    mb1.add(m);
    setJMenuBar(mb1);
    t.setEditable(false);
    add(t, BorderLayout.CENTER);
    // Настройка системы для переключения меню:
    b.addActionListener(new BL());
    b.setMnemonic(KeyEvent.VK_S);
    add(b, BorderLayout.NORTH);
    for(JMenuItem oth : other)
        fooBar.add(oth);
    fooBar.setMnemonic(KeyEvent.VK_B);
    mb2.add(fooBar);
}

```

```
public static void main(String[] args) {  
    run(new Menu(), 300, 200);  
}  
} ///:~
```

В данной программе я поместил пункты меню в массив, после чего перебрал элементы массива, вызывая метод `add()` для каждого объекта `JMenuItem`. Такая схема несколько упрощает монотонное добавление или удаление пунктов меню.

Программа создает не одну линейку меню `JMenuBar`, а целых две, чтобы продемонстрировать, как можно проводить перестановку меню прямо во время работы программы. Далее вы можете видеть, как линейка меню (`JMenuBar`) образуется из составных меню (`JMenu`), а те заполняются пунктами меню (`JMenuItem`), меню с флажками (`JCheckBoxMenuItem`) или другими объектами `JMenu` (для формирования подменю). Когда линейка меню `JMenuBar` будет собрана, ее можно подключить к программе вызовом метода `setJMenuBar()`. Заметьте, что при нажатии кнопки сначала с помощью метода `getJMenuBar()` проверяется, какое именно меню используется в данный момент в программе, и после этого другая полоса меню помещается на место старого меню.

Обратите внимание, что когда проводится проверка строки «Open», очень важны написание слова и соответствие регистра букв, однако при отсутствии совпадения Java не выдает никаких сообщений об ошибке. Подобные сравнения строк нередко становятся причиной ошибок программирования.

Установка и сброс флажков пунктов меню `JCheckBoxMenuItem` проводится автоматически. Код, обрабатывающий меню с флажками, определяет, что было помечено, двумя способами: посредством сравнения строк (как только что было замечено, не очень надежное решение, хотя и используется данной программой) и с помощью сравнения объектов-источников события. Как показано в программе, метод `getState()` возвращает состояние флажка `JCheckBoxMenuItem`. Также можно изменить это состояние программно, вызвав метод `setState()`.

События меню немного непоследовательны и опасны неоднозначными ситуациями: обычные меню `JMenuItem` связываются с `ActionListener`, однако меню с флажками `JCheckBoxMenuItem` используют слушатели `ItemListener`. Подменю `JMenu` также поддерживают `ActionListener`, однако для них эти слушатели не так часто применяются. Обычно каждому объекту `JMenuItem`, `JCheckBoxMenuItem` или `JRadioButtonMenuItem` дается свой собственный слушатель; здесь же показано, как можно определить единственного слушателя для обработки событий сразу нескольких компонентов меню.

Библиотека Swing поддерживает мнемоники, или так называемые «клавиатурные сокращения». С их помощью можно выбрать любой компонент, унаследованный от базовой кнопки `AbstractButton`, управляя программой с клавиатуры, без использования мыши¹. Делается это просто: для пункта меню `JMenuItem` существует перегруженный

¹ В Java быстрый доступ разделяется на мнемоники и акселераторы. Первые срабатывают для видимых пунктов меню, вторые (назначаются методом `setAccelerator`) — для скрытых пунктов меню (вложенных и не видимых в данный момент). Для установки акселераторов задается дополнительный параметр маски, указывающий одну из клавиш — `Ctrl`, `Alt` или `Shift`, нажимаемую в таком случае совместно с клавишей мнемоники. Таким образом, полноценный GUI всегда описывает как мнемоники, так и акселераторы. — *Примеч. ред.*

конструктор, во втором аргументе которого передается идентификатор клавиши. Впрочем, большая часть компонентов, унаследованных от класса `AbstractButton`, не имеет такого конструктора, из-за чего приходится прибегнуть к общему способу установки мнемоники — вызову метода `setMnemonic()`. Рассмотренный пример применяет мнемоники для кнопки и для некоторых пунктов меню; индикаторы «клавиатурных сокращений» автоматически выводятся на визуальных компонентах.

Также пример демонстрирует использование метода `setActionCommand()`. Здесь это выглядит немного странно, поскольку «команда действия» (action command) полностью совпадает с текстом самого пункта меню. Почему бы не использовать текст пункта меню вместо дополнительной строки? Проблема состоит в интернационализации. При переводе программы на другой язык изменяется только текст меню, однако код остается неизменным (его изменение наверняка привело бы к возникновению ошибок). Использование метода `setActionCommand()` позволяет сохранить неизменные «команды действия» при изменении текста меню. Весь код рассмотренной программы работает с ними и поэтому не зависит от языка. Заметьте, что в данной программе не все пункты меню используют «названия действий», поэтому события таких меню обрабатываются иным образом.

Остальная работа производится в слушателях. Слушатель `VL` осуществляет перестановку линеек меню `JMenuBar`. В слушателе `ML` используется подход «кто звонил?»: программа берет источник события `ActionEvent`, преобразует к пункту меню `JMenuItem` и получает строку передаваемой команды действия в серии команд `if`.

Слушатель `FL` достаточно прост, хотя он и обрабатывает все разнообразные сорта мороженого. Такой подход разумен тогда, когда обработка события достаточно проста и не требует много информации. Однако в основном используется отдельный слушатель для каждого пункта меню, как в данном примере (`Fool`, `BarL` и `BazL`), поскольку тогда не требуется определять, откуда пришло сообщение, — надо просто выполнить подходящее действие. Несмотря на увеличение количества классов (каждому пункту меню по классу), размер кода слушателя обычно уменьшается, и код становится элегантнее, а сам процесс более надежен.

Как нетрудно убедиться, код создания меню быстро разрастается и становится запутанным. В данном случае неплохо бы использовать автоматизированное средство построения GUI. Хороший инструментарий обязательно поддерживает создание меню произвольной сложности.

19. (3) Заместите в программе `Menus.java` меню с флажками на меню с переключателями.
20. (6) Создайте программу, разбивающую текстовый файл на слова. Распределите эти слова по надписями в меню и подменю.

Всплывающие меню

Всплывающее меню `JPopupMenu` обычно реализуется следующим образом: создается внутренний класс, расширяющий адаптер обработчика событий мыши `MouseAdapter`, после чего объект этого внутреннего класса добавляется ко всем компонентам, к которым вы хотели бы присоединить всплывающее меню:

```
//: gui/Popup.java
// Создание всплывающих меню библиотеки Swing.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```

import static net.mindview.util.SwingConsole.*;

public class Popup extends JFrame {
    private JPopupMenu popup = new JPopupMenu();
    private JTextField t = new JTextField(10);
    public Popup() {
        setLayout(new FlowLayout());
        add(t);
        ActionListener al = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                t.setText(((JMenuItem)e.getSource()).getText());
            }
        };
        JMenuItem m = new JMenuItem("Hither");
        m.addActionListener(al);
        popup.add(m);
        m = new JMenuItem("Yon");
        m.addActionListener(al);
        popup.add(m);
        m = new JMenuItem("Afar");
        m.addActionListener(al);
        popup.add(m);
        popup.addSeparator();
        m = new JMenuItem("Stay Here");
        m.addActionListener(al);
        popup.add(m);
        PopupListener pl = new PopupListener();
        addMouseListener(pl);
        t.addMouseListener(pl);
    }
    class PopupListener extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            maybeShowPopup(e);
        }
        public void mouseReleased(MouseEvent e) {
            maybeShowPopup(e);
        }
        private void maybeShowPopup(MouseEvent e) {
            if(e.isPopupTrigger())
                popup.show(e.getComponent(), e.getX(), e.getY());
        }
    }
    public static void main(String[] args) {
        run(new Popup(), 300, 200);
    }
} ///:~

```

Ко всем пунктам меню `JMenuItem` добавляется один и тот же слушатель `ActionListener`, который извлекает название пункта меню и помещает его в текстовое поле `JTextField`.

Рисование

В хорошей GUI-среде рисование должно быть достаточно простым — в библиотеке Swing рисовать действительно несложно. Проблема любого примера, вычерчивающего что-либо на экране, — относительно сложные вычисления, которые определяют, в каком месте экрана будет происходить графический вывод. Эти вычисления гораздо

сложнее вызова самих функций рисования, однако они часто помещаются прямо в вызов функций, и кажется, что функции рисования сложнее, чем они есть на самом деле.

Рассмотрим задачу отображения некоторых данных на экране — в нашем случае данные будут предоставлены встроенным методом `Math.sin()`, который возвращает значение синуса угла. Чтобы сделать программу интереснее, а также для включения в наше поле зрения дополнительных компонентов библиотеки Swing, добавим в нее регулятор `JSlider`, который будет располагаться в нижней части формы. Он позволяет динамически управлять периодом «волны» синусоиды. Вдобавок, если вы увеличите или уменьшите размеры окна, то увидите, что изображение функции автоматически приспособится к новым размерам.

Хотя для рисования подходит любой компонент, унаследованный от `JComponent`, обычно поверхностью для вывода служит панель `JPanel`. В ней необходимо переопределить один метод, с именем `paintComponent()`, который вызывается при перерисовке компонента (об этом можете не беспокоиться, библиотека Swing делает это автоматически). При вызове метода ему передается объект для рисования `Graphics`, который вы вправе использовать для рисования и графического вывода.

В следующем примере процесс рисования и связанные с ним вычисления проводятся в классе `SineDraw`; класс `SineWave` просто настраивает программу и создает регулятор `JSlider`. В классе `SineDraw` определен метод `setCycles()`, который позволяет другому объекту — в данном случае регулятору — управлять отображаемой синусоидой.

```
//: gui/SineWave.java
// Рисование средствами Swing, использование JSlider.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

class SineDraw extends JPanel {
    private static final int SCALEFACTOR = 200;
    private int cycles;
    private int points;
    private double[] sines;
    private int[] pts;
    public SineDraw() { setCycles(5); }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int maxWidth = getWidth();
        double hstep = (double)maxWidth / (double)points;
        int maxHeight = getHeight();
        pts = new int[points];
        for(int i = 0; i < points; i++)
            pts[i] =
                (int)(sines[i] * maxHeight/2 * .95 + maxHeight/2);
        g.setColor(Color.RED);
        for(int i = 1; i < points; i++) {
            int x1 = (int)((i - 1) * hstep);
            int x2 = (int)(i * hstep);
            int y1 = pts[i-1];
            int y2 = pts[i];
            g.drawLine(x1, y1, x2, y2);
        }
    }
}
```



```

    }
}
public void setCycles(int newCycles) {
    cycles = newCycles;
    points = SCALEFACTOR * cycles * 2;
    sines = new double[points];
    for(int i = 0; i < points; i++) {
        double radians = (Math.PI / SCALEFACTOR) * i;
        sines[i] = Math.sin(radians);
    }
    repaint();
}
}

public class SineWave extends JFrame {
    private SineDraw sines = new SineDraw();
    private JSlider adjustCycles = new JSlider(1, 30, 5);
    public SineWave() {
        add(sines);
        adjustCycles.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                sines.setCycles(
                    ((JSlider)e.getSource()).getValue());
            }
        });
        add(BorderLayout.SOUTH, adjustCycles);
    }
    public static void main(String[] args) {
        run(new SineWave(), 700, 400);
    }
} ///:~

```

Все поля и массивы программы предназначены для вычисления координат точек синусоиды на экране: переменная `cycles` показывает, сколько волн необходимо нарисовать на экране, переменная `points` хранит полное количество отображаемых точек, переменная `sines` содержит текущее значение синуса, а переменная `pts` — координаты точек по оси *Y*. Метод `setCycles()` создает массив, соответствующий количеству необходимых точек, и заполняет массив `sines` числовыми значениями. Вызывая метод `repaint()`, метод `setCycles()` неявно выполняет метод `paintComponent()`, который завершает вычисления и рисует синусоиду на экране.

При переопределении метода `paintComponent()` в первую очередь следует вызвать его базовую версию. После этого можно делать все, что потребуется; обычно выполняется рисование пикселей на `JPanel` с использованием методов класса `Graphics`, описанных в документации JDK для класса `java.awt.Graphics` (<http://java.sun.com>). Из рассмотренного примера видно, что большая часть кода занимается вычислениями; из «рисующих» методов вызываются только два: метод установки цвета `setColor()` и метод рисования линии `drawLine()`. При создании программ, выводящих на экран некоторые данные, обычно так и происходит — основное время уходит на выяснение того, что нужно рисовать, в то время как сам процесс рисования относительно прост.

При написании этой программы большая часть времени была затрачена на составление кода для отображения синусоиды. Когда эта часть была готова, я подумал, что неплохо было бы динамически изменять период волны синусоиды. Мой опыт

в программировании таких задач на других языках подсказывал, что это будет тяжело, однако оказалось, что в Java это самая простая часть проекта. Я создал регулятор `JSlider` (аргументы конструктора — это наименьшее, наибольшее и начальное значение шкалы регулятора, хотя в классе `JSlider` есть и другие конструкторы) и поместил его в `JFrame`. Затем я просмотрел документацию JDK и обнаружил, что для ползунка можно использовать только одного слушателя, с именем `addChangeListener`, который вызывается при изменении положения ползунка. Единственному методу интерфейса слушателя с именем `stateChanged()` передается аргумент-событие `ChangeEvent`, с помощью которого можно обратиться к источнику события и узнать новое положение ползунка. После этого остается вызвать метод `setCycles()` объекта `sines`, этот метод обновляет значение и перерисовывает панель `JPanel`.

Обычно все компоненты библиотеки Swing используются так же легко, даже если вы с ними до этого не работали. Надо посмотреть документацию, добавить компонент на форму, присоединить подходящего слушателя событий — вот и все.

Если ваша задача сложна и требует расширенных средств рисования, существует великое множество альтернатив, включая визуальные компоненты `JavaBean` от сторонних производителей и технологию сложного рисования `Java 2D API`. К сожалению, эти средства выходят за рамки темы книги, но если ваш код для рисования не справляется с поставленными задачами, вам непременно следует обратить на них внимание.

21. (5) Преобразуйте программу `SineWave.java` так, чтобы класс `SineDraw` стал компонентом `JavaBean`. Для этого определите в классе подходящие `get`- и `set`-методы.
22. (7) Создайте приложение с использованием `SwingConsole`. В нем должно быть три регулятора, каждый отвечает соответственно за интенсивность красной, синей и зеленой составляющей в цвете `java.awt.Color`. Остальную часть формы должна заполнить панель `JPanel`, закрашиваемая сформированным из значений трех регуляторов цветом. Также добавьте три текстовых поля, недоступных для редактирования, в которых укажите текущие числовые значения составляющих цвета RGB.
23. (8) Взяв за отправную точку пример `SineWaves.java`, создайте программу, которая рисует на экране вращающийся квадрат. Один регулятор должен управлять скоростью вращения, а второй — размером квадрата.
24. (7) Помните игрушку «волшебный экран» с двумя ручками — одна управляет вертикальным движением рисующей точки, другая управляет горизонтальным движением? Создайте подобие такой игрушки на компьютере, взяв за образец код `SineWave.java`. Место ручек должны занять ползунки регуляторов. Добавьте кнопку, которая будет полностью очищать экран.
25. (8) Взяв за отправную точку пример `SineWaves.java`, создайте программу (приложение с использованием класса `SwingConsole`), которая рисует анимированную синусоиду, пробегающую через все окно, как на осциллографе, используя для управления анимацией `java.util.Timer`. Скорость анимации должна задаваться регулятором `javax.swing.JSlider`.
26. (5) Измените упражнение 25 так, чтобы в приложении создавалось сразу несколько панелей с синусоидами. Количество панелей должно задаваться в командной строке.

- 27.** (5) Снова измените упражнение 25, чтобы для управления анимацией использовался таймер `javax.swing.Timer`. Отметьте отличия между ним и таймером `java.util.Timer`.
- 28.** (7) Создайте класс для моделирования кубика (просто класс, без графического интерфейса). Создайте пять объектов кубиков и проведите серию бросков. Выведите кривую с суммой очков при каждом броске; обеспечьте динамическое изменение кривой при новых бросках.

Диалоговые окна

Диалоговое окно — это окно, которое создается другим окном. Его предназначение — выполнить некоторые специфические действия, не загромождая родительского окна лишними подробностями. Диалоговые окна часто используются в оконных средах.

Для создания диалогового окна нужно использовать наследование от класса `JDialog`, который представляет собой еще один вид окна `Window`, как и окно с рамкой `JFrame`. Размещение элементов управления диалога `JDialog` определяется менеджером расположения (по умолчанию таковым является `BorderLayout`), для обработки событий к нему присоединяются слушатели. Вот очень простой пример:

```
//: gui/Dialogs.java
// Создание и использование диалоговых окон.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

class MyDialog extends JDialog {
    public MyDialog(JFrame parent) {
        super(parent, "My dialog", true);
        setLayout(new FlowLayout());
        add(new JLabel("Here is my dialog"));
        JButton ok = new JButton("OK");
        ok.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                dispose(); // Закрывает диалоговое окно
            }
        });
        add(ok);
        setSize(150,125);
    }
}

public class Dialogs extends JFrame {
    private JButton b1 = new JButton("Dialog Box");
    private MyDialog dlg = new MyDialog(null);
    public Dialogs() {
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                dlg.setVisible(true);
            }
        });
        add(b1);
    }
    public static void main(String[] args) {
```

продолжение ➤

```

    run(new Dialogs(), 125, 75);
}
} ///:~

```

После создания диалогового окна `JDialog` следует вызвать метод `setVisible(true)`, который выводит его на экран и активизирует. При закрытии диалогового окна необходимо освободить используемые им ресурсы вызовом `dispose()`.

Следующий пример более сложен; диалоговое окно заполняется сеткой (с использованием менеджера расположения `GridLayout`) из специальных кнопок, определенных в классе `ToeButton`. Такая кнопка рисует рамку, внутри которой помещается «крестик» или «нолик» (или вообще ничего не помещается, это зависит от состояния кнопки). Сначала кнопка «пуста», затем, в зависимости от того, чей сейчас ход, на ней рисуется «крестик» или «нолик». Вместе с тем игрок может щелкнуть кнопкой мыши на любой кнопке таблицы, чтобы изменить ее состояние. (Благодаря чему обычные правила игры в «крестики-нолики» приобретают оттенок вседозволенности.) Вдобавок пользователь может изменить количество строк и столбцов в диалоговом окне, изменяя числа в главном окне приложения.

```

//: gui/TicTacToe.java
// Диалоговые окна и создание собственных компонентов.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class TicTacToe extends JFrame {
    private JTextField
        rows = new JTextField("3"),
        cols = new JTextField("3");
    private enum State { BLANK, XX, OO }
    static class ToeDialog extends JDialog {
        private State turn = State.XX; // Начинаем с "крестиков"
        ToeDialog(int cellswide, int cellshigh) {
            setTitle("The game itself");
            setLayout(new GridLayout(cellswide, cellshigh));
            for(int i = 0; i < cellswide * cellshigh; i++)
                add(new ToeButton());
            setSize(cellswide * 50, cellshigh * 50);
            setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        }
    }
    class ToeButton extends JPanel {
        private State state = State.BLANK;
        public ToeButton() { addMouseListener(new ML()); }
        public void paintComponent(Graphics g) {
            super.paintComponent(g);
            int
                x1 = 0, y1 = 0,
                x2 = getSize().width - 1,
                y2 = getSize().height - 1;
            g.drawRect(x1, y1, x2, y2);
            x1 = x2/4;
            y1 = y2/4;
            int wide = x2/2, high = y2/2;
            if(state == State.XX) {
                g.drawLine(x1, y1, x1 + wide, y1 + high);
            }
        }
    }
}

```

```

        g.drawLine(x1, y1 + high, x1 + wide, y1);
    }
    if(state == State.OO)
        g.drawOval(x1, y1, x1 + wide/2, y1 + high/2);
    }
    class ML extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            if(state == State.BLANK) {
                state = turn;
                turn =
                    (turn == State.XX ? State.OO : State.XX);
            }
            else
                state =
                    (state == State.XX ? State.OO : State.XX);
            repaint();
        }
    }
}

class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JDialog d = new ToeDialog(
            new Integer(rows.getText()),
            new Integer(cols.getText()));
        d.setVisible(true);
    }
}

public TicTacToe() {
    JPanel p = new JPanel();
    p.setLayout(new GridLayout(2,2));
    p.add(new JLabel("Rows", JLabel.CENTER));
    p.add(rows);
    p.add(new JLabel("Columns", JLabel.CENTER));
    p.add(cols);
    add(p, BorderLayout.NORTH);
    JButton b = new JButton("go");
    b.addActionListener(new BL());
    add(b, BorderLayout.SOUTH);
}

public static void main(String[] args) {
    run(new TicTacToe(), 200, 200);
}

} ///:~

```

Статические члены допускаются только на внешнем уровне классов, внутренние классы не могут иметь статических данных или вложенных классов.

Метод `paintComponent()` рисует вокруг панели прямоугольную рамку, а также «крестик» или «нолик». В нем полно разных утомительных вычислений, но суть их вполне тривиальна.

Щелчки кнопок мыши обрабатываются слушателем `MouseListener`, который сначала проверяет, нарисовано ли что-нибудь на табличной кнопке. Если на ней ничего нет, то выдается запрос к родительскому окну и на основе полученной информации устанавливается состояние кнопки `ToeButton`. С помощью механизма внутренних классов `ToeButton` снова обращается к родительскому окну и изменяет результат

соответствующего хода. Если на кнопке уже отображался «крестик» или «нолик», происходит замена текущего состояния. В этих вычислениях очень удобен тройной оператор «если-то», описанный в главе 3. После модификации состояния кнопка перерисовывается.

Конструктор класса `ToeDialog` достаточно прост: он добавляет на форму заданное количество кнопок, после чего изменяет их размер так, чтобы на каждую сторону кнопки приходилось по 50 пикселей.

Класс `TicTacToe` вписывает в программу последние штрихи, создавая два текстовых поля `TextField` (для ввода количества строк и столбцов) и кнопку «Go», к которой присоединяется слушатель `ActionListener`. Во время обработки нажатия этой кнопки из текстовых полей извлекаются данные, и поскольку изначально они находятся в строковой форме, то далее преобразуются в целые числа `int` конструктором `Integer` с аргументом `String`.

Диалоговые окна выбора файлов

Некоторые операционные системы предоставляют особые встроенные диалоговые окна для выбора шрифтов, цветов, принтеров и т. п. Фактически все графические операционные системы поддерживают открытие и сохранение файлов, поэтому в библиотеке `Swing` появился класс `JFileChooser`, который инкапсулирует эти функции для удобства использования.

Следующее приложение использует два вида окна `JFileChooser`: в одном выбирается файл для открытия, в другом задается имя файла для сохранения. Большая часть кода не должна вызывать у вас вопросов, все интересные нам действия сосредоточены в слушателях `ActionListener` для двух кнопок:

```
//: gui/FileChooserTest.java
// Демонстрация диалогового окна выбора файлов.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class FileChooserTest extends JFrame {
    private JTextField
        fileName = new JTextField(),
        dir = new JTextField();
    private JButton
        open = new JButton("Open"),
        save = new JButton("Save");
    public FileChooserTest() {
        JPanel p = new JPanel();
        open.addActionListener(new OpenL());
        p.add(open);
        save.addActionListener(new SaveL());
        p.add(save);
        add(p, BorderLayout.SOUTH);
        dir.setEditable(false);
        fileName.setEditable(false);
    }
}
```

```
p = new JPanel();
p.setLayout(new GridLayout(2,1));
p.add(fileName);
p.add(dir);
add(p, BorderLayout.NORTH);
}
class OpenL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JFileChooser c = new JFileChooser();
        // Демонстрация диалогового окна открытия файлов:
        int rVal = c.showOpenDialog(FileChooserTest.this);
        if(rVal == JFileChooser.APPROVE_OPTION) {
            fileName.setText(c.getSelectedFile().getName());
            dir.setText(c.getCurrentDirectory().toString());
        }
        if(rVal == JFileChooser.CANCEL_OPTION) {
            fileName.setText("You pressed cancel");
            dir.setText("");
        }
    }
}
class SaveL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JFileChooser c = new JFileChooser();
        // Демонстрация диалогового окна сохранения файлов:
        int rVal = c.showSaveDialog(FileChooserTest.this);
        if(rVal == JFileChooser.APPROVE_OPTION) {
            fileName.setText(c.getSelectedFile().getName());
            dir.setText(c.getCurrentDirectory().toString());
        }
        if(rVal == JFileChooser.CANCEL_OPTION) {
            fileName.setText("You pressed cancel");
            dir.setText("");
        }
    }
}
public static void main(String[] args) {
    run(new FileChooserTest(), 250, 150);
}
} ///:~
```

Стоит заметить, что диалог `JFileChooser` обладает большими возможностями — например, он позволяет назначать фильтры для сокращения списка отображаемых файлов.

Для вызова окна открытия файла вы вызываете метод `showOpenDialog()`, а для окна сохранения файла — метод `showSaveDialog()`. Эти методы не возвращают управление до тех пор, пока диалоговое окно не будет закрыто. На выходе из диалога объект `JFileChooser` остается доступен, с его помощью можно получить различную информацию. Например, методы `getSelectedFile()` и `getCurrentDirectory()` представляют собой два способа запроса информации об имени выбранного файла и его расположения. Если они возвращают `null`, это значит, что пользователь отменил выбор файла и завершил диалог.

- 29.** (3) В документации JDK для пакета `javax.swing` найдите описание класса `JColorChooser`, который отображает диалоговое окно для выбора цвета. Напишите программу с кнопкой, нажатие на которую открывает это диалоговое окно на экране.

HTML для компонентов Swing

Все компоненты, которые отображают текст на экране, также могут получать данные в формате HTML, которые преобразуются согласно правилам данного языка разметки. Это значит, что вы можете легко и просто украсить компоненты Swing нестандартными надписями по своему вкусу. Например:

```
//: gui/HTMLButton.java
// Размещение текста в формате HTML
// на компонентах Swing.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class HTMLButton extends JFrame {
    private JButton b = new JButton(
        "<html><b><font size=+2>" +
        "<center>Hello!<br><i>Press me now!");
    public HTMLButton() {
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                add(new JLabel("<html>" +
                    "<i><font size=+4>Каров!");
                // Инициировать пересчет расположения
                // для включения новой надписи:
                validate();
            }
        });
        setLayout(new FlowLayout());
        add(b);
    }
    public static void main(String[] args) {
        run(new HTMLButton(), 200, 500);
    }
} ///:~
```

Текст должен начинаться с тега `<html>`, за которым уже могут следовать другие теги языка HTML. Заметьте, что использовать закрывающие теги не обязательно и их пропуск не считается ошибкой.

При своем вызове слушатель `ActionListener` добавляет на форму новую надпись `JLabel`, текст которой выводится в формате HTML. Но так как этот компонент был наложен на форму после конструирования, макет формы необходимо построить заново, для чего вызывается метод `validate()`.

Текст в формате HTML применим также для наборов вкладок (`JTabbedPane`), пунктов меню (`JMenuItem`), подсказок (`JToolTip`), переключателей (`JRadioButton`) и флажков (`JCheckBox`).

- 30.** (3) Напишите программу, демонстрирующую использование текста в формате HTML для всех компонентов, перечисленных в предыдущем абзаце.

Регуляторы и индикаторы выполнения

Регулятор `JSlider` (который уже использовался в примере `Sinewave.java`) позволяет пользователю вводить данные, перемещая ползунок по шкале, что в некоторых ситуациях очень удобно и интуитивно понятно (например, настройка громкости звука). Индикатор выполнения `JProgressBar` показывает данные в сравнении с некоторым эталоном, помогая пользователю оценить состояние выполнения, то есть пользователь наглядно получает информацию, какая относительная часть от максимального значения считается выполненной на данный момент. Мой любимый пример с участием этих элементов управления — присоединение регулятора `JSlider` к индикатору выполнения, чтобы при перемещении ползунка синхронно менялся бы и индикатор. В следующем примере также демонстрируется `ProgressMonitor` — монитор выполнения с расширенной функциональностью:

```
//: gui/Progress.java
// Использование регуляторов, индикаторов и мониторов выполнения.
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Progress extends JFrame {
    private JProgressBar pb = new JProgressBar();
    private ProgressMonitor pm = new ProgressMonitor(
        this, "Monitoring Progress", "Test", 0, 100);
    private JSlider sb =
        new JSlider(JSlider.HORIZONTAL, 0, 100, 60);
    public Progress() {
        setLayout(new GridLayout(2,1));
        add(pb);
        pm.setProgress(0);
        pm.setMillisToPopup(1000);
        sb.setValue(0);
        sb.setPaintTicks(true);
        sb.setMajorTickSpacing(20);
        sb.setMinorTickSpacing(5);
        sb.setBorder(new TitledBorder("Slide Me"));
        pb.setModel(sb.getModel()); // Общая модель
        add(sb);
        sb.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                pm.setProgress(sb.getValue());
            }
        });
    }
    public static void main(String[] args) {
        run(new Progress(), 300, 200);
    }
} ///:~
```

Для того чтобы ассоциировать два компонента (точнее, придать им одинаковое поведение), достаточно использовать одну и ту же модель поведения, что и делается в строке:

```
pb.setModel(sb.getModel());
```

Конечно, управлять одновременным изменением содержимого двух компонентов можно и с помощью слушателя, однако в таких простых программах это решение проще. Компонент `ProgressMonitor` не имеет модели, поэтому с ним приходится применять решение со слушателем. Следует учесть, что компонент `ProgressMonitor` поддерживает перемещение только вперед, а при достижении конца шкалы окно закрывается.

Индикатор выполнения (`JProgressBar`) относительно прост, а вот регулятор `JSlider` имеет множество настроек (таких, как ориентация, основные и второстепенные метки шкалы). Заметьте, как легко к нему присоединить рамку с заголовком.

31. (8) Создайте «асимптотический индикатор выполнения», который замедляется по мере приближения к конечной точке. Добавьте случайное поведение, чтобы периодически казалось, что он начинает ускоряться.
32. (6) Измените код `Progress.java`, чтобы вместо общей модели для связывания регулятора и индикатора выполнения использовался слушатель.

Выбор внешнего вида и поведения программы

«Модульный интерфейс пользователя» (`pluggable look & feel`) позволяет вашей программе эмулировать внешний вид и поведение различных операционных систем. Вы даже можете динамически изменять визуальное оформление и образ действий программы прямо во время ее выполнения. Впрочем, обычно выбирают одно из двух: либо предпочтение отдается платформенно-независимому пользовательскому интерфейсу (в библиотеке `Swing` его называют `Metal`), либо интерфейсу системы, под управлением которой выполняется программа, соответственно, она выглядит так, как будто была создана специально для этой системы (это почти всегда лучшее решение, потому что оно наверняка окажется более понятным для пользователя). Код для выбора используемого интерфейса довольно прост — однако выполнить его вы должны *до того*, как будут созданы графические компоненты, поскольку компоненты создаются на основе текущего внешнего вида и поведения программы и не изменятся сами собой, если вы вдруг надумаете заняться этим делом в процессе ее работы (изменить внешний вид и поведение уже созданных компонентов можно, но это довольно нелегко; за подробностями обращайтесь к специализированной литературе по библиотеке `Swing`).

Вообще говоря, если вы хотите использовать переносимый между платформами пользовательский интерфейс (`Metal`) библиотеки `Swing`, вам ничего не придется делать — он устанавливается «по умолчанию». Но если вам понадобится запустить вашу программу с интерфейсом определенной операционной системы¹, нужно будет выполнить следующий код (обычно его помещают прямо в метод `main()` или еще куда-то, где он будет выполнен перед добавлением компонентов):

```
try {
    UIManager.setLookAndFeel(
        UIManager.getSystemLookAndFeelClassName());
} catch (Exception e) {
    throw new RuntimeException(e);
}
```

¹ Насколько эффективно оформление `Swing` использует возможности вашей операционной системы? Решайте сами.

Включать какой-либо код в секцию `catch` не обязательно, так как класс `UIManager`, отвечающий за внешний вид и поведение программы, автоматически задействует кросс-платформенный интерфейс, если попытки установки другого интерфейса завершатся неудачей. Впрочем, в процессе отладки программы исключение может быть полезным — например, если вы захотите вывести в секции `catch` некоторое сообщение.

Следующая программа выбирает оформление программы в зависимости от полученного параметра командной строки и показывает, как выглядят некоторые компоненты в различных пользовательских интерфейсах.

```
//: gui/LookAndFeel.java
// Выбор оформления программы.
// {Args: motif}
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class LookAndFeel extends JFrame {
    private String[] choices =
        "Eeny Meeny Minnie Mickey Moe Larry Curly".split(" ");
    private Component[] samples = {
        new JButton("JButton"),
        new JTextField("JTextField"),
        new JLabel("JLabel"),
        new JCheckBox("JCheckBox"),
        new JRadioButton("Radio"),
        new JComboBox(choices),
        new JList(choices),
    };
    public LookAndFeel() {
        super("Look And Feel");
        setLayout(new FlowLayout());
        for(Component component : samples)
            add(component);
    }
    private static void usageError() {
        System.out.println(
            "Usage:LookAndFeel [cross|system|motif]");
        System.exit(1);
    }
    public static void main(String[] args) {
        if(args.length == 0) usageError();
        if(args[0].equals("cross")) {
            try {
                UIManager.setLookAndFeel(UIManager.
                    getCrossPlatformLookAndFeelClassName());
            } catch(Exception e) {
                e.printStackTrace();
            }
        } else if(args[0].equals("system")) {
            try {
                UIManager.setLookAndFeel(UIManager.
                    getSystemLookAndFeelClassName());
            } catch(Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

    } else if(args[0].equals("motif")) {
        try {
            UIManager.setLookAndFeel("com.sun.java."+
                "swing.plaf.motif.MotifLookAndFeel");
        } catch(Exception e) {
            e.printStackTrace();
        }
    } else usageError();
    // Заметьте, что тип пользовательского интерфейса
    // должен быть выбран перед созданием компонентов.
    run(new LookAndFeel(), 300, 300);
}
} ///:~

```

Пользовательский интерфейс программы можно также явно указать строкой с именем, как это сделано в нашей программе для `MotifLookAndFeel`. Однако такой подход применим при выборе только двух видов пользовательского интерфейса: этого и платформенно-независимого интерфейса `Metal`, выбираемого по умолчанию. Хотя для Windows и Macintosh также существуют строковые обозначения, использовать их можно только на соответствующих платформах (эти имена возвращаются методом `getSystemLookAndFeelClassName()` в процессе выполнения программы).

Также возможно создать собственный пакет оформления — например, если вам понадобится придать особый колорит программам какой-то компании. Однако это требует больших усилий и выходит за рамки темы данной книги (более того, часто эта тема не рассматривается даже в книгах, полностью посвященных библиотеке Swing!).

Деревья, таблицы и буфер обмена

Краткое введение и примеры по этим темам приведены в электронных приложениях этой главы по адресу www.MindView.net.

JNLP и Java Web Start

Апплет можно снабдить *цифровой подписью* по соображениям безопасности. Такая возможность продемонстрирована в приложении этой главы по адресу www.MindView.net. Подписанные апплеты обладают большими возможностями, сравнимыми с возможностями приложений, но они должны выполняться в браузере. Это требует от машины клиента дополнительных ресурсов для работы браузера, а также означает ограниченность пользовательского интерфейса апплета. Зачастую такой интерфейс сбивает пользователя с толку, так как у браузера имеются собственные меню и панели инструментов, которые появляются «над апплетом»¹.

Протокол запуска Java по сети (Java Network Launch Protocol, JNLP) решает проблему, не лишая вас преимуществ апплетов. Используя JNLP, вы сумеете загрузить и установить самостоятельное Java-приложение на машину клиента. Оно может быть запущено из командной строки, с рабочего стола или с помощью менеджера приложений,

¹ Этот раздел написан Джереми Мейером.

поставляемого вместе с имеющейся у вас реализацией JNLP. Приложение даже можно запустить с того сайта, откуда оно было изначально загружено.

Приложение JNLP способно динамически, прямо во время своей работы получать данные из Интернета и автоматически проверять версию. А это означает, что оно обладает всеми преимуществами апплетов, в добавление к достоинствам самостоятельного приложения.

Как и в случае с апплетами, система клиента должна с осторожностью относиться к приложениям JNLP. По этой причине на приложения JNLP распространяются те же ограничения системы безопасности, как и на апплеты. Как и апплеты, такие приложения могут поставляться в подписанных архивах JAR, давая пользователю возможность решить, стоит ли доверять стороне, выдавшей цифровую подпись. Но в отличие от апплетов, если приложение JNLP придет в неподписанном JAR-файле, у этого приложения все равно останется возможность получить доступ к отдельным ресурсам клиентской системы, посредством некоторых услуг интерфейса JNLP (пользователь должен будет давать разрешения программе во время ее работы).

Так как JNLP представляет собой протокол, а не реализацию, для его использования вам понадобится некоторая реализация. Компания Sun разработала пакет Java Web Start, сокращенно JAWS; это бесплатная, официальная и эталонная реализация, распространяемая в составе Java SE5. Если вы используете его при разработке, убедитесь в том, что файл JAR (`javaws.jar`) включен в `CLASSPATH`; проще всего добавить его из стандартного пути установки Java в `java/lib`. Если вы развертываете приложение JNLP с веб-сервера, следует убедиться в том, что он поддерживает тип `MIME application/x-java-jnlp-file`. При использовании новейшей версии сервера Tomcat (<http://jakarta.apache.org/tomcat>) об этом не нужно беспокоиться — в нем уже все настроено. За информацией о других серверах стоит обратиться к руководству пользователя.

Создать приложение JNLP нетрудно. Вы создаете обычное приложение, упаковываете его в архив JAR, а затем пишете файл запуска, который представляет собой обычный XML-файл, содержащий всю информацию, необходимую клиенту для загрузки и запуска вашего приложения. Если вы решите не подписывать свой файл, надо будет востребовать возможности протокола JNLP для получения доступа к тем ресурсам, которые будут вам необходимы на машине клиента.

Вот небольшая вариация примера с диалогами `JFileChooser`, но на этот раз мы используем для запуска примера протокол JNLP, так чтобы класс можно было развернуть как приложение JNLP в неподписанном архиве JAR.

```
//: gui/jnlp/JnlpFileChooser.java
// Открытие файлов на локальной машине с JNLP.
// {Requires: javax.jnlp.FileOpenService;
// You must have javaws.jar in your classpath}
// Чтобы создать файл jnlpfilechooser.jar выполните следующие команды:
// cd ..
// cd ..
// jar cvf gui/jnlp/jnlpfilechooser.jar gui/jnlp/*.class
package gui.jnlp;
import javax.jnlp.*;
import javax.swing.*;
import java.awt.*;
```

продолжение ➤

```

import java.awt.event.*;
import java.io.*;

public class JnlpFileChooser extends JFrame {
    private JTextField fileName = new JTextField();
    private JButton
        open = new JButton("Open"),
        save = new JButton("Save");
    private JEditorPane ep = new JEditorPane();
    private JScrollPane jsp = new JScrollPane();
    private FileContents fileContents;
    public JnlpFileChooser() {
        JPanel p = new JPanel();
        open.addActionListener(new OpenL());
        p.add(open);
        save.addActionListener(new SaveL());
        p.add(save);
        jsp.getViewPort().add(ep);
        add(jsp, BorderLayout.CENTER);
        add(p, BorderLayout.SOUTH);
        fileName.setEditable(false);
        p = new JPanel();
        p.setLayout(new GridLayout(2,1));
        p.add(fileName);
        add(p, BorderLayout.NORTH);
        ep.setContentType("text");
        save.setEnabled(false);
    }
    class OpenL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            FileOpenService fs = null;
            try {
                fs = (FileOpenService)ServiceManager.lookup(
                    "javax.jnlp.FileOpenService");
            } catch (UnavailableServiceException use) {
                throw new RuntimeException(use);
            }
            if(fs != null) {
                try {
                    fileContents = fs.openFileDialog(".",
                        new String[]{"txt", "*"});
                    if(fileContents == null)
                        return;
                    fileName.setText(fileContents.getName());
                    ep.read(fileContents.getInputStream(), null);
                } catch (Exception exc) {
                    throw new RuntimeException(exc);
                }
                save.setEnabled(true);
            }
        }
    }
}

class SaveL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        FileSaveService fs = null;
        try {
            fs = (FileSaveService)ServiceManager.lookup(
                "javax.jnlp.FileSaveService");

```

```

    } catch(UnavailableServiceException use) {
        throw new RuntimeException(use);
    }
    if(fs != null) {
        try {
            fileContents = fs.saveFileDialog(".",
                new String[]{"txt"},
                new ByteArrayInputStream(
                    ep.getText().getBytes()),
                fileContents.getName());
            if(fileContents == null)
                return;
            fileName.setText(fileContents.getName());
        } catch(Exception exc) {
            throw new RuntimeException(exc);
        }
    }
}
}
public static void main(String[] args) {
    JnlpFileChooser fc = new JnlpFileChooser();
    fc.setSize(400, 300);
    fc.setVisible(true);
}
} ///:~

```

Заметьте, что классы `FileOpenService` и `FileCloseService` импортируются из пакета `javax.jnlp` и что нигде в тексте программы нет прямого обращения к `JFileChooser`. Два вида сервиса, используемых нами, необходимо получать с помощью метода `ServiceManager.lookup()`, ресурсы на машине клиента доступны только через объекты, возвращаемые этим методом. В нашем примере чтение файлов и запись в них осуществлялись посредством интерфейса `FileContent`, предоставляемого JNLP. Попытки обращения к ресурсам напрямую (например, создание объектов `File` и `FileReader`) приведут к исключению `SecurityException`, точно так же, как это случилось бы при попытке использования этих объектов из неподписанного апплета. Если вы хотите работать с этими классами и не намерены ограничиваться услугами интерфейсов JNLP, необходимо подписать свой архив JAR.

Закомментированная команда `jar` в `JnlpFileChooser.java` создает необходимый файл JAR. Для нашего примера файл запуска выглядит так:

```

///:~ gui/jnlp/filechooser.jnlp
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec = "1.0+"
  codebase="file:C:/AAA-TIJ4/code/gui/jnlp"
  href="filechooser.jnlp">
  <information>
    <title>FileChooser demo application</title>
    <vendor>Mindview Inc.</vendor>
    <description>
      Jnlp File chooser Application
    </description>
    <description kind="short">
      Demonstrates opening, reading and writing a text file
    </description>
    <icon href="mindview.gif"/>
  </information>
</jnlp>

```

продолжение ➤

```

    <offline-allowed/>
</information>
<resources>
    <j2se version="1.3+"
      href="http://java.sun.com/products/autodl/j2se"/>
    <jar href="jnlptest.jar" download="eager"/>
</resources>
<application-desc
  main-class="gui.jnlp.JnlpFileChooser"/>
</jnlp>
///:~

```

Файл запуска включен в архив исходного кода этой книги (с файла *www.MindView.net*), сохраняется под именем `filechooser.jnlp` (без первой и последней строки) в том же каталоге, где находится архив формата JAR. Как видно из листинга, это файл формата XML, с одним тегом `<jnlp>`. У него есть несколько вложенных элементов, которые в основном понятны без комментариев.

Атрибут `spec` в элементе `jnlp` сообщает клиентской системе, какая версия JNLP требуется для запуска приложения. Атрибут `codebase` указывает на каталог расположения данного файла запуска и необходимых ресурсов. Обычно в нем содержится URL некоторого веб-сервера, но в нашем случае это каталог на локальной машине, что позволяет легко протестировать приложение. *Помните, что для успешного запуска программы необходимо изменить этот путь так, чтобы он обозначал соответствующий каталог на вашей машине.* Атрибут `href` должен указывать имя файла запуска.

Тег `information` включает в себя многочисленные подэлементы, предоставляющие информацию о приложении. Они используются административной консолью Java Web Start или ее эквивалентом, которые устанавливают приложение JNLP и позволяют пользователю запускать его из командной строки, создавать ярлыки и т. п.

Тег `resources` служит той же цели, что и тег `applet` в HTML. Подэлемент `j2se` указывает версию пакета J2SE, необходимую для запуска приложения, а в элементе `jar` задается имя архива формата JAR, в котором находятся классы. В элементе `jar` также имеется атрибут `download`, который может принимать значения `eager` или `lazy`. Он указывает реализации JNLP, нужно ли полностью загружать архив JAR для запуска приложения.

Атрибут `application-desc` сообщает реализации JNLP, какой класс является запускаемым в файле JAR (то есть определяет точку входа).

Еще один полезный подэлемент тега `jnlp` — элемент `security`, в нашем файле не показанный. Он мог бы выглядеть следующим образом:

```

<security>
  <all-permissions/>
</security>

```

Этот тег используется, когда ваше приложение разворачивается в виде подписанного архива JAR. В рассмотренном примере он не нужен, поскольку доступ ко всем локальным ресурсам осуществляется только через сервис JNLP.

Есть еще несколько тегов, подробнее о них можно узнать в спецификации по адресу <http://java.sun.com/products/javawebstart/download-spec.html>.

Чтобы запустить программу, необходимо создать страницу запуска с гипертекстовой ссылкой на файл `.jnlp`. Вот как он выглядит (без первой и последней строки):

