БИБЛИОТЕКА ПРОГРАММИСТА



Брюс Эккель

Философия Java

4-е издание

- Основы объектно-ориентированного программирования
- Синтаксис и семантика языка
- Интерфейсы и внутренние классы
- Система ввода-вывода
- Обработка ошибок и исключений
- Обнаружение проблем, анализ и планирование

PH PTR

LUXOFT
Expertise in Software Services

ПИТЕР

ПИТЕР®

Bruce Eckel

Thinking in Java

4th edition



Prentice Hall PTR
Upper Saddle River, New Jersey 07458
www.phptr.com



БИБЛИОТЕКА ПРОГРАММИСТА

Брюс Эккель

Философия Java

4-е издание

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2009

ББК 32.973 2-018.1
УДК 004.3
Э38

Эккель Б.
Э38 Философия Java. Библиотека программиста. 4-е изд. — СПб.: Питер, 2009. — 640 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-388-00003-3

Java нельзя понять, взглянув на него только как на коллекцию некоторых характеристик, – необходимо понять задачи этого языка как частные задачи программирования в целом. Эта книга — о проблемах программирования: почему они стали проблемами и какой подход использует Java в их решении. Поэтому обсуждаемые в каждой главе черты языка неразрывно связаны с тем, как они используются для решения определенных задач.

Эта книга, выдержавшая в оригинале не одно переиздание, благодаря глубокому и поистине философскому изложению тонкостей языка считается одним из лучших пособий для программирующих на Java.

ББК 32.973.2-018.1
УДК 004.3

Права на издание получены по соглашению с Prentice Hall PTR.
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 978-5-388-00003-3

© Prentice Hall PTR, 2006
© Перевод на русский язык ООО «Питер Пресс», 2009
© Издание на русском языке, оформление ООО «Питер Пресс», 2009

Уважаемый читатель!



Вы держите в руках 4-е издание популярной книги «Философия Java. Библиотека программиста».

Эта книга получила высшую оценку среди квалифицированных специалистов нашей компании. Нашему мнению можно доверять: Luxoft, европейский лидер среди фирм, занимающихся заказной разработкой программного обеспечения, является партнером многих крупнейших международных и российских компаний, таких, как Boeing, UBS, Deutsche Bank, Dell, Ренессанс Кредит и другие. Мы рекомендуем это издание в качестве настольной книги как опытным программистам, так и новичкам, только начинающим осваивать премудрости Java. На наш взгляд, «Философия Java. Библиотека программиста» — лучший учебник для программирования на Java. При этом ключевое слово тут «учебник». В большинстве известных нам книг содержится просто описание языка (список операторов, элементарные сведения о синтаксисе и т.п.). «Что еще нужно?» — может спросить опытный профессионал — и будет по-своему прав.

Но что, если книгу изучает новичок? В книге «Философия Java» материал изложен в порядке, удобном для изучения, и она содержит понятные и легко применимые на практике ответы на практически все вопросы, стоящие перед начинающими программистами на Java.

Подавляющее большинство профессиональных Java-разработчиков, знакомых с данной книгой, прочитало ее на английском языке — либо в формате pdf, либо купив за границей. Тот факт, что теперь эта книга доступна на русском, станет для многих приятным сюрпризом. Более того, мы считаем, это уникальная возможность повысить свою квалификацию для программистов, не владеющих английским языком свободно.

Мы искренне надеемся, что в результате ознакомления с данным изданием Вы приобретете новые навыки и почувствуете большую уверенность в своих способностях как IT-специалиста. А если Вы — командный игрок, человек увлеченный, талантливый, стремящийся применять свои умения и навыки в сложных проектах и реализовывать самые смелые профессиональные замыслы, то мы будем рады видеть Вас среди наших сотрудников.



С уважением,
Дмитрий Лощинин
Президент компании «Luxoft»

www.luxoft.ru
hr@luxoft.com

Содержание

Предисловие 13
Java SE5 и SE6 14
Благодарности 14

Глава 1. Введение в объекты 17
Развитие абстракции 18
Объект имеет интерфейс 20
Объект предоставляет услуги 22
Скрытая реализация 23
Повторное использование реализации 24
Наследование 25
Взаимозаменяемые объекты и полиморфизм 29
Однокорневая иерархия 33
Контейнеры 33
Параметризованные типы 35
Создание, использование объектов и время их жизни 36
Обработка исключений: борьба с ошибками 38
Параллельное выполнение 38
Java и Интернет 39
Резюме 47

Глава 2. Все является объектом 48
Для работы с объектами используются ссылки 48
Все объекты должны создаваться явно 49
Объекты никогда не приходится удалять 53
Создание новых типов данных 54
Методы, аргументы и возвращаемые значения 56
Создание программы на Java 58
Ключевое слово static 60
Наша первая программа на Java 61
Комментарии и встроенная документация 64
Стиль оформления программ 70
Резюме 70

Глава 3. Операторы 71
Простые команды печати 71
Операторы Java 72
Литералы 82
В Java отсутствует sizeof() 92
Резюме 100

Глава 4. Управляющие конструкции 101
Синтаксис foreach 105
return 107
break и continue 108
Нехорошая команда goto 109
Резюме 115

Глава 5. Инициализация и завершение 116
Конструктор гарантирует инициализацию 116
Перегрузка методов 118
Очистка: финализация и сборка мусора 130
Инициализация членов класса 137
Инициализация конструктором 140
Инициализация массивов 146
Резюме 151

Глава 6. Управление доступом 152
Пакет как библиотечный модуль 153
Спецификаторы доступа Java 159
Интерфейс и реализация 163
Доступ к классам 164
Резюме 167

Глава 7. Повторное использование классов 169
Синтаксис композиции 170
Синтаксис наследования 172
Делегирование 176
Сочетание композиции и наследования 178
Композиция в сравнении с наследованием 184
protected 185
Восходящее преобразование типов 186
Ключевое слово final 188
Инициализация и загрузка классов 195
Резюме 197

Глава 8. Полиморфизм 198
Снова о восходящем преобразовании 199
Особенности 201
Конструкторы и полиморфизм 208

Ковариантность возвращаемых типов . . . 216
Разработка с наследованием . . . 217
Резюме . . . 220

Глава 9. Интерфейсы . . . 221
Абстрактные классы и методы . . . 221
Интерфейсы . . . 224
Отделение интерфейса от реализации . . . 227
Расширение интерфейса через наследование . . . 233
Интерфейсы как средство адаптации . . . 236
Вложенные интерфейсы . . . 239
Интерфейсы и фабрики . . . 242
Резюме . . . 244

Глава 10. Внутренние классы . . . 245
Создание внутренних классов . . . 245
Связь с внешним классом . . . 246
Конструкции .this и .new . . . 248
Внутренние классы и восходящее преобразование . . . 249
Безымянные внутренние классы . . . 253
Внутренние классы: зачем? . . . 261
Наследование от внутренних классов . . . 272
Можно ли переопределить внутренний класс? . . . 272
Локальные внутренние классы . . . 274
Резюме . . . 276

Глава 11. Коллекции объектов . . . 277
Параметризованные и типизованные контейнеры . . . 277
Основные концепции . . . 280
Добавление групп элементов . . . 281
List . . . 285
Итераторы . . . 288
LinkedList . . . 291
Стек . . . 292
Множество . . . 294
Карта . . . 296
Очередь . . . 298
PriorityQueue . . . 299
Collection и Iterator . . . 301
Идиома «метод-адаптер» . . . 306
Резюме . . . 309

Глава 12. Обработка ошибок и исключения . . . 310
Основные исключения . . . 310
Перехват исключений . . . 312
Создание собственных исключений . . . 314

Спецификации исключений 319
Перехват произвольных исключений 320
Стандартные исключения Java 328
Завершение с помощью finally 330
Использование finally с return 334
Ограничения при использовании исключений 336
Конструкторы 339
Идентификация исключений 343
Альтернативные решения 344
Резюме 351

Глава 13. Информация о типах 352
Необходимость в динамическом определении типов (RTTI) 352
Регистрация фабрик 372
Рефлексия: динамическая информация о классе 376
Динамические посредники 380
Объекты с неопределенным состоянием 384
Интерфейсы и информация о типах 390
Резюме 394

Глава 14. Параметризация 397
Простая параметризация 398
Параметризованные интерфейсы 404
Параметризованные методы 407
Построение сложных моделей 419
Ограничения 437
Метасимволы 440
Резюме 452

Глава 15. Массивы 454
Особенности массивов 454
Массив как объект 456
Возврат массива 458
Многомерные массивы 460
Массивы и параметризация 463
Создание тестовых данных 465
Создание массивов с использованием генераторов 470
Вспомогательный инструментарий Arrays 474
Резюме 482

Глава 16. Система ввода/вывода Java 483
Класс File 484
Ввод и вывод 489
Добавление атрибутов и интерфейсов 491
Классы Reader и Writer 494

RandomAccessFile: сам по себе 497
Типичное использование потоков ввода/вывода 498
Средства чтения и записи файлов 505
Стандартный ввод/вывод 507
Новый ввод/вывод (nio) 510
Сжатие данных 531
Сериализация объектов 536
Предпочтения 553
Резюме 555

Глава 17. Параллельное выполнение 557
Класс Thread 559
Исполнители 561
Совместное использование ресурсов 578
Взаимодействие между потоками 598
Взаимная блокировка 602
Новые библиотечные компоненты 607
CountDownLatch 607
CyclicBarrier 609
DelayQueue 611
PriorityBlockingQueue 614
Семафоры 619
Exchanger 623
Моделирование 624
Резюме 629

Алфавитный указатель 631

Посвящается Дон

Предисловие

Поначалу я рассматривал Java как «очередной язык программирования»... которым он и является во многих отношениях.

Но время шло, я изучил Java глубже и начал понимать, что основной замысел этого языка отличается от всех других, известных мне.

Все программирование в той или иной степени связано с управлением сложными задачами: сложность решаемой проблемы накладывается на сложность системы, в которой она решается. Именно из-за этих трудностей большинство программных проектов завершаются неудачей. И до сих пор ни один из языков, которые я знаю, не был смоделирован и создан в расчете на преодоление сложности разработки и поддержки программ. Конечно, многие решения при создании языков принимались в расчете на разрешение сложностей, но при этом всегда находилось еще что-то, считаемое достаточно важным, чтобы учитывать это при планировании языка. Все это неизбежно заставляло программистов «биться головой о стену» при столкновении с проблемами. Например, C++ создавался в расчете на эффективность и совместимость с C (чтобы легко переходить с этого языка на C++). Оба решения, несомненно, полезны и стали одними из причин успеха C++, но также привели к дополнительным трудностям, что не позволило успешно воплотить в жизнь некоторые проекты (конечно, можно винить программистов и руководителей проектов, но, если язык в силах помочь в устранении ошибок, почему этим не воспользоваться?). Или другой пример: Visual Basic (VB) изначально был привязан к BASIC, в который не была заложена возможность расширения, из-за чего все дополнения, созданные для VB, имеют ужасный и сложно поддерживаемый синтаксис. С другой стороны, C++, VB, Perl и другие языки, подобные Smalltalk, частично фокусировались на преодолении трудностей и, как результат, стали успешными в решении определенных типов задач.

Больше всего удивило меня при ознакомлении с Java то, что его создатели из Sun, похоже, наряду с другими целями хотели по возможности облегчить *работу программиста*. Они словно говорили: «Мы хотим, чтобы вы могли быстрее

и проще написать надежный код». Раньше такое намерение приводило к тому, что быстродействие программ оставляло желать лучшего (хотя со временем ситуация улучшилась). И все же такой подход изумительно повлиял на сроки разработки программ; для разработки эквивалентной программы на C++ требуется вдвое или еще больше человеко-часов. Уже одно это приводит к экономии колоссальных денег и уймы времени, но Java не «застывает» в упоении достигнутым. Творцы языка идут дальше, встраивая поддержку технологий, ставших важными в последнее время (многозадачность, сетевое программирование), в сам язык или его библиотеки, что значительно упрощает решение этих задач. Наконец, Java энергично берется за действительно сложные проблемы: платформно-независимые программы, динамическое изменение кода и даже безопасность. Каждая из этих проблем способна задержать сроки сдачи вашего проекта, а может легко стать непреодолимым препятствием. Таким образом, несмотря на прошлые проблемы с производительностью, перспективы Java потрясают: он значительно повышает продуктивность нашей работы.

Во всех случаях — при создании программ, командной разработке проектов, конструировании пользовательских интерфейсов для общения программы с потребителем, запуске программ на разных типах компьютеров, простом написании программ, использующих Интернет, — Java расширяет «полосу пропускания» информации при коммуникациях *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, пытаясь обучать этому языку.

Java SE5 и SE6

В этом издании книги были учтены изменения, внесенные в Java в версии, которую фирма Sun изначально назвала JDK 1.5, затем переименовала в JDK5 или J2SE5 и наконец в Java SE5. Многие изменения Java SE5 создавались для удобства программиста. Как вы увидите, создатели Java не полностью преуспели на этом пути, но в общем сделали большие шаги в правильном направлении.

В этом издании я постарался полностью интегрировать усовершенствования Java SE5/6, включить и использовать их во всей книге. Таким образом, это издание «предназначено только для Java SE5/6», и большинство примеров не будет компилироваться в других версиях. Шаг довольно рискованный, и все же я полагаю, что преимущества того стоили.

Благодарности

Прежде всего выражаю благодарность моим друзьям и помощникам, которые работали со мной на семинарах и помогали создавать учебные проекты:

это Дэйв Бартлетт (Dave Bartlett), Билл Веннерс (Bill Venners), Чак Аллисон (Chuck Allison), Джереми Майер (Jeremy Meyer) и Джейми Кинг (Jamie King). Я благодарен им за терпение, с которым они относятся к моим попыткам построить наилучшую модель совместного существования нашего коллектива.

За последнее время (несомненно, из-за широкого распространения Интернета) ко мне обращалось множество людей, которые мне помогали, — особенно те, кто работает на дому. В прошлом мне пришлось бы нанимать огромный офис, чтобы вместить всех этих людей, но благодаря Сети, курьерской почте и телефону я мог пользоваться их содействием без дополнительных затрат. Пола Стойер (Paula Steuer) оказала неоценимую помощь: она взяла под свой контроль мой сомнительный трудовой график и привела его к нормальному виду (спасибо за то, что подталкивала меня, когда я чего-то не хотел делать, Пола). Джонатан Уилкокс (Jonathan Wilcox) просеял всю мою корпоративную структуру и отыскал все скрытые ловушки, которые могли помешать нормальному ходу дел (с юридической точки зрения). Спасибо за внимание и настойчивость. Шарлин Кобо (Sharlynn Cobaugh), эксперт по обработке звука, играла важную роль при создании мультимедийных семинаров, а также решении других проблем. Спасибо за скрупулезность, с которой она подходила к решению совершенно необъяснимых компьютерных проблем. Группа Amaio из Праги помогла мне в работе над некоторыми проектами. Дэниел Уилл-Харрис (Daniel Will-Harris) предложил изначальную идею «работы по Интернету»; и конечно, именно ему принадлежит решающее слово во всех решениях из области графического дизайна.

За прошедшие годы Джеральд Уайнберг (Gerald Weinberg) стал моим «неофициальным» учителем, и за это я благодарен ему.

Эрвин Варга (Ervin Varga) очень помог мне с технической правкой 4-го издания — хотя разные люди помогали мне с другими главами и примерами, Эрвин был главным техническим рецензентом книги. Он нашел многие ошибки и внес дополнения, значительно улучшившие мой текст. Его педантичность и внимание к деталям просто потрясают; бесспорно, это лучший технический редактор, с которым мне доводилось работать.

Мой блог на сайте Билла Веннерса www.Artima.com помог мне организовать обратную связь с читателями. Спасибо всем, кто помог мне прояснить некоторые концепции, — Джеймс Уотсон (James Watson), Говард Ловатт (Howard Lovatt), Майкл Баркер (Michael Barker) и многие другие... особенно те, кто помогал мне с проработкой темы параметризации.

Большое спасибо Марку Уэлшу (Mark Welsh) за его постоянную помощь.

Эван Кофски (Evan Cofsky), знаток всех тонкостей установки и сопровождения веб-серверов на базе Linux, помогает мне организовать нормальную работу сервера MindView.

Кафетерий Camp4 Coffee в Крестед-Бьют, штат Колорадо, стал стандартным местом проведения досуга посетителей семинаров MindView, а во время перерывов они обеспечивают отличное выездное обслуживание. Спасибо моему другу Элу Смиту (Al Smith) за то, что создал это заведение и сделал его таким отличным местом. Я также благодарен всем баристам Camp4, таким приветливым и дружелюбным.

Я благодарен всем сотрудникам Prentice Hall, которые снабжали меня всем необходимым и мирились с моими особыми запросами.

Некоторые программы оказались воистину бесценными в ходе работы над примерами, и я очень благодарен их создателям. Cygwin (www.cygwin.com) решает бесчисленные проблемы, которые Windows решить не может (не хочет), и я с каждым днем привязываюсь к этому пакету все сильнее. IBM Eclipse (www. eclipse.org) — совершенно замечательное творение для сообщества разработчиков. JetBrains IntelliJ Idea продолжает прокладывать новые творческие пути в области инструментариев разработчика.

Я начал использовать Enterprise Architect от Sparxsystems во время работы над этой книгой, и программа быстро стала моим основным UML-инструментом. Форматер кода Jalopy, созданный Марко Ханзикером (Marco Hunsicker) (www.triemax.com), часто оказывался очень полезным, а Марко помог настроить его под мои специфические потребности. Я также обнаружил, что JEdit с плагинами Славы Пестова (Slava Pestov) (www.jedit.org) оказывается весьма полезным в некоторых случаях; это вполне достойный редактор для начинающих Java-программистов.

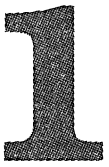
И конечно, я постоянно использую в своей повседневной работе Python (www.Python.org), творение моего друга Гидо Ван Россума (Guido Van Rossum) и группы безумных гениев, с которыми я провел множество замечательных дней. Спасибо всему сообществу Python, объединившему таких выдающихся людей!

Хочу отдельно поблагодарить всех своих учителей и учеников (которые на самом деле тоже являются учителями).

Кошка Молли часто сидела у меня на коленях, пока я работал над книгой. Так она вносила свой теплый, пушистый вклад в мою работу.

Напоследок перечислю лишь некоторых из моих друзей и помощников: Пэтти Гаст (Patty Gast) — ас в области массажа, Эндрю Бинсток (Andrew Binstock), Стив Синоски (Steve Sinoski), Джей Ди Хильдебрандт (JD Hildebrandt), Том Кеффер (Tom Keffer), Брайан Макэлхинни (Brian McElhinney), Бринкли Барр (Brinkley Barr), Билл Гейтс из «Midnight Engineering Magazine», Ларри Константин (Larry Constantine) и Люси Локвуд (Lucy Lockwood), Джин Ванг (Gene Wang), Дэйв Мейер (Dave Mayer), Дэвид Интерсимон (David Intersimone), Крис и Лора Стрэнд (Chris and Laura Strand), Элмквисты (Almquists), Брэд Джербик (Brad Jerbic), Мэрилин Цвитанич (Marylin Cvitanic), Марк Мабри (Mark Mabry), семья Роббинс (Robbins), семьи Мелтер (Moelter) и Макмиллан (McMillan), Майкл Вилк (Michael Wilk), Дэйв Стонер (Dave Stoner), Крэнстоны (Cranstons), Ларри Фогг (Larry Fogg), Майк Секейра (Mike Sequeira), Гэри Энтсмингер (Gary Entsminger), Кевин и Сонда Донован (Kevin and Sonda Donovan), Джо Лорди (Joe Lordi), Дэйв и Бренда Бартлетт (Dave and Brenda Bartlett), Блейк, Эннет и Джейд (Blacke, Annette & Jade), Рентшлеры (Rentschlers), Судеки (Sudeks), Дик (Dick), Патти (Patty) и Ли Экель (Lee Eckel), Линн и Тодд (Lynn and Todd) и их семьи. Ну и, конечно, мама с папой.

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 любит говорить Стив Джобс), и еще одно средство самовыражения. В результате инструменты программирования все меньше склоняются к машинам и все больше тяготеют к нашим умам, также 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 С. Если вам покажется, что перед прочтением этой книги вам не хватает познаний в программировании и синтаксисе С, воспользуйтесь мультимедийным семинаром *Thinking in C*, который можно загрузить с сайта www.MindView.net.

Настоящая глава содержит подготовительный и дополнительный материалы. Многие читатели предпочитают сначала представить себе общую картину, а уже потом разбираться в тонкостях ООП. Поэтому многие идеи в данной главе служат тому, чтобы дать вам цельное представление об ООП. Однако многие люди 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 — пропуск этой не будет препятствием для написания программ или изучения 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: «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 думать вас о структуре компьютера, а не о решаемой задаче. 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 ранее[1]. Таким

[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.

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», имеющих характеристики и поведение.

Алан Кей подвел итог и вывел пять основных черт 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 вы можете «обращаться с запросами» к объекту, требуя у него выполнить операции над собой. Теоретически абсолютно любой компонент решаемой задачи (собака, здание, услуга и т. п.) может быть представлен в виде объекта.
- **Программа — это группа объектов, указывающих друг другу, что делать, посредством сообщений.** Чтобы обратиться с запросом к объекту, вы «посылаете ему сообщение». Более наглядно можно представить сообщение как 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, скрыв общую сложность за простотой отдельных объектов.
- **У каждого объекта есть тип.** В других терминах, каждый объект является *экземпляром класса*, где «класс» является аналогом слова «тип». Важнейшее отличие классов друг от друга как раз и заключается в ответе на вопрос: «Какие сообщения можно посылать объекту?»
- **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 попадает под понятие фигуры. *Взаимозаменяемость* представляет одно из самых мощных понятий ООП.

Буч предложил еще более лаконичное описание объекта:
Объект обладает состоянием, поведением и индивидуальностью.

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 памяти[1].

[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.

Объект имеет интерфейс

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 наоборот[1].

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 расширяете язык программирования, добавляя новые типы данных, соответствующие вашим потребностям. Система программирования благосклонна к новым классам и уделяет им точно такое же внимание, как и встроенным типам.

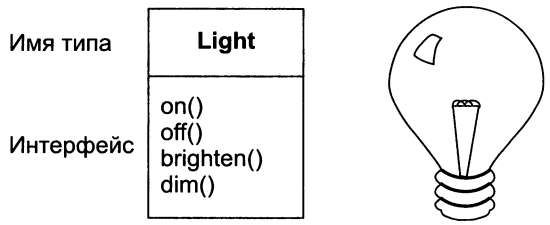
Объектно-ориентированный подход не ограничен построением моделей. Согласитесь вы или нет, что любая программа — модель разрабатываемой вами

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

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, *какие* запросы вы вправе делать 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 нужным запросом знаком точки. 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 распределить решение 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 полезно разбить на *создателей классов* (те, кто создает новые типы данных) и *программистов-клиентов* (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 прячущий 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. «Потаенная» 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 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 что они могут игнорировать.

Вторая причина появления ограничения доступа — стремление позволить разработчику библиотеки изменить внутренние механизмы класса, не беспокоясь о том, как это работает 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.

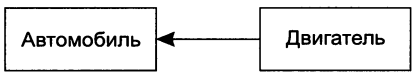
Java использует три явных ключевых слова, характеризующих уровень доступа: public, private и protected. Их предназначение и употребление очень просты. Эти *спецификаторы доступа* определяют, кто имеет право использовать следующие за ними определения. Слово public означает, что последующие определения доступны всем. Наоборот, слово private значит, что следующие за ним предложения доступны только создателю типа, внутри его методов. Термин private — «крепостная стена» между вами и программистом-клиентом. Если кто-то попытается использовать private-члены, он будет остановлен ошибкой компиляции. Спецификатор protected действует схоже с private, за одним исключением — производные классы имеют доступ к членам, помеченным 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 «имеет» (has-a), как, например, в предложении «у автомобиля есть двигатель».



(На UML-диаграммах композиция обозначается закрашенным ромбом. Я несколько упрощу этот формат: оставлю только простую линию, без ромба, чтобы обозначить связь[1].)

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

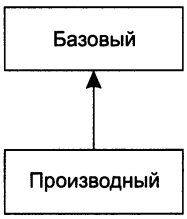
Композиция — очень гибкий инструмент. Объекты-члены вашего нового класса обычно объявляются закрытыми (private), что делает их недоступными для программистов-клиентов, использующих класс. Это позволяет вносить изменения в эти объекты-члены без модификации уже существующего клиентского кода. Вы можете также изменять эти члены во время исполнения программы, чтобы динамически управлять поведением вашей программы. Наследование, описанное 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 новичок может подумать, что наследование должно применяться повсюду. А это чревато созданием неуклюжих и 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 обидно 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 полученный клон. 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, *подклассом* или *дочерним* классом).

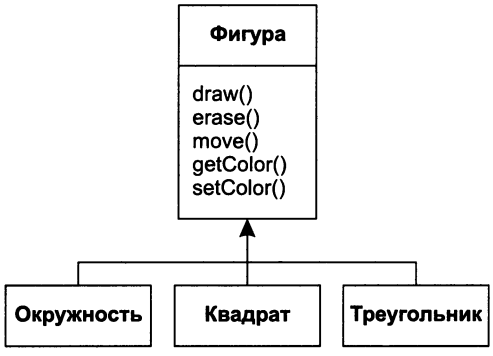


(Стрелка на UML-диаграмме направлена от производного класса к базовому классу. Как вы вскоре увидите, может быть и больше одного производного класса.)

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 способностью 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 создаете базовый тип, чтобы представить 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 черты поведения (алюминиевую банку можно смять, стальная банка притягивается магнитом). Вдобавок, некоторые черты 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, квадрат, 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.



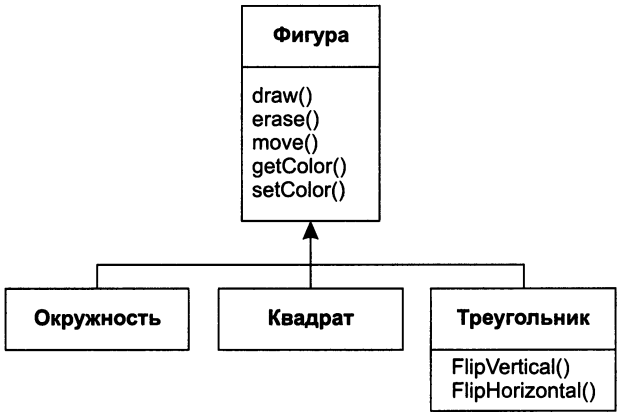
Приведение решения к понятиям, использованным в примере, чрезвычайно удобно, потому что вам не потребуется множество промежуточных моделей, связывающих описание решения с описанием задачи. При работе с объектами первичной моделью становится иерархия типов, так что вы переходите от описания системы реального 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.

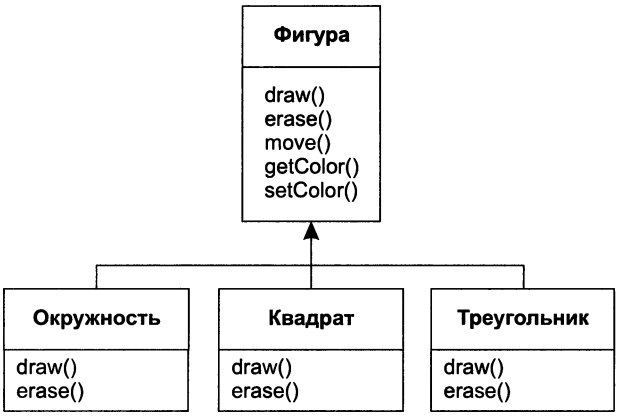
Используя наследование от существующего типа, вы создаете новый тип. Этот новый тип не только содержит все члены существующего типа (хотя члены, помеченные как private, скрыты и недоступны), но и, что еще важнее, повторяет интерфейс базового класса. Значит, все сообщения, которые вы могли посылать базовому классу, вы также вправе 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 методам. 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.



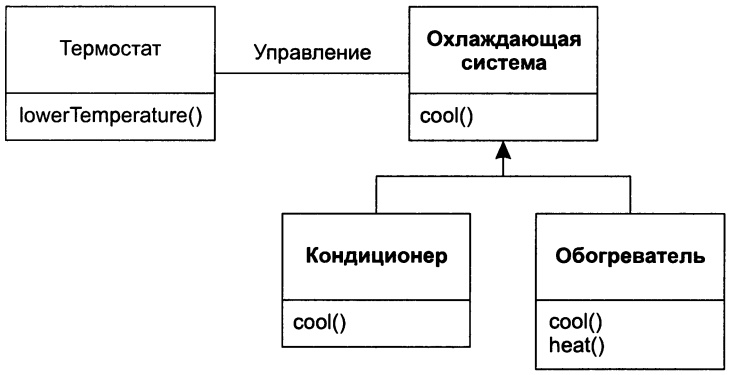
Для замещения метода нужно просто создать новое определение этого метода в производном классе. Вы как бы говорите: «Я использую тот же метод интерфейса, но хочу, чтобы он выполнял другие действия для моего нового типа».

Отношение «является» в сравнении с «похоже»

При использовании наследования встает очевидный вопрос: следует ли при наследовании переопределять *только* методы базового класса (и не добавлять новые методы, не существующие в базовом классе)? Это означало бы, 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 это способ наследования идеален. Подобный способ взаимосвязи базового и производного классов часто называют связью «*является тем-то*», поскольку можно сказать «круг *есть* фигура». 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

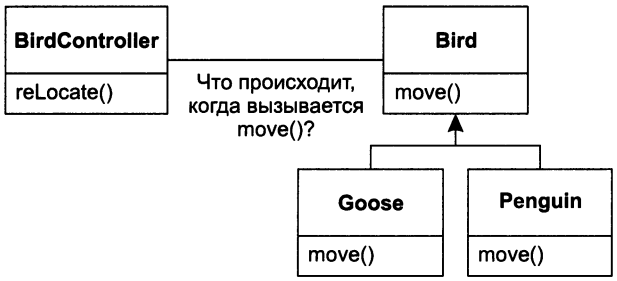
При использовании иерархий типов часто приходится обращаться с объектом определенного типа как с базовым типом. Это позволяет писать код, не зависящий от конкретных типов. Так, в примере с фигурами методы манипулируют просто фигурами, не обращая внимания на то, являются ли они окружностями, прямоугольниками, 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 поддержки 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 плывет, а объект `Penguin` (пингвин) бежит или плывет?



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 компонуются *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:

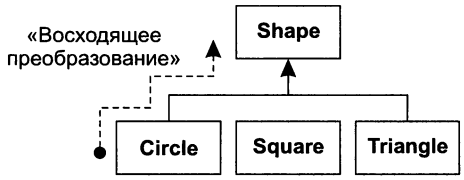
```
Circle circle = new Circle(), // окружность
Triangle triangle = new Triangle(); // треугольник
Line line = new Line(); // линия
doSomething(circle),
doSomething(triangle),
doSomething(line);
```

Вызовы метода doStuff() 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».



Объектно-ориентированная программа почти всегда содержит восходящее преобразование, потому что именно так вы избавляетесь от необходимости знать точный тип объекта, с которым работаете. 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 исполняется другой код, а не тот, что отрабатывает 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, эту роль могут исполнить 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 по необходимости расширяться, чтобы вместить все, что вы в него положите. Поэтому вам не нужно будет знать загодя, на сколько объектов рассчитана емкость контейнера. 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. Существует 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 объектов[1].

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 (от частного к общему), а по нисходящей (от общего к частному) линии. Данный способ называется *нисходящим преобразованием*. В случае восходящего 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, знающий тип хранимых объектов, и таким образом устраняющий необходимость преобразования типов и потенциальные ошибки? *параметризованные типы* представляют собой классы, которые компилятор может автоматически адаптировать для работы с определенными типами. Например, компилятор может настроить параметризованный контейнер на хранение и извлечение только фигур (Shape).

Одним из важнейших изменений Java SE5 является поддержка *параметризованных типов* (generics). Параметризованные 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.

---

[1] 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.

Создание, использование объектов и время их жизни

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, он должен быть уничтожен, 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 бродячих 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 система. 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 помещаются в стек (такие переменные называются *автоматическими*) или в область статического хранилища. Таким образом, основным фактором является скорость создания и уничтожения объектов, и это может быть неоценимо 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 — этот подход может оказаться чересчур ограниченным.

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 подход[1]. 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, как долго используется объект, и может автоматически уничтожить его. Однако при создании объекта в куче компилятор не имеет представления о сроках жизни объекта. В языках, подобных C++, уничтожение объекта должно быть явно оформлено в программе; если этого не сделать, возникает утечка памяти (обычная проблема в программах C++). В Java существует механизм, называемый *сборкой мусора*; он автоматически определяет, когда объект перестает использоваться, и уничтожает его. Сборщик мусора очень удобен, потому что он избавляет программиста от лишних хлопот. Что еще важнее, сборщик мусора дает гораздо большую уверенность в том, что в вашу программу не закралась коварная проблема утечки памяти (которая «поставила на колени» не один проект на языке C++).

В Java сборщик мусора спроектирован так, чтобы он мог самостоятельно решать проблему освобождения памяти (это не касается других аспектов завершения жизни объекта). Сборщик мусора «знает», когда объект перестает использоваться, и применяет свои знания для автоматического освобождения памяти. Благодаря этому факту (вместе с тем, что все объекты наследуются от единого базового класса `Object` и создаются только в куче) программирование на Java гораздо проще, чем программирование на C++. Разработчику приходится принимать меньше решений и преодолевать меньше препятствий.

---

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

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 возникновении 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 не похоже на числовой код ошибки, возвращаемый методом, или на флаг, устанавливаемый 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 операции. 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 раздельно выполняемых частей так, чтобы программа быстрее реагировала на внешние воздействия. Эти раздельно 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 потоки всего лишь определяют схему распределения времени на однопроцессорном компьютере. Но если операционная система поддерживает многопроцессорную обработку, каждый поток может быть назначен 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, возникают 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 он 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 Веб?

На первый взгляд Веб 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.

Веб как гигантский сервер

Фактически веб 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 веб.

Средства просмотра веб (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 отправку 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 веб-серверы, построенные на CGI-программах, тяжело обслуживать, 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 веб 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 решать широкий диапазон 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 самых 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 использовано в целях «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 брау- зерах (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 энтузиазма 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 % задач клиентского программирования. Ваша задача может как раз входить

в эти 80 %. 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 % задач клиентского программирования, кому же тогда «по зубам» остальные 20 %? 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 веб 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).

.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 рождения 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 интрасеть

Веб предоставляет 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 браузерами и программированием на стороне клиента. Когда технология веб используется для формирования информационной сети внутри компании, такая сеть называется *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 исполняется 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 объемом 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.

Несмотря на все разговоры о Java как языке Интернет-программирования, Java в действительности является полноценным языком программирования, способным решать 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 выглядит процедурная программа: 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.

Объектно-ориентированное программирование 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 вы 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.

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 языка C, он включает в себя 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 понятия 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 манипуляции данными. Иногда программисту приходится быть постоянно в курсе, какая именно манипуляция производится в программе. 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 со всем как с объектом, и поэтому повсюду используется единый последовательный синтаксис. Хотя

вы *обращаетесь* со всем как с объектом, идентификатор, которым вы манипулируете, на самом деле представляет собой *ссылку* на объект[1]. 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 уменьшении 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 означает 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 данном примере используется специальная возможность Java: инициализация строк текстом в кавычках. Обычно вы будете использовать более общий способ инициализации объектов.

Все объекты должны создаваться явно

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

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

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

---

[1] Этот вопрос очень важен. Существуют люди, утверждающие: «Ясно, 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 ссылки Java и C++ имеют несколько значительных различий. 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», предположив, что это поможет быстрее освоить новые особенности языка, в котором и без моих новых терминов много необычного. Однако есть люди, возражающие 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 (блюстители нравственности могут сказать, что я лгу вам, но я всегда могу возразить, что речь идет всего лишь о подходящей абстракции).

Конечно, кроме String, в Java имеется множество готовых типов. 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 сможете найти 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. **Постоянная память.** 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» системах.
5. **Не-оперативная память.** Если данные располагаются вне программы, они могут существовать и тогда, когда она не выполняется. Два основных примера: *потоковые объекты* (streamed objects), в которых объекты представлены в виде потока байтов, обычно используются для посылки 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). Причина для особого обращения состоит 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, бит	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-оберток», 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 устанавливается 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++ «прятать» переменные во 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("строка");
} // 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 *сборщик мусора* (garbage collector), который наблюдает за объектами, созданными оператором new, и определяет, на какие из них больше нет ссылок. Тогда он освобождает память от 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 отпадет, эти объекты исчезают сами по себе. При таком подходе исчезает целый класс проблем программирования: так называемые «утечки памяти», когда программист забывает освобождать 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, и поэтому 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 посылке 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». Например:

```
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
boolean	false
char	'\u0000' (null)
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d

Значения по умолчанию гарантируются Java только в том случае, если переменная используется *как член класса*. Тем самым обеспечивается обязательная инициализация элементарных типов (что 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 объекта[1], 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(арг1, арг2, арг3)
```

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 *посылкой сообщения объекту*. В примере выше сообщением является вызов f(), а объектом — a. Объектно-ориентированное программирование нередко характеризуется обобщающей формулой «посылка 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 ссылки[2]. Впрочем, тип ссылки должен соответствовать типу передаваемых данных. Если предполагается,

---

[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.

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:

```
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 бит, или 2 байта; это сделано для поддержки набора символов Unicode.) 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 посылать 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 «свалка» объектов со своими методами, которые принимают другие объекты в качестве аргументов и посылают 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 посылки сообщений.

Создание программы на Java

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

Видимость имен

Проблема управления именами присуща любому языку 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 C 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++ дозволяется 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 «наоборот», 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 значит, 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; вы просто записываете

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

чтобы сказать компилятору, что вы хотите использовать класс ArrayList. Впрочем, пакет util содержит множество классов, и вам могут понадобиться несколько из них. Чтобы избежать последовательного перечисления классов, используйте подстановочный символ *:

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

Как правило, импортируется целый набор классов именно таким образом, а не выписывается каждый класс 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*. Когда вы объявляете что-либо как static, это означает, что данные или метод не привязаны к определенному экземпляру этого класса. Поэтому, даже если вы никогда не создавали объектов класса, вы можете вызвать 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 расположены они в одном блоке памяти.

Существует два способа обратиться к статической переменной. Как было видно выше, вы можете указать ее с помощью объекта, например st2.i. Также можно обратиться к ней прямо по имени класса (для нестатических членов класса такая возможность отсутствует):

```
StaticTest i++.
```

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

Синтаксис с именем класса является предпочтительным, потому что он не только подчеркивает, что переменная описана как static, 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 increment() является статическим, можно вызвать его с прямым указанием класса:

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

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 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.

Наша первая программа на Java

Наконец, долгожданная программа. Она запускается, выводит на экран строку, а затем текущую дату, используя стандартный класс Date из стандартной библиотеки Java:

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

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 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 ее иным способом, обязательно это сделайте[1].) 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. Класса 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. 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 в книге. 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 ("что-то"), чтобы вывести сообщение на консоль.

[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.

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) {
```

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 число, 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, сборщик мусора заметит это, и в конце концов сам удалит его. Нам не нужно беспокоиться о его удалении самим.

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 программ[1], вам придется просмотреть их документацию, чтобы узнать, как компилировать и запускать программы.

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 установите 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.

---

[1] Часто используется компилятор IBM jikes, так как он работает намного быстрее компилятора javac от Sun. Также существуют проекты с открытыми исходными текстами, направленные на создание компиляторов, сред времени исполнения и библиотек Java.

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

```
java HelloDate
```

и вы увидите сообщение[1] 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.

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 */
```

---

[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 удается прочитать с экрана, они будут казаться иероглифами. 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/articles/ruschars/CodepagePrintStream.java, скомпилировать его и описать в переменной окружения. Данный путь лучше отложить до ознакомления с соответствующей темой (глава 12).) — *Примеч. ред.*

Второй вид комментария пришел из языка C++. Однострочный комментарий начинается с комбинации // и продолжается до конца строки. Такой стиль очень удобен и прост, 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 раздельно, 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 результае 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).

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). *Самостоятельные 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.

Существует 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 комментарий к методу тоже записывается 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 отвечал 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.

Примеры тегов

Далее описаны некоторые из тегов javadoc, используемых при документировании программы. Прежде чем применять javadoc для каких-либо серьезных целей, просмотрите 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 **информации о версии** содержит ту информацию, которую вы сочли нужным включить. Когда в командной строке **javadoc** 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 **javadoc** 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 «возбудить» (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** (расположенное на произвольном количестве

строк) 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. В Java SE5 тег @deprecated был заменен *директивой* @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 использована моя личная методика помещения специального маркера //: в комментарий как признака того, что в этой строке комментария содержится имя файла с исходным текстом. Здесь указывается путь к файлу (**object** означает эту главу) с последующим именем файла[1]. 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.

[1] Инструмент, который я создал на языке Python (подробнее на www.Python.org), распоряжается этой информацией для распределения файлов по папкам и создания файлов сборки. Вдобавок все файлы хранятся в системе CVS и автоматически вставляются в книгу с помощью макроса VBA (Visual Basic For Applications). Такой подход позволяет улучшить поддержку кода, особенно из-за использования CVS.

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.

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 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 {
     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().

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 присвоение (=) 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 создают немало затруднений). Кроме того, 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.

Присвоение

Присвоение выполняется оператором =. Трактуется 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 присвоение выполняется тривиально. Так как примитивный тип хранит данные, а не ссылку на объект, то присвоение 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 присвоении объектов все меняется. При выполнении операций с объектом вы в действительности работаете со ссылкой, поэтому присвоение «одного объекта другому» на самом деле 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
// Присвоение объектов имеет ряд хитростей
import static net.mindview.util.Print.*;

class Tank {
```

продолжение ⇩

```
    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 присвоения 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 присвоения и фактически потеряна; ее объект будет вскоре удален сборщиком 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 присвоение 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 разработки. Впрочем, это непростой вопрос, так что пока вам достаточно запомнить, что присвоение объектов может таить в себе немало сюрпризов.

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 — сложные темы. Будьте очень внимательными в таких случаях во избежание ловушек.

Арифметические операторы

Основные математические операторы остаются неизменными почти во всех языках программирования: сложение (+), вычитание (–), деление (/), умножение (*) и остаток от деления нацело (%). Деление нацело обрезает, а не округляет результат.

В Java также используется укороченная форма записи для того, чтобы одновременно произвести операцию и присвоение. 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 арифметических операций:

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

public class MathOps {
     public static void main(String[] args) {
              // Создание и раскрутка генератора случайных чисел
```

продолжение ⇩

```
        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.

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.

Унарные операторы плюс и минус

Унарные минус (-) и плюс (+) внешне не отличаются от аналогичнхы 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 в C, существует множество различных сокращений. Сокращения могут упростить написание кода, а также упростить или усложнить его чтение.

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

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

```
//: operators/AutoInc.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) {
```

продолжение ⇩

```
        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 начали: результатом будет false. Дело в том, что метод equals() по умолчанию сравнивает ссылки. Следовательно, пока вы не *переопределите* этот метод в вашем новом классе, 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.

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 C и C++. 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 примененяются 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.

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) {
```

продолжение

```
            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 удается. 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· 111111111111111
*///:~
```

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 присвоения понадобится сужающее приведение.

Восьмеричное представление (по основанию 8) обозначается начальным нулем в записи числа, состоящего из цифр от 0 до 7. Для литеральной записи чисел в двоичном представлении в Java, C и C++ поддержки нет. Впрочем, при работе с шестнадцатеричныыми и восьмеричными числами часто требуется получить двоичное представление результата. Задача легко решается методами `static toBinaryString()` классов `Integer` и `Long`.

Экспоненциальная запись

Экспоненциальные значения записываются, по-моему, очень неудачно: `1.39e-47f`. В науке и инженерном деле 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 неоднозначность[1]. Так или иначе, этот обычай был перенят в C, C++, а затем перешел в Java. Таким образом, 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 $1{,}39\times10^{-43}$.

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.

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 из входных битов был единицей, но не оба вместе. По-

[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 телетайпами, 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 70-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».

разрядный 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 и присвоение: &=, |= 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.

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, указанное после оператора. Оператор сдвига вправо (>>) сдвигает вправо операнд, находящийся слева от оператора, на количество битов, указанное 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 >>>=). Именующее выражение заменяется им же, но с проведенными над ним операциями сдвига. Однако при этом возникает проблема с оператором беззнакового правого сдвига, совмещенного с присвоением. 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;
               print(Integer.toBinaryString(i));
               i >>>= 10;
               print(Integer.toBinaryString(i));
               long l = -1;
               print(Long.toBinaryString(l));
               l >>>= 10;
               print(Long.toBinaryString(l));
               short s = -1;
               print(Integer.toBinaryString(s));
               s >>>= 10;
               print(Integer.toBinaryString(s));
               byte b = -1;
               print(Integer.toBinaryString(b));
               b >>>= 10;
               print(Integer.toBinaryString(b));
               b = -1;
               print(Integer.toBinaryString(b));
               print(Integer.toBinaryString(b>>>10));
     }
} /* Output:
11111111111111111111111111111111
11111111111111111111
11111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111
*///:~
```

В последней команде программы полученное значение не приводится обратно к b, поэтому получается верное действие.

Следующий пример демонстрирует использование всех операторов, так или иначе связанных с поразрядными операциями:

```
//: operators/BitManipulation.java
// Использование поразрядных операторов.
import java.util.*;
import static net.mindview.util.Print.*;

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

```
        int i = rand.nextInt();
        int j = rand.nextInt();
        printBinaryInt("-1", -1);
        printBinaryInt("+1", +1);
        int maxpos = 2147483647;
        printBinaryInt("макс  положит.", maxpos);
        int maxneg = -2147483648;
        printBinaryInt("макс  отрицат.", maxneg);
        printBinaryInt("i", i);
        printBinaryInt("~i", ~i);
        printBinaryInt("-i", -i);
        printBinaryInt("j", j);
        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("l", l);
        printBinaryLong("~l", ~l);
        printBinaryLong("-l", -l);
        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);
        monitor.expect("BitManipulation.out");
    }
    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:
   01111111111111111111111111111111
```

продолжение ⇩

```
макс. отрицат., int: -2147483648, двоичное·
   10000000000000000000000000000000
i, int· -1172028779, двоичное:
   10111010001001000100001010010101
~i, int. 1172028778, двоичное.
   01000101110110111011110101101010
-i, int. 1172028779, двоичное:
   01000101110110111011110101101011
j, int: 1717241110, двоичное:
   01100110010110110000010100010110
i & j, int· 570425364, двоичное:
   00010000000000000000000010100
i | j, int. -25213033, двоичное.
   11111110011111110100011110010111
i ^ j, int: -595638397, двоичное.
   11011100011111110100011110000011
i << 5, int: 1149784736, двоичное·
   01000100100010000101001010100000
i >> 5, int  -36625900, двоичное.
   11111101110100010010001000010100
(~i) >> 5, int  36625899, двоичное.
   00010111011011101111101011
i >>> 5, int. 97591828, двоичное.
   00101110001001000100000010100
(~i) >>> 5, int· 36625899, двоичное.
   00010111011011101111101011
..
*///.~
```

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 и 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

Тернарный оператор «если-иначе»

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 (описываемую 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.

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 механиз 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).

```
int x = 0, y = 1, z = 2;
String s = "x, y, z ";
System.out.println(s + x + y + z);
```

В данном случае компилятор Java приводит переменные x, y и z к их строковому представлению, вместо того чтобы сначала арифметически сложить их. А если вы запишете

```
System.out.println(x + s);
```

то и здесь Java преобразует x в строку.

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 присвоение. 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, упредив 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* (то есть 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
*///:~
```

Отсюда и ответ на наш вопрос — 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

метод round() из java.lang.Math. Так как этот метод является частью java.lang, дополнительное импортирование не потребуется.

Повышение

Вы можете обнаружить, что 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. (И тогда возможна потеря 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:
```

продолжение ⇩

```
        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);
```

продолжение ⇩

```
        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;
```

продолжение ⇩

```
        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 добавить логические переменные к другим типам или произвести 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 можно получить 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.

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).

Управляющие конструкции

4

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 языке 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`, 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)`), 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
```

и

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

Условие должно дать результат типа boolean. В секции *команда* располагается либо простая команда, завершенная точкой с запятой, либо 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 и C с 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 просмотреть результат) *///:~
```

В примере используется статический метод random() из библиотеки Math, который генерирует значение double, находящееся между 0 и 1 (включая 0, но не 1). Условие while означает: «повторять, пока condition() возвращает true». При каждом запуске программы будет выводиться различное количество чисел.

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 ложно, 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 i определяется в точке ее использования, в управляющем выражении цикла for, а не в начале блока, обозначенного фигурными скобками. Область действия для i — все выражения, принадлежащие циклу.

В программе также используется класс-«обертка» **java.lang.Character**, который не только позволяет 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.

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.