```
//:! gui/jnlp/filechooser.html
<html>
Follow the instructions in JnlpFileChooser.java to
build jnlpfilechooser.jar, then:
<a href="filechooser.jnlp">click here</a>
</html>
///:~
```

После того как приложение будет загружено, его можно будет настроить с помощью менеджера приложений (административной консоли). Если вы используете Java Web Start в Windows, то при втором запуске приложения пользователю предложат создать для него ярлык на рабочем столе. Это поведение может быть настроено.

В данном разделе были рассмотрены только два сервиса протокола JNLP, в текущем выпуске их поддерживается семь. Каждый из них отвечает за определенную задачу: печать, вставка и копирование в буфер обмена и т. д. Дополнительную информацию можно найти на сайте <http://java.sun.com>.

Параллельное выполнение и Swing

При общении с библиотекой Swing используются потоки. Вы убедились в этом в начале главы, когда узнали, что все задачи должны отправляться потоку диспетчеризации событий Swing вызовом `SwingUtilities.invokeLater()`. Однако тот факт, что вам не приходится явно создавать потоки (`Thread`), может обернуться внезапными проблемами, связанными с многозадачностью. Просто помните, что в Swing есть поток диспетчеризации событий Swing — он постоянно обрабатывает все события Swing, извлекая их из очереди и последовательно выполняя. Не забывайте об этом, если вы не хотите, чтобы ваше приложение оказалось в ситуации взаимной блокировки или в нем возникла ситуация гонки (`race condition`).

В этом разделе рассматриваются некоторые вопросы, которые необходимо учитывать при работе с потоками и библиотекой Swing.

Продолжительные задачи

Одна из основных ошибок, допускаемых при программировании графического интерфейса, — случайное использование потока диспетчеризации событий для выполнения продолжительной задачи. Простой пример:

```
//: gui/LongRunningTask.java
// Плохо написанная программа.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import static net.mindview.util.SwingConsole.*;
```

продолжение ➤

```

public class LongRunningTask extends JFrame {
    private JButton
        b1 = new JButton("Start Long Running Task"),
        b2 = new JButton("End Long Running Task");
    public LongRunningTask() {
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                try {
                    TimeUnit.SECONDS.sleep(3);
                } catch (InterruptedException e) {
                    System.out.println("Task interrupted");
                    return;
                }
                System.out.println("Task completed");
            }
        });
        b2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                // Прерваться?
                Thread.currentThread().interrupt();
            }
        });
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
    }
    public static void main(String[] args) {
        run(new LongRunningTask(), 200, 150);
    }
} ///:~

```

При нажатии кнопки `b1` поток диспетчеризации событий внезапно оказывается занятым выполнением продолжительной задачи. Кнопка так и остается в нажатом состоянии, потому что поток диспетчеризации событий, который обычно перерисовывает экран, занят. И вы не сможете сделать ничего больше (например, нажать кнопку `b2`), потому что программа не отреагирует на ваши действия до завершения задачи `b1` и освобождения потока диспетчеризации. Код `b2` безуспешно пытается решить проблему прерыванием потока диспетчеризации событий.

Конечно, проблема должна решаться выполнением продолжительных процессов в отдельных потоках. Здесь мы используем однопоточный объект `Executor`, который автоматически ставит ожидающие задачи в очередь и выполняет их одну за другой:

```

//: gui/InterruptedExceptionLongRunningTask.java
// Выполнение продолжительных задач в потоках.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import static net.mindview.util.SwingConsole.*;

class Task implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    public void run() {
        System.out.println(this + " started");
    }
}

```

```

    try {
        TimeUnit.SECONDS.sleep(3);
    } catch (InterruptedException e) {
        System.out.println(this + " interrupted");
        return;
    }
    System.out.println(this + " completed");
}
public String toString() { return "Task " + id; }
public long id() { return id; }
};

public class InterruptableLongRunningTask extends JFrame {
    private JButton
        b1 = new JButton("Start Long Running Task"),
        b2 = new JButton("End Long Running Task");
    ExecutorService executor =
        Executors.newSingleThreadExecutor();
    public InterruptableLongRunningTask() {
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Task task = new Task();
                executor.execute(task);
                System.out.println(task + " added to the queue");
            }
        });
        b2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                executor.shutdownNow(); // Силовое решение
            }
        });
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
    }
    public static void main(String[] args) {
        run(new InterruptableLongRunningTask(), 200, 150);
    }
} ///:~

```

Уже лучше, но при нажатии b2 для объекта `ExecutorService` вызывается метод `shutdownNow()`, который его закрывает. При попытке добавления новых задач вы получите исключение. Таким образом, при нажатии кнопки b2 программа становится неработоспособной. Нам требуется другое — закрыть текущую задачу (и отменить ожидающие задачи) без остановки всего остального. Механизм Java SE5 `Callable/Future`, описанный в главе 21, — именно то, что нужно. Мы определим новый класс с именем `TaskManager`, который содержит кортежи с объектом `Callable`, представляющим задачу, и объектом `Future`, поступающим от `Callable`. Необходимость использования кортежа объясняется тем, что он позволяет нам отслеживать исходную задачу, чтобы мы могли получить дополнительную информацию, недоступную в `Future`. Вот как это выглядит:

```

//: net/mindview/util/TaskItem.java
// Объект Future и объект Callable, который его создает.
package net.mindview.util;
import java.util.concurrent.*;

```

продолжение ➤

```

public class TaskItem<R,C extends Callable<R>> {
    public final Future<R> future;
    public final C task;
    public TaskItem(Future<R> future, C task) {
        this.future = future;
        this.task = task;
    }
} ///:~

```

В библиотеке `java.util.concurrent` задача не будет доступной через объект `Future` по умолчанию, потому что объект задачи может прекратить свое существование на момент получения результата от `Future`. Здесь мы заставляем задачу остаться, сохраняя ее.

Класс `TaskManager` включается в библиотеку `net.mindview.util`, чтобы его можно было использовать как вспомогательный класс общего назначения:

```

//: net/mindview/util/TaskManager.java
// Управление и выполнение очереди задач.
package net.mindview.util;
import java.util.concurrent.*;
import java.util.*;

public class TaskManager<R,C extends Callable<R>>
    extends ArrayList<TaskItem<R,C>> {
    private ExecutorService exec =
        Executors.newSingleThreadExecutor();
    public void add(C task) {
        add(new TaskItem<R,C>(exec.submit(task),task));
    }
    public List<R> getResults() {
        Iterator<TaskItem<R,C>> items = iterator();
        List<R> results = new ArrayList<R>();
        while(items.hasNext()) {
            TaskItem<R,C> item = items.next();
            if(item.future.isDone()) {
                try {
                    results.add(item.future.get());
                } catch (Exception e) {
                    throw new RuntimeException(e);
                }
                items.remove();
            }
        }
        return results;
    }
    public List<String> purge() {
        Iterator<TaskItem<R,C>> items = iterator();
        List<String> results = new ArrayList<String>();
        while(items.hasNext()) {
            TaskItem<R,C> item = items.next();
            // Сохранить завершенные задачи для вывода
            // информации о результатах:
            if(!item.future.isDone()) {
                results.add("Cancelling " + item.task);
                item.future.cancel(true); // Возможно прерывание
                items.remove();
            }
        }
    }
}

```

```

    return results;
}
} ///:~

```

Объект `TaskManager` представляет собой контейнер `ArrayList` с объектами `TaskItem`. Он также содержит однопоточный объект `Executor`, чтобы при вызове `add()` для `Callable` он отправлял `Callable` на выполнение и сохранял итоговый объект `Future` вместе с исходной задачей. Если вам потребуется что-нибудь сделать с задачей, у вас имеется ссылка на нее. В качестве простого примера в `purge()` используется метод `toString()` задачи.

Теперь этот класс может использоваться для управления продолжительными задачами в нашем примере:

```

//: gui/InterruptedExceptionLongRunningCallable.java
// Использование Callable для выполнения
// продолжительных задач.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

class CallableTask extends Task
implements Callable<String> {
    public String call() {
        run();
        return "Return value of " + this;
    }
}

public class
InterruptedExceptionLongRunningCallable extends JFrame {
    private JButton
        b1 = new JButton("Start Long Running Task"),
        b2 = new JButton("End Long Running Task"),
        b3 = new JButton("Get results");
    private TaskManager<String,CallableTask> manager =
        new TaskManager<String,CallableTask>();
    public InterruptedExceptionLongRunningCallable() {
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                CallableTask task = new CallableTask();
                manager.add(task);
                System.out.println(task + " added to the queue");
            }
        });
        b2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                for(String result : manager.purge())
                    System.out.println(result);
            }
        });
        b3.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // Пример вызова метода Task:
                for(TaskItem<String,CallableTask> tt :

```

продолжение ➤

```

        manager)
        tt.task.id(); // Преобразование типа не требуется
    for(String result : manager.getResults())
        System.out.println(result);
    }
});
setLayout(new FlowLayout());
add(b1);
add(b2);
add(b3);
}
public static void main(String[] args) {
    run(new InterruptableLongRunningCallable(), 200, 150);
}
} ///:~

```

Как видите, `CallableTask` делает точно то же самое, что `Task`, но при этом возвращает результат — в нашем случае объект `String`, идентифицирующий задачу.

Для решения сходной задачи были созданы средства, не входящие в Swing (и не являющиеся частью стандартной поставки Java): `SwingWorker` (на сайте Sun) и `Foxtrot` (<http://foxtrot.sourceforge.net>), но на момент написания книги они еще не были доработаны для использования механизма Java SE5 `Callable/Future`.

Часто бывает важно предоставить конечному пользователю какие-то визуальные признаки того, что задача продолжает выполняться, и информацию о ходе ее выполнения. Обычно для этой цели используется компонент `JProgressBar` или `ProgressMonitor`. В следующем примере используется `ProgressMonitor`:

```

//: gui/MonitoredLongRunningCallable.java
// Вывод информации о ходе выполнения
// с использованием ProgressMonitor.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

class MonitoredCallable implements Callable<String> {
    private static int counter = 0;
    private final int id = counter++;
    private final ProgressMonitor monitor;
    private final static int MAX = 8;
    public MonitoredCallable(ProgressMonitor monitor) {
        this.monitor = monitor;
        monitor.setNote(toString());
        monitor.setMaximum(MAX - 1);
        monitor.setMillisToPopup(500);
    }
    public String call() {
        System.out.println(this + " started");
        try {
            for(int i = 0; i < MAX; i++) {
                TimeUnit.MILLISECONDS.sleep(500);
                if(monitor.isCanceled())
                    Thread.currentThread().interrupt();
                final int progress = i;
            }
        }
    }
}

```

```

        SwingUtilities.invokeLater(
            new Runnable() {
                public void run() {
                    monitor.setProgress(progress);
                }
            }
        );
    }
} catch (InterruptedException e) {
    monitor.close();
    System.out.println(this + " interrupted");
    return "Result: " + this + " interrupted";
}
System.out.println(this + " completed");
return "Result: " + this + " completed";
}
public String toString() { return "Task " + id; }
};

public class MonitoredLongRunningCallable extends JFrame {
    private JButton
        b1 = new JButton("Start Long Running Task"),
        b2 = new JButton("End Long Running Task"),
        b3 = new JButton("Get results");
    private TaskManager<String, MonitoredCallable> manager =
        new TaskManager<String, MonitoredCallable>();
    public MonitoredLongRunningCallable() {
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                MonitoredCallable task = new MonitoredCallable(
                    new ProgressMonitor(
                        MonitoredLongRunningCallable.this,
                        "Long-Running Task", "", 0, 0)
                );
                manager.add(task);
                System.out.println(task + " added to the queue");
            }
        });
        b2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                for (String result : manager.purge())
                    System.out.println(result);
            }
        });
        b3.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                for (String result : manager.getResults())
                    System.out.println(result);
            }
        });
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
        add(b3);
    }
    public static void main(String[] args) {
        run(new MonitoredLongRunningCallable(), 200, 500);
    }
} ///:~

```

Конструктор `MonitoredCallable` получает объект `ProgressMonitor` в своем аргументе, а его метод `call()` обновляет `ProgressMonitor` каждые полсекунды. Обратите внимание: объект `MonitoredCallable` является отдельной задачей и поэтому не может напрямую управлять пользовательским интерфейсом, поэтому для передачи информации о ходе выполнения используется вызов `SwingUtilities.invokeLater()`. В учебнике *Swing Tutorial* компании Sun (на сайте <http://java.sun.com>) представлено альтернативное решение с исследованием класса `Swing Timer`, который проверяет состояние задачи и обновляет монитор.

Если нажата кнопка отмены, `monitor.isCanceled()` вернет `true`. В нашем случае задача просто вызывает `interrupt()` для своего потока; управление передается в блок `catch`, где монитор завершается методом `close()`.

Остальной код остался практически неизменным, если не считать создания `ProgressMonitor` в конструкторе `MonitoredLongRunningCallable`.

- 33.** (6) Измените код `InterruptedExceptionLongRunningCallable.java` так, чтобы все задачи выполнялись параллельно, а не последовательно.

Визуальные потоки

Следующий пример создает класс `JPanel`, реализующий `Runnable`, который окрашивает себя в разные цвета. Приложение получает из командной строки значения, определяющие размер сетки цветов и продолжительность задержки `sleep()` между изменениями. Экспериментируя с этими значениями, можно обнаружить некоторые интересные, а иногда и необъяснимые особенности реализации многопоточности на вашей платформе:

```
//: gui/ColorBoxes.java
// Визуальная демонстрация многопоточности.
// {Args: 12 50}
import javax.swing.*;
import java.awt.*;
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

class CBox extends JPanel implements Runnable {
    private int pause;
    private static Random rand = new Random();
    private Color color = new Color(0);
    public void paintComponent(Graphics g) {
        g.setColor(color);
        Dimension s = getSize();
        g.fillRect(0, 0, s.width, s.height);
    }
    public CBox(int pause) { this.pause = pause; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                color = new Color(rand.nextInt(0xFFFFFF));
                repaint(); // Асинхронный запрос paint()
                TimeUnit.MILLISECONDS.sleep(pause);
            }
        }
    }
}
```



```
    } catch(InterruptedException e) {  
        // Допустимый способ выхода  
    }  
}  
}  
  
public class ColorBoxes extends JFrame {  
    private int grid = 12;  
    private int pause = 50;  
    private static ExecutorService exec =  
        Executors.newCachedThreadPool();  
    public ColorBoxes() {  
        setLayout(new GridLayout(grid, grid));  
        for(int i = 0; i < grid * grid; i++) {  
            CBox cb = new CBox(pause);  
            add(cb);  
            exec.execute(cb);  
        }  
    }  
    public static void main(String[] args) {  
        ColorBoxes boxes = new ColorBoxes();  
        if(args.length > 0)  
            boxes.grid = new Integer(args[0]);  
        if(args.length > 1)  
            boxes.pause = new Integer(args[1]);  
        run(boxes, 500, 400);  
    }  
} ///:~
```

Класс `ColorBoxes` настраивает `GridLayout` так, чтобы компонент содержал `grid` ячеек в каждом измерении. Затем он добавляет соответствующее количество объектов `CBox` для заполнения сетки, передавая каждому значение `pause`. В методе `main()` полям `pause` и `grid` назначаются значения по умолчанию, которые можно изменить при передаче аргументов командной строки.

Вся основная работа выполняется в классе `CBox`, который является производным от `JPanel` и реализует интерфейс `Runnable`, чтобы каждый компонент `JPanel` также мог бы стать независимой задачей. Этими задачами управляет объект `ExecutorService`.

Текущим цветом ячейки является `cColor`. Цвета создаются конструктором `Color`, получающим 24-разрядное число (в нашем примере оно выбирается случайным образом).

Метод `paintComponent()` весьма прост; он просто назначает цвет `cColor` и заполняет этим цветом всю панель `JPanel`.

Метод `run()` содержит бесконечный цикл, который присваивает `cColor` новый случайный цвет, а затем вызывает `repaint()` для его отображения. Затем поток приостанавливается вызовом `sleep()` на промежуток времени, заданный в командной строке.

О вызове `repaint()` в `run()` стоит сказать особо. На первый взгляд может показаться, что мы создаем множество потоков, каждый из которых заставляет выполнить перерисовку. Казалось бы, это нарушает принцип, согласно которому задачи должны ставиться в очередь событий. Однако эти потоки в действительности не изменяют общий ресурс. Вызов `repaint()` не форсирует перерисовку, а всего лишь устанавливает флаг, который означает, что когда поток диспетчеризации событий в следующий раз будет

готов к перерисовке, эта область будет кандидатом на перерисовку. Таким образом, эта программа не создает проблем с потоками в Swing.

Когда поток диспетчеризации событий выполняет `paint()`, он сначала вызывает `paintComponent()`, затем `paintBorder()` и `paintChildren()`. Если вы должны переопределить `paint()` в производном компоненте, обязательно вызовите версию `paint()` базового класса, чтобы были выполнены все необходимые действия.

Именно благодаря гибкости этой архитектуры и привязке потоков к каждому экземпляру `JPanel` вы можете экспериментировать и создать столько потоков, сколько потребуется. (На практике количество потоков, с которыми нормально справляется JVM, ограничено.)

Программа также демонстрирует радикальные различия в производительности и поведении между разными реализациями JVM на разных платформах.

- 34.** (4) Измените пример `ColorBoxes.java` так, чтобы он сначала разбрасывал по экрану «звездочки», а затем случайным образом изменял их цвета.

Визуальное программирование и компоненты JavaBean

Читая эту книгу, вы имели возможность убедиться, как хорошо подходит язык Java для создания многократно используемых фрагментов кода. «Единицей многократного использования» является класс, поскольку он представляет собой связанный набор характеристик (полей) и действий (методов), а его многократное использование реализуется посредством композиции или наследования.

Наследование и полиморфизм — неотъемлемые составляющие объектно-ориентированного программирования, но в большинстве случаев при создании приложения нужны *компоненты*, которые просто делают то, что вам нужно. В идеале разработчик просто включает эти компоненты в программу — подобно инженеру-электронщику, который собирает микросхемы на печатной плате. Также кажется, что должен быть и способ ускорить процесс создания программы посредством аналогичной «модульной сборки».

«Визуальное программирование» стало популярным — *чрезвычайно* популярным — с появлением среды разработки программ Visual Basic (VB) от фирмы Microsoft, за которой последовала среда разработки второго поколения Delphi фирмы Borland (между прочим, это она стимулировала появление технологии JavaBeans). В этих программных средах компоненты представлены визуально, что вполне логично, так как обычно компоненты представляют собой некоторый элемент управления (кнопка, текстовое поле и т. д.). Общее визуальное представление компонента часто полностью совпадает с его внешним видом в работающей программе. Таким образом, часть процесса визуального программирования заключается в том, что вы перетаскиваете компонент с палитры и помещаете его на форму. Построитель приложений «пишет» код, выполняющий необходимые действия, и этот код при выполнении выводит на экран соответствующий компонент.

Простого перетаскивания компонента на форму обычно недостаточно для создания полноценной программы. Часто бывает нужно поменять характеристики компонента: его цвет, надпись, присоединенную базу данных и т. д. Характеристики компонента, которые изменяются во время разработки программы, называют еще *свойствами* (properties). Вы можете манипулировать свойствами компонента прямо в построителе программ, и когда вы создаете программу, полученная конфигурация запоминается, а затем, при запуске программы, восстанавливается.

К этому моменту вы уже должны смириться с тем, что объект — это не только набор характеристик, но еще и набор действий. Во время разработки программы действия визуального компонента частично представлены *событиями* (events), которые фактически означают, что с вашим компонентом что-то происходит. Обычно вы решаете, что должно происходить при возникновении события, закрепляя определенный код за этим событием.

Это очень важный момент: построитель приложений использует механизм отражения, чтобы динамически исследовать компонент и определить, какими свойствами он обладает и какие события поддерживает. Когда набор свойств будет определен, построитель может вывести их, чтобы вы могли внести необходимые изменения (и сохранить эти изменения), а также отобразить список событий. Как правило, разработчик выполняет некую операцию (чаще всего — двойной щелчок кнопкой мыши на нужном событии), а построитель программ создает тело необходимого метода и присоединяет его к этому событию. Вам остается написать код, который будет выполняться при возникновении события.

Таким образом, среда разработки выполняет за вас значительную часть работы. В результате вы можете сосредоточиться на том, как будет выглядеть программа и что она будет делать, и положиться на среду разработки в части рутинных подробностей. Огромная популярность средств визуального программирования обусловлена именно тем, что они значительно увеличивают скорость разработки приложений, — конечно, в первую очередь создание пользовательского интерфейса, однако и другие части программы тоже пишутся быстрее.

Что такое компонент JavaBean?

Итак, оказывается, в визуальных компонентах нет ничего экстраординарного; это всего лишь блок кода, который обычно представляется отдельным классом. Главное, чтобы построитель приложений был способен определить свойства компонента и его события. Для создания компонента в ранних версиях VB программисту приходилось писать довольно сложный код, который следовал определенным соглашениям, позволяющим определить свойства и события. Среда визуального программирования второго поколения Delphi во многом упростила процесс создания собственных визуальных компонентов. Но язык Java придал процессу создания визуальных компонентов новый импульс — технологию JavaBeans, где визуальный компонент Bean является простым классом. Вам не придется писать дополнительный код или использовать расширенные свойства языка, чтобы получить готовый компонент Bean. Все, что вам нужно сделать, — немного изменить способ именования методов своих классов. Именно

имя метода говорит строителю приложений, что является свойством компонента, что является его событием, а что является простым методом.

В документации JDK соглашение о записи имен ошибочно называют «паттерном проектирования». Это не очень мудрое решение и вносит только дополнительную путаницу; паттерны проектирования и без этих проблем достаточно трудны для изучения. Это не паттерн, это просто соглашение об именах, причем достаточно простое.

1. Для свойства с именем `xxx` обычно создаются два метода: метод для получения свойства — `getXxx()` и метод для установки свойства — `setXxx()`. Заметьте, что первая буква после слов `get` и `set` автоматически преобразуется к нижнему регистру для получения имени свойства. Тип, возвращаемый методом `getXxx()`, должен совпадать с типом аргумента метода `setXxx()`. Имя свойства не связано с типом `get/set`.
2. Для свойства типа `boolean` наряду с подходом, описанным в первом пункте, вместо префикса `get` можно использовать `is`.
3. Обычные методы компонента `Bean` не подчиняются описанной схеме имен, но они должны быть объявлены как открытые (`public`).
4. Для создания событий используется подход библиотеки `Swing` — слушатели. Описание имен точно такое же, что мы видели до этого во всех компонентах: например, для обработки события `BounceEvent` необходимо определить метод для добавления слушателя `addBounceListener(BounceListener)` и метод для удаления слушателя `removeBounceListener(BounceListener)`. В большинстве случаев хватает уже имеющихся событий и слушателей, но вы также можете создать свои собственные события и определить интерфейсы их слушателей.

Полученные рекомендации мы можем использовать для создания простого компонента `Bean`.

```
//: frogbean/Frog.java
// Простейший компонент JavaBean.
package frogbean;
import java.awt.*;
import java.awt.event.*;

class Spots {}

public class Frog {
    private int jumps;
    private Color color;
    private Spots spots;
    private boolean jmp;
    public int getJumps() { return jumps; }
    public void setJumps(int newJumps) {
        jumps = newJumps;
    }
    public Color getColor() { return color; }
    public void setColor(Color newColor) {
        color = newColor;
    }
    public Spots getSpots() { return spots; }
```

```
public void setSpots(Spots newSpots) {
    spots = newSpots;
}
public boolean isJumper() { return jmpr; }
public void setJumper(boolean j) { jmpr = j; }
public void addActionListener(ActionListener l) {
    //...
}
public void removeActionListener(ActionListener l) {
    // ...
}
public void addKeyListener(KeyListener l) {
    // ...
}
public void removeKeyListener(KeyListener l) {
    // ...
}
// "Обычный" открытый метод:
public void croak() {
    System.out.println("Ribbet!");
}
} ///:~
```

Во-первых, нетрудно убедиться, что это обыкновенный класс. Обычно все поля класса объявляются закрытыми (`private`) и изменяются только с помощью методов и свойств. По соглашению об именах свойства компонента именуются `jumps`, `color`, `spots` и `jumper` (заметьте изменение регистра первой буквы в имени свойства). Хотя в первых трех случаях имя внутреннего идентификатора класса совпадает с именем свойства, на примере свойства `jumper` вы можете видеть, что свойство не заставляет вас использовать какой-то конкретный идентификатор для внутренних переменных (более того, этих переменных может вообще *не быть*).

Данный компонент Bean позволяет обрабатывать события `ActionEvent` и `KeyEvent`, для чего были добавлены соответствующие методы *add* и *remove*. Наконец, обычный метод `croak()` также является частью этого компонента — просто потому, что он является открытым (`public`), а не потому, что его имя следует какой-то схеме.

Получение информации о компоненте Bean: инструмент Introspector

Один из самых важных моментов во всей механике визуальных компонентов JavaBean настает тогда, когда пользователь выбирает нужный ему компонент из палитры и помещает его на форму. Построитель приложений должен создать экземпляр выбранного компонента (это легко сделать, если у класса компонента имеется конструктор по умолчанию), а затем, не обращаясь к исходному коду класса, извлечь всю необходимую информацию о списке свойств и событий компонента.

Вероятно, часть решения уже очевидна, если вспомнить конец главы 14: механизм *отражения* позволяет определить все методы неизвестного класса. Он прекрасно подходит для поставленной задачи и не требует введения дополнительных ключевых слов, которые существуют в других визуальных языках программирования. Вообще говоря, одной из главных причин для добавления отражения в язык была поддержка

визуальных компонентов JavaBean (хотя оно также применяется при сериализации объектов и для удаленного вызова методов, не говоря уже об обычном программировании). Можно предположить, что создателю среды разработки придется применить отражение к каждому компоненту Bean, чтобы узнать его свойства и события.

Конечно, это возможно, однако создатели Java хотели произвести стандартный инструмент, который бы не только упрощал применение компонентов Bean, но и предоставлял механизм создания более сложных компонентов. Таким инструментом стал класс `Introspector`, а самым важным элементом данного класса является статический метод `getBeanInfo()`. Метод получает ссылку на объект `Class`, полностью исследует переданный ему класс и возвращает объект `BeanInfo`, из которого в дальнейшем можно получить информацию о свойствах, методах и событиях.

Обычно все это совершенно не нужно рядовому разработчику — чаще всего компоненты Bean используются «как есть», и задумываться о том, что происходит с ними внутри среды разработки, не приходится. Вы просто перетаскиваете компонент на форму, настраиваете его свойства и пишете код для событий, которые вас интересуют. Впрочем, будет весьма поучительно воспользоваться инструментом `Introspector` для получения информации о компоненте Bean:

```

//: gui/BeanDumper.java
// Исследование компонента Bean.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.lang.reflect.*;
import static net.mindview.util.SwingConsole.*;

public class BeanDumper extends JFrame {
    private JTextField query = new JTextField(20);
    private JTextArea results = new JTextArea();
    public void print(String s) { results.append(s + "\n"); }
    public void dump(Class<?> bean) {
        results.setText("");
        BeanInfo bi = null;
        try {
            bi = Introspector.getBeanInfo(bean, Object.class);
        } catch (IntrospectionException e) {
            print("Couldn't introspect " + bean.getName());
            return;
        }
        for (PropertyDescriptor d: bi.getPropertyDescriptors()){
            Class<?> p = d.getPropertyType();
            if (p == null) continue;
            print("Property type:\n " + p.getName() +
                "Property name:\n " + d.getName());
            Method readMethod = d.getReadMethod();
            if (readMethod != null)
                print("Read method:\n " + readMethod);
            Method writeMethod = d.getWriteMethod();
            if (writeMethod != null)
                print("Write method:\n " + writeMethod);
            print("=====");
        }
    }
}

```

```

print("Public methods:");
for(MethodDescriptor m : bi.getMethodDescriptors())
    print(m.getMethod().toString());
print("=====");
print("Event support:");
for(EventSetDescriptor e: bi.getEventSetDescriptors()){
    print("Listener type:\n " +
        e.getListenerType().getName());
    for(Method lm : e.getListenerMethods())
        print("Listener method:\n " + lm.getName());
    for(MethodDescriptor lmd :
        e.getListenerMethodDescriptors() )
        print("Method descriptor:\n " + lmd.getMethod());
    Method addListener= e.getAddListenerMethod();
    print("Add Listener Method:\n " + addListener);
    Method removeListener = e.getRemoveListenerMethod();
    print("Remove Listener Method:\n " + removeListener);
    print("=====");
}
}
class Dumper implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String name = query.getText();
        Class<?> c = null;
        try {
            c = Class.forName(name);
        } catch(ClassNotFoundException ex) {
            results.setText("Couldn't find " + name);
            return;
        }
        dump(c);
    }
}
public BeanDumper() {
    JPanel p = new JPanel();
    p.setLayout(new FlowLayout());
    p.add(new JLabel("Qualified bean name:"));
    p.add(query);
    add(BorderLayout.NORTH, p);
    add(new JScrollPane(results));
    Dumper dmpr = new Dumper();
    query.addActionListener(dmpr);
    query.setText("frogbean.Frog");
    // Программный запуск события
    dmpr.actionPerformed(new ActionEvent(dmpr, 0, ""));
}
public static void main(String[] args) {
    run(new BeanDumper(), 600, 500);
}
} ///:~

```

Всю работу выполняет метод `BeanDumper.dump()`. Сначала он пытается создать объект `BeanInfo`, и если это ему удастся, он вызывает методы класса `BeanInfo` для получения информации о свойствах, методах и событиях. При вызове метода `Introspector.getBeanInfo()` передается второй аргумент. Он сообщает `Introspector`, в каком месте иерархии наследования изучаемого объекта надо закончить исследования. В данном случае исследования прекращаются при достижении корневого класса `Object`, так как методы этого класса нас не интересуют.

Для получения свойств вызывается метод `getPropertyDescriptors()`, который возвращает массив объектов `PropertyDescriptor`. Для каждого объекта `PropertyDescriptor` вы можете вызвать метод `getPropertyType()` для нахождения класса объекта-свойства. Затем для каждого свойства вы можете узнать его псевдоним (получаемый из имен методов) с помощью метода `getName()`; метод для чтения свойства, с помощью метода `getReadMethod()`; и метод для записи свойства, с помощью метода `getWriteMethod()`. Последние два метода возвращают объект `Method`, который уже годится для непосредственного вызова соответствующего метода компонента (это часть механизма отражения).

Для открытых (`public`) методов (включая методы чтения и записи свойств) метод `getMethodDescriptors()` возвращает массив объектов с описаниями `MethodDescriptor`. Для каждого объекта `MethodDescriptor` можно получить ассоциированный с ним объект `Method` и вывести его имя.

Для получения информации о событиях используется метод `getEventSetDescriptors()`, который возвращает массив объектов `EventSetDescriptor`. К этим объектам можно обратиться с запросом для получения класса слушателя, списка методов класса слушателя, методов добавления и удаления слушателей. Программа `BeanDumper` распечатывает всю эту информацию.

При запуске программа самостоятельно анализирует созданный ранее компонент `frogbean.Frog`. Результат работы программы после удаления ненужных подробностей таков:

```
Property type:
    Color
Property name:
    color
Read method:
    public Color getColor()
Write method:
    public void setColor(Color)
=====
Property type:
    boolean
Property name:
    jumper
Read method:
    public boolean isJumper()
Write method:
    public void setJumper(boolean)
=====
Property type:
    int
Property name:
    jumps
Read method:
    public int getJumps()
Write method:
    public void setJumps(int)
=====
Property type:
    frogbean.Spots
Property name:
    spots
```



```

Read method:
    public frogbean.Spots getSpots()
Write method:
    public void setSpots(frogbean.Spots)
=====
Public methods:
public void setSpots(frogbean.Spots)
public void setColor(Color)
public void setJumps(int)
public boolean isJumper()
public frogbean.Spots getSpots()
public void croak()
public void addActionListener(ActionListener)
public void addKeyListener(KeyListener)
public Color getColor()
public void setJumper(boolean)
public int getJumps()
public void removeActionListener(ActionListener)
public void removeKeyListener(KeyListener)
=====
Event support:
Listener type:
    KeyListener
Listener method:
    keyPressed
Listener method:
    keyReleased
Listener method:
    keyTyped
Method descriptor:
    public abstract void keyPressed(KeyEvent)
Method descriptor:
    public abstract void keyReleased(KeyEvent)
Method descriptor:
    public abstract void keyTyped(KeyEvent)
Add Listener Method:
    public void addKeyListener(KeyListener)
Remove Listener Method:
    public void removeKeyListener(KeyListener)
=====
Listener type:
    ActionListener
Listener method:
    actionPerformed
Method descriptor:
    public abstract void actionPerformed(ActionEvent)
Add Listener Method:
    public void addActionListener(ActionListener)
Remove Listener Method:
    public void removeActionListener(ActionListener)
=====

```

Из результата работы программы видно, какую информацию извлекает `Introspector` из компонента `Bean` при создании объекта `BeanInfo`. Нетрудно заметить, что тип свойства и его имя не зависят друг от друга. Обратите внимание на автоматическое преобразование первого символа имени свойства в нижний регистр (кроме имен свойств, начинающихся с нескольких заглавных букв). И помните, что имена методов, которые

вы здесь видите (подобные методам для чтения и записи свойств), на самом деле получены из объектов `Method`, которые можно использовать для запуска ассоциированных с ними методов объекта.

Список открытых (`public`) методов включает в себя как обыкновенные методы, не связанные со свойствами или событиями (например, метод `sleep()`), так и методы, закрепленные за ними. Это все методы, которые можно программно вызвать для компонента `Bean`, и среда разработки готова выдать список доступных методов, чтобы упростить вашу работу.

Наконец, мы видим, что все события досконально «препарируются»: выдается вся информация о слушателе, о его методах, о методах для добавления и удаления слушателя. В принципе, при наличии объекта `BeanInfo` можно получить всю существенную информацию о компоненте `Bean`. Вы даже можете вызывать методы этого компонента, хотя у вас нет ничего, кроме простого объекта (здесь снова помогает отражение).

Более сложный компонент `Bean`

Следующий пример посложнее, хотя его и не назовешь серьезным. Панель `JPanel` всего-навсего рисует небольшую окружность вокруг курсора мыши при ее передвижениях. При нажатии кнопки мыши в центре рабочего окна появляется слово «Бум!» и инициируется событие.

Настраиваемые свойства этого компонента таковы: диаметр окружности, цвет текста, размер шрифта и сам текст, который выводится на экран при нажатии кнопки мыши. Компонент `BangBean` также определяет методы `addActionListener()` и `removeActionListener()`, поэтому вы вправе присоединить к нему слушателя действий, который будет срабатывать все при том же нажатии пользователем кнопки мыши. Вероятно, свойства и поддержка событий вам покажутся знакомыми:

```
//: bangbean/BangBean.java
// Графический компонент Bean.
package bangbean;
import javax.swing.*.*;
import java.awt.*.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;

public class
BangBean extends JPanel implements Serializable {
    private int xm, ym;
    private int cSize = 20; // Диаметр окружности
    private String text = "Bang!";
    private int fontSize = 48;
    private Color tColor = Color.RED;
    private ActionListener actionListener;
    public BangBean() {
        addMouseListener(new ML());
        addMouseMotionListener(new MML());
    }
    public int getCircleSize() { return cSize; }
    public void setCircleSize(int newSize) {
```

```

    cSize = newSize;
}
public String getBangText() { return text; }
public void setBangText(String newText) {
    text = newText;
}
public int getFontSize() { return fontSize; }
public void setFontSize(int newSize) {
    fontSize = newSize;
}
public Color getTextColor() { return tColor; }
public void setTextColor(Color newColor) {
    tColor = newColor;
}
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.BLACK);
    g.drawOval(xm - cSize/2, ym - cSize/2, cSize, cSize);
}
// Одноадресный слушатель, простейшая форма
// управления слушателями:
public void addActionListener(ActionListener l)
throws TooManyListenersException {
    if(actionListener != null)
        throw new TooManyListenersException();
    actionListener = l;
}
public void removeActionListener(ActionListener l) {
    actionListener = null;
}
class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        Graphics g = getGraphics();
        g.setColor(tColor);
        g.setFont(
            new Font("TimesRoman", Font.BOLD, fontSize));
        int width = g.getFontMetrics().stringWidth(text);
        g.drawString(text, (getSize().width - width) / 2,
            getSize().height/2);
        g.dispose();
        // Вызов метода слушателя:
        if(actionListener != null)
            actionListener.actionPerformed(
                new ActionEvent(BangBean.this,
                    ActionEvent.ACTION_PERFORMED, null));
    }
}
class MML extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
        xm = e.getX();
        ym = e.getY();
        repaint();
    }
}
public Dimension getPreferredSize() {
    return new Dimension(200, 200);
}
} ///:~

```

Во-первых, вы заметите, что класс `BangBean` реализует интерфейс `Serializable`. Это значит, что среда разработки способна «законсервировать» информацию о компоненте `BangBean` с помощью механизма сериализации объектов Java, чтобы сохранить измененные пользователем свойства компонента. Когда компонент `Bean` будет применен в качестве части запускаемого приложения, эти «законсервированные» свойства будут восстановлены, и компонент станет выглядеть так, как он был настроен.

Взглянув на описание метода `addActionListener()`, вы увидите, что он может возбуждать исключение `TooManyListenersException`. Таким образом, этот метод поддерживает *одноадресных* (unicast) слушателей, это значит, что о событии уведомляется только один слушатель. Обычно создаются *многоадресные* (multicast) события, о которых уведомляется сразу группа слушателей. Однако при создании таких событий требуется учитывать вопросы многозадачности, чуть позже в этой главе мы рассмотрим создание данного рода компонентов (см. далее раздел «Компоненты JavaBean и синхронизация»). Создание одноадресных событий позволяет обойти эту проблему.

При нажатии кнопки в центре компонента `BangBean` размещается текст, и если поле `actionListener` не содержит ссылку `null`, вызывается его метод `actionPerformed()`, для которого динамически вырабатывается новый объект-событие `ActionEvent`. При передвижениях мыши программа извлекает ее текущие координаты, после чего холст (панель `JPanel`) перерисовывается (соответственно нарисованная ранее окружность стирается).

Класс `BangBeanTest` позволяет вам протестировать созданный компонент:

```

//: bangbean/BangBeanTest.java
// {Timeout: 5} Отмена через 5 секунд
package bangbean;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

public class BangBeanTest extends JFrame {
    private JTextField txt = new JTextField(20);
    // Во время тестирования сообщать о действиях:
    class BBL implements ActionListener {
        private int count = 0;
        public void actionPerformed(ActionEvent e) {
            txt.setText("BangBean action " + count++);
        }
    }
    public BangBeanTest() {
        BangBean bb = new BangBean();
        try {
            bb.addActionListener(new BBL());
        } catch (TooManyListenersException e) {
            txt.setText("Too many listeners");
        }
        add(bb);
        add(BorderLayout.SOUTH, txt);
    }
}

```

```
public static void main(String[] args) {  
    run(new BangBeanTest(), 400, 500);  
}  
} ///:~
```

Внутри среды разработки этот класс для компонента `BangBean` не используется, но тем не менее он достаточно полезен и позволяет провести быстрое тестирование каждого вашего компонента `Bean`. Класс `BangBeanTest` помещает компонент `BangBean` в `JFrame` и присоединяет к нему простого слушателя. Слушатель (`ActionListener`) выводит количество обработанных событий `ActionEvent` в текстовое поле (`JTextField`). Обычно большую часть такого кода автоматически пишет среда разработки, в которой используется этот компонент.

Когда вы будете исследовать компонент `BangBean` с помощью программы `BeanDumper` или подключите этот компонент к любой среде визуальной разработки приложений, вы увидите, что у него имеется больше свойств и событий, чем можно обнаружить по приведенному выше коду. Ведь компонент `BangBean` унаследован от панели `JPanel`, которая также является визуальным компонентом `Bean`; следовательно, вы увидите и ее свойства и события.

- 35.** (6) Найдите и загрузите из Интернета один или несколько бесплатных построителей графического интерфейса или же приобретите коммерческий продукт. Определите, что необходимо сделать для добавления компонента `BangBean` в эту среду программирования, и выполните эти действия.

Компоненты JavaBean и синхронизация

При создании компонента `Bean` всегда следует учитывать возможность его использования в многозадачном окружении. А это значит следующее.

1. Где это только возможно, все открытые (`public`) методы компонента `Bean` должны быть синхронизированными (`synchronized`). Конечно, тогда придется мириться с издержками синхронизации (которые были значительно уменьшены в последних выпусках пакета `JDK`). Если это создает проблемы, методы, которые не причиняют проблем в критических секциях, можно оставить без синхронизации, но помните, что такие методы не всегда очевидны. Обычно они невелики (такие, как метод `getCircleSize()` в следующем примере) и/или «атомарны», то есть их вызов выполняется в таком маленьком количестве кода, что изменения состояния объекта за это время не происходит (но просмотрите главу 21 — то, что вы считаете атомарностью, может таковой не быть). Даже если не синхронизировать такие методы, это вряд ли окажет значительное воздействие на производительность программы. С таким же успехом можно сделать все открытые методы компонента синхронизированными; синхронизация убирается только в том случае, если вы точно знаете, что это даст какой-то положительный результат и не принесет никакого вреда.
2. При инициировании многоадресного события слушателям, заинтересованным в этом событии, нужно иметь в виду возможность добавления или удаления слушателей из списка во время перемещения по списку.

Первый пункт достаточно очевиден, а вот над вторым утверждением надо немного поразмыслить. В примере `BangBean.java` мы обошли проблему синхронизации, игнорируя ключевое слово `synchronized` и используя одноадресные события. Вот этот пример, измененный так, чтобы компонент можно было использовать в многозадачной среде (к тому же теперь в нем используется групповая обработка событий):

```
//: gui/BangBean2.java
// Вам следует создавать свои компоненты Bean
// именно таким способом, чтобы их можно было
// использовать в многозадачном окружении.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

public class BangBean2 extends JPanel
implements Serializable {
    private int xm, ym;
    private int cSize = 20; // Диаметр окружности
    private String text = "Bang!";
    private int fontSize = 48;
    private Color tColor = Color.RED;
    private ArrayList<ActionListener> actionListeners =
        new ArrayList<ActionListener>();
    public BangBean2() {
        addMouseListener(new ML());
        addMouseMotionListener(new MM());
    }
    public synchronized int getCircleSize() { return cSize; }
    public synchronized void setCircleSize(int newSize) {
        cSize = newSize;
    }
    public synchronized String getBangText() { return text; }
    public synchronized void setBangText(String newText) {
        text = newText;
    }
    public synchronized int getFontSize(){ return fontSize; }
    public synchronized void setFontSize(int newSize) {
        fontSize = newSize;
    }
    public synchronized Color getTextColor(){ return tColor;}
    public synchronized void setTextColor(Color newColor) {
        tColor = newColor;
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.BLACK);
        g.drawOval(xm - cSize/2, ym - cSize/2, cSize, cSize);
    }
    // Многоадресный слушатель - явление более типичное, чем
    // одноадресный подход, использованный в BangBean.java:
    public synchronized void
    addActionListener(ActionListener l) {
        actionListeners.add(l);
    }
}
```

```
public synchronized void
removeActionListener(ActionListener l) {
    actionListeners.remove(l);
}
// Обратите внимание: метод не объявлен synchronized:
public void notifyListeners() {
    ActionEvent a = new ActionEvent(BangBean2.this,
        ActionEvent.ACTION_PERFORMED, null);
    ArrayList<ActionListener> lv = null;
    // Создаем поверхностную копию ArrayList
    // на тот случай, если слушатель будет добавлен
    // в момент вызова:
    synchronized(this) {
        lv = new ArrayList<ActionListener>(actionListeners);
    }
    // Вызов методов всех слушателей:
    for(ActionListener al : lv)
        al.actionPerformed(a);
}
class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        Graphics g = getGraphics();
        g.setColor(tColor);
        g.setFont(
            new Font("TimesRoman", Font.BOLD, fontSize));
        int width = g.getFontMetrics().stringWidth(text);
        g.drawString(text, (getSize().width - width) / 2,
            getSize().height/2);
        g.dispose();
        notifyListeners();
    }
}
class MM extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
        xm = e.getX();
        ym = e.getY();
        repaint();
    }
}
public static void main(String[] args) {
    BangBean2 bb2 = new BangBean2();
    bb2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.out.println("ActionEvent" + e);
        }
    });
    bb2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.out.println("BangBean2 action");
        }
    });
    bb2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.out.println("More action");
        }
    });
    JFrame frame = new JFrame();
    frame.add(bb2);
}
```