Синтаксис foreach

В Java SE5 появилась новая, более компактная форма for для перебора элементов массивов и контейнеров (см. далее). Эта упрощенная форма, называемая *синтаксисом foreach*, не требует ручного изменения служебной переменной 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)
```

продолжение ⇩

```
                    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 данной разновидностью for. 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++)
```

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:

```
//: 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
*///:~
```

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

return

Следующая группа ключевых слов обеспечивает *безусловный переход*, 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 имеется 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 <@062> 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.

break и continue

В теле любого из циклов вы можете управлять потоком программы, используя специальные ключевые слова 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 теоретически 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.

Нехорошая команда goto

Ключевое слово goto появилось одновременно с языками программирования. Действительно, безусловный переход заложил основы принятия решений в языке ассемблера: «если условие A, 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 ей 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 укромным углам.

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

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

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
               //...
               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 существования 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.

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';
                        printnb((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.

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.

Инициализация и завершение

5

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 сборщик 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.

Рассмотрим определение простого класса с конструктором:

```
//. initialization/SimpleConstructor.java
// Демонстрация простого конструктора

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.

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 + " ");
     }
}
```

продолжение ⇩

```
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 его можно было бы выбирать, то компилятору пришлось бы как-то объяснять, что же делать с этим значением.

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 этими действиями. Большинство человеческих языков несет избыточность, и даже при пропуске некоторых слов определить

смысл не так сложно. Уникальные имена не обязательны — сказанное можно понять из контекста.

Большинство языков программирования (и в особенности C) требовали использования уникальных имен для всех функций. 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: один из них не имеет аргументов (конструктор *по умолчанию*[1], также называемый конструктором *без аргументов* (no-arg)), а другой получает 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 аргументами[2]. И хотя перегрузка методов обязательна только для конструкторов, она удобна в принципе и может быть применена к любому методу.

Следующая программа показывает 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) {
```

продолжение ⇩

[1] Иногда в документации фирмы Sun можно встретить неуклюжий, но точный термин «конструктор без аргументов». Однако термин «конструктор по умолчанию» употребляется уже много лет, и я придерживаюсь его.

[2] 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.*

```
                print(s + ": Дерево высотой " + height + " м.");
        }
}

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 привнести 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)"); }
```

продолжение ⇩

```
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) { printnb("f4(long)"); }
void f4(float x) { printnb("f4(float)"); }
void f4(double x) { printnb("f4(double)"); }

void f5(long x) { printnb("f5(long)"); }
void f5(float x) { printnb("f5(float)"); }
void f5(double x) { printnb("f5(double)"); }

void f6(float x) { printnb("f6(float)"); }
void f6(double x) { printnb("f6(double)"); }

void f7(double x) { printnb("f7(double)"); }

void testConstVal() {
        printnb("5: ");
        f1(5);f2(5);f3(5);f4(5);f5(5),f6(5);f7(5);print();
}
void testChar() {
        char x = 'x';
        printnb("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;
        printnb("short: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);print();
}
void testInt() {
        int x = 0;
        printnb("int: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);print();
}
void testLong() {
        long x = 0;
        printnb("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;
        printnb("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)
*///:~
```

Если вы рассмотрите результат работы программы, то увидите, 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 соответственным 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:

```
//: c04:Demotion.java
// Понижение примитивов и перегрузка.
import com.bruceeckel.simpletest.*;

public class Demotion {
    static Test monitor = new Test();

    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)"); }
```

продолжение ⇩

```
    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;
            System.out.println("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();
            monitor.expect(new String[] {
                                                "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

Вполне 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 отличить друг от друга:

```
void f() {}
int f() {}
```

Такой подход прекрасно сработает в ситуации, в которой компилятор может однозначно выбрать нужную версию метода, 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».

Ключевое слово this

Если у вас есть два объекта одинакового типа с именами a и b, вы, возможно, заинтересуетесь, 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 представитьткак:

```
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 this вам 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 необходимости[1]. 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
// Simple use of the "this" keyword.

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");
     }
}

class Peeler {
     static Apple peel(Apple apple) {
```

продолжение ⇨

[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.

```
                //  .. 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.

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
// Calling constructors with "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);
                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.

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 статических[1] (хотя обратное позволено), и статические методы можно вызывать для имени класса, без каких-либо объектов. Статические методы отчасти напоминают глобальные функции языка C, 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,

[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.

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 сборка 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 сборщик мусора, освобождающий память от ненужных объектов. Но представим себе необычную ситуацию. Предположим, что объект выделяет «специальную» память без использования оператора new. Сборщик 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 сборщик 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 выполнять завершающие действия *во время работы сборщика 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 разрушаются всегда* (в правильно написанной программе), в то время как в Java объекты удаляются сборщиком мусора не во всех случаях. Другими словами:

ВНИМАНИЕ

1. Ваши объекты могут быть и не переданы сборщику мусора.
2. Сборка мусора не является удалением.

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 сборщиком 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 сборщик 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 сборка мусора сопровождается весомыми издержками, и если сборщик 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?

Запомните третье правило:

ВНИМАНИЕ

3. Процесс сборки мусора относится только к памяти.

Единственная причина существования сборщика мусора — освобождение памяти, которая перестала использоваться вашей программой. Поэтому все действия, так или иначе связанные со сбором 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 — сборщик 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 нечасто[1]. И правда, это не то место, где следует проводить рутинные операции очистки. Но где же тогда эти обычные операции будут уместны?

[1] Джошуа Блош в своей книге (в разделе «Избегайте финализаторов») высказывается еще решительнее: «Финализаторы непредсказуемы, зачастую опасны и чаще всего не нужны». *Effective Java*, стр. 20 (издательство Addison-Wesley, 2001).

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 происходит 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 сборщик мусора и без того выполнит освобождение памяти. Значит, в несколько упрощенном изложении можно утверждать, что деструктор в Java отсутствует из-за присутствия сборщика мусора. Но в процессе чтения книги вы еще не раз убедитесь, что наличие сборщика 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 потребуется 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 сборка 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 сборки 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 применение метода finalize(), не зависящее от того, вызывается ли 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

[1] Этот термин предложил Билл Веннерс (www.artima.com) во время семинара, который мы проводили с ним вместе.

закрепленной за ним памяти безопасно. Например, если объект представляет открытый файл, то он должен быть соответствующим образом закрыт, перед тем как его «приберет» сборщик 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 сборка 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 сборщика мусора, но в методе main() программист ошибся и не отметил один из объектов Book. Если бы в методе finalize() 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 сборщик 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.

Как работает сборщик 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 сборщика 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 похоже 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 заметно снижает производительность), а затем и вовсе исчерпается. Как раз здесь в действие вступает сборщик мусора; во время своей работы он компактно размещает объекты кучи, как бы смещая «указатель кучи» ближе к началу «ленты», тем самым предотвращая фрагментацию памяти. Сборщик мусора реструктуризует внутреннее расположение объектов в памяти и позволит получить высокоскоростную модель кучи для резервирования памяти.

Чтобы понять, как работает сборка мусора в Java, необходимо узнать, как устроены реализации сборщиков мусора (СМ) в других системах. Простой, но медленный механизм СМ 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. Сборщик 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 сборщика мусора достаточно времени. Подсчет ссылок часто используется для объяснения принципов процесса сборки мусора, но, судя по всему, он не используется ни в одной из виртуальных машин Java.

В более быстрых схемах сборка 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 сборщика мусора автоматически.

В описанном здесь подходе работает *адаптивный* механизм сбора 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 сборку 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 сборщики» 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 сборщик 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 сборщик решит «упаковать» фрагментированную кучу, то делается это перемещением объектов внутри нее.

Идея «остановиться-и-копировать» несовместима с фоновым процессом сборки мусора; в начале уборки программа останавливается. В литературе фирмы Sun можно найти немало заявлений о том, что сборка 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 сборщик 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 СМ 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 сборки 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 сборки 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, СМ 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. Заставить программиста присвоить переменной значение по умолчанию — значит предотвратить 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;
```

продолжение ⇩

```
        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 значение ноль, которое отображается в виде пробела.) По крайней мере, нет опасности случайного использования неинициализированной переменной.

Если ссылка на объект, определямая 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 всего 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 получат 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 минует двойную инициализацию, перед вызовом конструктора и во время него. (Первый объект теряется, и со временем его уничтожит сборщик мусора.) Поначалу это может показаться неэффективным, но такой подход гарантирует верную инициализацию — что произошло бы, 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 + ")"),
```

продолжение ⇩

```
        }
        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()
Bowl(3)
```

```
Cupboard()
f1(2)
f2(1)
f3(1)
*///:~
```

Класс Bowl позволяет проследить за процессом создания классов; классы Table и Cupboard содержат определения статических объектов Bowl. Заметьте, что в классе Cupboard создается нестатическая переменная Bowl bowl3, хотя все остальные определения — статические.

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.bowl1 или Table.bowl2, то, соответственно, не будет и объектов static Bowl bowl1 и static Bowl bowl2. Они инициализируются только при создании *первого* объекта Table (или при первом обращении к статическим данным). После этого статические объекты повторно не переопределяются.

Сначала инициализируются static-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 Bowl, загружается класс Bowl. 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 позже) 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 объект класса никогда 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)
*///:~
```

Статический инициализатор класса Cups выполняется 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, будут ли исполнены одна или обе строки (2) программы — static-инициализация все равно выполняется только один раз.

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)
```

продолжение

```
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.

Инициализация массивов

Массив представляет собой последовательность объектов или примитивов, относящихся к одному типу, обозначаемую одним идентификатором. Массивы определяются и используются с помощью *оператора индексирования* [ ]. Чтобы объявить массив, 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/ArraysOfPrimitives.java
// Массивы простейших типов.
import static net.mindview.util.Print.*;

public class ArraysOfPrimitives {
    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, 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++)
                         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.

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 неизвестное заранее количество аргументов неизвестного типа. Так как абсолютно все классы унаследованы 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" });
               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
*///:~
```

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 сборщиком мусора, и аналоги деструкторов обычно не нужны. В таких случаях сборщик мусора Java значительно упрощает процесс программирования и к тому же добавляет так необходимую безопасность при освобождении ресурсов. Некоторые сборщики мусора могут проводить завершающие действия даже с такими ресурсами, как графические и файловые дескрипторы. Однако сборщики 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.

Управление доступом

6

Важнейшим фактором объектно-ориентированной разработки является отделение переменных составляющих от постоянных.

Это особенно важно для библиотек. Пользователь (*программист-клиент*) библиотеки зависит от неизменности некоторого аспекта вашего кода. 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, 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 остутствию 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. 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;

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 важны управление пространствами имен в Java и возможность создания уникальных идентификаторов для всех классов.

До этого 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 в книге, 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.

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 интерпретацию[1] этих файлов.

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 mypackage и, если кто-то захочет использовать его, ему придется полностью записать или 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;
```

[1] Использовать Java-интерпретатор не обязательно. Существует несколько компиляторов, создающих единый исполняемый файл.

```
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 подключится к Интернету и начнет 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 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 слеши 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 все сказанное, 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
```

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, найдет каталог net/mindview/simple, а затем переходит к поиску компилированных 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.

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 пришлось бы 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).

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
```

Теперь, когда бы вы ни придумали новый интересный инструмент, вы всегда можете добавить его в свою библиотеку.

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.

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. Не существует волшебного способа «ворваться» внутрь него. 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.

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(Strıng[] args) {
              Cookie x = new Cookie();
              //! x.bite(); // Обращение невозможно
     }
} /* Output:
Конструктор Cookie
*/// ~
```

то можно 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* — здесь 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; напомню, 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. далее).

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 полезен: 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 верно 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 доступ к конкретному методу производным классам, но закрыть его от всех остальных. Именно для этой задачи используется ключевое слово protected. Спецификатор protected также предоставляет доступ в пределах пакета — то есть члены с этим спецификатором доступны для других классов из того же пакета.

Интерфейс и реализация

Контроль над доступом часто называют *сокрытием реализации*. Помещение данных и методов в классы в комбинации с сокрытием реализации часто называют

инкапсуляцией. В результате появляется тип данных, обладающий характеристиками и поведением.

Доступ к типам данных ограничивается по двум причинам. Первая причина — чтобы программисту-клиенту знать, что он может использовать, а что не может. Вы вольны 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 можно 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.

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 у вас есть лишь такой выбор при задании доступа к классу: в пределах пакета или открытый (public). Если вы хотите перекрыть доступ к классу для всех, объявите все его конструкторы со спецификатором private, соответственно, запретив 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, т.к  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();
}
```

[1] На самом деле доступ private или protected могут иметь *внутренние классы*, но это особый случай (см. главу 8).

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, объявив 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, пэтому он всегда существует только в одном экземпляре и его невозможно получить без вызова открытого метода access().

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.

Существует 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.

Повторное использование классов

7

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 не 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 статическим методам print() и printnb(), используемым 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 объекты 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·
```

продолжение ⇘

```
        { 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.

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 {
     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 встраивать 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 так-

же 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, что 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()).

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.

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.*;
```

продолжение ⇩

```
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.)

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 приказать 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");
```

продолжение ⇨

```
            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.

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);
        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.

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 сборщика 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 сборщик 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 10, 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 10: 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 являются проблемой; достаточно дать сборщику мусора выполнить свою работу. Но уж если понадобилось провести их явно, сделайте это со всей возможной тщательностью и вниманием, так как в процессе сборки мусора трудно в чем-либо быть уверенным. Сборщик мусора вообще может не вызываться, а если он начнет работать, то объекты будут уничтожаться в произвольном порядке. Лучше не полагаться на сборщик 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().

Сокрытие 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 запись **@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)");
     }
} ///:~
```

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 применяется, когда в новом классе необходимо использовать 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 чуть-чуть подумать, то вы 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.

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 собою 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;
     }
}
```

продолжение ⇩

```
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.

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 имеет *тип* 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) {
             // ...
```

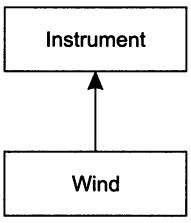
```
            i.play();
        }
}

// 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 с готовностью берет один тип вместо другого. Но стоит вспомнить, что объект Wind также является объектом 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 к 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 это возможно. Наоборот, пользоваться им следует осмотрительно — только там, где полезность наследования не вызывает сомнений. Один из хороших критериев выбора между композицией и наследованием — спросить себя, собираеесь ли вы впоследствии проводить восходящее преобразование от производного класса к базовому классу. Если восходящее преобразование актуально, выбирайте наследование, а если нет — подумайте, нельзя ли поступить иначе.

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 факически были константными.) Данное ограничение относится и к массивам, которые тоже являются объектами.

Следующий пример демонстрирует использование final для полей классов:

```
// 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); // OK - 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
```

продолжение ⇩

```
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.

Пустые константы

В Java разрешается создавать *пустые константы* — поля, объявленные как final, которым, однако, 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.

Неизменные аргументы

Java позволяет вам объявлять неизменными аргументы метода, объявляя их с ключевым словом 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);
```

продолжение ⇩

```
            bf with(null);
        }
} ///.~
```

Методы f() и g() показывают, что происходит при передаче методу примитивов с пометкой final: их значение можно прочитать, но изменить его не удастся.

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 превращать 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.

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 выглядет 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()");
     }
     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 privat-метод недоступен и фактически невидим для окружающего мира, он не влияет ни на что, кроме внутренней организации кода в классе, где он был описан.

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, что 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.

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. 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.)

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 методам.

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.

Резюме

Как наследование, так и композиция позволяют создавать новые типы на основе уже существующих. 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, разбив 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.

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 неотъемлемой чертой объектно-ориентированного языка, 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. Сокрытие 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 дает возможность 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) {
```

продолжение ⇘

```
            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 это 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, 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. Однако большинство программистов с опытом работы на процедурных языках при работе с полиморфизмом испытывают некоторые затруднения.

Особенности

Сложности с программой Music.java обнаруживаются после ее запуска. Она выводит строку Wind.play(). Именно это и требуется, но не понятно, 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 тип 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.

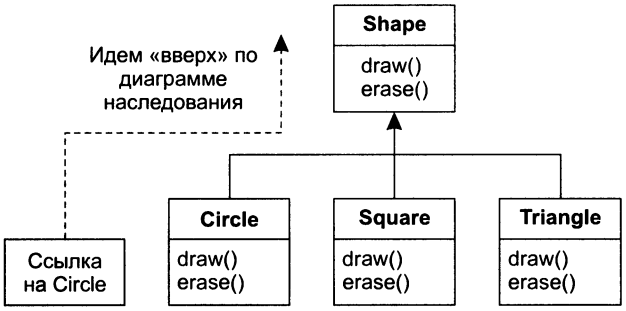
Зачем объявлять метод как final? Как уже было замечено в предыдущей главе, это запрещает переопределение соответствующего метода. Что еще важнее, это фактически «отключает» позднее связывание или, скорее, указывает компилятору на то, что позднее связывание не является необходимым. Поэтому для методов 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:

```
//: polymorphism/shape/Shapes.java
package polymorphism.shape;

public class Shape {
     public void draw() {}
     public void erase() {}
} ///:~

//: polymorphism/shape/Circle.java
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()"); }
```

продолжение ⇩

```
     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.

Расширяемость

Теперь вернемся к программе Music.java. Благодаря полиморфизму вы можете добавить в нее сколько угодно новых типов, не изменяя метод tune(). В хорошо спланированной 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.

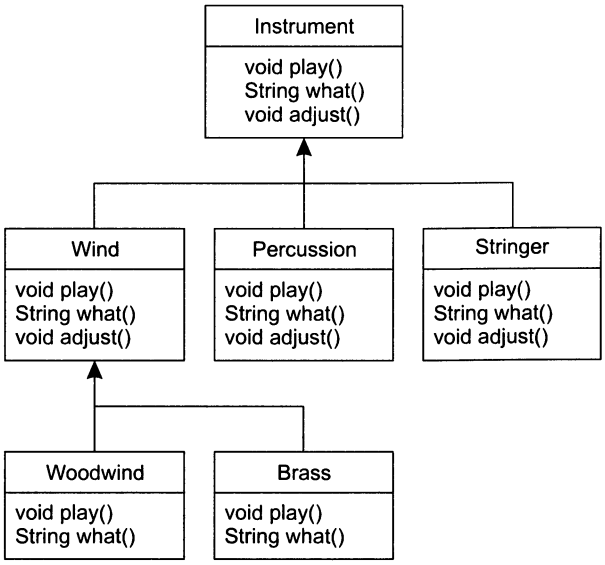
Давайте возьмем пример с объектами Instrument и включим дополнительные методы в базовый класс, а также определим несколько новых классов. Рассмотрим диаграмму (см. рисунок на обороте).

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 {
```

продолжение




```
      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"), }
}

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».

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();
     }
```

продолжение ⇩

```
}

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.

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 объ-

ектами. Результат виден при создании объекта Sandwich в методе main(). Это значит, что конструкторы для сложного объекта вызываются в следующей 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.

Наследование и завершающие действия

Если при создании нового класса используется композиция и наследование, обычно вам не приходится беспокоиться о проведении завершающих действий — подобъекты уничтожаются сборщиком 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() {
```

продолжение ⇩

```
                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 финализованы в порядке, противоположном очередности их создания.

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) {
```

продолжение ⇩

```
            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.

Поведение полиморфных методов при вызове из конструкторов

В иерархиях конструкторов возникает интересный вопрос. Что происходит, 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 только 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. Однако 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 программ 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».

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.

Разработка с наследованием

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, 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/Transmogrify.java
// Динамическое изменение поведения объекта
// с помощью композиции (шаблон проектирования «Состояние»)
import static net.mindview.util.Print.*;

class Actor {
     public void act() {}
}

class HappyActor extends Actor {
     public void act() { print("HappyActor"), }
}
```

продолжение ⇩

```
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 Transmogrify {
     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.

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 «больший» 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, послав 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(); // Нисх. 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.

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.

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 прибегнув 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.

Интерфейсы 9

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». Попытка вызова такого метода означала, что в программе произошла какая-то ошибка. Это объяснялось 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:

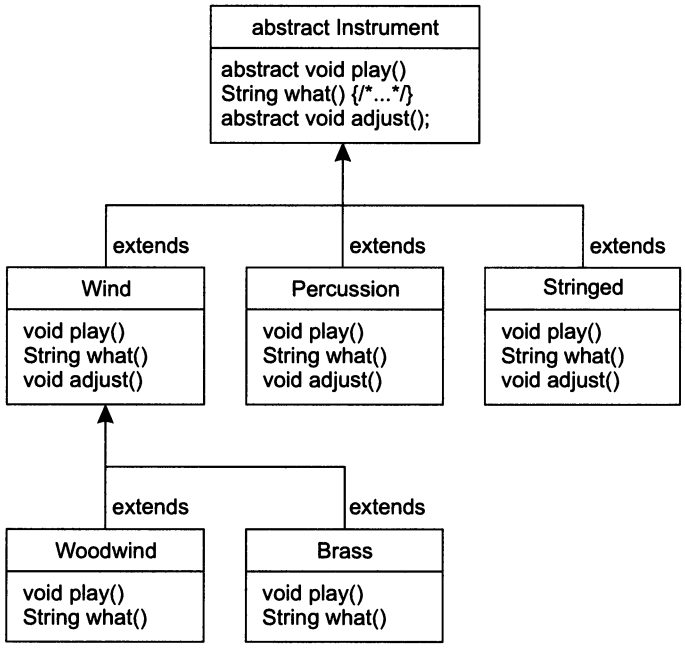
```
abstract void f();
```

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 этого не сделать, 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:



[1] Аналог *чисто виртуальных методов* языка C++.

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,
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:
```

продолжение ⇩

```
        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, объем изменений минимален.

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.

Интерфейсы

Ключевое слово interface становится следующим шагом на пути к абстракции. Оно 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.

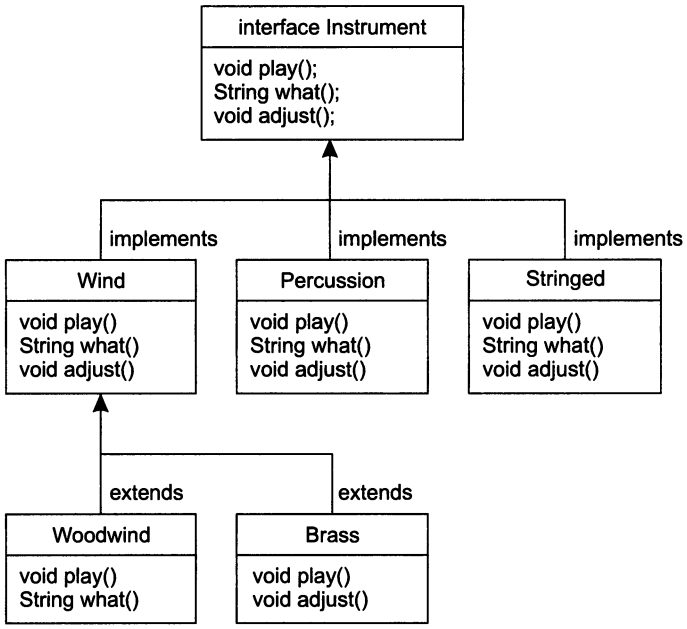
Ключевое слово interface фактически означает: «Именно так должны выглядеть все классы, которые *реализуют* данный интерфейс». Таким образом, любой код, использующий конкретный интерфейс, знает только то, какие методы вызываются для этого интерфейса, но не более того. Интерфейс определяет своего рода «протокол взаимодействия» между классами.

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 специфи-

катор 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 сейчас 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. Заметьте, 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 класс Instrument «обычным» или абстрактным, или это вообще не класс, а интерфейс.

Отделение интерфейса от реализации

В любой ситуации, когда метод работает с классом вместо интерфейса, вы ограничены использованием этого класса или его субклассов. 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; }
```

продолжение ⇩

```
}

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. 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 «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);
    }
}

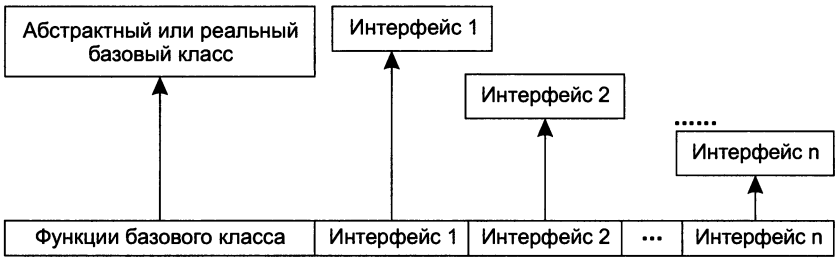
public class FilterProcessor {
    public static void main(String[] args) {
            Waveform w = new Waveform();
            Apply.process(new FilterAdapter(new LowPass(1.0)), w);
            Apply.process(new FilterAdapter(new HighPass(2.0)), w);
            Apply.process(
                    new 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.

«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 «реальным» (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 реального 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 реальный класс ActionCharacter с интерфейсами CanFight, CanSwim и CanFly. При объединении реального класса с интерфейсами на первом месте должен стоять реальный 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 реальный 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 можно 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.

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(),
}

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.

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 архитектурном паттерне *«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** 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
Waqizeyoy
*/// ~
```

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 схемой адаптера, 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()) + " ",
```

продолжение ⇨

```
            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 заключается преимущество интерфейсов перед классами.

Поля в интерфейсах

Так как все поля, помещаемые в интерфейсе, 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++:

```
//: 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.