```

    run(frame, 300, 300);
  }
} ///:~

```

Добавить к методам ключевое слово `synchronized` было несложно. Однако отметьте тот факт, что теперь в методах `addActionListener()` и `removeActionListener()` для слушателей используется список `ArrayList`, так что вы легко сможете закрепить за новым компонентом столько слушателей, сколько потребуется.

Из примера видно, что метод `notifyListeners()` *не* синхронизируется и может вызываться несколькими потоками одновременно. Может оказаться, что метод `addActionListener()` или метод `removeActionListener()` будет вызван прямо во время выполнения метода `notifyListeners()`, что чревато неприятностями, поскольку эти методы обращаются к списку слушателей и изменяют его. Чтобы предотвратить проблему, список `ArrayList` клонируется в секции `synchronized`, после чего для запуска событий используется клон. Таким образом, операции с оригинальным списком `ArrayList` не повлияют на работу метода `notifyListeners()`.

Метод `paintComponent()` также не синхронизирован. Определить, надо ли синхронизировать переопределенные методы или нет, не так просто, как в случае с собственными методами. В этом примере обе версии метода `paintComponent()` — и синхронизированная, и нет — работали прекрасно. Но в любом случае надо учесть следующие факторы.

1. Изменяет ли метод состояние «критических» переменных объекта? Чтобы определить, какие переменные являются «критическими», вы должны определить, используются ли они для чтения и записи другими потоками программы. (В таком случае чтение практически всегда осуществляется синхронизированными методами, и отыскать их не сложно.) В случае с методом `paintComponent()` изменений таких переменных не происходит.
2. Зависит ли метод от состояния этих «критических» переменных? Если некий синхронизированный метод изменяет переменную, которую использует ваш метод, то естественно было бы синхронизировать и ваш метод. На этом основании можно сказать, что раз переменная `cSize` используется синхронизированным методом, то неплохо было бы объявить метод `paintComponent()` как `synchronized`. Впрочем, на этом этапе неплохо спросить самого себя: «Что может произойти, если значение переменной `cSize` изменится во время работы метода `paintComponent()`?» Если ничего страшного не предвидится, можно оставить метод `paintComponent()` без синхронизации, тем самым сэкономя на времени вызова этого метода.
3. Также нужно проверить, синхронизирован ли метод `paintComponent()` из базового класса; оказывается, не синхронизирован. Это не определяющий аргумент, а просто еще один фактор. В нашем случае поле, которое *меняется* синхронизированными методами (то есть поле `cSize`), сплетается с оригинальным методом `paintComponent()` и может в корне изменить ситуацию. Заметьте, однако, что синхронизация не передается по наследству, то есть если метод был синхронизирован в базовом классе, его переопределенная версия в производном классе *не будет* автоматически объявлена синхронизированной.

4. Методы `paint()` и `paintComponent()` должны выполняться как можно быстрее. Все, что требует дополнительных издержек, обязано из них удаляться, так что если вы видите, что такие методы необходимо синхронизировать, это может свидетельствовать о неудачной структуре программы.

Код тестирования `main()` посредством `BangBeanTest` был изменен относительно предшествующего варианта — теперь в нем демонстрируется многоадресная обработка событий, примененная в новой версии компонента `BangBean2`, посредством добавления новых слушателей.

Упаковка компонента Bean

Чтобы компонент Bean можно было использовать в системе визуальной разработки приложений с поддержкой JavaBeans, его необходимо упаковать в стандартный контейнер, используемый для визуальных компонентов. Это файл формата JAR, который наряду с файлами `.class` содержит *манифест* (manifest), утверждающий, что данный класс является компонентом Bean. Файл манифеста — это простой текстовый файл, имеющий определенную структуру. Для класса `BangBean` файл манифеста выглядит следующим образом:

```
Manifest-Version: 1.0
```

```
Name: bangbean/BangBean.class
```

```
Java-Bean: True
```

Первая строка содержит версию используемой схемы манифеста, которая в данный момент равна 1.0. Вторая строка (пустые строки игнорируются) указывает на имя файла `BangBean.class`, и третья строка утверждает, что в указанном файле находится компонент Bean. Если третья строка будет опущена, среда разработки не распознает класс как визуальный компонент.

Единственное место, где часто возникают ошибки, — поле с именем `Name:`, в котором нужно написать правильный путь к файлу с классом. Если вы посмотрите на программу `BangBean.java`, то увидите, что ее класс должен располагаться в пакете `bangbean` (и соответственно, в подкаталоге с именем `bangbean`, который не описан в переменной окружения `CLASSPATH`). Имя файла с классом в манифесте обязательно должно включать в себя имя пакета. Вдобавок файл с манифестом должен находиться в каталоге уровнем *выше* того, в котором находятся файлы пакета; в нашем случае это родительский каталог каталога `bangbean`. Дальше только остается запустить утилиту `jar` из каталога, в котором хранится файл с манифестом:

```
jar cfm BangBean.jar BangBean.mf bangbean
```

Предполагается, что конечному файлу в формате JAR должно быть присвоено имя `BangBean.jar`, а манифест находится в файле с именем `BangBean.mf`.

Вы можете спросить: «А что же все остальные файлы с классами, которые появились после компиляции программы `BangBean.java`?» Они все располагаются в подкаталоге `bangbean`, и вы видите, что на него ссылается последний параметр для запуска `jar`. Когда вы передаете программе `jar` имя подкаталога, она упаковывает все содержащиеся в нем файлы (независимо от типа) в результирующий архив формата JAR (включая

в таком случае и файл с исходным текстом программы `BangBean.java` — возможно, вы не стали бы распространять исходные тексты со своими собственными компонентами Bean). Вдобавок, если вы распакуете только что созданный вами файл JAR, то не найдете в нем файла с манифестом; вместо этого программа `jar` создает свой вариант манифеста (большой частью основанный на оригинале) в файле `MANIFEST.MF`, который размещает в подкаталоге `META-INF` (для «метаданных»). Открыв этот файл, вы увидите, что программа `jar` добавила к каждому файлу цифровую подпись в следующей форме:

```
Digest-Algorithms: SHA MD5
SHA-Digest: pDpEAG9NaeCx8aFtqPI4udSX/00=
MD5-Digest: 04NcS1hE3Smnzlp2hj6qeg==
```

Обычно вам не придется близко знакомиться с содержимым цифровых подписей, а если вы измените компонент, надо будет соответствующим образом отредактировать манифест и заново вызвать упаковщик `jar`, чтобы он создал новый файл JAR для вашего компонента Bean. В файл формата JAR легко добавить другие компоненты Bean — просто присоедините их описание в файл с манифестом.

Стоит отметить, что размещать компоненты Bean в отдельных подкаталогах выгодно, поскольку их имена при упаковке можно будет передать программе `jar`, и `jar` автоматически включит в создаваемый файл JAR все содержимое этих подкаталогов. Вы могли видеть, что известные вам по этой главе компоненты `Frog` и `BangBean` размещены в отдельных каталогах.

После того как компонент Bean будет помещен в файл JAR, вы сможете загрузить его в среде разработки с поддержкой JavaBean. В каждой конкретной среде разработки это делается по-разному, у фирмы Sun есть бесплатная программа для проверки компонентов Bean, ее название `Bean Builder` (доступна для загрузки с <http://java.sun.com/beans>). Для того чтобы проверить работоспособность вашего компонента, достаточно скопировать файл JAR в нужный каталог.

- 36.** (4) Добавьте файл `Frog.class` в файл с манифестом, как показано в этой главе, и запустите инструмент `jar` с целью получения файла формата JAR, который должен содержать как компонент `Frog`, так и компонент `BangBean`. Затем загрузите и установите пакет `Bean Builder` от фирмы Sun или используйте свое собственное средство визуального программирования с поддержкой JavaBeans. Добавьте полученный файл JAR в свое окружение и протестируйте оба компонента Bean.
- 37.** (5) Создайте ваш собственный компонент Bean с названием `Valve`, в котором определите два свойства: логическое (`boolean`) значение с именем `on` и целое (`int`) с именем `level`. Подготовьте манифест, запустите инструмент `jar` для упаковки компонента Bean, затем загрузите полученный файл в программу `Bean Builder` или в среду разработки с поддержкой JavaBeans для последующего тестирования.

Поддержка более сложных компонентов Bean

Как вы могли убедиться, создать компонент Bean несложно, однако ваши возможности не ограничены тем, что было показано выше. Архитектура технологии JavaBeans позволяет быстро создавать простые компоненты, но при необходимости можно перейти и к более сложным решениям. Такие решения выходят за рамки темы этой книги, однако коротко мы их охарактеризуем. Подробности вы найдете на сайте <http://java.sun.com/beans>.

Одно из направлений повышения гибкости — свойства компонента. Рассмотренные нами примеры использовали только одиночные свойства, однако допустимы и массивы свойств, отсортированных по номерам. Такие свойства называются *индексированными*. Вы просто определяете подходящие методы (следуя соглашению об именах методов), а Introspector распознает индексированные свойства, чтобы среда разработки могла адекватно отреагировать.

Свойства могут быть *связанными* (bound) (друг с другом), это значит, что они будут оповещать другие объекты о своих изменениях с помощью события `PropertyChangeEvent`. Другие объекты способны изменять свое состояние в зависимости от характера изменений в компоненте Bean.

Свойства могут быть *ограниченными* (constrained), это значит, что другие объекты вправе запрещать изменение свойства, если сочтут его неприемлемым. Другие объекты оповещаются об изменениях свойства с помощью события `PropertyChangeEvent`, неприемлемые модификации блокируются посредством возбуждения исключения `PropertyVetoException`.

Также возможно переопределить способ представления компонента Bean во время разработки программы.

1. Можно предоставить собственный список свойств для определенного компонента Bean. Для всех остальных компонентов Bean будет использоваться обычный список свойств, но при выборе этого компонента Bean автоматически будет появляться специальный список.
2. Можно создать специальный редактор для определенного свойства; тогда будет использоваться обычный список свойств компонента, однако при редактировании этого свойства будет автоматически запускаться ваш редактор.
3. Разрешается применять свой собственный класс с информацией о компоненте `BeanInfo`, который будет возвращать информацию, отличную от той, которую возвращает по умолчанию Introspector.
4. Также позволено включить и выключить режим «эксперта» в описаниях `FeatureDescriptor`, чтобы отделить основные свойства компонента от дополнительных, более сложных его возможностей.

Больше о компонентах Bean

О технологии JavaBeans написано немало книг, например книга Эллиота Расти Гарольда *JavaBeans* (издательство IDG, 1998).

Альтернативы для Swing

Хотя Sun рекомендует применять библиотеку Swing для построения графического интерфейса, этот способ создания графических интерфейсов ни в коем случае не является единственным. Среди альтернативных технологий следует выделить Macromedia Flash с использованием системы программирования Macromedia Flex (для графических интерфейсов в веб-приложениях на стороне клиента) и библиотеку с открытым кодом Eclipse Standard Widget Toolkit (SWT) для настольных приложений.

Зачем рассматривать альтернативы, спросите вы? Для веб-клиентов довольно убедительным аргументом может стать неудача апплетов. Апплеты существуют уже давно, их появление сопровождалось шумихой и массой обещания, — а веб-приложения с использованием апплетов встречаются довольно редко. Даже Sun не везде применяет апплеты, вот пример:

<http://java.sun.com/developer/onlineTraining/new2java/javamap/intro.html>

Интерактивная карта возможностей Java на сайте Sun кажется очень подходящим кандидатом для апплета Java, и все же она сделана на базе Flash. Этот факт, похоже, подтверждает, что технология апплетов провалилась. Что еще важнее, проигрыватель Flash Player установлен на 98 процентах вычислительных платформ, поэтому его можно считать фактическим стандартом. Как вы вскоре увидите, система Flex предоставляет очень мощные средства программирования на стороне клиента — безусловно более мощные, чем JavaScript, а также превосходит апплеты в отношении оформления и поведения. Чтобы использовать апплеты, вам придется убедить клиента загрузить JRE, тогда как Flash Player относительно компактен и быстро загружается.

У применения Swing для настольных приложений есть один недостаток: пользователи *видят*, что они используют приложение другого типа, потому что по своему оформлению и поведению приложения Swing отличаются от классических настольных приложений. Пользователи обычно не стремятся к новому оформлению и поведению; они хотят побыстрее выполнить свою работу и предпочитают, чтобы приложение выглядело и работало как другие приложения. Приложения SWT выглядят традиционно, а поскольку библиотека по возможности использует системные компоненты, приложения SWT обычно работают быстрее эквивалентных приложений Swing.

Построение веб-клиентов Flash с использованием Flex

Так как легковесные виртуальные машины Macromedia Flash получили повсеместное распространение, большинство пользователей сможет использовать интерфейс на базе Flash без установки дополнительных компонентов, и такие приложения будут выглядеть и работать одинаково на всех системах и платформах¹.

Используя технологию Macromedia Flex, вы можете разрабатывать пользовательские интерфейсы Flash для ваших Java-приложений. Flex состоит из модели программирования на базе XML и сценариев, сходной с такими моделями, как HTML и JavaScript, а также мощной библиотеки компонентов. Для управления макетом и размещением виджетов применяется синтаксис MXML, а технология динамических сценариев используется для добавления кода обработки событий и обращений к сервисным функциям, который связывает пользовательский интерфейс с классами Java, моделями данных, веб-службами и т. д. Компилятор Flex берет разметку MXML и файлы сценариев и компилирует их в байт-код. Виртуальная машина Flash на стороне клиента

¹ Основной материал этого раздела предоставлен Шоном Невиллом.

работает по тому же принципу, что и виртуальная машина Java: она тоже интерпретирует откомпилированный байт-код. Формат байт-кода Flash называется SWF, а файлы SWF создаются компилятором Flex.

Учтите, что у Flex существует альтернатива с открытым кодом на сайте <http://open-laszlo.org>; по своей структуре она близка к Flex, но, возможно, кому-то этот вариант покажется предпочтительным. Также существуют другие инструменты для создания Flash-приложений.

Первое приложение Flex

Следующий код MXML определяет пользовательский интерфейс (обратите внимание: первая и последняя строки отсутствуют в коде, который вы загрузите в составе пакета исходного кода книги):

```
//:! gui/flex/helloflex1.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.macromedia.com/2003/mxml"
  backgroundColor="#ffffff">
  <mx:Label id="output" text="Hello, Flex!" />
</mx:Application>
///:~
```

Файлы MXML содержат данные в формате XML, поэтому они начинаются с директивы XML, определяющей версию и кодировку. Внешний элемент MXML — `Application` — является визуальным и логическим контейнером пользовательского интерфейса Flex верхнего уровня. Внутри элемента `Application` размещаются теги, представляющие визуальные элементы управления (такие, как приведенный выше элемент `Label`). Элементы управления всегда размещаются в контейнерах, а контейнеры, среди прочего, содержат менеджеры расположения, управляющие размещением находящихся в них элементов. В простейшем случае, как в нашем примере, `Application` выполняет функции контейнера. Менеджер расположения `Application` по умолчанию просто выстраивает элементы управления сверху вниз в порядке их объявления.

ActionScript — версия ECMAScript, или JavaScript, которая имеет немало общего с Java и поддерживает классы и сильную типизацию. Включая сценарии в пример, можно добавить в него новое поведение. Здесь элемент управления MXML Script используется для включения ActionScript прямо в файл MXML:

```
//:! gui/flex/helloflex2.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.macromedia.com/2003/mxml"
  backgroundColor="#ffffff">
  <mx:Script>
    <![CDATA[
      function updateOutput() {
        output.text = "Hello! " + input.text;
      }
    ]]>
  </mx:Script>
  <mx:TextInput id="input" width="200"
```

продолжение ➤

```

    change="updateOutput()" />
    <mx:Label id="output" text="Hello!" />
</mx:Application>
///:~

```

Элемент управления `TextInput` получает данные, введенные пользователем, а `Label` отображает данные в процессе ввода. Обратите внимание: атрибут `id` каждого элемента управления становится доступным в сценарии как имя переменной, так что сценарий может ссылаться на экземпляры тегов MXML. В поле `TextInput` атрибут `change` связывается с функцией `updateOutput()`, чтобы эта функция вызывалась при любых изменениях.

Компилирование MXML

Работу с Flex проще всего начать с бесплатной пробной версии, которую можно загрузить на сайте www.macromedia.com/software/flex/trial¹. Продукт распространяется в нескольких версиях, от бесплатных пробных до корпоративных серверных; кроме того, Macromedia предоставляет дополнительные инструменты для разработки приложений Flex. Состав каждой версии может изменяться, за подробностями обращайтесь на сайт Macromedia. Возможно, вам также придется внести изменения в файле `jvm.config` из каталога `bin` установки Flex.

Чтобы откомпилировать код MXML в байт-код Flash, у вас имеются два варианта.

1. Поместите файл MXML в веб-приложение Java вместе со страницами HTML и JSP в файл WAR и организуйте компиляцию запросов файла `.mxmlc` во время выполнения при каждом запросе URL-адреса документа MXML из браузера.
2. Откомпилируйте файл MXML при помощи компилятора командной строки Flex `mxmlc`.

Для первого варианта (компиляции во время выполнения) помимо Flex потребуется контейнер сервлетов (такой, как Apache Tomcat). Файл(-ы) WAR контейнера сервлетов необходимо обновить данными конфигурации Flex, и он должен включать JAR-файлы Flex — все эти операции выполняются автоматически при установке Flex. После настройки файла WAR вы можете поместить файлы MXML в веб-приложение и запросить URL-адрес документа в любом браузере. Flex откомпилирует приложение по первому запросу (по аналогии с моделью JSP) и в дальнейшем будет поставлять откомпилированный и кэшированный SWF-контейнер в оболочке HTML.

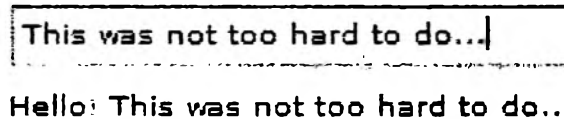
Для второго варианта сервер не понадобится. При вызове компилятора Flex `mxmlc` из командной строки создаются файлы SWF, которые вы можете развертывать по своему усмотрению. Исполняемый файл `mxmlc` находится в каталоге `bin` установки Flex, а при запуске без аргументов он выводит список допустимых параметров командной строки. Обычно в командной строке указывается местоположение библиотеки клиентских компонентов Flex (значение параметра `-flexlib`), но в очень простых случаях — как в первых двух примерах — компилятор Flex может попытаться определить местоположение библиотеки компонентов самостоятельно. Итак, первые два примера можно откомпилировать следующими командами:

¹ Обратите внимание: загружать надо Flex, а не Flex Builder (это среда разработки).

```
mxmlc.exe helloflex1.mxml  
mxmlc.exe helloflex2.mxml
```

Команды создают файл `helloflex2.swf`, который можно запустить в Flash или разместить вместе с HTML на любом сервере HTTP (после того, как поддержка Flash будет загружена в браузере, часто бывает достаточно сделать двойной щелчок на файле SWF, чтобы запустить его в браузере).

Для файла `helloflex2.swf` пользовательский интерфейс, отображаемый в Flash Player, будет выглядеть так.



В более сложных приложениях MXML может отделяться от ActionScript, для чего в разметку включаются функции из внешних файлов Android. В MXML для элементов управления Script используется следующий синтаксис:

```
<mx:Script source="MyExternalScript.as" />
```

Этот код позволяет элементам управления MXML ссылаться на функции, содержащиеся в файле с именем `MyExternalScript.as`, как если бы они находились в файле MXML.

MXML и ActionScript

MXML представляет собой декларативную сокращенную запись для классов ActionScript. Везде, где вы видите тег MXML, существует класс ActionScript с тем же именем. Разбирая разметку MXML, компилятор Flex сначала преобразует XML в ActionScript и загружает классы ActionScript, упоминаемые в ссылках, после чего компилирует и компоует ActionScript в SWF.

Приложение Flex может быть написано полностью на ActionScript, без использования MXML. Таким образом, MXML — всего лишь вспомогательный инструмент. Компоненты пользовательского интерфейса — такие, как контейнеры и элементы управления, — обычно объявляются в MXML, тогда как логика обработки событий и другая клиентская логика обеспечивается в коде ActionScript и Java.

Вы можете создать собственные элементы управления MXML и включить ссылки на них в разметку MXML, написав для этого классы ActionScript. Также можно объединить существующие элементы управления и контейнеры MXML в новом документе MXML, ссылка на который может быть включена в виде тега в другой документ MXML. О том, как это делается, более подробно рассказано на веб-сайте Macromedia.

Контейнеры и элементы управления

Визуальное ядро библиотеки компонентов Flex состоит из набора контейнеров, управляющих расположением, и элементов управления, размещаемых в этих контейнерах. К числу контейнеров относятся панели, вертикальные и горизонтальные поля, сетки

и т. д. Элементы управления представляют собой виджеты пользовательского интерфейса: кнопки, текстовые поля, регуляторы, календари, сетки данных и т. д.

В оставшейся части этого раздела будет представлено приложение Flex, которое отображает и сортирует список аудиофайлов. Оно демонстрирует использование контейнеров, элементов управления, а также связывание с кодом Java из Flash.

Работа над файлом MXML начинается с размещения элемента управления DataGrid (один из относительно сложных элементов управления Flex) в контейнере Panel:

```
//:| gui/flex/songs.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.macromedia.com/2003/mxml"
  backgroundColor="#B9CAD2" pageTitle="Flex Song Manager"
  initialize="getSongs()">
  <mx:Script source="songScript.as" />
  <mx:Style source="songStyles.css"/>
  <mx:Panel id="songListPanel"
    titleStyleDeclaration="headerText"
    title="Flex MP3 Library">
    <mx:HBox verticalAlign="bottom">
      <mx>DataGrid id="songGrid"
        cellPress="selectSong(event)" rowCount="8">
        <mx:columns>
          <mx:Array>
            <mx>DataGridColumn columnName="name"
              headerText="Song Name" width="120" />
            <mx>DataGridColumn columnName="artist"
              headerText="Artist" width="180" />
            <mx>DataGridColumn columnName="album"
              headerText="Album" width="160" />
          </mx:Array>
        </mx:columns>
      </mx>DataGrid>
      <mx:VBox>
        <mx:HBox height="100" >
          <mx:Image id="albumImage" source=""
            height="80" width="100"
            mouseOverEffect="resizeBig"
            mouseOutEffect="resizeSmall" />
          <mx:TextArea id="songInfo"
            styleName="boldText" height="100%" width="120"
            vScrollPolicy="off" borderStyle="none" />
        </mx:HBox>
        <mx:MediaPlayback id="songPlayer"
          contentPath=""
          mediaType="MP3"
          height="70"
          width="230"
          controllerPolicy="on"
          autoPlay="false"
          visible="false" />
      </mx:VBox>
    </mx:HBox>
    <mx:ControlBar horizontalAlign="right">
      <mx:Button id="refreshSongsButton"
```