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 интерфейсы[1]. 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 {
```

продолжение ⇩

[1] Благодарю Мартина Даннера за то, что он задал этот вопрос на семинаре.

```
                public void f() {}
        }
        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() {}
        }
        // Private-интерфейс не может быть реализован нигде,
        //  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();
            // Член интерфейса недоступен
            //! 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. Вложенные интерфейсы, как и обычные, могут иметь «пакетную» 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). Для чего нужен закрытый вложенный интерфейс? Может показаться, 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. Факт реализации private-интерфейса не может упоминаться в программе, поэтому реализация такого интерфейса — просто способ принудительного определения методов этого интерфейса без добавления информации о дополнительном типе (то есть восходящее преобразование становится невозможным).

Метод getD() усугубляет сложности, связанные с private-интерфейсом, — это открытый (public) метод, возвращающий ссылку на закрытый (private) интерфейс. Что можно сделать с возвращаемым значением этого метода? В методе main() мы видим несколько попыток использовать это возвращаемое значение, и все они оказались неудачными. Заставить метод работать можно только одним способом — передать возвращаемое значение некоторому объекту, которому разрешено его использование (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 внутрь другого интерфейса, автоматически объявляется открытым и его нельзя сделать закрытым (private).

Пример NestingInterfaces демонстрирует разнообразные способы реализации вложенных интерфейсов. Особо стоит отметить тот факт, что при реализации

интерфейса вы не обязаны реализовывать вложенные в него интерфейсы. Также закрытые (private) интерфейсы нельзя реализовать за пределами классов, в которых они описываются.

Интерфейсы и фабрики

Предполагается, что интерфейс предоставляет своего рода «шлюз» к нескольким альтернативным реализациям. Типичным способом получения объектов, соответствующих интерфейсу, является паттерн *«фабрика»*. Вместо того, чтобы вызывать конструктор напрямую, вы вызываете метод объекта-фабрики, который предоставляет реализацию интерфейса — в этом случае программный код теоретически отделяется от реализации интерфейса, благодаря чему становится возможной совершенно прозрачная замена реализации. Следующий пример демонстрирует типичную структуру фабрики:

```
//: 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 применения фабрики вам пришлось бы где-то указать точный тип создаваемого объекта Service, чтобы он мог вызвать подходящий конструктор.

Но зачем вводить лишний уровень абстракции? Данный паттерн часто применяется при создании библиотек. Допустим, вы создаете систему для игр, которая позволяла бы играть в шашки и шахматы на одной доске:

```
//: interfaces/Games.java
// Игровая библиотека с использованием фабрики
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();
```

продолжение ⇩

```
            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, такое решение позволит повторно использовать его для разных типов игр.

В следующей главе будет представлен более элегантный способ реализации фабрик на базе анонимных внутренних классов.

Резюме

После первого знакомства интерфейсы выглядят так хорошо, что может показаться, будто им всегда следует отдавать предпочтение перед реальными классами. Конечно, в любой ситуации, когда вы создаете класс, вместо него можно создать интерфейс и фабрику.

Многие программисты поддались этому искушению. Тем не менее любая абстракция должна быть мотивирована реальной потребностью. Основное назначение интерфейсов — возможность переработки реализации в случае необходимости, а не введение лишнего уровня абстракции вместе с дополнительными сложностями. Дополнительные сложности могут оказаться довольно существенными. А представьте, что кто-то будет вынужден разбираться в вашем коде и в конечном итоге поймет, что интерфейсы были добавлены «на всякий случай», без веских причин — в таких ситуациях все проектирование, которое выполнялось данным разработчиком, начинает выглядеть довольно сомнительно.

В общем случае рекомендуется *отдавать предпочтение классам перед интерфейсами*. Начните с классов, а если необходимость интерфейсов станет очевидной — переработайте архитектуру своего проекта. Интерфейсы — замечательный инструмент, но ими нередко злоупотребляют.

Внутренние классы 10

Определение класса может размещаться внутри определения другого класса. Такие классы называются *внутренними* (inner class).

Внутренние классы весьма полезны, так как они позволяют группировать классы, логически принадлежащие друг другу, и управлять доступом к ним. Однако следует понимать, что внутренние классы заметно отличаются от композиции.

На первый взгляд создается впечатление, что внутренние классы представляют собой простой механизм сокрытия кода. Однако вскоре вы узнаете, что возможности внутренних классов гораздо шире (они знают о существовании внешних классов и могут работать с ними), а программный код с внутренними классами часто бывает более элегантным и понятным (хотя конечно, этого никто не гарантирует).

В этой главе подробно исследуется синтаксис внутренних классов. Эти возможности представлены для полноты материала, хотя, скорее всего, на первых порах они вам не понадобятся. Возможно, начальные разделы этой главы содержат все, что вам действительно необходимо знать на этой стадии, а к более подробным объяснениям можно относиться как к справочному, дополнительному материалу.

Создание внутренних классов

Внутренние классы создаются в точности так, как и следовало ожидать, — определение класса размещается внутри окружающего класса:

```
//: innerclasses/Parcel1.java
// Создание внутренних классов.

public class Parcel1 {
```

продолжение ⇩

```
        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 имеет много общего
        // с использованием 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("Тасмания");
        }
} /* Output:
Тасмания
*///:~
```

Если вам понадобится создать объект внутреннего класса где-либо, кроме как в не-статическом методе внешнего класса, тип этого объекта должен задаваться в формате *ИмяВнешнегоКласса.ИмяВнутреннегоКласса*, что и делается в методе main().

Связь с внешним классом

Пока что внутренние классы выглядят как некоторая схема для сокрытия имен и организации кода — полезная, но не особенно впечатляющая. Однако есть еще один нюанс. Объект внутреннего класса связан с *внешним объектом-создателем* и может обращаться к его членам без каких-либо дополнительных описаний. Вдобавок для внутренних классов доступны все без исключения элементы внешнего класса[1]. Следующий пример иллюстрирует сказанное:

```
//: innerclasses/Sequence.java
// Хранение последовательности объектов

interface Selector {
     boolean end();
     Object current();
     void next();
}

public class Sequence {
```

[1] Эта концепция внутренних классов сильно отличается от концепции *вложенных классов* C++, которые представляют собой простой механизм для сокрытия имен. Вложенные классы C++ не имеют связи с объектом-оболочкой и прав доступа к его элементам.

```
    private Object[] objects;
    private int next = 0;
    public Sequence(int size) { items = new Object[size]; }
    public void add(Object x) {
            if(next < items.length)
                    items[next++] = x;
    }
    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.getSelector();
            while(!selector.end()) {
                    System.out.println(selector.current() + " ");
                    selector.next();
            }
    }
} /* Output:
0 1 2 3 4 5 6 7 8 9
*///:~
```

Класс Sequence — не более чем «оболочка» для массива с элементами Object, имеющего фиксированный размер. Для добавления новых объектов в конец последовательности (при наличии свободного места) используется метод add(). Для выборки каждого объекта в последовательности Sequence предусмотрен интерфейс с именем Selector. Он позволяет узнать, достигнут ли конец последовательности (метод end()), обратиться к текущему объекту (метод current()) и перейти к следующему объекту последовательности (метод next()). Так как Selector является интерфейсом, другие классы вправе реализовать его по-своему, а передача его в параметре методов повышает универсальность кода.

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 вызывается метод getSelector() для получения интерфейса Selector, который используется для перемещения по последовательности и выбора ее элементов.

На первый взгляд создание SequenceSelector напоминает создание обычного внутреннего класса. Но присмотритесь к нему повнимательнее. Заметьте, что в каждом из методов end(), current() и next() присутствует ссылка на items, а это не одно из полей класса SequenceSelector, а закрытое (private) поле объемлющего класса. Внутренний класс может обращаться ко всем полям и методам внешнего класса-оболочки, как будто они описаны в нем самом. Это весьма удобно, и вы могли в этом убедиться, изучая рассмотренный пример.

Итак, внутренний класс автоматически получает доступ к членам объемлющего класса. Как же это происходит? Внутренний класс содержит скрытую ссылку на определенный объект окружающего класса, ответственный за его создание. При обращении к члену окружающего класса используется эта (скрытая) ссылка. К счастью, все технические детали обеспечиваются компилятором, но теперь вы знаете, что объект внутреннего класса можно создать только в сочетании с объектом внешнего класса (как будет показано позже, если внутренний класс не является статическим). Конструирование объекта внутреннего класса требует наличия ссылки на объект внешнего класса; если ссылка недоступна, компилятор выдаст сообщение об ошибке. Большую часть времени весь процесс происходит без всякого участия со стороны программиста.

Конструкции .this и .new

Если вам понадобится получить ссылку на объект внешнего класса, запишите имя внешнего класса, за которым следует точка, а затем ключевое слово this. Полученная ссылка автоматически относится к правильному типу, известному и проверяемому на стадии компиляции, поэтому дополнительные издержки на стадии выполнения не требуются. Следующий пример показывает, как использовать конструкцию .this:

```
//: innerclasses/DotThis.java
// Обращение к объекту внешнего класса.

public class DotThis {
     void f() { System.out.println("DotThis.f()"); }
     public class Inner {
               public DotThis outer() {
                         return DotThis this,
                         // A plain "this" would be Inner's "this"
               }
     }
     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 бывает нужно приказать другому объекту создать объект одного из его внутренних классов. Для этого перед .new указывается ссылка на другой объект внешнего класса:

```
//: innerclasses/DotNew.java
// Непосредственное создание внутреннего класса в синтаксисе .new

public class DotNew {
     public class Inner {}
     public static void main(String[] args) {
```

```
        DotNew dn = new DotNew();
        DotNew.Inner dni = dn.new Inner();
    }
} /// ~
```

При создании объекта внутреннего класса указывается не имя внешнего класса DotNew, как можно было бы ожидать, а имя *объекта* внешнего класса. Это также решает проблему видимости имен для внутреннего класса, поэтому мы не используем (а вернее, не можем использовать) запись вида dn.new DotNew.Inner().

Невозможно создать объект внутреннего класса, не имея ссылки на внешний класс. Но если создать *вложенный класс* (статический внутренний класс), ссылка на объект внешнего класса не нужна.

Рассмотрим пример использования .new в примере Parcel:

```
// innerclasses/Parcel3.java
// Использование .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();
        // Для создания экземпляра внутреннего класса
        // необходимо использовать экземпляр внешнего класса:
        Parcel3.Contents c = p.new Contents();
        Parcel3.Destination d = p.new Destination("Танзания");
    }
} /// ~
```

Внутренние классы и восходящее преобразование

Мощь внутренних классов по-настоящему проявляется при выполнении восходящего преобразования к базовому классу, и в особенности к интерфейсу. (Получение ссылки на интерфейс по ссылке на реализующий его объект ничем принципиально не отличается от восходящего преобразования к базовому классу.) Причина в том, что внутренний класс — реализация интерфейса — может быть абсолютно невидимым и недоступным окружающему миру, а это очень удобно для сокрытия реализации. Все, что вы при этом получаете, — ссылку на базовый класс или интерфейс.

Для начала определим интерфейсы для предыдущих примеров:

```
// innerclasses/Destination.java
public interface Destination {
```

продолжение

```
    String readLabel();
} ///:~

//: innerclasses/Contents.java
public interface Contents {
    int value();
} ///:~
```

Теперь интерфейсы Contents и Destination доступны 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 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("Тасмания");
            // Запрещено - нет доступа к private-классу:
            //! Parcel4.PContents pc = p.new PContents();
    }
} ///:~
```

В класс Parcel4 было добавлено кое-что новое: внутренний класс PContents является закрытым (private), поэтому он недоступен для всех, кроме внешнего класса Parcel4. Класс PDestination объявлен как protected, следовательно, доступ к нему имеют только класс Parcel4, классы из одного пакета с Parcel4 (так как спецификатор protected также дает доступ в пределах пакета) и наследники класса Parcel4. Таким образом, программист-клиент обладает ограниченной информацией и доступом к этим членам класса. Более того, нельзя даже выполнить нисходящее преобразование к закрытому (private) внутреннему классу (или protected, кроме наследников), поскольку его имя недоступно, как показано

в классе Test. 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. Наконец, у компилятора Java появится возможность оптимизировать код.

Внутренние классы в методах и областях действия

Ранее мы рассмотрели ряд типичных применений внутренних классов. В основном ваш код будет содержать «простые» внутренние классы, смысл которых понять нетрудно. Однако синтаксис внутренних классов скрывает множество других, не столь тривиальных способов их использования: внутренние классы можно создавать внутри метода или даже в пределах произвольного блока. На то есть две причины:

- как было показано ранее, вы реализуете некоторый интерфейс, чтобы затем создавать и возвращать ссылку его типа;
- вы создаете вспомогательный класс для решения сложной задачи, но при этом не хотите, чтобы этот класс был открыт для посторонних.

В следующих примерах рассмотренная недавно программа будет изменена, благодаря чему у нас появятся:

- класс, определенный внутри метода;
- класс, определенный внутри области действия (блока), которая находится внутри метода;
- безымянный класс, реализующий интерфейс;
- безымянный класс, расширяющий класс, у которого отсутствует конструктор по умолчанию;
- безымянный класс, выполняющий инициализацию полей;
- безымянный класс, осуществляющий конструирование посредством инициализации экземпляра (безымянные внутренние классы не могут иметь конструкторы).

Первый пример демонстрирует создание целого класса в контексте метода (вместо создания в контексте другого класса). Такие внутренние классы называются *локальными*:

```
//· innerclasses/Parcel5.java
// Вложение класса в тело метода.

public class Parcel5 {
     public Destination dest(String s) {
              class PDestination implements Destination {
                       private String label;
                       private PDestination(String whereTo) {
                                label = whereTo;
                       }
                       public String readLabel() { return label; }
```

продолжение ⇩

```
            }
            return new PDestination(s);
        }
        public static void main(String[] args) {
            Parcel5 p = new Parcel5();
            Destination d = p.destination("Тасмания");
        }
    } ///:~
```

Теперь класс PDestination является частью метода destination(), а не частью класса Parcel5. Поэтому доступ к классу PDestination возможен только из метода destination(). Обратите внимание на восходящее преобразование, производимое в команде return, — из метода возвращается лишь ссылка на базовый класс Destination, и ничего больше. Конечно, тот факт, что имя класса PDestination находится внутри метода destination(), не означает, что объект PDestination после выхода из этого метода станет недоступным.

Идентификатор PDestination может использоваться для внутренних классов каждого отдельного класса в одном подкаталоге, без порождения конфликта имен.

Следующий пример демонстрирует, как можно вложить внутренний класс 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("ожидание");
            String s = ts.getSlip();
        }
        // Здесь использовать класс нельзя!
        // Вне области видимости.
        //! 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 значит, что *класс* создается в зависимости от условия — он компилируется вместе со всем остальным кодом. Однако при этом он недоступен вне контекста, в котором был определен. В остальном он выглядит точно так же, как и обычный класс.

Безымянные внутренние классы

Следующий пример выглядит немного странно:

```
// innerclasses/Parcel7.java
// Метод возвращает экземпляр безымянного внутреннего класса

public class Parcel7 {
     public Contents contents() {
               return new Contents() { // Вставить определение класса
                         private int i = 11;
                         public int value() { return i; }
                         }; // В данной ситуации точка с запятой необходима
     }
     public static void main(String[] args) {
               Parcel7 p = new Parcel7();
               Contents c = p.contents();
     }
} ///:~
```

Метод contents() совмещает создание возвращаемого значения с определением класса, который это возвращаемое значение и представляет! Вдобавок, этот класс является *безымянным* — у него отсутствует имя. Ситуация запутывается еще тем, что поначалу мы будто бы приступаем к созданию объекта Contents, а потом, остановившись перед точкой с запятой, говорим: «Стоп, а сюда я подкину определение класса».

Такая необычная форма записи значит буквально следующее: «Создать объект безымянного класса, который унаследован от Contents». Ссылка, которая возвращается при этом из выражения new, автоматически повышается до базового типа Contents. Синтаксис записи безымянного внутреннего класса является укороченной формой записи такой конструкции:

```
//: 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 безымянном внутреннем классе базовый класс Contents создается с использованием конструктора по умолчанию. Следующая программа показывает, как следует поступать, если базовый класс требует вызова конструктора с аргументами:

```
// innerclasses/Parcel8.java
// Вызов конструктора базового класса.

public class Parcel8 {
```

продолжение ⇩

```
    public Wrapping wrapping(int x) {
            // Вызов конструктора базового класса.
            return new Wrapping(x) { // аргумент конструктора
                    public int value() {
                            return super.value() * 47;
                    }
            }; // Требуется точка с запятой
    }
    public static void main(String[] args) {
            Parcel8 p = new Parcel8();
            Wrapping w = p.wrapping(10);
    }
} ///:~
```

Требуемый аргумент просто передается в конструктор базового класса, как в рассмотренном примере `x` в выражении `new Wrapping(x)`. Хотя это обычный класс с реализацией, `Wrapping` также используется в качестве общего «интерфейса» для своих производных классов:

```
//: innerclasses/Wrapping.java
public class Wrapping {
    private int i;
    public Wrapping(int x) { i = x; }
    public int value() { return i; }
} ///:~
```

Класс `Wrapping` имеет конструктор с аргументом — просто для того, чтобы ситуация стала чуть более интересной.

Точка с запятой в конце безымянного внутреннего класса поставлена вовсе не для того, чтобы обозначить конец тела класса (как делается в C++). Вместо этого она указывает на конец выражения, в котором содержится внутренний класс. Таким образом, в данном случае ее использование ничем не отличается от обычного.

Инициализацию также можно провести в точке определения полей безымянного класса:

```
//: innerclasses/Parcel9.java
// Безымянный внутренний класс, выполняющий инициализацию.
// Более короткая версия программы Parcel5 java

public class Parcel9 {
    // Для использования в безымянном внутреннем классе
    // аргументы должны быть неизменны (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("Тасмания");
    }
} ///:~
```

Если вы определяете безымянный внутренний класс и хотите при этом использовать объекты, определенные вне этого внутреннего класса, компилятор требует, чтобы переданные на них ссылки объявлялись неизменными (final), как это сделано аргументе destination(). Без такого объявления вы получите сообщение об ошибке при компиляции программы.

Пока мы ограничиваемся простым присваиванием значений полям, указанный подход работает. А если понадобится выполнить некоторые действия, свойственные конструкторам? В безымянном классе именованный конструктор определить нельзя (раз у самого класса нет имени!), но *инициализация экземпляра* (instance initialization) фактически позволяет добиться желаемого эффекта:

```
//. innerclasses/AnonymousConstructor.java
// Создание конструктора для безымянного внутреннего класса.
import static net.mindview.util.Print.*.

abstract class Base {
     public Base(int i) {
             print("Конструктор Base. i = " + i);
     }
     public abstract public void f();
}

public class AnonymousConstructor {
     public static Base getBase(int i) {
             return new Base(i) {
                     { print("Инициализация экземпляра"); }
                     public void f() {
                             print("Безымянный f()").
                     }
             }.
     }
     public static void main(String[] args) {
             Base base = getBase(47);
             base.f();
                     }).
     }
} /* Output.
Конструктор Base. i = 47
Инициализация экземпляра
Безымянный f()
*///.~
```

В таком случае переменная i *не обязана* быть неизменной (final). И хотя i передается базовому конструктору безымянного класса, она никогда не используется напрямую *внутри* безымянного класса.

Вернемся к нашим объектам Parcel, на этот раз выполнив для них инициализацию экземпляра. Отметьте, что параметры метода destination() должны быть объявлены неизменными, так как они используются внутри безымянного класса:

```
//· innerclasses/Parcel10.java
// Демонстрация "инициализации экземпляра" для
// конструирования безымянного внутреннего класса.
```

продолжение ⇩

```
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("Тасмания", 101 395F);
     }
} /* Output:
Превышение бюджета!
*/// ~
```

Внутри инициализатора экземпляра виден код, недоступный при инициализации полей (то есть команда if). Поэтому инициализатор экземпляра фактически является конструктором безымянного внутреннего класса. Конечно, возможности его ограничены; перегружать такой инициализатор нельзя, и поэтому он будет присутствовать в классе только в единственном числе.

Возможности безымянных внутренних классов несколько ограничены по сравнению с обычным наследованием — они могут либо расширять класс, либо реализовывать интерфейс, но не то и другое одновременно. А если вы выберете второй вариант, реализовать можно только один интерфейс.

Снова о методе-фабрике

Посмотрите, насколько приятнее выглядит пример interfaces/Factories.java при использовании безымянных 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();
                        }
                };
}

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 фабрику необязательно оформлять в виде именованного класса. Кроме того, часто бывает достаточно одного фабричного объекта, поэтому в данном случае он создается как статическое поле в реализации **Service**. Наконец, итоговый синтаксис выглядит более осмысленно.

Пример **interfaces/Games.java** тоже можно усовершенствовать с помощью безымянных внутренних классов:

```
//. innerclasses/Games.java
// Использование анонимных внутренних классов в библиотеке 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;
```

продолжение ⇩

```
        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
*///·~
```

Вспомните совет, данный в конце предыдущей главы: *отдавать предпочтение классам перед интерфейсами.* Если архитектура системы требует применения интерфейса, вы это поймете. В остальных случаях не применяйте интерфейсы без крайней необходимости.

Вложенные классы

Если связь между объектом внутреннего класса и объектом внешнего класса не нужна, можно сделать внутренний класс статическим (объявить его как static). Часто такой класс называют *вложенным*[1] (nested). Чтобы понять смысл ключевого

[1] Близкий аналог вложенных классов C++, за тем исключением, что в Java вложенные классы способны обращаться к закрытым членам внешнего класса.

слова static в отношении внутренних классов, следует вспомнить, что в объекте обычного внутреннего класса тайно хранится ссылка на объект создавшего его объемлющего внешнего класса. При использовании статического внутреннего класса такой ссылки не существует. Применение статического внутреннего класса означает следующее:

- для создания объекта статического внутреннего класса не нужен объект внешнего класса;
- из объекта вложенного класса нельзя обращаться к не-статическим членам внешнего класса.

Есть и еще одно различие между вложенными и обычными внутренними классами. Поля и методы обычного внутреннего класса определяются только на уровне внешнего класса, поэтому обычные внутренние классы не могут содержать статические данные, поля и классы. Но вложенные классы не имеют таких ограничений:

```
//· innerclasses/Parcel11.java
// Вложенные (статические внутренние) классы

public class Parcel11 {
     private static class PContents 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 cont() {
               return new ParcelContents(),
     }
     public static void main(String[] args) {
               Contents c = contents(),
               Destination d = destination("Тасмания"),
     }
} ///:~
```

В методе main() не требуется объекта класса Parcel11; вместо этого для вызова методов, возвращающих ссылки на Contents и Destination, используется обычный синтаксис обращения к статическим членам класса.

Как было сказано ранее, в обычном (не-статическом) внутреннем классе для обращения к объекту внешнего класса используется специальная ссылка this. Во вложенном классе такая ссылка недействительна (по аналогии со статическими методами).

Классы внутри интерфейсов

Обычно интерфейс не может содержать программный код, но вложенный класс *может* стать его частью. Любой класс, размещенный внутри интерфейса, автоматически является public и static. Так как класс является статическим, он не нарушает правил обращения с интерфейсом — этот вложенный класс просто использует пространство имен интерфейса. Во внутреннем классе даже можно реализовать окружающий интерфейс:

```
//· 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 может пригодиться для создания обобщенного кода, используемого с разными реализациями этого интерфейса.

Ранее в книге я предлагал помещать в каждый класс метод main(), позволяющий при необходимости протестировать данный класс. Недостатком такого подхода является дополнительный скомпилированный код, увеличивающий размеры программы. Если для вас это нежелательно, используйте статический внутренний класс для хранения тестового кода:

```
//· innerclasses/TestBed.java
// Помещение тестового кода во вложенный класс
// {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 создается 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; файл TestBed$Tester.class можно просто удалить перед окончательной сборкой программы.

Доступ вовне из многократно вложенных классов

Независимо от глубины вложенности, внутренний класс всегда может напрямую обращаться ко всем членам всех классов, в которые он встроен. Следующая программа демонстрирует этот факт[1]:

```
//: innerclasses/MultiNestingAccess.java
// Вложенные классы могут обращаться ко всем членам всех
// классов, в которых они находятся.