```
        label="Refresh Songs" width="100"  
        tooltip="Refresh Song List"  
        click="songService.getSongs()" />  
    </mx:ControlBar>  
</mx:Panel>  
<mx:Effect>  
    <mx:Resize name="resizeBig" heightTo="100"  
        duration="500"/>  
    <mx:Resize name="resizeSmall" heightTo="80"  
        duration="500"/>  
</mx:Effect>  
<mx:RemoteObject id="songService"  
    source="gui.flex.SongService"  
    result="onSongs(event.result)"  
    fault="alert(event.fault.faultstring, 'Error')">  
    <mx:method name="getSongs"/>  
</mx:RemoteObject>  
</mx:Application>  
///:~
```

Элемент управления `DataGrid` содержит вложенные теги для своего массива столбцов. Когда вы видите атрибут или вложенный тег у элемента управления, вы знаете, что он соответствует некоторому свойству, событию или инкапсулированному объекту нижележащего класса `ActionScript`. `DataGrid` имеет атрибут `id` со значением `songGrid`, поэтому теги MXML и `ActionScript` могут обращаться к сетке на программном уровне, используя имя переменной `songGrid`. У `DataGrid` намного больше свойств, чем показано в нашем примере; полное описание API элементов управления MXML и контейнеров можно найти по адресу http://livedocs.macromedia.com/flex/15/asdocs_en/index.html.

За сеткой `DataGrid` следует элемент `VBox`, в котором содержится изображение (`Image`) с обложкой альбома и информация о песне, а также элемент управления `MediaPlayback` для воспроизведения файлов MP3. В этом примере осуществляется потоковая загрузка контента для сокращения размера откомпилированного файла SWF. Изображения, аудио- и видеофайлы, встроенные в приложение Flex, становятся частью откомпилированного файла SWF и устанавливаются вместе с ресурсами пользовательского интерфейса — в отличие от потоковой загрузки «по требованию» во время выполнения.

Flash Player содержит встроенные кодеки для воспроизведения и потоковой загрузки аудио- и видеоматериалов в разных форматах. Flash и Flex поддерживают использование наиболее распространенных графических форматов Web, а Flex также предоставляет возможность преобразования файлов SVG (Scalable Vector Graphics) в ресурсы SWF, которые могут встраиваться в клиентов Flex.

Эффекты и стили

Flash Player выводит графику с использованием векторных операций, что позволяет выполнять выразительные преобразования во время выполнения. *Эффекты* Flex дают некоторое представление о таких анимациях. Эти эффекты применяются к событиям мыши, поддерживаемым элементом управления `Image`, для `albumImage`.

Flex также предоставляет эффекты для таких распространенных анимаций, как переходы, постепенная смена изображений и модуляция альфа-каналов. Помимо встроенных

эффектов, Flex поддерживает API графического вывода Flash для действительно интересных анимаций. Более глубокое изложение этой темы требует знакомства с основами компьютерной графики и анимации и выходит за рамки данного раздела.

Стандартное стилевое оформление основано на поддержке каскадных таблиц стилей (CSS, Cascading Style Sheets) в Flex. Если присоединить файл CSS к файлу MXML, определенные в нем стили будут применяться к элементам управления Flex. Например, файл `songStyles.css` содержит следующее объявление CSS:

```

//:! gui/flex/songStyles.css
.headerText {
    font-family: Arial, "_sans";
    font-size: 16;
    font-weight: bold;
}

.boldText {
    font-family: Arial, "_sans";
    font-size: 11;
    font-weight: bold;
}
///:~

```

Этот файл импортируется и используется в приложении, для чего в файл MXML включается тег `Style`. После импортирования таблицы стилей ее объявления применяются к элементам управления Flex в файле MXML. Например, объявление `boldText` из таблицы стилей используется элементом управления `TextArea` с идентификатором `songInfo`.

События

Пользовательский интерфейс может рассматриваться как конечный автомат; он выполняет действия при изменении состояний. В Flex для управления этими изменениями используются события. Библиотека классов Flex содержит обширный набор элементов управления с многочисленными событиями, описывающими все аспекты перемещения мыши и использования клавиатуры.

Например, атрибут `click` элемента управления `Button` представляет одно из событий элемента управления. Значение, присвоенное `click`, представляет функцию или встроенный фрагмент сценария. Например, в файле MXML элемент управления `ControlBar` содержит кнопку `refreshSongsButton` для обновления списка файлов. Из тега видно, что при возникновении события `click` вызывается метод `songService.getSongs()`. В данном примере событие `click` элемента управления `Button` содержит ссылку на объект `RemoteObject`, соответствующий методу Java.

Связывание с Java

Тег `RemoteObject` в конце файла MXML устанавливает связь с внешним классом Java `gui.flex.SongService`. Клиент Flex использует метод `getSongs()` класса Java для получения данных `DataGrid`. Для этого он должен выглядеть как *служба* (service) — конечная точка, с которой клиент может обмениваться сообщениями. Служба, определяемая в теге `RemoteObject`, содержит атрибут `source`, обозначающий класс Java для `RemoteObject`,

а также задает функцию обратного вызова ActionScript с именем `onSongs()`, который должен вызываться при возвращении управления методом Java. Вложенный тег `method` объявляет метод `getSongs()`, который делает этот метод Java доступным для остального приложения Flex.

Все обращения к службам в Flex возвращают управление асинхронно — через события, иницилируемые для функций обратного вызова. Объект `RemoteObject` также открывает диалоговое окно с оповещением в случае ошибки.

Теперь метод `getSongs()` может вызываться из Flash средствами ActionScript:

```
songService.getSongs();
```

При заданной конфигурации MXML это приведет к вызову метода `getSongs()` класса `SongService`:

```
//: gui/flex/SongService.java
package gui.flex;
import java.util.*;

public class SongService {
    private List<Song> songs = new ArrayList<Song>();
    public SongService() { fillTestData(); }
    public List<Song> getSongs() { return songs; }
    public void addSong(Song song) { songs.add(song); }
    public void removeSong(Song song) { songs.remove(song); }
    private void fillTestData() {
        addSong(new Song("Chocolate", "Snow Patrol",
            "Final Straw", "sp-final-straw.jpg",
            "chocolate.mp3"));
        addSong(new Song("Concerto No. 2 in E", "Hilary Hahn",
            "Bach: Violin Concertos", "hahn.jpg",
            "bachviolin2.mp3"));
        addSong(new Song("'Round Midnight", "Wes Montgomery",
            "The Artistry of Wes Montgomery",
            "wesmontgomery.jpg", "roundmidnight.mp3"));
    }
} ///:~
```

Каждый объект `Song` представляет собой обычный контейнер данных:

```
//: gui/flex/Song.java
package gui.flex;

public class Song implements java.io.Serializable {
    private String name;
    private String artist;
    private String album;
    private String albumImageUrl;
    private String songMediaUrl;
    public Song() {}
    public Song(String name, String artist, String album,
        String albumImageUrl, String songMediaUrl) {
        this.name = name;
        this.artist = artist;
        this.album = album;
        this.albumImageUrl = albumImageUrl;
    }
}
```

продолжение ➤

```

    this.songMediaUrl = songMediaUrl;
}
public void setAlbum(String album) { this.album = album;}
public String getAlbum() { return album; }
public void setAlbumImageUrl(String albumImageUrl) {
    this.albumImageUrl = albumImageUrl;
}
public String getAlbumImageUrl() { return albumImageUrl;}
public void setArtist(String artist) {
    this.artist = artist;
}
public String getArtist() { return artist; }
public void setName(String name) { this.name = name; }
public String getName() { return name; }
public void setSongMediaUrl(String songMediaUrl) {
    this.songMediaUrl = songMediaUrl;
}
public String getSongMediaUrl() { return songMediaUrl; }
} ///:~

```

При инициализации приложения или при нажатии кнопки `refreshSongsButton` вызывается метод `getSongs()`, а при возвращении управления вызывается обработчик `ActionScript onSongs(event.result)` для заполнения `songGrid`.

Ниже приведен код `ActionScript`, включаемый при помощи элемента управления `Script` в файле `MXML`:

```

//: gui/flex/songScript.as
function getSongs() {
    songService.getSongs();
}

function selectSong(event) {
    var song = songGrid.getItemAt(event.itemIndex);
    showSongInfo(song);
}

function showSongInfo(song) {
    songInfo.text = song.name + newline;
    songInfo.text += song.artist + newline;
    songInfo.text += song.album + newline;
    albumImage.source = song.albumImageUrl;
    songPlayer.contentPath = song.songMediaUrl;
    songPlayer.visible = true;
}

function onSongs(songs) {
    songGrid.dataProvider = songs;
} ///:~

```

Чтобы обрабатывать выделение ячеек `DataGrid`, мы добавляем атрибут события `cellPress` в объявление `DataGrid` в файл `MXML`:

```
cellPress="selectSong(event)"
```

Когда пользователь щелкает на песне в сетке `DataGrid`, это приводит к вызову `selectSong()` в приведенном выше коде `ActionScript`.

Модели данных и связывание данных

Элементы управления могут напрямую вызывать службы, а функции обратного вызова ActionScript дают возможность на программном уровне обновлять состояние визуальных элементов при возвращении данных службами. Хотя сценарный код, обновляющий элементы, достаточно тривиален, он быстро разрастается и становится громоздким; с другой стороны, эта задача встречается настолько часто, что Flex обрабатывает это поведение автоматически посредством *связывания данных* (data binding).

В простейшей форме связывание данных позволяет элементам управления напрямую обращаться к данным без вспомогательного кода, копирующего данные в элемент управления. При обновлении данных элемент, ссылающийся на них, также обновляется автоматически, без вмешательства со стороны программиста. Инфраструктура Flex корректно реагирует на события изменения данных и обновляет все элементы, связанные с данными.

Простой пример синтаксиса связывания данных:

```
<mx:Slider id="mySlider"/>
<mx:Text text="{mySlider.value}"/>
```

Чтобы выполнить связывание данных, следует заключить ссылку в фигурные скобки: {}. Все, что находится в фигурных скобках, интерпретируется как выражение, которое должно быть обработано Flex.

Значение первого элемента управления — виджета `Slider` — отображается во втором элементе, поле `Text`. При изменении `Slider` свойство `text` поля `Text` обновляется автоматически. При этом разработчику не нужно обрабатывать события изменения `Slider`, чтобы обновить поле `Text`.

С некоторыми элементами управления — такими, как `Tree` и `DataGrid` в нашем приложении, — дело обстоит сложнее. Они предоставляют свойство `dataProvider`, упрощающее связывание с наборами данных. Функция ActionScript `onSongs()` показывает, как метод `SongService.getSongs()` связывается со свойством `dataProvider` элемента `DataGrid`. Как указано в теге `RemoteObject` из файла MXML, ActionScript использует эту функцию для обратного вызова при возвращении из метода Java.

Более сложное приложение с более сложной моделью данных — например, корпоративное приложение, использующее объекты передачи данных (DTO), или система передачи сообщений с данными, соответствующими сложной схеме, — могут стимулировать дальнейшее отделение источника данных от элементов управления. В программировании Flex для этого объявляется «объект модели» — обобщенный контейнер MXML для хранения данных. Модель не содержит собственной логики. Она напоминает объекты передачи данных в корпоративных приложениях или структуры из других языков программирования. Элементы управления связываются с моделью, и в то же время модель связывает свои свойства с входом и выходом служб. Таким образом источники данных — службы — отделяются от визуальных потребителей данных, что упрощает использование паттерна «Модель-Представление-Контроллер» (MVC). В больших, более сложных приложениях возрастание сложности, обусловленное введением модели, часто с избытком компенсируется пользой от четкости логического деления в приложениях MVC.

Кроме объектов Java, Flex также может обращаться к веб-службам на базе SOAP и REST-совместимым службам HTTP с использованием элементов управления `WebService` и `HttpService` соответственно. Доступ ко всем службам осуществляется с учетом ограничений системы безопасности.

Построение и развертывание

В предыдущих примерах мы обходились без флага `-flexlib` в командной строке, но для компиляции этой программы необходимо задать местонахождение файла `flex-config.xml`. В моей установке следующая команда работает, но вам придется привести ее в соответствие со своей конфигурацией (команда состоит из одной строки, разбитой для удобства чтения):

```
//:! gui/flex/build-command.txt
mxmmlc -flexlib C:/Program
Files"/Macromedia/Flex/jrun4/servers/default/flex/WEB-
INF/flex songs.mxml
///:~
```

Команда строит из приложения файл SWF, который можно просматривать в браузере, но архив кода книги не содержит файлов MP3 или JPG, поэтому при запуске приложения вы не увидите ничего, кроме общей структуры.

Кроме того, для успешного взаимодействия с файлами Java из приложения Flex необходимо выполнить соответствующую настройку сервера. Пробный пакет Flex включает сервер JRun, который можно запустить из меню компьютера после установки Flex или же из командной строки:

```
jrun -start default
```

Чтобы убедиться в том, что сервер был успешно запущен, откройте в браузере страницу <http://localhost:8700/samples> и просмотрите примеры (а заодно ознакомьтесь с возможностями Flex).

Вместо того чтобы компилировать приложение в командной строке, вы также можете откомпилировать его через сервер. Для этого перенесите исходные файлы, таблицу стилей CSS и т. д. в каталог `jrun4/servers/default/flex` и обратитесь к ним из браузера по адресу <http://localhost:8700/flex/songs.mxml>.

Чтобы успешно запустить приложение, необходимо выполнить настройку как на стороне Java, так и на стороне Flex.

Java: откомпилированные файлы `Song.java` и `SongService.java` должны находиться в каталоге `WEB-INF/classes`. Именно здесь должны находиться классы WAR согласно спецификации J2EE. Также можно упаковать файлы в формат JAR и перенести их в `WEB-INF/lib`. Если вы используете JRun, они будут находиться в `jrun4/servers/default/flex/WEB-INF/classes/gui/flex/Song.class` и `jrun4/servers/default/flex/WEB-INF/classes/gui/flex/SongService.class` соответственно. Также необходимо предоставить веб-приложению доступ к файлам графики и MP3 (для JRun корневым каталогом веб-приложения является `jrun4/servers/default/flex`).

Flex: по соображениям безопасности Flex не может обращаться к объектам Java, если только вы явно не разрешите такие обращения, внося соответствующие изменения

в файл `flex-config.xml`. Для JRun он находится в каталоге `jrun4/servers/default/flex/WEB-INF/flex/`. Найдите в файле раздел `<remote-objects>`, найдите в нем элемент `<whitelist>` с записью следующего содержания:

```
<!--
```

По соображениям безопасности "белый список" по умолчанию заблокирован. Раскомментируйте элемент `source`, чтобы разрешить доступ ко всем классам в ходе разработки.

Настоятельно не рекомендуется предоставлять доступ к исходным файлам в окончательной версии, так как это откроет доступ к системным классам Java и Flex.

```
<source>*</source>
```

```
-->
```

Раскомментируйте запись `<source>`, чтобы разрешить доступ (строка должна принять вид `<source>*</source>`). Смысл этой и других записей описан в документации по конфигурации Flex.

- 38.** (3) Постройте «простой пример синтаксиса связывания данных», приведенный выше.
- 39.** (4) Архив исходного кода этой книги не включает файлы MP3 и JPG, упомянутые в `SongService.java`. Найдите какие-нибудь файлы MP3 и JPG, отредактируйте `SongService.java`, загрузите пробную версию Flex и построьте приложение.

Создание приложений SWT

Как упоминалось ранее, в Swing все компоненты пользовательского интерфейса строятся «с нуля», чтобы они были доступны разработчику независимо от того, поддерживаются такие компоненты операционной системой или нет. В SWT выбран промежуточный вариант: она использует «родные» компоненты, если они предоставляются операционной системой, и синтезирует отсутствующие компоненты. В результате приложение выглядит более знакомо для пользователя и часто обладает заметно более высокой производительностью, чем эквивалентная программа Swing. Кроме того, модель программирования SWT обычно проще модели Swing, что может оказаться положительным фактором во многих приложениях¹.

Так как SWT старается использовать средства ОС для выполнения как можно большей части своей работы, эта среда может автоматически использовать возможности ОС, недоступные для Swing, — например, в Windows реализован режим «субпиксельной визуализации», в котором шрифты более четко отображаются на ЖК-экранах.

SWT даже может использоваться для создания апплетов.

Этот раздел не содержит подробного введения в SWT; он всего лишь дает общее представление о технологии и показывает, чем SWT отличается от Swing. Вы узнаете, что в SWT существует много виджетов и все они достаточно просты в использовании. Поподробности можно найти в полной документации и многочисленных примерах на сайте www.eclipse.org. Вышло несколько книг, посвященных программированию для SWT, и за ними следуют другие.

¹ Крис Гриндстафф помог адаптировать примеры и предоставил информацию о SWT.

Установка SWT

Для работы приложений SWT необходимо загрузить и установить библиотеку SWT с сайта проекта Eclipse. Откройте страницу www.eclipse.org/downloads/ и выберите зеркальный сайт. Перейдите по ссылкам к текущей сборке Eclipse и найдите сжатый файл с именем, которое начинается с «swt» и включает имя платформы (например, «win32»). В этом файле вы найдете файл `swt.jar`. В простейшем варианте установки файл `swt.jar` просто помещается в каталог `jre/lib/ext` (при этом вам не придется вносить никаких изменений в `CLASSPATH`). При распаковке библиотеки SWT могут обнаружиться дополнительные файлы, которые необходимо установить в соответствующих местах вашей платформы. Например, в поставку Win32 входят файлы DLL, которые должны находиться где-то в `java.library.path` (обычно совпадает с переменной окружения `PATH`, но вы можете выполнить программу `object/ShowProperties.java` для получения фактического значения `java.library.path`). После того как это будет сделано, вы сможете прозрачно компилировать и выполнять приложения SWT так, как любые другие программы Java.

Документация SWT находится в отдельном загружаемом пакете.

Другой возможный путь — установка редактора Eclipse, включающего SWT и документацию SWT, для просмотра которой можно воспользоваться справочной системой Eclipse.

Первое приложение

Начнем с простейшего приложения в стиле канонической программы «Hello World»:

```
//: swt/HelloSWT.java
// {Requires: org.eclipse.swt.widgets.Display; You must
// install the SWT library from http://www.eclipse.org }
import org.eclipse.swt.widgets.*;

public class HelloSWT {
    public static void main(String [] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Hi there, SWT!"); // Заголовок
        shell.open();
        while(!shell.isDisposed())
            if(!display.readAndDispatch())
                display.sleep();
        display.dispose();
    }
} //::~
```

Загрузив исходный код этой книги, вы увидите, что директива-комментарий `Requires` попадает в файл Ant `build.xml` как предусловие для построения подкаталога `swt`; все файлы, импортирующие `org.eclipse.swt`, требуют установки библиотеки SWT с сайта www.eclipse.org.

Класс `Display` управляет связью между SWT и операционной системой — это часть «моста» между операционной системой и SWT. `Shell` представляет окно верхнего

уровня, в котором строятся все остальные компоненты. При вызове `setText()` аргумент становится текстом надписи в заголовке окна.

Чтобы вывести на экран окно (а следовательно, и приложение), необходимо вызвать `open()` для `Shell`.

Swing скрывает цикл обработки событий от разработчика, но в SWT он должен быть создан явно. В начале цикла вы проверяете, был ли объект `Shell` уничтожен, — это позволяет вставить код для выполнения завершающих действий. Но это также означает, что поток `main()` является потоком пользовательского интерфейса. В Swing «за кулисами» создается второй поток диспетчеризации событий, но в SWT с пользовательским интерфейсом работает главный поток. Так как по умолчанию используется только один поток, а не два, это несколько снижает вероятность того, что пользовательский интерфейс будет парализован потоками.

Заметьте, что вам не нужно заниматься отправкой задач потоку пользовательского интерфейса, как это делалось в Swing. SWT не только делает это за вас, но и возбуждает исключение при попытке выполнить операцию с виджетом из неверного потока. Однако если вам потребуется породить другие потоки для выполнения продолжительных операций, изменения все равно придется отправлять так же, как при использовании Swing. Для этого SWT предоставляет три метода, вызываемых для объекта `Display`: `asyncExec(Runnable)`, `syncExec(Runnable)` и `timerExec(int, Runnable)`.

Метод `main()` на этой стадии вызывает `readAndDispatch()` для объекта `Display` (это означает, что в приложении может быть только один объект `Display`). Метод `readAndDispatch()` возвращает `true`, если в очереди имеются другие события, ожидающие обработки. В этом случае он должен быть немедленно вызван повторно. Но если ожидающих событий нет, вызывается метод `sleep()` объекта `Display`, который делает непродолжительную паузу, прежде чем снова проверять очередь событий.

Когда программа будет завершена, объект `Display` необходимо явно освободить вызовом `dispose()`. SWT часто заставляет явно освобождать ресурсы, потому что обычно используются ресурсы операционной системы, которые могут быть исчерпаны.

Чтобы показать, что объект `Shell` представляет главное окно программы, следующая программа создает несколько объектов `Shell`:

```
///  
swt/ShellsAreMainWindows.java  
import org.eclipse.swt.widgets.*;  
  
public class ShellsAreMainWindows {  
    static Shell[] shells = new Shell[10];  
    public static void main(String [] args) {  
        Display display = new Display();  
        for(int i = 0; i < shells.length; i++) {  
            shells[i] = new Shell(display);  
            shells[i].setText("Shell #" + i);  
            shells[i].open();  
        }  
        while(!shellsDisposed())  
            if(!display.readAndDispatch())  
                display.sleep();  
    }  
}
```

продолжение ➤

```

        display.dispose();
    }
    static boolean shellsDisposed() {
        for(int i = 0; i < shells.length; i++)
            if(shells[i].isDisposed())
                return true;
        return false;
    }
} ///:~

```

При запуске открываются десять главных окон. Если вы закроете одно из них, это приведет к закрытию всех окон. SWT также использует менеджеры расположения — отличные от тех, которые используются в Swing, но работающие по тому же принципу. Ниже приведен более сложный пример, который получает текст от `System.getProperties()` и добавляет его в Shell:

```

//: swt/DisplayProperties.java
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import java.io.*;

public class DisplayProperties {
    public static void main(String [] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Display Properties");
        shell.setLayout(new FillLayout());
        Text text = new Text(shell, SWT.WRAP | SWT.V_SCROLL);
        StringWriter props = new StringWriter();
        System.getProperties().list(new PrintWriter(props));
        text.setText(props.toString());
        shell.open();
        while(!shell.isDisposed())
            if(!display.readAndDispatch())
                display.sleep();
        display.dispose();
    }
} ///:~

```

В SWT все виджеты должны иметь родительский объект типа `Composite`; родитель передается в первом аргументе конструктора виджета. Соответствующий пример представлен в конструкторе `Text`, где `shell` передается в первом аргументе. Практически все конструкторы также получают набор флагов, который позволяет передать любое количество стилевых директив (в зависимости от возможностей конкретного виджета). Стилевые директивы объединяются операцией «поразрядного ИЛИ», как показано в этом примере.

При настройке объекта `Text()` я добавил флаги переноса текста и автоматического добавления вертикальной полосы прокрутки. Вы увидите, что в SWT конструкторы играют очень важную роль; многие атрибуты виджетов трудно или невозможно изменить без использования конструктора. Информацию о поддерживаемых флагах всегда можно получить из документации конструктора виджета. Обратите внимание: некоторые конструкторы должны получать аргумент с флагами, даже если «допустимые» флаги в документации не указаны. Таким образом обеспечивается возможность будущих расширений без изменения интерфейса.

Устранение избыточного кода

Прежде чем следовать дальше, обратите внимание на то, что некоторые операции должны выполняться в каждом приложении SWT (по аналогии с повторяющимися действиями в программах Swing). Для SWT вы всегда создаете объект `Display`, создаете `Shell` на основе `Display`, создаете цикл `readAndDispatch()` и т. д. Конечно, в некоторых особых случаях можно обойтись и без них, и все эти операции достаточно распространены, чтобы избавиться от избыточного кода, как это было сделано в `net.mindview.util.SwingConsole`.

Каждое приложение должно реализовать единый интерфейс:

```
//: swt/util/SWTApplication.java
package swt.util;
import org.eclipse.swt.widgets.*;

public interface SWTApplication {
    void createContents(Composite parent);
} ///:~
```

Приложение получает объект `Composite` (субклассом которого является `Shell`) и должно использовать его для создания всего содержимого внутри `createContents()`. `SWTConsole.run()` вызывает `createContents()` в соответствующей точке, задает размер `shell` в соответствии с аргументами, переданными `run()`, открывает `shell` и запускает цикл событий, и наконец, уничтожает `shell` при выходе из программы:

```
//: swt/util/SWTConsole.java
package swt.util;
import org.eclipse.swt.widgets.*;

public class SWTConsole {
    public static void
    run(SWTApplication swtApp, int width, int height) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText(swtApp.getClass().getSimpleName());
        swtApp.createContents(shell);
        shell.setSize(width, height);
        shell.open();
        while(!shell.isDisposed()) {
            if(!display.readAndDispatch())
                display.sleep();
        }
        display.dispose();
    }
} ///:~
```

Программа также заполняет заголовок именем класса `SWTApplication` и задает ширину и высоту `Shell`.

Мы можем создать версию `DisplayProperties.java`, которая выводит состояние окружения машины с использованием `SWTConsole`:

```
//: swt/DisplayEnvironment.java
import swt.util.*;
import org.eclipse.swt.*;
```

продолжение ➤

```

import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import java.util.*;

public class DisplayEnvironment implements SWTApplication {
    public void createContents(Composite parent) {
        parent.setLayout(new FillLayout());
        Text text = new Text(parent, SWT.WRAP | SWT.V_SCROLL);
        for(Map.Entry entry: System.getenv().entrySet()) {
            text.append(entry.getKey() + ": " +
                entry.getValue() + "\n");
        }
    }
    public static void main(String [] args) {
        SWTConsole.run(new DisplayEnvironment(), 800, 600);
    }
} ///:~

```

Класс `SWTConsole` позволяет разработчику сосредоточиться на творческих аспектах приложения, а не на рутинном коде.

- 40.** (4) Измените пример `DisplayProperties.java` так, чтобы в нем использовался класс `SWTConsole`.
- 41.** (4) Измените пример `DisplayEnvironment.java` так, чтобы в нем не использовался класс `SWTConsole`.

Меню

Следующий пример демонстрирует работу с простейшими меню: он читает свой исходный код, разбивает его на слова, после чего заполняет меню этими словами:

```

//: swt/Menus.java
// Развлечения с меню.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import java.util.*;
import net.mindview.util.*;

public class Menus implements SWTApplication {
    private static Shell shell;
    public void createContents(Composite parent) {
        shell = parent.getShell();
        Menu bar = new Menu(shell, SWT.BAR);
        shell.setMenuBar(bar);
        Set<String> words = new TreeSet<String>(
            new TextFile("Menus.java", "\\W+"));
        Iterator<String> it = words.iterator();
        while(it.next().matches("[0-9]+"))
            ; // Числа пропускаются.
        MenuItem[] mItem = new MenuItem[7];
        for(int i = 0; i < mItem.length; i++) {
            mItem[i] = new MenuItem(bar, SWT.CASCADE);
            mItem[i].setText(it.next());
            Menu submenu = new Menu(shell, SWT.DROP_DOWN);
            mItem[i].setMenu(submenu);
        }
    }
}

```

```

    int i = 0;
    while(it.hasNext()) {
        addItem(bar, it, mItem[i]);
        i = (i + 1) % mItem.length;
    }
}
static Listener listener = new Listener() {
    public void handleEvent(Event e) {
        System.out.println(e.toString());
    }
};
void
addItem(Menu bar, Iterator<String> it, MenuItem mItem) {
    MenuItem item = new MenuItem(mItem.getMenu(), SWT.PUSH);
    item.addListener(SWT.Selection, listener);
    item.setText(it.next());
}
public static void main(String[] args) {
    SWTConsole.run(new Menus(), 600, 200);
}
} ///:~

```

Объект `Menu` добавляется в `Shell`, а класс `Composite` позволяет получить связанный с ним объект `Shell` методом `getShell()`. Класс `TextFile` взят из пакета `net.mindview.util`, описанного ранее в книге; в данном случае класс `TreeSet` заполняется словами, чтобы они упорядочивались по алфавиту. Числа при этом пропускаются. Из потока слов выбираются имена меню верхнего уровня; затем создаются подменю, которые заполняются оставшимися словами.

При выборе одной из команд меню `Listener` просто выводит объект события, чтобы вы видели, какая информация в нем содержится. При запуске программы становится видно, что эта информация включает текст меню, поэтому вы можете реагировать на команду в зависимости от этого текста — или же предоставить разных слушателей для разных меню (более надежное решение в контексте интернационализации).

Вкладки, кнопки и события

SWT содержит обширный набор элементов управления, которые называются *виджетами* (widgets). Описания основных виджетов представлены в документации `org.eclipse.swt.widgets`, а сложные виджеты рассматриваются в документации `org.eclipse.swt.custom`.

Чтобы продемонстрировать несколько основных виджетов, этот пример размещает несколько «подпримеров» на вкладках. Заодно вы увидите, как создавать объекты `Composite` (примерные аналоги объектов `Swing JPanel`) для размещения компонентов внутри других компонентов.

```

//: swt/TabbedPane.java
// Размещение компонентов SWT на вкладках.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.browser.*;

```

продолжение ➤

```

public class TabbedPane implements SWTApplication {
    private static TabFolder folder;
    private static Shell shell;
    public void createContents(Composite parent) {
        shell = parent.getShell();
        parent.setLayout(new FillLayout());
        folder = new TabFolder(shell, SWT.BORDER);
        labelTab();
        directoryDialogTab();
        buttonTab();
        sliderTab();
        scribbleTab();
        browserTab();
    }

    public static void labelTab() {
        TabItem tab = new TabItem(folder, SWT.CLOSE);
        tab.setText("A Label"); // Text on the tab
        tab.setToolTipText("A simple label");
        Label label = new Label(folder, SWT.CENTER);
        label.setText("Label text");
        tab.setControl(label);
    }

    public static void directoryDialogTab() {
        TabItem tab = new TabItem(folder, SWT.CLOSE);
        tab.setText("Directory Dialog");
        tab.setToolTipText("Select a directory");
        final Button b = new Button(folder, SWT.PUSH);
        b.setText("Select a Directory");
        b.addListener(SWT.MouseDown, new Listener() {
            public void handleEvent(Event e) {
                DirectoryDialog dd = new DirectoryDialog(shell);
                String path = dd.open();
                if(path != null)
                    b.setText(path);
            }
        });
        tab.setControl(b);
    }

    public static void buttonTab() {
        TabItem tab = new TabItem(folder, SWT.CLOSE);
        tab.setText("Buttons");
        tab.setToolTipText("Different kinds of Buttons");
        Composite composite = new Composite(folder, SWT.NONE);
        composite.setLayout(new GridLayout(4, true));
        for(int dir : new int[]{
            SWT.UP, SWT.RIGHT, SWT.LEFT, SWT.DOWN
        }) {
            Button b = new Button(composite, SWT.ARROW | dir);
            b.addListener(SWT.MouseDown, listener);
        }
        newButton(composite, SWT.CHECK, "Check button");
        newButton(composite, SWT.PUSH, "Push button");
        newButton(composite, SWT.RADIO, "Radio button");
        newButton(composite, SWT.TOGGLE, "Toggle button");
        newButton(composite, SWT.FLAT, "Flat button");
    }
}

```

```

    tab.setControl(composite);
}

private static Listener listener = new Listener() {
    public void handleEvent(Event e) {
        MessageBox m = new MessageBox(shell, SWT.OK);
        m.setMessage(e.toString());
        m.open();
    }
};

private static void newButton(Composite composite,
    int type, String label) {
    Button b = new Button(composite, type);
    b.setText(label);
    b.addListener(SWT.MouseDown, listener);
}

public static void sliderTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Sliders and Progress bars");
    tab.setToolTipText("Tied Slider to ProgressBar");
    Composite composite = new Composite(folder, SWT.NONE);
    composite.setLayout(new GridLayout(2, true));
    final Slider slider =
        new Slider(composite, SWT.HORIZONTAL);
    final ProgressBar progress =
        new ProgressBar(composite, SWT.HORIZONTAL);
    slider.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent event) {
            progress.setSelection(slider.getSelection());
        }
    });
    tab.setControl(composite);
}

public static void scribbleTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Scribble");
    tab.setToolTipText("Simple graphics: drawing");
    final Canvas canvas = new Canvas(folder, SWT.NONE);
    ScribbleMouseListener sml= new ScribbleMouseListener();
    canvas.addMouseListener(sml);
    canvas.addMouseMoveListener(sml);
    tab.setControl(canvas);
}

private static class ScribbleMouseListener
    extends MouseAdapter implements MouseMoveListener {
    private Point p = new Point(0, 0);
    public void mouseMove(MouseEvent e) {
        if((e.stateMask & SWT.BUTTON1) == 0)
            return;
        GC gc = new GC((Canvas)e.widget);
        gc.drawLine(p.x, p.y, e.x, e.y);
    }
}

```

```

        gc.dispose();
        updatePoint(e);
    }

    public void mouseDown(MouseEvent e) { updatePoint(e); }
    private void updatePoint(MouseEvent e) {
        p.x = e.x;
        p.y = e.y;
    }
}

public static void browserTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("A Browser");
    tab.setToolTipText("A Web browser");
    Browser browser = null;
    try {
        browser = new Browser(folder, SWT.NONE);
    } catch(SWTError e) {
        Label label = new Label(folder, SWT.BORDER);
        label.setText("Could not initialize browser");
        tab.setControl(label);
    }

    if(browser != null) {
        browser.setUrl("http://www.mindview.net");
        tab.setControl(browser);
    }
}

public static void main(String[] args) {
    SWTConsole.run(new TabbedPane(), 800, 600);
}
} ///:~

```

Вызов `createContents()` создает макет, после чего вызывает методы создания всех вкладок. Текст на каждой вкладке задается методом `setText()` (вы также можете создавать на вкладках кнопки и графику); также для каждой вкладки задается текст всплывающей подсказки. В конце каждого метода находится вызов `setControl()`, который помещает элемент управления, созданный методом, в пространство этой конкретной вкладки.

Метод `labelTab()` демонстрирует использование простых надписей. Метод `directoryDialogTab()` содержит кнопку, открывающую стандартный объект `DirectoryDialog` для выбора каталога. Полученный результат используется для задания текста кнопки.

Метод `buttonTab()` демонстрирует использование основных кнопок. Метод `sliderTab()` повторяет приведенный ранее пример для Swing, связывающий регулятор с индикатором прогресса.

Метод `scribbleTab()` предоставляет любопытный пример работы с графикой. Фактически простейший графический редактор создается всего несколькими строками кода.

Наконец, метод `browserTab()` демонстрирует мощь компонента SWT Browser — полноценного браузера в одном компоненте.

Графика

Вот как выглядит программа Swing `SineWave.java`, переработанная для SWT:

```
//: swt/SineWave.java
// SWT-версия программы Swing SineWave.java.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;

class SineDraw extends Canvas {
    private static final int SCALEFACTOR = 200;
    private int cycles;
    private int points;
    private double[] sines;
    private int[] pts;
    public SineDraw(Composite parent, int style) {
        super(parent, style);
        addPaintListener(new PaintListener() {
            public void paintControl(PaintEvent e) {
                int maxWidth = getSize().x;
                double hstep = (double)maxWidth / (double)points;
                int maxHeight = getSize().y;
                pts = new int[points];
                for(int i = 0; i < points; i++)
                    pts[i] = (int)((sines[i] * maxHeight / 2 * .95)
                        + (maxHeight / 2));
                e.gc.setForeground(
                    e.display.getSystemColor(SWT.COLOR_RED));
                for(int i = 1; i < points; i++) {
                    int x1 = (int)((i - 1) * hstep);
                    int x2 = (int)(i * hstep);
                    int y1 = pts[i - 1];
                    int y2 = pts[i];
                    e.gc.drawLine(x1, y1, x2, y2);
                }
            }
        });
        setCycles(5);
    }

    public void setCycles(int newCycles) {
        cycles = newCycles;
        points = SCALEFACTOR * cycles * 2;
        sines = new double[points];
        for(int i = 0; i < points; i++) {
            double radians = (Math.PI / SCALEFACTOR) * i;
            sines[i] = Math.sin(radians);
        }
        redraw();
    }
}
```

продолжение ➤

```

    }
}

public class SineWave implements SWTApplication {
    private SineDraw sines;
    private Slider slider;
    public void createContents(Composite parent) {
        parent.setLayout(new GridLayout(1, true));
        sines = new SineDraw(parent, SWT.NONE);
        sines.setLayoutData(
            new GridData(SWT.FILL, SWT.FILL, true, true));
        sines.setFocus();
        slider = new Slider(parent, SWT.HORIZONTAL);
        slider.setValues(5, 1, 30, 1, 1, 1);
        slider.setLayoutData(
            new GridData(SWT.FILL, SWT.DEFAULT, true, false));
        slider.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                sines.setCycles(slider.getSelection());
            }
        });
    }

    public static void main(String[] args) {
        SWTConsole.run(new SineWave(), 700, 400);
    }
} ///:~

```

Вместо JPanel в SWT в качестве базовой поверхности графического вывода используется компонент Canvas.

Сравнив эту версию программы с версией для Swing, вы увидите, что реализация SineDraw почти не изменилась. В SWT мы получаем графический контекст gc от объекта события, переданного PaintListener, а в Swing объект Graphics передается методу paintComponent() напрямую. Но действия, выполняемые с графическим объектом, остаются неизменными, и метод setCycles() выглядит так же.

Метод createContents() содержит чуть больше кода, чем Swing-версия, в связи с дополнительной настройкой регулятора и его слушателя, но основные операции снова остаются неизменными.

Параллельное выполнение в SWT

Хотя технология AWT/Swing является однопоточной, эту однопоточность легко нарушить с потерей детерминированности программы. В общем случае графический вывод из нескольких потоков нежелателен, потому что вывод накладывается непредсказуемым образом.

В SWT это запрещено — при попытке графического вывода из нескольких потоков возбуждается исключение. Эта мера безопасности не позволит неопытному программисту случайно создать в программе трудноуловимую ошибку.

Ниже приведена программа `Swing ColorBoxes.java`, переработанная для SWT:

```
//: swt/ColorBoxes.java
// SWT-версия программы Swing ColorBoxes.java.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

class CBox extends Canvas implements Runnable {
    class CBoxPaintListener implements PaintListener {
        public void paintControl(PaintEvent e) {
            Color color = new Color(e.display, cColor);
            e.gc.setBackground(color);
            Point size = getSize();
            e.gc.fillRect(0, 0, size.x, size.y);
            color.dispose();
        }
    }

    private static Random rand = new Random();
    private static RGB newColor() {
        return new RGB(rand.nextInt(255),
            rand.nextInt(255), rand.nextInt(255));
    }

    private int pause;
    private RGB cColor = newColor();
    public CBox(Composite parent, int pause) {
        super(parent, SWT.NONE);
        this.pause = pause;
        addPaintListener(new CBoxPaintListener());
    }

    public void run() {
        try {
            while(!Thread.interrupted()) {
                cColor = newColor();
                getDisplay().asyncExec(new Runnable() {
                    public void run() {
                        try { redraw(); } catch(SWTException e) {}
                        // Исключение SWTException допустимо
                        // при завершении родителя.
                    }
                });
                TimeUnit.MILLISECONDS.sleep(pause);
            }
        }
    }
}
```

продолжение ↗

```

    } catch(InterruptedException e) {
        // Допустимый способ выхода
    } catch(SWTException e) {
        // Допустимый способ выхода: родитель
        // был завершен.
    }
}
}

public class ColorBoxes implements SWTApplication {
    private int grid = 12;
    private int pause = 50;
    public void createContents(Composite parent) {
        GridLayout gridLayout = new GridLayout(grid, true);
        gridLayout.horizontalSpacing = 0;
        gridLayout.verticalSpacing = 0;
        parent.setLayout(gridLayout);
        ExecutorService exec = new DaemonThreadPoolExecutor();
        for(int i = 0; i < (grid * grid); i++) {
            final CBox cb = new CBox(parent, pause);
            cb.setLayoutData(new GridData(GridData.FILL_BOTH));
            exec.execute(cb);
        }
    }