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();
     }
} ///.~
```

Как видно из примера, в классе MNA.A.B методы f() и g() вызываются без дополнительных описаний (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. Синтаксис .new обеспечивает правильную область действия, и вам не приходится уточнять имя класса при вызове конструктора.

Внутренние классы: зачем?

К настоящему моменту мы подробно рассмотрели синтаксис и семантику работы внутренних классов, но это не дало ответа на вопрос, зачем они вообще нужны.

[1] Снова благодарю Мартина Даннера.

Что же заставило создателей Java добавить в язык настолько фундаментальное свойство?

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 резонный вопрос: «Если мне понадобится ссылка на интерфейс, почему бы внешнему классу не реализовать этот интерфейс?» Ответ: «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 такова:

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 возможности внутренних классов наследовать реализацию более чем одного реального или абстрактного класса некоторые задачи планирования и программирования становятся практически неразрешимыми. Поэтому внутренний класс выступает как «довесок» решения проблемы множественного наследования. Интерфейсы берут на себя часть этой задачи, тогда как внутренние классы фактически обеспечивают «множественное наследование реализации». Другими словами, внутренние классы позволяют наследовать от нескольких не-интерфейсов.

Чтобы понять все сказанное до конца, рассмотрим ситуацию, где два интерфейса тем или иным способом должны быть реализованы в классе. Вследствие гибкости интерфейсов возможен один из двух способов решения: отдельный одиночный класс или внутренний класс:

```
//: innerclasses/MultiInterfaces.java
// Два способа реализации нескольких интерфейсов.

interface A {}
interface B {}

class X implements A, B {}

class Y implements A {
     B makeB() {
               // Безымянный 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 других ограничений оба подхода, использованные в рассмотренном примере, ничем не отличаются с точки зрения реализации. Оба они работают.

Но если вместо интерфейсов имеются реальные или абстрактные классы и новый класс должен как-то реализовать функциональность двух других, придется прибегнуть к внутренним классам:

```
//: innerclasses/MultiImplementation.java
// При использовании реальных или абстрактных классов
// "множественное наследование реализации" возможно
// только с применением внутренних классов
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 нет необходимости решать задачу «множественного наследования реализации», скорее всего, вы без особого труда напишите программу, не прибегая к особенностям внутренних классов. Однако внутренние классы открывают перед вами ряд дополнительных возможностей:

- У внутреннего класса может существовать произвольное количество экземпляров, каждый из которых обладает собственной информацией состояния, не зависящей от состояния объекта внешнего класса.
- Один внешний класс может содержать несколько внутренних классов, по-разному реализующих один и тот же интерфейс или наследующих от единого базового класса. Вскоре мы рассмотрим пример такой конструкции.
- Место создания объекта внутреннего класса не привязано к месту и времени создания объекта внешнего класса.

- Внутренний класс не использует тип отношений классов «является тем-то», способных вызвать недоразумения; он представляет собой отдельную сущность.

Например, если бы в программе Sequence.java отсутствовали внутренние классы, 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 обеспечивают только внутренние классы.

Замыкания и обратные вызовы

Замыканием (closure) называется вызываемый объект, который сохраняет информацию о контексте, он был создан. Из этого определения видно, что внутренний класс является объектно-ориентированным замыканием, поскольку он не только содержит информацию об объекте внешнего класса («место создания»), но к тому же располагает ссылкой на весь объект внешнего класса, с помощью которой он может манипулировать всеми членами этого объекта, в том числе и закрытыми (private).

При обсуждении того, стоит ли включать в Java некое подобие указателей, самым веским аргументом «за» была возможность *обратных вызовов* (callback). В механизме обратного вызова некоторому стороннему объекту передается информация, позволяющая ему затем обратиться с вызовом к объекту, который произвел изначальный вызов. Это очень мощная концепция программирования, к которой мы еще вернемся. С другой стороны, при реализации обратного вызова на основе указателей вся ответственность за его правильное использование возлагается на программиста. Как было показано ранее, язык Java ориентирован на безопасное программирование, поэтому указатели в него включены не были.

Замыкание, предоставляемое внутренним классом, — хорошее 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();
}

// Простая реализация интерфейса:
class Callee1 implements Incrementable {
     private int i = 0;
     public void increment() {
               i++;
               print(i);
     }
}
```

```
class MyIncrement {
     public void increment() { System.out.println("Другая операция") };
     public static void f(MyIncrement mi) { mi.increment(); }
}

// Если 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;
                         // в противном случае возникает 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 демонстрирует различия между реализацией интерфейса внешним или внутренним классом. Класс Callee1 — наиболее очевидное решение задачи с точки зрения программирования. 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 наследуется в Callee2, метод increment() нельзя переопределить для использования в качестве метода интерфейса Incrementable, поэтому нам приходится предоставлять отдельную реализацию во внутреннем классе. Также отметьте, что создание внутреннего класса не затрагивает и не изменяет существующий интерфейс внешнего класса.

Все элементы, за исключением метода getCallbackReference(), в классе Callee2 являются закрытыми. Для *любой* связи с окружающим миром необходим интерфейс Incrementable. Здесь мы видим, как интерфейсы позволяют полностью отделить интерфейс 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 угодно).

Класс Caller получает ссылку на Incrementable в своем конструкторе (хотя передача ссылки для обратного вызова может происходить в любое время), а после этого использует ссылку для «обратного вызова» объекта Callee.

Главным достоинством обратного вызова является его гибкость — вы можете динамически выбирать функции, выполняемые во время работы программы.

Внутренние классы и система управления

В качестве более реального пример использования внутренних классов мы рассмотрим 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. При работе с каркасами приложений обычно используется наследование от одного или нескольких классов, с переопределением некоторых методов. Код переопределенных методов адаптирует типовое решение, предоставляемое каркасом приложения, к вашим конкретным потребностям. Система управления представляет собой определенный тип каркаса приложения, основным движущим механизмом которого является обработка событий. Такие системы называются *системами, управляемыми по событиям* (event-driven system). Одной из самых типичных задач в прикладном программировании является создание графического интерфейса пользователя (GUI), всецело и полностью ориентированного на обработку событий.

Чтобы на наглядном примере увидеть, как с применением внутренних классов достигается простота создания и использования библиотек, мы рассмотрим систему, ориентированную на обработку событий по их «готовности». Хотя в практическом смысле под «готовностью» может пониматься все, что угодно, в нашем случае она будет определяться по показаниям счетчика времени. Далее

приводится общее описание управляющей системы, никак не зависящей от того, чем именно она управляет. Нужная информация предоставляется посредством наследования, при реализации метода action().

Начнем с определения интерфейса, описывающего любое событие системы. Вместо интерфейса здесь используется абстрактный класс, поскольку по умолчанию управление координируется по времени, а следовательно, присутствует частичная реализация:

```
//: 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() {      // Позволяет перезапуск
              eventTime = System nanoTime() + delayTime;
     }
     public boolean ready() {
              return System.nanoTime() >= eventTime;
     }
     public abstract void action();
} ///:~
```

Конструктор просто запоминает время (от момента создания объекта), через которое должно выполняться событие Event, и после этого вызывает метод start(), который прибавляет к текущему времени интервал задержки, чтобы вычислить время возникновения события. Метод start() отделен от конструктора, благодаря чему становится возможным «перезапуск» события после того, как его время уже истекло; таким образом, объект Event можно использовать многократно. Скажем, если вам понадобится повторяющееся событие, достаточно 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 событие Event активизируется не по времени, а по иному условию.

Следующий файл описывает саму систему управления, которая распоряжается событиями и инициирует их. Объекты Event содержатся в контейнере List<Event>. На данный момент достаточно знать, что метод add() присоединяет объект Event к концу контейнера с типом List, метод size() возвращает количество элементов в контейнере, синтаксис foreach() осуществляет последовательную выборку элементов List, а метод remove() удаляет заданный элемент из контейнера:

```
//: innerclasses/controller/Controller.java
// Обобщенная система управления
package innerclasses.controller;
import java.util.*;
```

продолжение ⇩

```
public class Controller {
     // Класс из пакета java.util для хранения событий Event:
     private List<Event> eventList = new ArrayList<Event>();
     public void addEvent(Event c) { eventList.add(c); }
     public void run() {
              while(eventList.size() > 0) {
                       for(Event e : new ArrayList<Event>(eventList))
                                if(e.ready()) {
                                         System.out.println(e);
                                         e.action();
                                         eventList.remove(e);
                                }
     }
} ///:~
```

Метод run() в цикле перебирает копию eventList в поисках событий Event, готовых для выполнения. Для каждого найденного элемента он выводит информацию об объекте методом toString(), вызывает метод action(), а после этого удаляет событие из списка.

Заметьте, что в этой архитектуре совершенно неважно, *что* конкретно выполняет некое событие Event. В этом и состоит «изюминка» разработанной системы; она отделяет постоянную составляющую от изменяющейся. «Вектором изменения» являются различные действия разнообразных событий Event, выражаемые посредством создания разных субклассов Event.

На этом этапе в дело вступают внутренние классы. Они позволяют добиться двух целей:

1. Вся реализация системы управления создается в одном классе, с полной инкапсуляцией всей специфики данной реализации. Внутренние классы используются для представления различных разновидностей action(), необходимых для решения задачи.
2. Внутренние классы помогают избежать громоздкой, неудобной реализации, так как у них есть доступ к внешнему классу. Без этой возможности программный код очень быстро станет настолько неприятным, что вам захочется поискать другие альтернативы.

Рассмотрим конкретную реализацию системы управления, разработанную для управления функциями оранжереи[1]. Все события — включение света, воды и нагревателей, звонок и перезапуск системы — абсолютно разнородны. Однако система управления разработана так, что различия в коде легко изолируются. Внутренние классы помогают унаследовать несколько производных версий одного базового класса Event в пределах одного класса. Для каждого типа события от Event наследуется новый внутренний класс, и в его реализации action() записывается управляющий код.

Как это обычно бывает при использовании каркасов приложений, класс GreenhouseControls наследует от класса Controller:

[1] Я всегда решал эту задачу с особым удовольствием; она впервые появилась в одной из первых моих книг *C++ Inside & Out*, но Java-реализация выглядит гораздо элегантнее.

```
//: 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() {
                         // Сюда помещается аппаратный вызов, выполняющий
                         // физическое включение света.
                         light = true;
               }
               public String toString() { return "Свет включен"; }
     }
     public class LightOff extends Event {
               public LightOff(long delayTime) { super(delayTime); }
               public void action() {
                         // Сюда помещается аппаратный вызов, выполняющий
                         // физическое 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 включения
                         // системы полива.
                         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 выключения
                         // системы полива
                         water = false;
               }
               public String toString() {
                         return "Полив отключен";
               }
     }
     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";
               }
```

продолжение ⇩

```
                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
        // 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 каждый раз
                                addEvent(e);
                        }
                        start(); // Возвращаем это событие Event
                        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**, **thermostat** и **ring** принадлежат внешнему классу **GreenhouseControls**, и все же внутренние классы имеют возможность обращаться к ним, не используя особой записи и не запрашивая особых разрешений. Большинство методов **action()** требует управления оборудованием оранжереи, что, скорее всего, привлечет в программу сторонние низкоуровневые вызовы.

В основном классы Event похожи друг на друга, однако классы Bell и Restart представляют собой особые случаи. Bell выдает звуковой сигнал и добавляет себя в список событий, чтобы звонок позднее сработал снова. Заметьте, что внутренние классы действуют *почти* как множественное наследование: классы Bell и Restart имеют доступ ко всем методам класса Event, а также ко всем методам внешнего класса GreenhouseControls.

Классу Restart передается массив объектов 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 шаблона проектирования *«команда»* — каждый объект в EventList представляет собой запрос, инкапсулированный в объекте:

```
//: c08:GreenhouseController.java
// Настраивает и запускает систему управления.
// {Args: 5000}
import innerclasses.controller.*;

public class GreenhouseController {
     public static void main(String[] args) {
               GreenhouseControls gc = new GreenhouseControls();
               // Вместо жесткого кодирования фиксированных данных
               // можно было бы считать информацию для настройки
               // 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 отключен
Термостат использует дневной режим
Перезапуск системы
Отключение
///:~
```

Класс инициализирует систему, включая в нее нужные события. Если передать программе параметр командной строки, она завершается по истечении заданного количества миллисекунд (используется при тестировании). Конечно, чтобы программа стала более гибкой, описания событий следовало бы не включать в программный код, а загружать из файла.

Этот пример поможет понять всю ценность механизма внутренних классов, особенно в случае с системами управления.

Наследование от внутренних классов

Так как конструктор внутреннего класса связывается со ссылкой на окружающий внешний объект, наследование от внутреннего класса получается чуть сложнее, чем обычное. Проблема состоит в том, что «скрытая» ссылка на объект объемлющего внешнего класса *должна быть* инициализирована, а в производном классе больше не существует объемлющего объекта по умолчанию. Для явного указания объемлющего внешнего объекта применяется специальный синтаксис:

```
//: 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 дело доходит до создания конструктора, предлагаемый по умолчанию конструктор не подходит, и вы не можете просто передать ссылку на внешний объект. Необходимо включить в тело конструктора выражение

```
ссылкаНаОбъемлющийКласс.super();
```

в теле конструктора. Оно обеспечит недостающую ссылку, и программа откомпилируется.

Можно ли переопределить внутренний класс?

Что происходит, если вы создаете внутренний класс, затем наследуете от его внешнего класса, а после этого заново описываете внутренний класс в производном

классе? Другими словами, можно ли переопределить 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. Можно подумать, что при создании объекта **BigEgg** должен использоваться «переопределенный» класс **Yolk**, но это отнюдь не так, как видно из результата работы программы.

Этот пример просто показывает, что при наследовании от внешнего класса ничего особенного с внутренними классами не происходит. Два внутренних класса — совершенно отдельные составляющие, с независимыми пространствами имен. Впрочем, возможность явного наследования от внутреннего класса сохранилась:

```
//: 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().
```

продолжение ⇩

```
        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() { System.out.println("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** повысить один из своих объектов **Yolk** до ссылки **y** в классе **Egg2**, поэтому при вызове **y.f()** в методе **g()** используется переопределенная версия **f()**. Второй вызов **Egg2.Yolk()** — это вызов конструктора базового класса из конструктора класса **BigEgg2.Yolk**. Мы также видим, что при вызове метода **g()** используется «обновленная» версия метода **f()**.

Локальные внутренние классы

Как было замечено ранее, внутренние классы также могут создаваться в блоках кода — чаще всего в теле метода. Локальный внутренний класс не может иметь спецификатора доступа, так как он не является частью внешнего класса, но для него доступны все неизменные (**final**) переменные текущего блока и все члены внешнего класса. Следующий пример сравнивает процессы создания локального внутреннего класса и безымянного 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() {
                        printnb(name); // неизменный аргумент
                        return count++;
                }
        }
        return new LocalCounter();
    }
    // То же самое с безымянным внутренним классом:
    Counter getCounter2(final String name) {
        return new Counter() {
                // У безымянного внутреннего класса не может быть
                // именованного конструктора, «легальна» только
                // инициализация экземпляром:
                {
                        print("Counter()");
                }
                public int next() {
                        printnb(name); // неизменный аргумент
                        return count++;
                }
        };
    }
    public static void main(String[] args) {
        LocalInnerClass lic = new LocalInnerClass();
        Counter
                c1 = lic.getCounter(" локальный"),
                c2 = lic.getCounter2(" безымянный");
        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
безымянный 5
безымянный 6
безымянный 7
безымянный 8
безымянный 9
*///:~
```

Объект **Counter** возвращает следующее по порядку значение. Он реализован и как локальный класс, и как безымянный внутренний класс, с одинаковым поведением и характеристиками. Поскольку имя локального внутреннего класса недоступно за пределами метода, доводом для применения локального класса вместо безымянного внутреннего может быть необходимость в именованном

конструкторе и (или) перегруженных конструкторах; безымянные внутренние классы допускают только инициализацию экземпляром.

Другая причина для использования локального внутреннего класса вместо безымянного внутреннего — необходимость создания более чем одного объекта такого класса.

Идентификаторы внутренних классов

Так как каждый класс компилируется в файл с расширением .class, содержащий полную информацию о создании его экземпляров (эта информация помещается в «мета-класс», называемый объектом Class), напрашивается предположение, что внутренние классы также создают файлы .class для хранения информации о *своих* объектах Class. Имена этих файлов-классов строятся по жестко заданной схеме: имя объемлющего внешнего класса, затем символ $ и 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 безымянными, компилятор использует в качестве их идентификаторов номера. Если внутренние классы вложены в другие внутренние классы, их имена просто присоединяются после символа $ и идентификаторов всех внешних классов.

Хотя такая схема построения внутренних имен проста и прямолинейна, она вполне надежна и работает практически в любых ситуациях. Так как она является стандартной для языка Java, все получаемые файлы автоматически становятся платформно-независимыми.

Резюме

Интерфейсы и внутренние классы — весьма нетривиальные концепции, 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 решить с применением множественного наследования. Однако множественное наследование C++ создает массу проблем; по сравнению с ним интерфейсы и внутренние классы Java гораздо более доступны.

Хотя сами по себе эти механизмы не так уж сложны, решение об их использовании принимается на уровне проектирования (как и в случае с полиморфизмом). Со временем вы научитесь сразу оценивать, где большую выгоду даст интерфейс, где внутренний класс, а где нужны обе возможности сразу. А пока достаточно хотя бы в общих чертах ознакомиться с их синтаксисом и семантикой.

Коллекции объектов

11

Ограниченное количество объектов с фиксированным временем жизни характерно разве что для относительно простых программ.

В основном ваши программы будут создавать новые объекты на основании критериев, которые станут известны лишь во время их работы. До начала выполнения программы вы не знаете ни количества, ни даже типов нужных вам объектов. Следовательно, использовать именованную ссылку для каждого из возможных объектов не удастся:

```
MyType aReference;
```

так как заранее неизвестно, сколько таких ссылок реально потребуется.

В большинстве языков существуют некоторые пути решения этой крайне насущной задачи. В Java предусмотрено несколько способов хранения объектов (или, точнее, ссылок на объекты). Встроенным типом является массив, который мы уже рассмотрели. Библиотека утилит Java (`java.util.*`) также содержит достаточно полный набор *классов контейнеров* (также известных, как *классы коллекций*, но, поскольку имя `Collection` (коллекция) используется для обозначения определенного подмножества библиотеки Java, я буду употреблять общий термин «контейнер»). Контейнеры обладают весьма изощренными возможностями для хранения объектов и работы с ними, и с их помощью удается решить огромное количество задач.

Параметризованные и типизованные контейнеры

Одна из проблем, существовавших при работе с контейнерами до выхода Java SE5, заключалась в том, что компилятор позволял вставить в контейнер объект неверного типа. Для примера рассмотрим один из основных рабочих контейнеров

ArrayList, в котором мы собираемся хранить объекты Apple. Пока рассматривайте ArrayList как «автоматически расширяемый массив». Работать с ним несложно: создайте объект, вставляйте объекты методом add(), обращайтеь к ним методом get(), используйте индексирование — так же, как для массивов, но без квадратных скобок. ArrayList также содержит метод size(), который возвращает текущее количество элементов в массиве.

В следующем примере в контейнере размещаются объекты Apple и Orange, которые затем извлекаются из него. Обычно компилятор Java выдает предупреждение, потому что в данном примере не используется параметризация, однако в Java SE5 существует специальная *директива* @SuppressWarnings для подавления предупреждений. Директивы начинаются со знака @ и могут получать аргументы; в данном случае аргумент означает, что подавляются только «непроверяемые» предупреждения:

```
//: holding/ApplesAndOrangesWithoutGenerics.java
// Простой пример работы с контейнером
// (компилятор выдает предупреждения).
// {ThrowsException}
import java.util.*;

class Apple {
     private static long counter;
     private final long id = counter++;
     public long id() { return id; }
}

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 препятствует добавлению объекта Orange:
               apples.add(new Orange());
               for(int i = 0; i < apples.size(); i++)
                        ((Apple)apples.get(i)).id();
                        // Объект Orange обнаруживается только во время выполнения
     }
}
///:~
```

Директивы Java SE5 будут рассмотрены позднее.

Apple и Orange — совершенно разные классы; они не имеют ничего общего, кроме происхождения от Object (напомню: если в программе явно не указан базовый класс, то в этом качестве используется Object). Так как в ArrayList хранятся объекты Object, метод add() может добавлять в контейнер не только объекты Apple, но и Orange, без ошибок компиляции или времени выполнения. Но при вызове метода get() класса ArrayList вы вместо объекта Apple получаете ссылку на Object, которую необходимо преобразовать в Apple. Все выражение должно быть заключено в круглые скобки, чтобы преобразование было выполнено

перед вызовом метода id() класса Apple. Во время выполнения, при попытке преобразования объекта Orange в Apple, произойдет исключение.

В главе «параметризованные типы» вы узнаете, что *создание* классов, использующих механизм параметризации, может быть довольно сложной задачей. С другой стороны, с *применением* готовых параметризованных классов проблем обычно не бывает. Например, чтобы определить объект ArrayList, предназначенный для хранения объектов Apple, достаточно использовать вместо имени ArrayList запись ArrayList<Apple>. В угловых скобках перечисляются *параметры типов* (их может быть несколько), указывающие тип объектов, хранящихся в данном экземпляре контейнера.

Механизм параметризации предотвращает занесение объектов неверного типа в контейнер на стадии компиляции. Рассмотрим тот же пример, но с использованием параметризации:

```
// holding/ApplesAndOrangesWithGenerics.java
import java.util.*;

public class ApplesAndOrangesWithGenerics {
    public static void main(String[] args) {
        ArrayList<Apple> apples = new ArrayList<Apple>();
        for(int i = 0; i < 3; i++)
            apples.add(new Apple());
        // Ошибка 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
*///:~
```

На этот раз компилятор не разрешит поместить объекты Orange в контейнер apples, поэтому вы получите ошибку на стадии компиляции (а не на стадии выполнения).

Также обратите внимание на то, что выборка данных из List не требует преобразования типов. Поскольку контейнер знает тип хранящихся в нем элементов, он автоматически выполняет преобразование при вызове get(). Таким образом, параметризация не только позволяет компилятору проверять тип объектов, помещаемых в контейнеры, но и упрощает синтаксис работы с объектами в контейнере. Пример также показывает, что, если индексы элементов вам не нужны, для перебора можно воспользоваться синтаксисом foreach.

Вы не обязаны точно соблюдать тип объекта, указанный в качестве параметра типа. Восходящее преобразование работает с параметризованными контейнерами точно так же, как и с 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
*///:~
```

Мы видим, что в контейнер, рассчитанный на хранение объектов Apple, можно помещать объекты типов, производных от Apple.

В результатах, полученных с использованием метода toString() объекта Object, выводится имя класса с беззнаковым шестнадцатеричным представлением *хеш-кода* объекта (сгенерированного методом hashCode()).

Основные концепции

В библиотеке контейнеров Java проблема хранения объектов делится на две концепции, выраженные в виде базовых интерфейсов библиотеки:

- **Коллекция**: группа отдельных элементов, сформированная по некоторым правилам. Класс List (список) хранит элементы в порядке вставки, в классе Set (множество) нельзя хранить повторяющиеся элементы, а класс Queue (очередь) выдает элементы в порядке, определяемом спецификой очереди (обычно это порядок вставки элементов в очередь).
- **Карта**: набор пар объектов «ключ-значение», с возможностью выборки по ключу. ArrayList позволяет искать объекты по порядковым номерам, поэтому в каком-то смысле он связывает числа с объектами. Класс Map (карта — также встречаются термины *ассоциативный массив* и *словарь*) позволяет искать объекты по другим объектам — например, получить объект значения по объекту ключа, по аналогии с поиском определения по слову.

Хотя на практике это не всегда возможно, в идеале весь программный код должен писаться в расчете на взаимодействие с этими интерфейсами, а точный

тип указывается только в точке создания. Следовательно, объект List может быть создан так:

```
List<Apple> apples = new ArrayList<Apple>();
```

Обратите внимание на восходящее преобразование ArrayList к List, в отличие от предыдущих примеров. Если позднее вы решите изменить реализацию, достаточно сделать это в точке создания:

```
List<Apple> apples = new LinkedList<Apple>();
```

Итак, в типичной ситуации вы создаете объект реального класса, повышаете его до соответствующего интерфейса, а затем используете интерфейс во всем остальном коде.

Такой подход работает не всегда, потому что некоторые классы обладают дополнительной функциональностью. Например, LinkedList содержит дополнительные методы, не входящие в интерфейс List, а TreeMap — методы, не входящие в Map. Если такие методы используются в программе, восходящее преобразование к обобщенному интерфейсу невозможно.

Интерфейс Collection представляет концепцию *последовательности* как способа хранения группы объектов. В следующем простом примере интерфейс Collection (представленный контейнером ArrayList) заполняется объектами Integer, с последующим выводом всех элементов полученного контейнера:

```
//: 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,
*///:~
```

Поскольку в этом примере используются только методы Collection, подойдет объект любого класса, производного от Collection, но ArrayList является самым простейшим типом последовательности.

Все коллекции поддерживают перебор в синтаксисе foreach, как в приведенном примере. Позднее в этой главе будет рассмотрена другая, более гибкая концепция *итераторов*.

Добавление групп элементов

Семейства Arrays и Collections в java.util содержат вспомогательные методы для включения групп элементов в коллекции. Метод Arrays.asList() получает либо массив, либо список элементов, разделенных запятыми, и преобразует его в объект List. Метод Collections.addAll() получает объект Collection и либо массив,

либо список, разделенный запятыми, и добавляет элементы в Collection. Пример:

```
//· 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 намного быстрее, но таким способом
               // невозможно сконструировать Collection:
               Collections.addAll(collection, 11, 12, 13, 14, 15);
               Collections.addAll(collection, moreInts);
               // Создает список на основе массива:
               List<Integer> list = Arrays.asList(16, 17, 18, 19, 20);
               list.set(1, 99); // Можно - изменение элемента
               // list.add(21); // Ошибка времени выполнения - нижележащий
                                // массив не должен изменяться в размерах
     }
} ///:~
```

Конструктор Collection может получать другой объект Collection, используемый для его инициализации, поэтому для передачи исходных данных можно воспользоваться методом Arrays.asList(). Однако метод Collections.addAll() работает намного быстрее, и вы с таким же успехом можете сконструировать Collection без элементов, а затем вызвать Collections.addAll — этот способ считается предпочтительным.

Методу Collection.addAll() в аргументе может передаваться только другой объект Collection, поэтому он уступает в гибкости методам Arrays.asList() и Collections.addAll(), использующим переменные списки аргументов.

Также можно использовать вывод Arrays.asList() напрямую, в виде List, но в этом случае нижележащим представлением будет массив, не допускающий изменения размеров. Вызов add() или delete() для такого списка приведет к попытке изменения размера массива, а это приведет к ошибке во время выполнения.

Недостаток Arrays.asList() заключается в том, что он пытается «вычислить» итоговый тип List, не обращая внимания на то, что ему присваивается. Иногда это создает проблемы:

```
//: holding/AsListInference.java
// Arrays.asList() makes its best guess about type.
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 компилируется:
               // List<Snow> snow2 = Arrays.asList(
               //   new Light(), new Heavy());
               // Сообщение компилятора:
               // found     java.util.List<Powder>
               // required: java.util.List<Snow>

               // Collections.addAll() работает нормально:
               List<Snow> snow3 = new ArrayList<Snow>();
               Collections.addAll(snow3, new Light(), new Heavy());

               // Передача информации посредством уточнения
               // типа аргумента
               List<Snow> snow4 = Arrays.<Snow>asList(
                         new Light(), new Heavy());
     }
} ///:~
```

При попытке создания **snow2**, **Arrays.asList()** создает **List<Powder>** вместо **List <Snow>**, тогда как **Collections.addAll()** работает нормально, потому что целевой тип определяется первым аргументом. Как видно из создания **snow4**, в вызов **Arrays.asList()** можно вставить «подсказку», которая сообщает компилятору фактический тип объекта **List**, производимого **Arrays.asList()**.

С контейнерами **Map** дело обстоит сложнее, и стандартная библиотека Java не предоставляет средств их автоматической инициализации, кроме как по содержимому другого объекта **Map**.

Вывод содержимого контейнеров

Для получения печатного представления массива необходимо использовать метод **Arrays.toString**, но контейнеры отлично выводятся и без посторонней помощи. Следующий пример демонстрирует использование основных типов контейнеров:

```
//: c11:PrintingContainers.java
// Вывод контейнеров по умолчанию
import java.util.*;
import static net.mindview.util.Print.*;

public class PrintingContainers {
     static Collection fill(Collection<String> collection) {
               collection.add("rat");
               collection.add("cat");
               collection.add("dog");
               collection.add("dog");
               return collection;
     }
     static Map fill(Map<String,String> map) {
               map.put("rat", "Fuzzy");
```

продолжение ⇩

```
                map.put("cat", "Rags");
                map.put("dog", "Bosco");
                map.put("dog", "Spot");
                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:
[rat, cat, dog, dog]
[rat, cat, dog, dog]
[dog, cat, rat]
[cat, dog, rat]
[rat, cat, dog]
{dog=Spot, cat=Rags, rat=Fuzzy}
{cat=Rags, dog=Spot, rat=Fuzzy}
{rat=Fuzzy, cat=Rags, dog=Spot}
*///:~
```

Как уже было упомянуто, в библиотеке контейнеров Java существует две основные категории, различающиеся прежде всего тем, сколько в одной ячейке контейнера «помещается» элементов. Коллекции (Collection) содержат только один элемент в каждой ячейке. К этой категории 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). В контейнерах Map (карта) хранятся два объекта: ключ и связанное с ним значение.

Из выходных данных программы видно, что вывод по умолчанию (обеспечиваемый методом toString() каждого контейнера) дает вполне приличные результаты. Содержимое Collection выводится в квадратных скобках, с разделением элементов запятыми. Содержимое Map заключается в фигурные скобки, ключи и значения разделяются знаком равенства (ключи слева, значения справа).

Контейнеры ArrayList и LinkedList принадлежат к семейству List, и из выходных данных видно, что элементы в них хранятся в порядке вставки. Они различаются не только скоростью выполнения тех или иных операций, но и тем, что LinkedList содержит больше операций, чем ArrayList.

HashSet, TreeSet и LinkedHashSet относятся к семейству Set. Из выходных данных видно, что в множествах Set каждый элемент хранится только в одном экземпляре, а разные реализации Set используют разный порядок хранения элементов. В HashSet порядок элементов определяется по довольно сложному алгоритму — пока достаточно знать, что этот алгоритм обеспечивает минимальное время выборки элементов, но порядок следования элементов на первый взгляд выглядит хаотично. Если порядок хранения для вас важен, используйте

контейнер TreeSet, в котором объекты хранятся отсортированными по возрастанию в порядке сравнения, или LinkedHashSet с хранением элементов в порядке добавления.

Карта (Map) позволяет искать объекты по *ключу*, как несложная база данных. Объект, ассоциированный с ключом, называется *значением*. (Карты также называют *ассоциативными массивами*.)

В нашем примере используются три основные разновидности Map: HashMap, TreeMap и LinkedHashMap. Как и HashSet, HashMap обеспечивает максимальную скорость выборки, а порядок хранения его элементов не очевиден. TreeMap хранит ключи отсортированными по возрастанию, а LinkedHashMap хранит ключи в порядке вставки, но обеспечивает скорость поиска HashMap.

List

Контейнеры List гарантируют определенный порядок следования элементов. Интерфейс List дополняет Collection несколькими методами, обеспечивающими вставку и удаление элементов в середине списка.

Существует две основные разновидности List:

- Базовый контейнер ArrayList, оптимизированный для произвольного доступа к элементам, но с относительно медленнными операциями вставки (удаления) элементов в середине списка.
- Контейнер LinkedList, оптимизированный для последовательного доступа, с быстрыми операциями вставки (удаления) в середине списка. Произвольный доступ к элементам LinkedList выполняется относительно медленно, но по широте возможностей он превосходит ArrayList.

В следующем примере используется библиотека typenfo.pets из главы «Информация о типе». Она содержит иерархию классов домашних животных Pet, а также ряд вспомогательных средств для случайного построения объектов Pet. Пока достаточно знать, что (1) библиотека содержит класс Pet и производные типы, и (2) статический метод Pets.arrayList() возвращает контейнер 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); // Автоматическое изменение размера
               print("2: " + pets);
               print("3: " + pets.contains(h));
               pets.remove(h); // Удаление объекта
               Pet p = pets.get(2);
```

продолжение ⇩

```
        print("4: " +  p + " " + pets.indexOf(p));
        Pet cymric = new Cymric();
        print("5: " + pets.indexOf(cymric));
        print("6: " + pets.remove(cymric));
        // Точно заданный объект:
        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("subList: " + sub);
        print("10: " + pets.containsAll(sub));
        Collections.sort(sub); // Сортировка "на месте"
        print("sorted subList: " + sub);
        // Для containsAll() порядок неважен:
        print("11: " + pets.containsAll(sub));
        Collections.shuffle(sub, rand); // Случайная перестановка
        print("shuffled subList: " + 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); // Получение новой копии
        copy remove(2); // Удаление по индексу
        print("14: " + copy);
        copy.removeAll(sub); // Удаление заданных элементов
        print("15: " + copy);
        copy.set(1, new Mouse()); // Замена элемента
        print("16: " + copy);
        copy.addAll(2, sub); // Вставка в середину списка
        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]
subList: [Manx, Mutt, Mouse]
10: true
sorted subList: [Manx, Mouse, Mutt]
11: true
```

```
shuffled subList: [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 вывода пронумерованы, чтобы вам было удобнее связывать результат с исходным кодом.

В первой строке выводится исходный контейнер List с объектами Pets. В отличие от массивов, List поддерживает добавление и удаление элементов с изменением размеров списка. Результат добавления Hamster виден в строке 2: объект появляется в конце списка.

Метод contains() проверяет, присутствует ли объект в списке. Чтобы удалить объект, передайте ссылку на него методу remove(). Кроме того, при наличии ссылки на объект можно узнать его индекс в списке при помощи метода indexOf(), как показано в строке 4.

При проверке вхождения элемента в List, проверке индекса элемента и удаления элемента из List по ссылке используется метод equals() (из корневого класса Object). Все объекты Pet считаются уникальными, поэтому несмотря на присутствие двух объектов Cymric в списке, если я создам новый объект Cymric и передам его indexOf(), результат будет равен −1 (элемент не найден), а вызов remove() вернет false. Для других классов метод equals() может быть определен иначе — например, объекты String считаются равными в случае совпадения содержимого.

В строках 7 и 8 из List успешно удаляется заданный объект.

Строка 9 и предшествующий ей код демонстрируют вставку элемента в середину списка. Метод subList() позволяет легко создать «срез» из подмножества элементов списка; естественно, при передаче его методу containsAll() большего списка будет получен истинный результат. Вызовы Collections.sort() и Collections.shuffle() для sub не влияют на результат вызова containsAll().

Метод retainAll() фактически выполняет операцию «пересечения множеств», то есть определения всех элементов copy, которые также присутствуют в sub. И снова поведение метода зависит от реализации equals().

В строке 14 представлен результат удаления элемента по индексу — это проще, чем удаление по ссылке на объект, потому что вам не придется беспокоиться о поведении equals().

Работа метода removeAll() также зависит от equals(). Как подсказывает название, метод удаляет из List все объекты, входящие в List-аргумент.

Название метода set() выбрано неудачно, потому что оно совпадает с именем класса Set — возможно, лучше было бы назвать метод «replace», потому что он заменяет элемент с заданным индексом (первый аргумент) вторым аргументом.

В строке вывода 17 показано, что для List существует перегруженный метод addAll(), вставляющий новый список в середину исходного списка (вместо простого добавления в конец методом addAll(), унаследованным от Collection).

В строках 18–20 представлен результат вызова методов isEmpty() и clear(). Строки 22 и 23 демонстрируют, что любой объект Collection можно преобразовать в массив с использованием toArray().

Итераторы

У любого контейнера должен существовать механизм вставки и выборки элементов. В конце концов, контейнер предназначен именно для хранения объектов. При работе с List для вставки может использоваться метод add(), а для выборки — метод get() (впрочем, существуют и другие способы).

Если взглянуть на ситуацию с более высокого уровня, обнаруживается проблема: чтобы использовать контейнер в программе, необходимо знать его точный тип. Что, если вы начали использовать в программе контейнер List, а затем обнаружили, что в вашем случае будет удобнее применить тот же код к множеству (Set)? Или если вы хотите написать универсальный код, который не зависит от типа контейнера и может применяться к любому контейнеру?

С данной абстракцией хорошо согласуется концепция *итератора* (iterator). Итератор — это 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» (lightweight) объектом: его создание должно обходиться без заметных затрат ресурсов. Из-за этого итераторы часто имеют ограничения; например, Iterator в Java поддерживает перемещение только в одном направлении. Его возможности не так уж широки, но с его помощью можно сделать следующее:

- Запросить у контейнера итератор вызовом метода iterator(). Полученный итератор готов вернуть начальный элемент последовательности при первом вызове своего метода next().
- Получить следующий элемент последовательности вызовом метода next().
- Проверить, остались ли еще объекты в последовательности (метод hasNext()).
- Удалить из последовательности последний элемент, возвращенный итератором, методом remove().

Чтобы увидеть итератор в действии, мы снова воспользуемся иерархией Pets:

```
// 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 способ:
        for(Pet p : pets)
            System.out.print(p.id() + ":" + p + " ");
        System.out.println();
        // Итератор также способен 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]
*///:~
```

Мы видим, что с Iterator можно не беспокоиться о количестве элементов в последовательности. Проверка осуществляется методами hasNext() и next().

Если вы просто перебираете элементы списка в одном направлении, не пытаясь модифицировать его содержимое, «синтаксис foreach» обеспечивает более компактную запись.

Iterator удаляет последний элемент, полученный при помощи next(), поэтому перед вызовом remove() необходимо вызвать next().

Теперь рассмотрим задачу создания метода display(), не зависящего от типа 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) {
```

продолжение ⇩

```
        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
*///:~
```

В методе **display()** отсутствует информация о типе последовательности, и в этом проявляется истинная мощь итераторов: операция перемещения по последовательности отделяется от фактической структуры этой последовательности. Иногда говорят, что итераторы *унифицируют доступ к контейнерам.*

ListIterator

ListIterator — более мощная разновидность **Iterator**, поддерживаемая только классами **List**. Если **Iterator** поддерживает перемещение только вперед, **ListIterator** является двусторонним. Кроме того, он может выдавать индексы следующего и предыдущего элементов по отношению к текущей позиции итератора в списке и заменять последний посещенный элемент методом **set()**. Вызов **listIterator()** возвращает **ListIterator**, указывающий в начало **List**, а для создания итератора **ListIterator**, изначально установленного на элемент с индексом **n**, используется вызов **listIterator(n)**. Все перечисленные возможности продемонстрированы в следующем примере:

```
//: 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 списке, начиная с позиции 3 и далее.

LinkedList

LinkedList тоже реализует базовый интерфейс List, как и ArrayList, но выполняет некоторые операции (например, вставку и удаление в середине списка) более эффективно, чем ArrayList. И наоборот, операции произвольного доступа выполняются им с меньшей эффективностью.

Класс LinkedList также содержит методы, позволяющие использовать его в качестве стека, очереди (Queue) или двусторонней очереди (дека).

Некоторые из этих методов являются псевдонимами или модификациями для получения имен, более знакомых в контексте некоторого использования. Например, методы getFirst() и element() идентичны — они возвращают начало (первый элемент) списка без его удаления и выдают исключение NoSuchElementException для пустого списка. Метод peek() представляет собой небольшую модификацию этих двух методов: он возвращает null для пустого списка.

Метод addFirst() вставляет элемент в начало списка. Метод offer() делает то же, что add() и addLast() — он добавляет элемент в конец списка. Метод removeLast() удаляет и возвращает последний элемент списка.

Следующий пример демонстрирует схожие и различающиеся аспекты этих методов:

```
// 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());
               // Различие проявляется только для пустых списков:
               print("pets peek(): " + pets.peek());
               // Идентично, удаление и возврат первого элемента.
               print("pets remove(): " + pets.remove());
               print("pets removeFirst(): " + pets.removeFirst());
               // Различие проявляется только для пустых списков:
```

продолжение ⇩

```
            print("pets poll()  " + pets poll()).
            print(pets).
            pets addFirst(new Rat()).
            print("After addFirst()  " + pets).
            pets offer(Pets randomPet()).
            print("After offer()  " + pets).
            pets.add(Pets randomPet()).
            print("After add()  " + pets).
            pets addLast(new Hamster()).
            print("After 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]
After addFirst(): [Rat, Mutt, Pug]
After offer(): [Rat, Mutt, Pug, Cymric]
After add(): [Rat, Mutt, Pug, Cymric, Pug]
After addLast()· [Rat, Mutt, Pug, Cymric, Pug, Hamster]
pets removeLast(). Hamster
*///:~
```

Результат Pets.arrayList() передается конструктору LinkedList для заполнения. Присмотревшись к интерфейсу Queue, вы найдете в нем методы element(), offer(), peek(), poll() и remove(), добавленные в LinkedList для использования в реализации очереди (см. далее).

Стек

Стек часто называют контейнером, работающим по принципу «первым вошел, последним вышел» (LIFO). То есть элемент, последним занесенный в стек, будет первым, полученным при извлечении из стека.

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 лучше справляется с задачей:

```
//. net/mindview/util/Stack java
// Создание стека из списка 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(). }
} ///:~
```

Это простейший пример определения класса с использованием параметризации. Суффикс **<T>** после имени класса сообщает компилятору, что тип является *параметризованным* по типу T — при использовании класса на место T будет подставлен фактический тип. Фактически такое определение означает: «Мы определяем класс Stack для хранения объектов типа T». Stack реализуется на базе LinkedList, также предназначенного для хранения типа T. Обратите внимание: метод push() получает объект типа T, а методы peek() и pop() возвращают объект типа T. Метод peek() возвращает верхний элемент без извлечения из стека, а метод pop() удаляет и возвращает верхний элемент.

Простой пример использования нового 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 хотите использовать класс Stack в своем коде, вам придется либо полностью указать пакет, либо изменить имя класса при создании объекта; в противном случае, скорее всего, возникнет конфликт с классом Stack из пакета java.util. Пример использования имен пакетов при импортировании java.util.* в предыдущем примере:

```
//: 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·
```

продолжение ⇨

```
fleas has dog My
fleas has dog My
*///:~
```

В java.util нет общего интерфейса Stack — вероятно, из-за того, что имя было задействовано в исходной, неудачно спроектированной версии java. util.Stack для Java 1.0. Хотя класс java.util.Stack существует, LinkedList обеспечивает более качественную реализацию стека, и решение net.mindview.util. Stack является предпочтительным.

Множество

В множествах (Set) каждое значение может храниться только в одном экземпляре. Попытки добавить новый экземпляр эквивалентного объекта блокируются. Множества часто используются для проверки принадлежности, чтобы вы могли легко проверить, принадлежит ли объект заданному множеству. Следовательно, важнейшей операцией Set является операция поиска, поэтому на практике обычно выбирается реализация HashSet, оптимизированная для быстрого поиска.

Set имеет такой же интерфейс, что и Collection. В сущности, Set и является Collection, но обладает несколько иным поведением (кстати, идеальный пример использования наследования и полиморфизма: выражение разных концепций поведения). Пример использования HashSet 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 множество включаются десять тысяч случайных чисел от 0 до 29; естественно, числа должны многократно повторяться. Но при этом мы видим, что в результатах каждое число присутствует только в одном экземпляре.

Также обратите внимание на непредсказуемый порядок следования чисел в выводе. Это объясняется тем, что HashSet использует *хеширование* для ускорения выборки. Порядок, поддерживаемый HashSet, отличается от порядка TreeSet или LinkedHashSet, поскольку каждая реализация упорядочивает элементы по-своему. Если вы хотите, чтобы результат был отсортирован, воспользуйтесь 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 наиболее распространенных операций со множествами является проверка принадлежности методом **contains()**, но существуют и другие операции, которые напомнят вам диаграммы Венна из школьного курса:

```
// 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]
*///:~
```

Имена методов говорят за себя. Информацию о других методах **Set** можно найти в документации JDK.

Карта

Возможность отображения одних объектов на другие (ассоциация) чрезвычайно полезна при решении широкого класса задач программирования. В качестве примера рассмотрим программу, анализирующую качество распределения класса Java Random. В идеале класс Random должен выдавать абсолютно равномерное распределение чисел, но чтобы убедиться в этом, необходимо сгенерировать большое количество случайных чисел и подсчитать их количество в разных интервалах. Множества упрощают эту задачу: ключом в данном случае является число, сгенерированное при помощи Random, а значением — количество его вхождений:

```
//  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++) {
                       // Получение случайного числа от 0 до 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 main() механизм автоматической упаковки преобразует случайно сгенерированое целое число в ссылку на Integer, которая может использоваться с HashMap (контейнеры не могут использоваться для хранения примитивов). Метод get() возвращает null, если элемент отсутствует в контейнере (то есть если число было сгенерировано впервые. В противном случае метод get() возвращает значение Integer, связанное с ключом, и последнее увеличивается на 1 (автоматическая упаковка снова упрощает вычисления, но в действительности при этом выполняются преобразования к Integer и обратно).

Следующий пример демонстрирует поиск объектов Pet по строковому описанию String. Он также показывает, как проверить присутствие некоторого ключа или значения в 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
*///:~
```

Map, по аналогии с массивами и Collection, легко расширяются до нескольких измерений; достаточно создать Map со значениями типа Map (причем значениями *этих* Map могут быть другие контейнеры, и даже другие Map). Контейнеры легко комбинируются друг с другом, что позволяет быстро создавать сложные структуры данных. Например, если нам потребуется сохранить информацию о владельцах сразу нескольких домашних животных, для этого будет 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);
```

продолжение ⇩

```
            }
        }
} /* 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
*/// ~
```

Map может вернуть множество (Set) своих ключей, коллекцию (Collection) значений или множество (Set) всех пар «ключ-значение». Метод keySet() создает множество всех ключей, которое затем используется в синтаксисе foreach для перебора Map.

Очередь

Очередь обычно представляет собой контейнер, работающий по принципу «первым вошел, первым вышел» (*FIFO*). Иначе говоря, элементы заносятся в очередь с одного «конца» и извлекаются с другого в порядке их поступления. Очереди часто применяются для реализации надежной передачи объектов между разными областями программы.

Класс LinkedList содержит методы, поддерживающие поведение очереди, и реализует интерфейс Queue, поэтому LinkedList может использоваться в качестве реализации Queue. В следующем примере LinkedList повышается восходящим преобразованием до Queue:

```
//: holding/QueueDemo.java
// Восходящее преобразование LinkedList в Queue
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(), один из методов Queue, вставляет элемент в конец очереди, а если вставка невозможна — возвращает false. Методы peek() и element() возвращают начальный элемент *без его удаления из очереди*, но peek() для пустой очереди возвращает null, а element() выдает исключение NoSuchElementException. Методы poll() и remove() удаляют и возвращают начальный элемент очереди, но poll() для пустой очереди возвращает null, а remove() выдает NoSuchElementException.

Автоматическая упаковка преобразует результат int вызова nextInt() в объект Integer, необходимый для queue, а char c — в объект Character, необходимый для qc. Интерфейс Queue сужает доступ к методам LinkedList так, что доступными остаются только соответствующие методы и у пользователя остается меньше возможностей для вызова методов LinkedList (конечно, queue можно преобразовать обратно в LinkedList, но это создает дополнительные затруднения).

PriorityQueue

Принцип FIFO описывает наиболее типичную *организацию очереди*. Именно организация очереди определяет, какой элемент будет следующим для заданного состояния очереди. Правило FIFO означает, что следующим элементом будет тот, который дольше всего находится в очереди.

В *приоритетной очереди* следующим элементом считается элемент, обладающий наивысшим приоритетом. Например, в аэропорту пассажира, самолет которого скоро улетит, могут пропустить без очереди. В системах обработки сообщений некоторые сообщения могут быть важнее других и должны обрабатываться как можно скорее, независимо от момента их поступления. Параметризованный класс PriorityQueue был добавлен в Java SE5 как механизм автоматической реализации этого поведения.

При помещении объекта в PriorityQueue вызовом offer() объект сортируется в очереди. По умолчанию используется *естественный порядок* помещения объектов в очередь, однако вы можете изменить его, предоставив собственную реализацию Comparator. PriorityQueue гарантирует, что при вызове peek(), poll() или remove() вы получите элемент с наивысшим приоритетом.

Создание приоритетной очереди для встроенных типов — Integer, String, Character и т. д. — является делом тривиальным. В следующем примере используются те же значения, что и в предыдущем, но PriorityQueue выдает их в другом порядке:

```
//. holding/PriorityQueueDemo.java
import java util *;

public class PriorityQueueDemo {
     public static void main(String[] args) {
               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); // Автоматическая упаковка
               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 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
*///:~
```

Мы видим, что дубликаты разрешены, а меньшие значения обладают более высокими приоритетами. Чтобы показать, как изменить порядок элементов посредством передачи собственного объекта Comparator, при третьем вызове конструктора PriorityQueue<Integer> и втором — PriorityQueue<String> используется

Comparator с обратной сортировкой, полученный вызовом Collections.reverseOrder() (одно из новшеств Java SE5).

В последней части добавляется HashSet для уничтожения дубликатов Character — просто для того, чтобы пример был чуть более интересным.

Integer, String и Character изначально работают с PriorityQueue, потому что они обладают «встроенным» естественным упорядочением. Если вы хотите использовать собственный класс с PriorityQueue, включите дополнительную реализацию естественного упорядочения или предоставьте собственный объект Comparator.

Collection и Iterator

Collection — корневой интерфейс, описывающий общую функциональность всех последовательных контейнеров. Его можно рассматривать как «вторичный интерфейс», появившийся вследствие сходства между другими интерфейсами. Кроме того, класс java.util.AbstractCollection предоставляет реализацию Collection по умолчанию, поэтому вы можете создать новый подтип AbstractCollection без избыточного дублирования кода.