    public static void main(String[] args) {
        ColorBoxes boxes = new ColorBoxes();
        if(args.length > 0)
            boxes.grid = new Integer(args[0]);
        if(args.length > 1)
            boxes.pause = new Integer(args[1]);
        SWTConsole.run(boxes, 500, 400);
    }
} ///:~

```

Как и в предыдущем примере, для управления графическим выводом создается слушатель `PaintListener` с методом `paintControl()`, который вызывается, когда поток SWT готов к прорисовке вашего компонента. Слушатель `PaintListener` регистрируется в конструкторе `CBox`.

В этой версии `CBox` существенно отличается метод `run()`, который не может просто вызвать `redraw()` напрямую, а должен отправить запрос `redraw()` методу `asyncExec()` объекта `Display` — примерному аналогу `SwingUtilities.invokeLater()`. Если заменить его прямым вызовом `redraw()`, то программа просто перестает работать.

При запуске этой программы видны небольшие визуальные артефакты — горизонтальные линии, время от времени пробегающие по полю. Это объясняется тем, что по умолчанию SWT не использует двойную буферизацию — в отличие от Swing. Попробуйте запустить версию для Swing рядом с версией для SWT, и отличия станут более очевидными. Впрочем, вы можете написать код двойной буферизации для SWT; примеры доступны на сайте www.eclipse.org.

- 42.** (4) Измените пример `swt/ColorBoxes.java` так, чтобы он сначала разбрасывал по экрану «звездочки», а затем случайным образом изменял их цвета.

SWT или Swing?

Трудно представить полную картину по такому короткому введению, но вы, вероятно, хотя бы примерно представили, что SWT во многих ситуациях позволяет написать более прямолинейный код, чем Swing. Однако программирование графического интерфейса в SWT может быть достаточно сложным, поэтому первоочередной причиной для использования SWT, вероятно, должна быть привычность приложения для пользователя (потому что приложение выглядит/работает как остальные приложения для этой платформы), а во-вторых, если важна скорость реакции, обеспечиваемая SWT. Для остальных случаев может быть достаточно Swing.

- 43.** (6) Выберите один из примеров Swing, которые не были переработаны в этом разделе, и преобразуйте его для SWT.

Резюме

Из всех библиотек языка Java именно библиотека графического пользовательского интерфейса (GUI) претерпела наиболее значительные изменения в ходе эволюции языка. Библиотека AWT из языка Java версии 1.0 постоянно подвергалась критике как одна из самых плохих библиотек, когда-либо созданных в программном обеспечении, и хотя она позволяла создавать переносимые программы, программы эти были «одинаково посредственными на всех платформах». К тому же она была ограниченной, запутанной и попросту неэлегантной в сравнении с инструментами создания приложений для определенных платформ.

Когда с новым выпуском Java версии 1.1 появилась улучшенная модель событий и технология JavaBeans, начался новый этап — теперь стало реально создавать визуальные компоненты, которые можно было легко перетаскивать и помещать на форму в средствах визуальной разработки приложений. Вдобавок, разработка модели событий и компонентов JavaBean подтвердила тенденцию к упрощению процесса программирования и поддержки кода (то, что было забыто в AWT версии 1.0). Но только после появления библиотеки классов JFC/Swing работа была закончена. Теперь с помощью компонентов библиотеки Swing независимое от платформы программирование GUI перестало быть мечтой.

Настоящая революция произошла в области сред разработки. Если вы хотите, чтобы коммерческая среда разработки закрытого языка улучшилась, вам остается только надеяться, чтобы его производитель внес нужные изменения. Но Java — открытая среда, и это не только дает широкие возможности для конкуренции в области средств визуального создания программ, но и поощряет эту конкуренцию. Чтобы такие средства воспринимались серьезно, в них просто необходимо встраивать поддержку технологии JavaBeans. Это значит, что для всех существуют равные условия: с появлением более совершенной среды разработки никто не привязан к старой системе — можно легко ее бросить и перейти к более перспективному инструменту, чтобы повысить свою производительность. Такое здоровое соперничество в области производства визуальных средств разработки программ сулит программистам только хорошее.

Данная глава всего лишь дает начальные сведения о программировании GUI. Она показывает, как легко можно создавать большие приложения, затрачивая минимум усилий. Того, что вы увидели и о чем узнали, должно хватить для решения большей части задач, связанных с созданием пользовательского интерфейса. Однако Swing, SWT и Flash/Flex способны на большее. Вероятно, вы найдете в них средства для решения любых задач, которые только сможете представить.

Ресурсы

Презентации Бена Гэлбрайта на сайте www.galbraiths.org/presentations неплохо описывают возможности Swing и SWT.



Приложение

У книги существует ряд дополнений, включая данные, семинары и услуги, доступные на сайте компании MindView.

В этом приложении описаны дополнения к книге, а насколько они полезны именно для вас — решайте сами.

Хотя семинары обычно проводятся как открытые мероприятия, также возможно их проведение в закрытом режиме на территории заказчика.

Приложения, доступные для загрузки

Весь код, приведенный в книге, можно загрузить на сайте www.MindView.net. В частности, в него включены файлы построения Ant и другие вспомогательные файлы, необходимые для успешного построения и выполнения всех примеров в книге.

Кроме того, некоторые части книги были переведены в электронную форму. К ним относятся следующие темы:

- ☐ Клонирование объектов.
- ☐ Передача и возвращение объектов.
- ☐ Анализ и проектирование.

Это части других глав 3-го издания книги, которые мы сочли недостаточно важными для включения в печатную версию 4-го издания книги.

Thinking in C: Foundations for Java

На сайте www.MindView.net вы найдете семинар *Thinking in C*, доступный для бесплатной загрузки. Этот мультимедийный курс на базе Flash, созданный Чаком Эллисоном и разработанный компанией MindView, познакомит вас с синтаксисом, операторами

и функциями языка C, на котором основан синтаксис Java. Учтите, что для воспроизведения *Thinking in C* в системе необходимо установить проигрыватель Flash Player с сайта www.Macromedia.com.

Семинар «Разработка Объектов и Систем»

Этот семинар представляет основные концепции объектно-ориентированного программирования с точки зрения проектировщика. В нем исследуется процесс разработки и построения систем, а основное внимание будет сосредоточено на так называемых «гибких методах» или «легковесных методологиях», и особенно на экстремальном программировании (XP). Мы расскажем о методологиях в целом, о небольших инструментах, таких как техника планирования «индексные карточки», описанная в книге *Planning Extreme Programming* (авторы Бек и Фаулер, 2002), о карточках CRC для проектирования объектов, о парном программировании, итерационном планировании, модульном тестировании, об автоматизированной сборке, управлении версиями и о других подобных вопросах. Курс включает в себя XP-проект, который будет разрабатываться всю неделю.

Если вы запускаете проект и хотите освоить использование приемов объектно-ориентированного проектирования, вы сможете использовать этот проект в качестве примера и получить готовый прототип к концу недели.

За информацией о расположении, расписанием, отзывами и подробностями обращайтесь на сайт www.MindView.net.

Б

Приложение. Ресурсы

Программные средства

- ❑ **Пакет разработки JDK** с сайта <http://java.sun.com>. Даже если вы используете среду разработки программ от стороннего производителя, все равно рядом следует держать пакет JDK, на тот случай, если вы найдете возможную ошибку в компиляторе. Пакет JDK — это своего рода эталон, и если вы обнаружите ошибку в нем, скорее всего, она уже хорошо известна.
- ❑ **Документация JDK в формате HTML** с сайта <http://java.sun.com>. Мне еще не приходилось видеть справочник по стандартным библиотекам Java, данные которого не устарели и не упустили ни одной детали. Хотя HTML-документация фирмы Sun просто переполнена различными мелкими неточностями, и иногда до неприличия немногословна, по крайней мере, она *содержит* описания всех классов и методов. Поначалу может показаться, что использовать интерактивную документацию неудобно по сравнению с традиционным печатным справочником, но, по крайней мере, во время просмотра HTML-документов вы сможете составить для себя общую картину. Если это вам не поможет, обратитесь к печатным книгам.

Редакторы и среды разработки

В этой области существует здоровая конкуренция. Многие предложения бесплатны (а коммерческие обычно имеют бесплатный пробный период), так что вам лучше всего просто опробовать их и остановиться на том, которое лучше всего подходит для ваших потребностей. Несколько вариантов:

- ❑ **JEedit**, бесплатный редактор Славы Пестова, написан на Java, так что вы увидите настольное Java-приложение в действии. Редактор широко использует подключаемые модули (плагины), многие из которых были созданы активным сообществом. Загружается с сайта www.jedit.org.

- ❑ **NetBeans**, бесплатная среда разработки от Sun (www.netbeans.org), предназначенная для визуального построения графического интерфейса, редактирования кода, отладки и т. д.
- ❑ **Eclipse**, проект с открытым кодом при поддержке компании IBM (среди прочих). Платформа Eclipse также задумана как расширяемая основа, так что вы можете строить собственные независимые приложения на базе Eclipse. В частности, этот проект породил библиотеку SWT, описанную в главе 22. Загружается с сайта www.eclipse.org.
- ❑ **IntelliJ IDEA**, коммерческая среда, пользующаяся популярностью у многих программистов Java, которые считают, что IDEA постоянно опережает Eclipse, — возможно, потому что IntelliJ не стремится создать одновременно IDE и платформу разработки, а только ограничивается созданием IDE. Бесплатную пробную версию можно загрузить с сайта www.jetbrains.com.

КНИГИ

- ❑ **Core Java 2** (7-е издание), авторы Хорстманн и Корнелл, 2 тома (издательство Prentice-Hall, 2005). Гигантская, всеобъемлющая книга; именно в ней я первым делом ищу ответы на возникающие у меня вопросы. Эту книгу я могу порекомендовать вам, если вы закончили чтение *Thinking in Java* и хотели бы выйти на новый уровень.
- ❑ **The Java Class Libraries: An Annotated Reference**, авторы Патрик Чан и Розанна Ли (издательство Addison-Wesley, 1997). Как ни печально, книга немного устарела, но именно такой *должна* быть хорошая интерактивная документация: достаточно подробные объяснения для практической работы. Один из рецензентов *Thinking in Java* сказал буквально следующее: «Если бы мне разрешили пользоваться только одной книгой по Java, я бы выбрал эту (в дополнение к вашей, конечно)». Я не разделяю его восхищения относительно этой книги. Она большая, дорогая, и качество примеров меня не удовлетворяет. Но в нее стоит заглянуть, если вы ищете ответ на сложный вопрос; она глубже (и попросту объемнее) других книг. Тем не менее в Core Java 2 приведено более актуальное описание многих библиотечных компонентов.
- ❑ **Java Network Programming**, 2-е издание, автор Эллиот Расти Гарольд (издательство O'Reilly, 2000). Я не понимал сетевых механизмов Java, пока не обнаружил эту книгу. Ко всему прочему, сайт автора этой книги весьма хорош, он содержит описание текущих и предстоящих событий в области Java-разработок и не симпатизирует какому-либо гиганту индустрии. Этот сайт постоянно обновляется и никогда не отстает от времени. См. <http://cafeaulait.org>.
- ❑ **Приемы объектно-ориентированного проектирования**, авторы Гамма, Гельм, Джонсон и Влиссидес (Питер, 2013). Родоначальник концепции паттернов и моделей в программировании; упоминается во многих главах этой книги.
- ❑ **Refactoring to Patterns**, автор Джошуа Кериевски (издательство Addison-Wesley, 2005). Книга связывает рефакторинг с паттернами проектирования. Самая

полезная особенность этой книги заключается в том, что она показывает, как организовать эволюцию архитектуры посредством интеграции паттернов по мере надобности.

- ❑ **The Art of UNIX Programming**, автор Эрик Рэймонд (Addison-Wesley, 2004). Хотя Java является платформенно-независимым языком, из-за его доминирования на серверах разработчикам желательно разбираться в Unix/Linux. Книга Эрика содержит отличный вводный курс истории и философии этой операционной системы. Это весьма увлекательное чтение для тех, кто хочет разобраться в базовых концепциях компьютерных технологий.

Анализ и планирование

- ❑ **Extreme Programming Explained**, 2-е издание, авторы Кент Бек и Синтия Андрес (издательство Addison-Wesley, 2005). Я всегда думал, что должен быть совершенно другой, гораздо лучший процесс разработки программ, и, честное слово, экстремальное программирование (XP) близко к этому. Существует еще только одна книга, оказавшая на меня такое же сильное впечатление, — это *Peopleware* (описанная чуть ниже), но она в основном посвящена рабочему окружению и корпоративной культуре. *Extreme Programming Explained* говорит о программировании и переводит все его понятия, даже совсем недавние «открытия», на свой язык. Они идут настолько далеко, что иные осмеливаются утверждать, что рисовать схемы и диаграммы можно, но только если не тратить на это слишком много времени, в противном случае их надо просто выбросить. (Вы заметите, что на данной книге нет штампа «Одобрено. UML».) Я видел разработчиков, которые поступали на работу в компанию только потому, что практиковалось экстремальное программирование. Небольшая книга, маленькие главы, не требующие больших усилий для прочтения и захватывающие ваше внимание с первой страницы. Вы начинаете представлять себя в такой рабочей атмосфере, и это открывает перед вами абсолютно новый мир.
- ❑ **UML Distilled**, 2-е издание, автор Мартин Фаулер (издательство Addison-Wesley, 2000). Когда вы в первый раз сталкиваетесь с UML, кажется, что его невозможно понять — такое там множество диаграмм и мелочей. Автор утверждает, что большая часть всего этого попросту никому не нужна, поэтому он отбрасывает малозначительные детали и говорит только о самом главном. При разработке большинства проектов требуются лишь несколько инструментов построения диаграмм, и цель книги — научить читателя создавать хорошую структуру и план проекта, не беспокоясь при этом о мелких деталях. Отличная, тоненькая, «читабельная» книга; если вам понадобится понять UML, начните с нее.
- ❑ **Domain-Driven Design**, автор Эрик Эванс (издательство Addison-Wesley, 2004). Центральное место в этой книге занимает модель предметной области как основной артефакт процесса проектирования. Я всегда считал, что это важное смещение акцента, которое помогает проектировщикам держаться на правильном уровне абстракции.
- ❑ **The Unified Software Development Process**, авторы Ивар Якобсен, Гради Буч и Джеймс Рамбо (издательство Addison-Wesley, 1999). Перед тем как начать читать

эту книгу, я приготовился к самому худшему. Казалось, она целиком состоит из скучного научного текста. Я был приятно удивлен — лишь некоторые места книги содержали объяснения, которые были написаны так, будто сами авторы не совсем понимали, о чем идет речь. Большая часть книги не то чтобы совсем понятна, но, по крайней мере, ее чтение доставляет удовольствие. И что лучше всего, сведения, содержащиеся в данной книге, имеют большое практическое значение. Конечно, программирование, описываемое здесь, не сравнить с экстремальным программированием (тесты не являются тут главным оружием), но это тоже часть мира UML. Даже если вы не приемлете экстремальное программирование, у движения «UML — это хорошо» появилось очень много сторонников (даже если их *практический* опыт в этой области не так уж велик), так что вам тоже стоит приобщиться. Думаю, эту книгу можно по праву называть флагом UML и именно ее стоит прочитать после труда Фаулера *UML Distilled*, если вам понадобятся дальнейшие подробности.

Перед тем как окончательно остановиться на том или ином методе разработки, вам следует прислушаться к мнениям тех людей, что сами не рекламируют методы разработки. Начать использовать метод, не понимая, в чем, собственно, состоят его преимущества и недостатки, очень просто. Ведь его уже кто-то востребовал, а это само собой является хорошей характеристикой. Впрочем, психология людей обладает загадочной особенностью: если они убеждены, что нечто обязательно им поможет, они непременно попытаются использовать это. (Это эксперимент, и это хорошо.) Но если попытка им не поможет, они удвоят свои усилия, а потом начнут во весь голос заявлять, что обнаружили нечто замечательное. Вероятно, они считают, что всегда приятнее находиться в одной лодке с другими людьми, потому что вам не будет одиноко — даже если лодка никуда не плывет (или тонет).

Я вовсе не утверждаю, что методологии никуда не ведут, просто вы должны быть буквально вооружены до зубов различными правилами, которые помогут остаться вам в режиме эксперимента («Это не сработало; что же, давайте попробуем что-нибудь другое») и избежать отрицания («Нет, это на самом деле не проблема. Все прекрасно, нам не нужны перемены»). Я думаю, что нижеследующие книги, если вы прочитаете их до того, как выбрать метод, обеспечат вас такими правилами.

- ❑ **Software Creativity**, автор Роберт Гласс (издательство Prentice-Hall, 1995). Из тех книг, что я видел, эта лучше других обсуждает *перспективы* методологии в целом. Это сборник статей и эссе, которые были в разное время написаны как самим Глассом, так и другими авторами (П. Дж. Плогер — один из них). Они отражают многолетний опыт преподавания предмета данной книги. Эти статьи довольно занимательны и говорят только то, что нужно сказать; автор не перескакивает с одной мысли на другую и не надоедает читателю. Гласс не просто высказывает свои измышления; в книге содержатся сотни ссылок на статьи и монографии других авторов. Рекомендую прочитать эту книгу всем программистам и менеджерам, перед тем как с головой окунуться в мир методологий.
- ❑ **Software Runaways: Monumental Software Disasters**, автор Роберт Гласс (издательство Prentice-Hall, 1998). В этой книге замечательно то, что она выводит на передний план моменты, о которых мы обычно забываем: она описывает не просто тот факт, что огромное количество проектов проваливаются, но еще и то, с каким

треском они проваливаются. Я обнаружил, что мы в большинстве своем думаем: «Со мной этого не случится» (или «Больше этого не повторится»); при таком подходе мы изначально ставим себя в невыгодное положение. Если помнить, что все в любой момент может пойти не так, как намечалось, можно обеспечить для себя более выгодные начальные условия.

- ❑ **Peopleware**, 2-е издание, авторы Том ДеМарко и Тимоти Листер (издательство Dorset House, 1999). *Обязательно* прочитайте эту книгу. Она не только увлекательно читается, она перевернет ваши представления и разрушит многие предубеждения. Несмотря на то что авторы обладают солидным опытом в области разработки программного обеспечения, эта книга посвящена проектам и командам в целом. Однако она старается сконцентрироваться на *людях* и их нуждах, а не на технологии и ее требованиях. Она говорит о том, как создать среду, в которой люди будут счастливы и продуктивны, а не о том, какие правила надо устанавливать, чтобы люди становились безотказными винтиками большой машины (все равно это не поможет; программисты улыбаются и одобрительно кивают при обсуждении нового метода разработки, а потом спокойно продолжают делать то, что они делали раньше).
- ❑ **Secrets of Consulting, A Guide to Giving & Getting Advice Successfully**, автор Джеральд М. Вайнберг (издательство Dorset House, 1985). Великолепная книга, одна из самых моих любимых. Идеально подойдет, если вы собираетесь стать консультантом *или* работаете с консультантами, пытаетесь улучшить собственную работу. Короткие главы, насыщенные историями и житейскими примерами, обучающие вас самой сути дела с минимальными усилиями. Смотрите также книгу **More Secrets of Consulting**, опубликованную в 2002 году, или любую другую книгу этого же автора.
- ❑ **Complexity**, автор М. Митчелл Валдроп (издательство Simon & Schuster, 1992). Здесь рассказано о том, как однажды группа ученых из совершенно разных областей науки собралась вместе в Санта Фе, чтобы обсудить некоторые проблемы, которые конкретная область науки неспособна была решить самостоятельно (анализ рынка ценных бумаг — в экономике, зарождение жизни — в биологии, почему люди поступают именно так, а не иначе, — в социологии и т. д.). Скрещивая физику, экономику, химию, математику, информатику, социологию и другие науки, объединившиеся ученые пытались выработать новый способ решения задач. В это время, и это очень важно, возникал новый способ *мышления*: все более удаляющийся от математического детерминизма, иллюзии того, что вы можете написать уравнение, способное предсказать все возможные реакции, и больше тяготеющий к *наблюдению*, поиску системы, структуры и последующим попыткам изобразить эту структуру любыми доступными средствами. (В этой книге запечатлен, например, момент возникновения генетических алгоритмов.) Я полагаю, что такой способ мышления становится все более полезен, по мере того как мы разрабатываем все более и более сложные программные проекты.

Python

- ❑ **Learning Python**, 2-е издание, авторы Марк Лутц и Дэвид Эшер (издательство O'Reilly, 2003). Отлично подходящее для программистов введение в мой любимый язык программирования Python, прекрасно сочетающийся с Java. В книге коротко

рассказано о системе JPython, которая позволяет комбинировать два языка (Java и Python) в одной программе (компилятор JPython компилирует код в самый настоящий байт-код Java, и ничего особенного для этого делать не нужно). Альянс двух языков чрезвычайно перспективен.

СПИСОК МОИХ КНИГ

Книги перечислены в порядке их публикации. Не все упомянутые здесь книги общедоступны, но их можно найти в букинистических магазинах.

- ❑ **Computer Interfacing with Pascal & C** (напечатано по собственной инициативе и на свои средства в издательстве Eisis imprint, 1988. Найти ее можно только на сайте www.MindView.net). Введение в компьютерные средства тех незапамятных времен, когда система CP/M занимала ведущие позиции, а DOS была просто очередной выскочкой. Я использовал языки высокого уровня, зачастую вместе с параллельным портом компьютера, чтобы управлять различными электронными системами. Книга составлена на основе выдержек из моих статей в первом и лучшем журнале, для которого я когда-либо писал, *Micro Cornucopia*. Увы, Микро С забыли еще задолго до появления Интернета. Создание и публикация этой книги доставили мне массу удовольствия.
- ❑ **Using C++** (издательство Osborne/McGraw-Hill, 1989). Одна из моих первых книг, посвященных C++. Она была полностью распродана и больше не печаталась и была заменена вторым изданием, с новым именем *C++ Inside & Out*.
- ❑ **C++ Inside & Out** (издательство Osborne/McGraw-Hill, 1993). Как уже было сказано, на самом деле это второе издание *Using C++*. Язык C++ в этой книге описан достаточно точно и тщательно, но разобрана его версия примерно 1992 года, таким образом, новая книга *Thinking in C++* призвана заменить данную. Узнать больше об этой книге и загрузить для нее исходные тексты примеров можно на сайте www.MindView.net.
- ❑ **Thinking in C++**, 1-е издание (издательство Prentice-Hall, 1995). Обладатель премии *Software Development Magazine Jolt Award* как лучшая книга года.
- ❑ **Thinking in C++**, 2-е издание, том 1 (издательство Prentice-Hall, 2000). Можно загрузить с сайта www.MindView.net. Материал приведен в соответствие с завершенным стандартом языка.
- ❑ **Thinking in C++**, 2-е издание, том 2, написан совместно с Чаком Эллисоном (издательство Prentice-Hall, 2003). Можно загрузить с сайта www.MindView.net.
- ❑ **Black Belt C++, the Master's Collection**, под редакцией Брюса Эккеля (издательство M&T Books, 1994). Полностью распродана. Сборник глав, написанных различными светилами в области языка C++. Основана на презентациях, проведенных ими на конференции Software Development Conference, где я заведовал направлением C++. Увидев великолепную обложку этой книги, я решил в дальнейшем всегда самостоятельно руководить процессом создания обложек для своих книг.
- ❑ **Thinking in Java**, 1-е издание (издательство Prentice-Hall, 1998). Первое издание этой книги получило награду за продуктивность от журнала *Software Development*

Magazine, премию редакционной коллегии журнала *Java Developer's Journal*, звание лучшей книги *JavaWorld Reader's Choice Award*. Книгу можно найти на сайте www.MindView.net.

- ❑ **Thinking in Java**, 2-е издание (издательство Prentice-Hall, 2000). Это издание получило награду *JavaWorld Editor's Choice Award*. Можно загрузить с сайта www.MindView.net.
- ❑ **Thinking in Java**, 3-е издание (издательство Prentice-Hall, 2003). Это издание получило награду *Software Development Magazine* как «книга года», а также ряд других наград. Можно загрузить с сайта www.MindView.net.

Брюс Эккель
Философия Java.
4-е полное издание

Перевел с английского Е. Матвеев

Заведующий редакцией	<i>П. Щеголев</i>
Руководитель проекта	<i>П. Щеголев</i>
Ведущий редактор	<i>Ю. Сергиенко</i>
Художественный редактор	<i>Л. Адуевская</i>
Корректоры	<i>С. Беллева, В. Листова</i>
Верстка	<i>Л. Родионова</i>

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 —

Книги печатные профессиональные, технические и научные.

Подписано в печать 22.10.14. Формат 70х100/16. Усл. п. л. 94,170. Тираж 1500. Заказ 5939

Отпечатано в полном соответствии с качеством предоставленных издательством материалов
в Чеховский Печатный Двор. 142300, Чехов, Московская область, г. Чехов, ул. Полиграфистов, д. 1.

Отпечатано способом ролевой струйной печати

в ОАО «Первая Образцовая типография»

Филиал «Чеховский Печатный Двор»

142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1

Сайт: www.chpd.ru, E-mail: sales@chpd.ru, т/ф. 8(496)726-54-10

КНИГА-ПОЧТОЙ



ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **postbook@piter.com**
- по телефону: **(812) 703-73-74**
- по почте: **197198, Санкт-Петербург, а/я 127, ООО «Питер Мейл»**
- по ICQ: **413763617**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

-  Наложным платежом с оплатой при получении в ближайшем почтовом отделении.
-  С помощью банковской карты. Во время заказа Вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
-  Электронными деньгами. Мы принимаем к оплате все виды электронных денег: от традиционных Яндекс.Деньги и Web-money до USD E-Gold, MoneyMail, INOCard, RBK Money (RuPay), USD Bets, Mobile Wallet и др.
-  В любом банке, распечатав квитанцию, которая формируется автоматически после совершения Вами заказа.

Все посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку Ваших покупок максимально быстро. Дату отправления Вашей покупки и предполагаемую дату доставки Вам сообщат по e-mail.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают профессиональную и популярную литературу по различным направлениям: история и публицистика, экономика и финансы, менеджмент и маркетинг, компьютерные технологии, медицина и психология.

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электrozаводская», Семеновская наб., д. 2/1, стр. 1
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: voronej@piter.com

Екатеринбург: ул. Бебеля, д. 11а
тел./факс: (343) 378-98-41, 378-98-42; e-mail: office@ekat.piter.com

Нижний Новгород: тел.: 8 960 187-85-50; e-mail: nnovgorod@piter.com

Новосибирск: Комбинатский пер., д. 3
тел./факс: (383) 279-73-92; e-mail: sib@nsk.piter.com

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 229-68-09; e-mail: samara@piter.com

УКРАИНА

Киев: Московский пр., д. 6, корп. 1, офис 33
тел./факс: (044) 490-35-69, 490-35-68; e-mail: office@kiev.piter.com

Харьков: ул. Суздальские ряды, д. 12, офис 10
тел./факс: (057) 7584145, +38 067 545-55-64; e-mail: piter@kharkov.piter.com

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163
тел./факс: (517) 208-80-01, 208-81-25; e-mail: minsk@piter.com

≡ Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок
тел./факс: (812) 703-73-73; e-mail: spb@piter.com

≡ Издательский дом «Питер» приглашает к сотрудничеству авторов
тел./факс издательства: (812) 703-73-72, (495) 974-34-50

≡ Заказ книг для вузов и библиотек
тел./факс: (812) 703-73-73, доб. 6250; e-mail: uchebnik@piter.com

≡ Заказ книг по почте: на сайте www.piter.com; по тел.: (812) 703-73-74, доб. 6225



САЛД

САНКТ-ПЕТЕРБУРГСКАЯ
АНТИВИРУСНАЯ
ЛАБОРАТОРИЯ
ДАНИЛОВА

www.SALD.ru
8 (812) 336-3739

АНТИВИРУСНЫЕ
ПРОГРАММНЫЕ ПРОДУКТЫ

Впервые читатель может познакомиться с полной версией этого классического труда, который ранее на русском языке печатался в сокращении. Книга, выдержавшая в оригинале не одно переиздание, за глубокое и поистине философское изложение тонкостей языка Java считается одним из лучших пособий для программистов.

Чтобы по-настоящему понять язык Java, необходимо рассматривать его не просто как набор неких команд и операторов, а понять его «философию», подход к решению задач, в сравнении с таковыми в других языках программирования. На этих страницах автор рассказывает об основных проблемах написания кода: в чем их природа и какой подход использует Java в их разрешении. Поэтому обсуждаемые в каждой главе черты языка неразрывно связаны с тем, как они используются для решения определенных задач.

Блог Издательского дома «Питер» на Хабрахабре: <http://habrahabr.ru/company/piter/>



ok@piter.com

лог книг и интернет-магазин

ISBN: 978-5-496-01127-3



9 785496 011273