Один из доводов в пользу интерфейсов заключается в том, что они позволяют создавать более универсальный код. Код, написанный для интерфейса, а не для его реализации, может быть применен к более широкому кругу объектов. Таким образом, если я пишу метод, которому при вызове передается Collection, этот метод будет работать с любым типом, реализующим Collection, — следовательно, если новый класс реализует Collection, он будет совместим с моим методом. Однако интересно заметить, что стандартная библиотека C++ не имеет общего базового класса для своих контейнеров — вся общность контейнеров обеспечивается итераторами. Казалось бы, в Java будет логично последовать примеру C++ и выражать сходство между контейнерами при помощи итераторов, а не Collection. Тем не менее эти два подхода взаимосвязаны, потому что 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();
    }
    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);
```

продолжение ⇩

```
                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 как с объектами **Map**, так и с подтипами **Collection**; при этом как **Collection**, так и **Iterator** изолируют методы **display()** от знания конкретной реализации используемого контейнера.

В данном случае два решения примерно равноценны. Использование **Iterator** становится предпочтительным при реализации постороннего класса, для которого реализация интерфейса **Collection** затруднена или нежелательна. Например, если мы создаем реализацию **Collection** наследованием от класса, содержащего объекты **Pet**, нам придется реализовать все методы **Collection**, даже если они не будут использоваться в методе **display()**. Хотя проблема легко решается наследованием от **AbstractCollection**, вам все равно придется реализовать **iterator()** вместе с **size()**, чтобы предоставить методы, не реализованные **AbstractCollection**, но используемые 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*. В нашем примере реализовывать его не нужно, и в случае вызова он выдает исключение.

Из приведенного примера видно, что при реализации **Collection** вы также реализуете **iterator()**, а простая отдельная реализация **iterator()** требует чуть меньших усилий, чем наследование от **AbstractCollection**. Но, если класс уже наследует от другого класса, наследование еще и от **AbstractCollection** невозможно. В этом случае для реализации **Collection** придется реализовать все методы интерфейса, и тогда гораздо проще ограничиться наследованием и добавить возможность 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() { // Не реализован
                            throw new UnsupportedOperationException();
                    }
            };
    }
    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
*///:~
```

Создание Iterator обеспечивает минимальную логическую привязку между последовательностью и методом, использующим эту последовательность, а также налагает гораздо меньше ограничений на класс последовательности, реализующий Collection.

Синтаксис foreach и итераторы

До настоящего момента «синтаксис foreach» использовался в основном с массивами, но он также будет работать с любым объектом Collection. Некоторые примеры уже встречались нам при работе с ArrayList, но можно привести и более общее подтверждение:

```
//: holding/ForEachCollections.java
// Синтаксис foreach работает с любыми коллекциями
import java.util.*;

public class ForEachCollections {
     public static void main(String[] args) {
               Collection<String> cs = new LinkedList<String>();
               Collections.addAll(cs,
                         "Take the long way home".split(" "));
               for(String s : cs)
                         System.out.print("'" + s + "' ");
     }
} /* Output:
'Take' 'the' 'long' 'way' 'home'
*///:~
```

Поскольку cs является Collection, этот пример показывает, что поддержка foreach является характеристикой всех объектов Collection.

Работа этой конструкции объясняется тем, что в Java SE5 появился новый интерфейс Iterable, который содержит метод iterator() для создания Iterator, и именно интерфейс Iterable используется при переборе последовательности в синтаксисе foreach. Следовательно, создав любой класс, реализующий Iterable, вы сможете использовать его в синтаксисе foreach:

```
//: holding/IterableClass.java
// Anything Iterable works with 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() { // Not implemented
                                   throw new UnsupportedOperationException();
                         }
               };
     }
```

```
    public static void main(String[] args) {
            for(String s : new IterableClass())
                    System.out.print(s + " ");
    }
} /* Output:
And that is how we know the Earth to be banana-shaped.
*///:~
```

Метод iterator() возвращает экземпляр анонимной внутренней реализации Iterator<string>, последовательно доставляющей каждое слово в массиве. В main() мы видим, что IterableClass действительно работает в синтаксисе foreach.

В Java SE5 многие классы реализуют Iterable, прежде всего все классы Collection (но не Map). Например, следующий код выводит все переменные окружения (environment) 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 увидеть результат) *///:~
```

System.getenv() возвращает Map, entrySet() создает Set с элементами Map.Entry, а Set поддерживает Iterable и поэтому может использоваться в цикле foreach.

Синтаксис foreach работает с массивами и всем, что поддерживает Iterable, но это не означает, что массив автоматически поддерживает Iterable:

```
//: holding/ArrayIsNotIterable.java
import java.util.*;

public class ArrayIsNotIterable {
    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 к Iterable:
            test(Arrays.asList(strings));
    }
} /* Output:
1 2 3 A B C
*///:~
```

Попытка передачи массива в аргументе Iterable завершается неудачей. Автоматическое преобразование в Iterable не производится; его необходимо выполнять вручную.

Идиома «метод-адаптер»

Что делать, если у вас имеется существующий класс, реализующий Iterable, и вы хотите добавить новые способы использования этого класса в синтаксисе foreach? Допустим, вы хотите иметь возможность выбора между перебором списка слов в прямом или обратном направлении. Если просто воспользоваться наследованием от класса и переопределить метод iterator, то существующий метод будет заменен и никакого выбора не будет.

Одно из решений этой проблемы основано на использовании идиомы, которую я называю «методом-адаптером». Термин «адаптер» происходит от одноименного паттерна: вы должны предоставить интерфейс, необходимый для работы синтаксиса foreach. Если у вас имеется один интерфейс, а нужен другой, проблема решается написанием адаптера. В данном случае требуется *добавить* к стандартному «прямому» итератору обратный, так что переопределение исключено. Вместо этого мы добавим метод, создающий объект Iterable, который может использоваться в синтаксисе foreach. Как будет показано далее, это позволит нам предоставить несколько вариантов использования foreach:

```
//: holding/AdapterMethodIdiom.java
// Идиома "метод-адаптер" позволяет использовать foreach
// с дополнительными 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(" ")));
               // Получаем обычный итератор, полученный при помощи iterator():
               for(String s : ral)
                         System.out.print(s + " ");
               System.out.println();
```

```
            // Передаем выбранный нами Iterable
            for(String s · ral.reversed())
                    System.out.prınt(s + " "),
    }
} /* Output
To be or not to be
be to not or be To
*/// ~
```

Если просто поместить объект ral в синтаксис foreach, мы получим (стандартный) «прямой» итератор. Но если вызвать для объекта reversed(), поведение изменится.

Использовав этот прием, можно добавить в пример IterableClass.java два метода-адаптера:

```
//  holding/MultiIterableClass.java
// Adding several Adapter Methods.
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(),
                                    }
                            };
                    }
            },
    }
    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();
```

продолжение ⇩

```
            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
*///:~
```

Из выходных данных видно, что метод Collections.shuffle не изменяет исходный массив, а только переставляет ссылки в shuffled. Так происходит только потому, что метод randomized() создает для результата Arrays.asList() «обертку» в виде ArrayList. Если бы операция выполнялась непосредственно с объектом List, полученным от Arrays.asList(), то это привело бы к изменению нижележащего массива:

```
//: 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 перестановки. " + list1);
            Collections.shuffle(list1, rand);
            System.out.println("После перестановки: " + list1);
            System.out.println("Массив: " + Arrays.toString(ia));

            List<Integer> list2 = Arrays.asList(ia);
            System.out.println("До перестановки: " + list2);
            Collections.shuffle(list2, rand);
            System.out.println("После перестановки: " + list2);
            System.out.println("Массив: " + Arrays.toString(ia));
    }
} /* Output:
До перестановки: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
После перестановки: [4, 6, 3, 1, 8, 7, 2, 5, 10, 9]
Массив: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
До перестановки: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
После перестановки: [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(), а последний создает объект ArrayList, ссылающийся на элементы ia. Перестановка этих ссылок не изменяет массива. Но, если мы используем результат Arrays.asList(ia) напрямую, перестановка изменит порядок ia. Важно учитывать, что Arrays.asList() создает объект List, который использует нижележащий массив в качестве своей физической реализации. Если с этим объектом List выполняются какие-либо изменяющие операции, но вы не хотите изменения исходного массива, создайте копию в другом контейнере.

Резюме

В Java существует несколько способов хранения объектов:

- В массивах объектам назначаются числовые индексы. Массив содержит объекты заранее известного типа, поэтому преобразование типа при выборке объекта не требуется. Массив может быть многомерным и может использоваться для хранения примитивных типов. Тем не менее изменить размер созданного массива невозможно.
- В Collection хранятся отдельные элементы, а в Map — пары ассоциированных элементов. Механизм параметризации позволяет задать тип объектов, хранимых в контейнере, поэтому поместить в контейнер объект неверного типа невозможно, и элементы не нуждаются в преобразовании типа при выборке. И Collection, и Map автоматически изменяются в размерах при добавлении новых элементов. В контейнерах не могут храниться примитивы, но механизм автоматической упаковки автоматически создает объектные «обертки», сохраняемые в контейнере.
- В контейнере List, как и в массиве, объектам назначаются числовые индексы — таким образом, массивы и List являются упорядоченными контейнерами.
- Используйте ArrayList при частом использовании произвольного доступа к элементам или LinkedList при частом выполнении операций вставки и удаления в середине списка.
- Поведение очередей и стеков обеспечивается контейнером LinkedList.
- Контейнер Map связывает с объектом не целочисленный индекс, а *другой объект*. Контейнеры HashMap оптимизированы для быстрого доступа, а контейнер TreeMap хранит ключи в отсортированном порядке, но уступает по скорости HashMap. В контейнере LinkedHashMap элементы хранятся в порядке вставки, но хеширование обеспечивает быстрый доступ.
- В контейнере Set каждый объект может храниться только в одном экземпляре. Контейнер HashSet обеспечивает максимальную скорость поиска, а в TreeSet элементы хранятся в отсортированном порядке. В контейнере LinkedHashSet элементы хранятся в порядке вставки.
- Использовать старые классы Vector, Hashtable и Stack в новом коде не нужно.

Контейнеры Java — необходимый инструмент, которым вы будете постоянно пользоваться в своей повседневной работе; благодаря им ваш код станет более простым, мощным и эффективным. Возможно, на освоение некоторых аспектов контейнеров потребуется время, но вы быстро привыкнете к классам этой библиотеки и начнете использовать их.

12 Обработка ошибок и исключения

Один из основополагающих принципов философии Java состоит в том, что «плохо написанная программа не должна запускаться».

В идеале ошибки должны обнаруживаться во время компиляции, перед запуском программы. Однако не все ошибки удается выявить в это время. Остальные проблемы приходится решать во время работы программы, с помощью механизма, который позволяет источнику ошибки передать необходимую информацию о ней получателю — а последний справляется с возникшими трудностями.

Усовершенствованная система восстановления после ошибок входит в число важнейших факторов, влияющих на надежность кода. Восстановление особенно важно в языке Java, на котором часто пишутся программные компоненты, используемые другими сторонами. *Надежная система может быть построена только из надежных компонентов.* Унифицированная модель передачи информации об ошибках в Java позволяет компонентам передавать информацию о возникших проблемах в клиентский код.

Механизм исключений значительно упрощает создание больших надежных программ, уменьшает объем необходимого кода и повышает уверенность в том, что в приложении не будет необработанной ошибки. Освоить работу с исключениями несложно, и это одна из языковых возможностей, способных принести немедленную и значительную выгоду в ваших проектах. В этой главе будет показано, как правильно организовать обработку исключений в программе, а также как сгенерировать собственное исключение, если какой-то из ваших методов сталкивается с непредусмотренными трудностями.

Основные исключения

Исключительной ситуацией называется проблема, из-за которой нормальное продолжение работы метода или части программы, выполняющихся в данный момент, становится невозможным. Важно отличать исключительную ситуацию

от «обычных» ошибок, когда в текущем контексте имеется достаточно информации для преодоления затруднений. В исключительной ситуации обработать исключение в *текущем контексте* невозможно, потому что вы не располагаете необходимой информацией. Остается единственный выход — покинуть текущий контекст и передать проблему на более высокий уровень. Именно это и происходит при выдаче исключения.

Простейшим примером является деление. Потенциальное деление на нуль может быть выявлено проверкой соответствующего условия. Но что делать, если знаменатель оказался нулем? Возможно, в контексте текущей задачи известно, как следует поступить с нулевым знаменателем. Но, если нулевой знаменатель возник неожиданно, деление в принципе невозможно, и тогда необходимо возбудить исключение, а не продолжать исполнение программы.

При возбуждении исключения 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 места программы для передачи исключения. Таким местом является *обработчик исключений*, который пытается решить возникшую проблему так, чтобы программа могла снова попытаться выполнить проблемную операцию или просто продолжила свое выполнение.

В качестве простого примера выдачи исключения представьте ссылку на объект t. Возможно, полученная вами ссылка не была инициализирована; стоит проверить это обстоятельство, прежде чем вызывать методы с использованием этой ссылки. Чтобы передать информацию об ошибке на более высокий уровень, создайте объект, представляющий передаваемую информацию, и «запустите» его из текущего контекста. Тем самым вы *возбудите исключение*. Вот как это выглядит:

```
if(t == null)
     throw new NullPointerException( );
```

Вырабатывается исключение, которое позволяет вам — в текущем контексте — переложить с себя ответственность, не задумываясь о будущем. Ошибка, словно по волшебству, обрабатывается где-то в другом месте (вскоре мы узнаем, где именно).

Один из основополагающих аспектов исключений состоит в том, что при возникновении нежелательных ситуаций выполнение программы не продолжается по обычному пути. Исключения позволяют вам (в крайнем случае) остановить программу и сообщить о возникших трудностях или (в идеале) разобраться с проблемой и вернуть программу в стабильное состояние.

Аргументы исключения

Исключения, как и любые объекты Java, создаются в куче оператором new, который выделяет память и вызывает конструктор. У всех стандартных исключений существует два конструктора: стандартный (по умолчанию) и другой,

со строковым аргументом, в котором можно разместить подходящую информацию об исключении:

```
throw new NullPointerException("t = null");
```

Переданная строка потом может быть извлечена различными способами, о чем будет рассказано позже.

Ключевое слово throw влечет за собой ряд довольно интересных действий. Как правило, сначала new используется для создания объекта, представляющего условие происшедшей ошибки. Ссылка на указанный объект передается команде throw. Фактически этот объект «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 (впрочем, с этой аналогией не стоит заходить слишком далеко). Возбуждение исключений также позволяет выходить из простых блоков видимости. В обоих случаях возвращается объект исключения и происходит выход из текущего метода или блока.

Но все сходство с обычным возвратом из метода на этом заканчивается, поскольку при возврате из исключения вы попадаете совсем не туда, куда попали бы при нормальном вызове метода. (Обработчик исключения может находиться очень «далеко» — на расстоянии нескольких уровней в стеке вызова — от метода, где возникла исключительная ситуация.)

Вообще говоря, можно возбудить любой тип исключений, происходящих от объекта Throwable (корневой класс иерархии исключений). Обычно для разных типов ошибок возбуждаются разные типы исключений. Информация о случившейся ошибке как содержится внутри объекта исключения, так и указывается косвенно в самом типе этого объекта, чтобы кто-то на более высоком уровне сумел выяснить, как поступить с исключением. (Нередко именно тип объекта исключения является единственно доступной информацией об ошибке, в то время как внутри объекта никакой полезной информации нет.)

Перехват исключений

Чтобы увидеть, как перехватываются ошибки, сначала следует усвоить понятие *защищенной секции* — той части программы, в которой могут произойти исключения и за которой следует специальный блок, отвечающий за обработку этих исключений.

Блок try

Если вы «находитесь» внутри метода и инициируете исключение (или это делает другой вызванный метод), этот метод завершит работу при возникновении исключения. Но если вы не хотите, чтобы оператор throw завершил работу метода, разместите в методе специальный блок для перехвата исключения — так называемый *блок try*. Этот блок представляет собой простую область действия, которой предшествует ключевое слово try:

```
try {
     // Фрагмент, способный возбуждать исключения
}
```

Если бы не обработка исключений, для тщательной проверки ошибок вам пришлось бы добавить к вызову каждого метода дополнительный код для проверки ошибок — даже при многократном вызове одного метода. С обработкой исключений весь код размещается в блоке 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 в конечном итоге должно быть где-то обработано. Этим местом является *обработчик исключений*, который создается для каждого исключения, которое вы хотите перехватить. Обработчики исключений размещаются прямо за блоком try и обозначаются ключевым словом catch:

```
try {
     // Часть программы, способная возбуждать исключения
} catch(Type1 id1) {
     // Обработка исключения Type1
} catch(Type2 id2) {
     // Обработка исключения Type2
} catch(Type3 id3) {
     // Обработка исключения Type3
}
// и т  д.
```

Каждое предложение catch (обработчик исключения) напоминает маленький метод, принимающий один и только один аргумент определенного типа. Идентификатор (id1, id2 и т. д.) может использоваться внутри обработчика точно так же, как и метод распоряжается своими аргументами. Иногда этот идентификатор остается невостребованным, так как тип исключения дает достаточно информации для его обработки, но тем не менее присутствует он всегда.

Обработчики всегда следуют прямо за блоком try. При возникновении исключения механизм обработки исключений ищет первый из обработчиков исключений, аргумент которого соответствует текущему типу исключения. После этого он передает управление в блок catch, и таким образом исключение считается обработанным. После выполнения предложения catch поиск обработчиков исключения прекращается. Выполняется только одна секция catch, соответствующая типу исключения; в этом отношении обработка исключений отличается от команды switch, где нужно дописывать break после каждого case, чтобы предотвратить исполнение всех прочих case.

Заметьте также, что внутри блока try могут вызываться различные методы, способные породить одинаковые типы исключения, но обработчик понадобится всего один.

Прерывание в сравнении с возобновлением

В теории обработки исключений имеется две основные модели. Модель *прерывания* (которое используется в Java и C++) предполагает, что ошибка настолько серьезна, что при возникновении исключения продолжить исполнение невозможно. Кто бы ни возбудил исключение, сам факт его выдачи означает, что исправить ситуацию «на месте» невозможно и возвращать управление обратно *не нужно*.

Альтернативная модель называется *возобновлением*. Она подразумевает, что обработчик ошибок сделает что-то для исправления ситуации, после чего предпринимается попытка повторить неудавшуюся операцию в надежде на успешный исход. В таком случае исключение больше напоминает вызов метода — чтобы применить модель возобновления в Java, вам придется пойти именно по этому пути (то есть не возбуждать исключение, а вызвать метод, способный решить проблему). Также можно создать блок try внутри цикла while, который станет снова и снова обращаться к этому блоку, 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. Хотя 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 усложняет написание и поддержку программ, особенно для больших систем, где исключения могут быть сгенерированы во многих различных местах.

Создание собственных исключений

Ваш выбор не ограничивается использованием уже существующих в Java исключений. Иерархия исключений JDK не может предусмотреть все возможные ошибки, поэтому вы вправе создавать собственные типы исключений для обозначения специфических ошибок вашей программы.

Для создания собственного класса исключения вам придется определить его производным от уже существующего типа — желательно наиболее близкого к вашей ситуации (хоть это и не всегда возможно). В простейшем случае создается класс с конструктором по умолчанию:

```
//· exceptions/InheritingExceptions.java
// Создание собственного исключения

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 вызывает конструктор базового класса. Конечно, в этом случае вы лишаетесь 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 исключений, похожих на созданное выше, будет достаточно.

В примере результаты работы выводятся на консоль. Впрочем, их также можно направить в стандартный поток ошибок, что достигается использованием класса **System.err**. Обычно это правильнее, чем выводить в поток **System.out**, который может быть перенаправлен. При выводе результатов с помощью **System.err** пользователь заметит их скорее, чем при выводе в **System.out**.

Также можно создать класс исключения с конструктором, получающим аргумент **String**:

```
//: 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.err);
            }
            try {
                    g();
            } catch(MyException e) {
                    e.printStackTrace(System.err);
            }
    }
} /* Output:
Возбуждаем MyException из f()
```

продолжение ⇩

```
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)
 ///·~
```

Изменения незначительны — появилось два конструктора, определяющие 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** (базового для **Exception**). Этот метод выводит информацию о последовательности вызовов, которая привела к точке возникновения исключения. В нашем примере информация направляется в **System.out**, но вызов по умолчанию направляет информацию в стандартный поток ошибок:

```
e.printStackTrace(),
```

Регистрация исключений

Вспомогательное пространство имен **java.util.logging** позволяет зарегистрировать информацию об исключениях в журнале. Базовые средства регистрации достаточно просты:

```
//  exceptions/LoggingExceptions.java
// Регистрация исключений с использованием 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**, ассоциируемый с аргументом **String** (обычно имя пакета и класса, к которому относятся ошибки); объект передает свой вывод в **System.err**. Простейший способ записи информации в **Logger** заключается в вызове метода, соответствующего уровню ошибки; в нашем примере используется метод **severe()**. Нам хотелось бы создать **String** для регистрируемого сообщения из результатов трассировки стека, но метод **printStackTrace()** по умолчанию не создает **String**. Для получения **String** необходимо использовать перегруженную версию **printStackTrace()** с аргументом **java.io.PrintWriter** (за подробными объяснениями обращайтесь к главе «Ввод/вывод»). Если передать конструктору **PrintWriter** объект **java.io. StringWriter**, для получения вывода в формате **String** достаточно вызвать **toString()**.

Подход **LoggingException** чрезвычайно удобен (вся инфраструктура регистрации встроена в само исключение, и все работает автоматически без вмешательства со стороны клиента), однако на практике чаще применяется перехват и регистрация «сторонних» исключений, поэтому сообщение должно генерироваться в обработчике исключения:

```
//: 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
```

продолжение ⇩

```
        at LoggingExceptions2 main(LoggingExceptions2 java:16)
*///.~
```

На этом процесс создания собственных исключений не заканчивается — исключение можно снабдить дополнительными конструкторами и элементами:

```
//: exceptions/ExtraFeatures.java
// Дальнейшее расширение 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("MyException2 в f()");
              throw new MyException2();
     }
     public static void g() throws MyException2 {
              System out.println("MyException2 в g()");
              throw new MyException2("Возбуждено в g()");
     }
     public static void h() throws MyException2 {
              System out.println("MyException2 в h()");
              throw new MyException2("Возбуждено 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:
MyException2 в f()
MyException2: Подробное сообщение: 0 null
              at ExtraFeatures.f(ExtraFeatures.java·22)
```

```
                at ExtraFeatures.main(ExtraFeatures.java.34)
MyException2 в g()
MyException2: Подробное сообщение: 0 Возбуждено в g()
                at ExtraFeatures.g(ExtraFeatures.java:26)
                at ExtraFeatures.main(ExtraFeatures.java:39)
MyException2: Подробное сообщение: 47 Возбуждено в h()
                at ExtraFeatures.h(ExtraFeatures.java:30)
                at ExtraFeatures.main(ExtraFeatures.java:44)
e.val() = 47
*///:~
```

Было добавлено поле данных x вместе с методом, считывающим его значение, а также дополнительный конструктор для инициализации x. Переопределенный метод **Throwable.getMessage()** выводит более содержательную информацию об исключении. Метод **getMessage()** для классов исключений — аналог **toString()** в обычных классах.

Так как исключение является просто видом объекта, расширение возможностей классов исключений можно продолжить. Однако следует помнить, что все эти программисты, использующие ваши библиотеки, могут попросту проигнорировать все «украшения» — нередко программисты ограничиваются проверкой типа исключения (как чаще всего бывает со стандартными исключениями Java).

Спецификации исключений

В языке Java желательно сообщать программисту, вызывающему ваш метод, об исключениях, которые данный метод способен возбуждать. Пользователь, вызывающий метод, сможет написать весь необходимый код для перехвата возможных исключений. Конечно, когда доступен исходный код, программист-клиент может пролистать его в поиске предложений **throw**, но библиотеки не всегда поставляются с исходными текстами. Чтобы библиотека не превращалась в «черный ящик», в Java добавили синтаксис (*обязательный* для использования), при помощи которого вы сообщаете клиенту об исключениях, возбуждаемых методом, чтобы он сумел правильно обработать их. Этот синтаксис называется *спецификацией исключений* (exception specification), входит в объявление метода и следует сразу за списком аргументов.

Спецификация исключений состоит из ключевого слова **throws**, за которым перечисляются все возможные типы исключений. Примерное определение метода будет выглядеть так:

```
void f() throws TooBig, TooSmall, DivZero { //...
```

Однако запись

```
void f() { // ...
```

означает, что метод не вырабатывает исключений. (*Кроме* исключений, производных от **RuntimeException**, которые могут быть возбуждены практически в любом месте — об этом еще будет сказано.)

Обойти спецификацию исключений невозможно — если ваш метод возбуждает исключения и не обрабатывает их, компилятор предложит либо обработать исключение, либо включить его в спецификацию. Жестким контролем за соблюдением правил сверху донизу Java гарантирует правильность использования механизма исключений во время компиляции программы.

Впрочем, «обмануть» компилятор все же можно: вы вправе объявить о возбуждении исключения, которого на самом деле нет. Компилятор верит вам на слово и заставляет пользователей метода поступать так, как будто им и в самом деле необходимо перехватывать исключение. Таким образом можно «зарезервировать» исключение на будущее и уже потом возбуждать его, не изменяя описания готовой программы. Такая возможность может пригодиться и для создания абстрактных базовых классов и интерфейсов, в производных классах которых может возникнуть необходимость в возбуждении исключений.

Исключения, которые проверяются и навязываются еще на этапе компиляции программы, называют *контролируемыми* (checked).

Перехват произвольных исключений

Можно создать универсальный обработчик, перехватывающий любые типы исключения. Осуществляется это перехватом базового класса всех исключений Exception (существуют и другие базовые типы исключений, но класс Exception является базовым практически для всех программных исключительных ситуаций):

```
catch(Exception e) {
     System.out.println("Перехвачено исключение");
}
```

Подобная конструкция не упустит ни одного исключения, поэтому ее следует размещать в самом конце списка обработчиков, во избежание блокировки следующих за ней обработчиков исключений.

Поскольку класс Exception является базовым для всех классов исключений, интересных программисту, сам он не предоставит никакой полезной информации об исключительной ситуации, но можно вызвать методы из *его* базового типа Throwable:

- String getMessage(), String getLocalizedMessage() возвращают подробное сообщение об ошибке (или сообщение, локализованное для текущего контекста);
- String toString() возвращает короткое описание объекта Throwable, включая подробное сообщение, если оно присутствует;
- void printStackTrace(), void printStackTrace(PrintStream), void printStackTrace(java.io.PrintWriter) выводят информацию об объекте Throwable и трассировку стека вызовов для этого объекта. Трассировка стека вызовов показывает последовательность вызова методов, которая привела к точке возникновения исключения. Первый вариант отправляет информацию в стандартный поток ошибок, второй и третий — в поток по

вашему выбору (в главе «Ввод/вывод» вы поймете, почему типов потоков два);

- Throwable fillInStackTrace() записывает в объект Throwable информацию о текущем состоянии стека. Метод используется при повторном возбуждении ошибок или исключений.

Вдобавок в вашем распоряжении находятся методы типа Object, базового для Throwable (и для всех остальных классов). При использовании исключений может пригодиться метод getClass(), который возвращает информацию о классе объекта. Эта информация заключена в объекте типа Class. Например, вы можете узнать имя класса вместе с информацией о пакете методом getName() или получить только имя класса методом 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)
*///:~
```

Как видите, методы последовательно расширяют объем выдаваемой информации — всякий последующий фактически является продолжением предыдущего.

Трассировка стека

Информацию, предоставляемую методом printStackTrace(), также можно получить напрямую вызовом getStackTrace(). Метод возвращает массив элементов трассировки, каждый из которых представляет один кадр стека. Нулевой элемент представляет вершину стека, то есть последний вызванный метод последовательности (точка, в которой был создан и инициирован объект Throwable).

Соответственно, последний элемент массива представляет «низ» стека, то есть первый вызванный элемент последовательности. Рассмотрим простой пример:

```
//: exceptions/WhoCalled.java
// Программный доступ к данным трассировки стека

public class WhoCalled {
     static void f() {
               // Выдача исключения для заполнения трассировочных данных
               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
*///:~
```

Повторное возбуждение исключения

В некоторых ситуациях требуется заново возбудить уже перехваченное исключение; чаще всего это происходит при использовании **Exception** для перехвата всех исключений. Так как ссылка на текущее исключение уже имеется, вы попросту возбуждаете исключение по этой ссылке:

```
catch(Exception e) {
     System.out.println("Было возбуждено исключение");
     throw e;
}
```

При повторном возбуждении исключение передается в распоряжение обработчика более высокого уровня. Все остальные предложения **catch** текущего блока **try** игнорируются. Вся информация из объекта, представляющего исключение, сохраняется, и обработчик более высокого уровня, перехватывающий подобные исключения, сможет ее извлечь.

Если вы просто заново возбуждаете исключение, информация о нем, выводимая методом printStackTrace(), будет по-прежнему относиться к месту возникновения исключения, но не к месту его повторного возбуждения. Если вам понадобится использовать новую трассировку стека, вызовите метод fillInStackTrace(), который возвращает исключение (объект Throwable), созданное на базе старого с помещением туда текущей информации о стеке. Вот как это выглядит:

```
//  exceptions/Rethrowing.java
// Демонстрация метода 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()
java lang Exception  thrown from f()
        at Rethrowing.f(Rethrowing.java 7)
        at Rethrowing g(Rethrowing java:11)
        at Rethrowing.main(Rethrowing.java:29)
main. printStackTrace()
java.lang.Exception: thrown from f()
```

продолжение ⇩

```
        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. thrown from f()
        at Rethrowing.f(Rethrowing.java·7)
        at Rethrowing.h(Rethrowing.java:20)
        at Rethrowing main(Rethrowing.java·35)
main· printStackTrace()
java lang.Exception: thrown from f()
        at Rethrowing.h(Rethrowing.java·24)
        at Rethrowing.main(Rethrowing.java:35)
*///:~
```

Строка с вызовом fillInStackTrace() становится новой точкой выдачи исключения.

Выдаваемое исключение также может отличаться от исходного. В этом случае эффект получается примерно таким же, как при использовании fillInStackTrace() — информация о месте зарождения исключения теряется, а остается информация, относящаяся к новой команде throw.

```
//: exceptions/RethrowNew java
// Повторное возбуждение объекта,
// отличающегося от первоначального

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(
                                         "Во внутреннем блоке try,
e printStackTrace()"),  .
                                e.printStackTrace(System.out);
                                throw new TwoException("из внутреннего блока try"),
                       }
              } catch(TwoException e) {
                       System.out.println(
                                "Во внешнем блоке try, e.printStackTrace()"),
                       e.printStackTrace(System.out),
              }
     }
} /* Output:
```

```
создание исключения в f()
Во внутреннем блоке try, e.printStackTrace()
OneException· thrown from f()
              at RethrowNew.f(RethrowNew.java·15)
              at RethrowNew.main(RethrowNew.java·20)
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 поступило из внутреннего блока try, но не из метода f().

Вам никогда не придется заботиться об удалении предыдущих исключений, и исключений вообще. Все они являются объектами, созданными в общей куче оператором new, и сборщик мусора уничтожает их автоматически.

Цепочки исключений

Зачастую необходимо перехватить одно исключение и возбудить следующее, не потеряв 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 конструкторам всех подклассов Throwable может передаваться объект-*причина* (cause). Предполагается, что *причиной* является изначальное исключение и передача ее в новый объект обеспечивает трассировку стека вплоть до самого его начала, хотя при этом создается и возбуждается 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. Для организации цепочек из других типов исключений придется использовать метод initCause(), а не конструктор.

Следующий пример демонстрирует динамическое добавление полей в объект DynamicFields во время работы программы:

```
//. exceptions/DynamicFields.java
// Динамическое добавление полей в класс.
// Пример использования цепочки 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(),
```

продолжение ⇩

```
        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. Добавим 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
                // класса 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("число", 47);
                    df.setField("число2", 48);
                    print(df);
                    df.setField("d", "Новое значение d"),
                    df setField("число3", 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
число: 47
число2: 48

df· d· Новое значение d
число: 47
число:2 48
число3· 11

Значение 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)      )
            ..
*///·~
```

Каждый объект DynamicFields содержит массив пар Object-Object. Первый объект содержит идентификатор поля (String), а второй объект — значение поля, которое может быть любого типа, кроме неупакованных примитивов. При создании объекта необходимо оценить примерное количество полей. Метод setField() либо находит уже существующее поле с заданным именем, либо создает новое поле и сохраняет значение. Когда пространство для полей заканчивается,

метод наращивает его, создавая массив размером на единицу больше и копируя в него старые элементы. При попытке размещения пустой ссылки null метод инициирует исключение DynamicFieldsException, создавая объект нужного типа и передавая методу initCause() в качестве причины исключение NullPointerException.

Для возвращаемого значения метод setField() использует старое значение поля, получая его методом getField(), который может возбудить исключение NoSuchFieldException. Если метод getField() вызывает программист-клиент, то он ответственен за обработку возможного исключения NoSuchFieldException, однако, если последнее возникает в методе setField(), это является ошибкой программы; соответственно, полученное исключение преобразуется в исключение RuntimeException с помощью конструктора, принимающего аргумент-причину.

Для создания результата toString() использует объект StringBuilder. Этот класс будет подробно рассмотрен при описании работы со строками.

Стандартные исключения Java

Класс Java Throwable описывает все объекты, которые могут возбуждаться как исключения. Существует две основные разновидности объектов Throwable (то есть ветви наследования). Тип Error представляет системные ошибки и ошибки времени компиляции, которые обычно не перехватываются (кроме нескольких особых случаев). Тип Exception может быть возбужден из любого метода стандартной библиотеки классов Java или пользовательского метода в случае неполадок при исполнении программы. Таким образом, для программистов интерес представляет прежде всего тип Exception.

Лучший способ получить представление об исключениях — просмотреть документацию JDK. Стоит сделать это хотя бы раз, чтобы получить представление о различных видах исключений, но вскоре вы убедитесь в том, что наиболее принципиальным различием между разными исключениями являются их имена. К тому же количество исключений в Java постоянно растет, 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.

Основной принцип заключается в том, что имя исключения относительно полно объясняет суть возникшей проблемы. Не все исключения определены в пакете java.lang, некоторые из них созданы для поддержки других библиотек, таких как util, net и io, как 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();
```

Только представьте, как ужасно было бы проверять таким образом *каждую* ссылку, переданную вашему методу. К счастью, делать это не нужно — такая проверка автоматически выполняется во время исполнения Java-программы, и при попытке использования null-ссылок автоматически возбуждается NullPointerException. Таким образом, использованная в примере конструкция избыточна.

Есть целая группа исключений, принадлежащих к этой категории. Они всегда возбуждаются в Java автоматически, и вам не придется включать их в спецификацию исключений. Все они унаследованы от одного базового класса RuntimeException, что дает нам идеальный пример наследования: семейство классов, имеющих общие характеристики и поведение. Вам также не придется создавать спецификацию исключений, указывающую на возбуждение методом RuntimeException (или любого унаследованного от него исключения), 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. Проверка RuntimeException привела бы к излишнему загромождению программы. И хотя вам обычно не требуется перехватывать RuntimeException, возможно, вы посчитаете нужным возбуждать некоторые из них в своих собственных библиотеках программ.

Что же происходит, когда подобные исключения не перехватываются? Так как компилятор не заставляет перечислять такие исключения в спецификациях, можно предположить, что исключение RuntimeException проникнет прямо в метод main(), и не будет перехвачено. Чтобы увидеть все в действии, испытайте следующий пример:

```
//: exceptions/NeverCaught.java
// Игнорирование RuntimeExceptions.
// {ThrowsException}

public class NeverCaught {
     static void f() {
              throw new RuntimeException("Из f()");
     }
     static void g() {
              f();
     }
     public static void main(String[] args) {
              g();
     }
} ///.~
```

Можно сразу заметить, что RuntimeException (и все от него унаследованное) является специальным случаем, так как компилятор не требует для него спецификации исключения. Выходные данные выводятся в System.err:

```
Exception in thread "main" java.lang.RuntimeException: Из f()
        at NeverCaught.f(NeverCaught.java:7)
        at NeverCaught.g(NeverCaught.java:10)
        at NeverCaught.main(NeverCaught.java:13)
```

Мы приходим к ответу на поставленный вопрос: если RuntimeException добирается до main() без перехвата, то работа программы завершается с вызовом метода printStackTrace().

Помните, что только исключения типа RuntimeException (и производных классов) могут игнорироваться во время написания текста программы, в то время как остальные действия компилятор осуществляет в обязательном порядке. Это объясняется тем, что RuntimeException является следствием ошибки программиста, например:

- ошибки, которую невозможно предвидеть (к примеру, получение null-ссылки в вашем методе, переданной снаружи);
- ошибки, которую вы как программист должны были проверить в вашей программе (подобной ArrayIndexOutOfBoundsException, с проверкой размера массива). Ошибки первого вида часто становятся причиной ошибок второго вида.

В подобных ситуациях исключения оказывают неоценимую помощь в отладочном процессе.

Назвать механизм исключений Java узкоспециализированным инструментом было бы неверно. Да, он помогает справиться с досадными ошибками на стадии исполнения программы, которые невозможно предусмотреть заранее, но при этом данный механизм помогает выявлять многие ошибки программирования, выходящие за пределы возможностей компилятора.

Завершение с помощью finally

Часто встречается ситуация, когда некоторая часть программы должна выполняться независимо от того, было или нет возбуждено исключение внутри блока try. Обычно это имеет отношение к операциям, не связанным с освобождением памяти (так как это входит в обязанности сборщика мусора). Для достижения желаемой цели необходимо разместить блок finally[1] после всех обработчиков исключений. Таким образом, полная конструкция обработки исключения выглядит так:

```
try {
     // Защищенная секция: рискованные операции,
     // которые могут породить исключения A, B, или C
} catch(A a1) {
     // Обработчик для ситуации A
} catch(B b1) {
     // Обработчик для ситуации B
} catch(C c1) {
     // Обработчик для ситуации C
} finally {
     // Действия, производимые в любом случае
}
```

Чтобы продемонстрировать, что блок finally выполняется всегда, рассмотрим следующую программу:

```
//: exceptions/FinallyWorks.java
// Блок finally выполняется всегда
```

[1] Механизм обработки исключений в языке C++ не имеет аналога finally, поскольку опирается на деструкторы в такого рода действиях.

```
class ThreeException extends Exception {}

public class FinallyWorks {
     static int count = 0;
     public static void main(String[] args) {
              while(true) {
                       try {
                                // Операция постфиксного приращения, в первый раз 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; // вне цикла "while"
                       }
              }
     }
} /* Output:
ThreeException
В блоке finally
Нет исключения
В блоке finally
*///~
```

Результат работы программы показывает, что вне зависимости от того, было ли возбуждено исключение, предложение finally выполняется всегда.

Данный пример также подсказывает, как справиться с тем фактом, что Java не позволяет вернуться к месту возникновения исключения, о чем говорилось ранее. Если расположить блок try в цикле, можно также определить условие, на основании которого будет решено, должна ли программа продолжаться. Также можно добавить статический счетчик или иной механизм для проверки нескольких разных решений, прежде чем отказаться от попыток восстановления. Это один из способов обеспечения повышенной отказоустойчивости программ.

Для чего нужен блок finally?

В языках без сборки мусора *и без* автоматических вызовов деструкторов[1] блок finally гарантирует освобождение ресурсов и памяти независимо от того, что случилось в блоке try. В Java существует сборщик мусора, поэтому с освобождением памяти проблем не бывает. Также нет необходимости вызывать деструкторы, их просто нет. Когда же нужно использовать finally в Java?

Блок finally необходим тогда, когда в исходное состояние вам необходимо вернуть что-то *другое*, а не память. Это может быть, например, открытый файл или сетевое подключение, часть изображения на экране или даже какой-то физический переключатель, вроде смоделированного в следующем примере:

[1] *Деструктор* — специальная функция, вызываемая при завершении работы с объектом. Всегда точно известно, где и когда вызывается деструктор. В языке C# (который гораздо больше схож с Java) реализовано автоматическое уничтожение объектов.

```
//: exceptions/Switch.java
import static net.mindview.util.Print.*;

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 ? "on" : "off"; }
} ///:~

//: exceptions/OnOffException1.java
public class OnOffException1 extends Exception {} ///:~

//: exceptions/OnOffException2.java
public class OnOffException2 extends Exception {} /// ~

//: exceptions/OnOffSwitch.java
// Для чего нужно finally?

public class OnOffSwitch {
     private static Switch sw = new Switch();
     static void f()
     throws OnOffException1, OnOffException2 {}
     public static void main(String[] args) {
               try {
                         sw.on();
                         // Код, способный возбуждать исключения...
                         f();
                         sw.off();
               } catch(OnOffException1 e) {
                         System.out.println("OnOffException1");
                         sw.off();
               } catch(OnOffException2 e) {
                         System.out.println("OnOffException2");
                         sw.off();
               }
     }
} /* Output:
on
off
*///:~
```

Наша цель — убедиться в том, что переключатель был выключен по завершении метода **main()**, поэтому в конце блока **try** и в конце каждого обработчика исключения помещается вызов **sw.off()**. Однако в программе может возникнуть неперехватываемое исключение, и тогда вызов **sw.off()** будет пропущен. Однако благодаря **finally** завершающий код можно поместить 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:
on
off
*///:~
```

Здесь вызов метода sw.off() просто перемещен в то место, где он гарантированно будет выполнен.

Даже если исключение не перехватывается в текущем наборе условий 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 исполняется при использовании команд break и continue. Заметьте, что комбинация finally в сочетании с break и continue с метками снимает в Java всякую необходимость в операторе goto.

Использование finally с return

Поскольку секция finally выполняется всегда, важные завершающие действия будут выполнены даже при возврате из нескольких точек метода:

```
//· 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("Завершение");
               }
     }
     public static void main(String[] args) {
               for(int i = 1; i <= 4; i++)
                         f(i);
     }
} /* Output:
Инициализация, требующая завершения
Точка 1
Завершение
Инициализация, требующая завершения
Точка 1
Точка 2
Завершение
Инициализация, требующая завершения
Точка 1
Точка 2
Точка 3
Завершение
Инициализация, требующая завершения
Точка 1
Точка 2
Точка 3
Конец
Завершение
*///:~
```

Из выходных данных видно, что выполнение finally не зависит от того, в какой точке защищенной секции была выполнена команда return.

Проблема потерянных исключений

К сожалению, реализация механизма исключений в Java не обошлась без изъяна. Хотя исключение сигнализирует об аварийной ситуации в программе и никогда

не должно игнорироваться, оно может быть потеряно. Это происходит при использовании finally в конструкции определенного вида:

```
//: exceptions/LostMessage.java
// Как теряются исключения.

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
*///:~
```

В выводе нет никаких признаков VeryImportantException, оно было просто 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. Например, в C++ подобная ситуация (возбуждение второго исключения без обработки первого) рассматривается как грубая ошибка программиста. Возможно, в новых версиях Java эта проблема будет решена (впрочем, любой метод, способный возбуждать исключения — такой, как dispose() в приведенном примере — обычно заключается в конструкцию try-catch).

Еще проще потерять исключение простым возвратом из finally:

```
//: exceptions/ExceptionSilencer.java

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

продолжение

```
                throw new RuntimeException();
            } finally {
                // Команда 'return' в блоке finally
                // прерывает обработку исключения
                return;
            }
        }
} ///.~
```

Запустив эту программу, вы увидите, что она ничего не выводит — несмотря на исключение.

Ограничения при использовании исключений

В переопределенном методе можно 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, унаследованные 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 {
        // Реальное исключение не возбуждается
    }
    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 нужно учитывать и исключения базового конструктора:
    public StormyInning()
        throws RainedOut, BaseballException {}
```

```
    public StormyInning(String s)
            throws Foul, BaseballException {}
    // Обычные методы должны соответствовать базовым:
    //! 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 {}
    // Метод может 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("Обобщенное исключение ");
            }
            // Strike не возбуждается в производной версии.
            try {
                    // Что произойдет при восходящем преобразовании?
                    Inning i = new StormyInning();
                    i.atBat();
                    // Необходимо 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("Обобщенное исключение");
            }
    }
} ///:~
```

В классе **Inning** и конструктор, и метод **event()** объявляют, что будут возбуждать исключения, но в действительности этого не делают. Это допустимо, поскольку подобный подход заставляет пользователя перехватывать все виды исключений, которые потом могут быть добавлены в переопределенные версии метода **event()**. Данный принцип распространяется и на абстрактные методы, 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** расширяет **Inning** и реализует

интерфейс Storm, выясняется, что метод event() из Storm не способен изменить тип исключения для метода event() класса Inning. Опять-таки это вполне разумно, так как иначе вы бы никогда не знали, перехватываете ли нужное исключение в случае работы с базовым классом. Конечно, когда метод, описанный в интерфейсе, отсутствует в базовом классе (как rainHard()), никаких проблем с возбуждением исключений нет.

Метод StormyInning.walk() не компилируется из-за того, что он возбуждает исключение, тогда 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 при подстановке объекта класса, производного от Inning, возникли бы исключения, нарушающие работу программы. Таким образом, принудительно обеспечивая соответствие спецификаций исключений в производных и базовых версиях методов, Java добивается взаимозаменяемости объектов.

Переопределенный метод event() показывает, что метод производного класса может вообще не возбуждать исключений, даже если это делается в базовой версии. Опять-таки это нормально, так как не влияет на уже написанный код — подразумевается, что метод базового класса возбуждает исключения. Аналогичная логика применима для метода atBat(), возбуждающего исключение PopFoul, производное от Foul, которое возбуждается базовой версией atBat(). Итак, если вы пишете код, работающий с Inning и вызывающий atBat(), то он должен перехватывать исключение Foul. Так как PopFoul наследует от Foul, обработчик исключения для Foul перехватит и PopFoul.

Последний интересный момент встречается в методе main(). Мы видим, 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 заставляет перехватывать исключения из базового класса. Все эти ограничения значительно повышают ясность и надежность кода обработки исключений[1].

Хотя компилятор заставляет описывать исключения при наследовании, спецификация исключений не является частью объявления (сигнатуры) метода, которое состоит только из имени метода и типов аргументов. Соответственно, нельзя переопределять методы только по спецификациям исключений. Вдобавок, даже если спецификация исключения присутствует в методе базового класса, это вовсе не гарантирует его существования в методе производного класса. Данная практика сильно отличается от правил наследования, по которым метод базового класса обязательно присутствует и в производном 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.

[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.

Конструкторы

При программировании обработки исключений всегда спрашивайте себя: «Если произойдет исключение, будет ли все корректно завершено?» Чаще все идет более или менее безопасно, но с конструкторами возникает проблема. 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 исключение произойдет в конструкторе, эти финальные действия могут быть исполнены ошибочно. А это означает, что при написании конструкторов необходимо быть особенно внимательным.

Казалось бы, блок **finally** решает все проблемы. Но в действительности все сложнее — ведь **finally** выполняется *всегда*, и даже тогда, когда завершающий код не должен активизироваться до вызова какого-то метода. Если сбой в конструкторе произойдет где-то на середине, может оказаться, что часть объекта, освобождаемая в **finally**, еще не была создана.

В следующем примере создается класс, названный **InputFile**, который открывает файл и позволяет читать из него по одной строке. Он использует классы **FileReader** и **BufferedReader** из стандартной библиотеки ввода/вывода Java, которая будет изучена далее, но эти классы достаточно просты, и у вас не возникнет особых сложностей при работе с ними:

```
//  exceptions/InputFile.java
// Специфика исключений 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("Невозможно открыть " + fname);
                       // Файл не открывался, поэтому не может быть закрыт
                       throw e;
              } catch(Exception e) {
                       // При других исключениях файл должен быть закрыт
                       try {
                                in.close();
                       } catch(IOException e2) {
                                System.out.println("in.close() исполнен неудачно");
                       }
                       throw e; // Повторное возбуждение
              } finally {
                       // Не закрывайте файл здесь!!!
              }
     }
     public String getLine() {
              String s;
              try {
                       s = in.readLine();
```

продолжение ⇩

```
                } catch(IOException e) {
                        throw new RuntimeException("readLine() исполнен неудачно");
                }
                return s;
        }
        public void dispose() {
                try {
                        in.close();
                        System.out.println("dispose() успешен");
                } catch(IOException e2) {
                        throw new RuntimeException("in.close() исполнен неудачно");
                }
        }
} ///:~
```

Конструктор InputFile получает в качестве аргумента строку (String) с именем открываемого файла. Внутри блока try он создает объект FileReader для этого файла. Класс FileReader не особенно полезен сам по себе, поэтому мы встраиваем его в созданный BufferedReader, с которым и работаем, — одно из преимуществ InputFile состоит в том, что он объединяет эти два действия.

Если при вызове конструктора FileReader произойдет сбой, возбуждается исключение FileNotFoundException. В этом случае закрывать файл не нужно, так как он и не открывался. Все *остальные* блоки catch обязаны закрыть файл, так как он *уже был* открыт во время входа в них. (Конечно, все было бы сложнее в случае, если бы несколько методов могли возбуждать FileNotFoundException. В таких ситуациях обычно требуется несколько блоков try.) Метод close() тоже может возбудить исключение, которое также проверяется и перехватывается — несмотря на то, что вызов находится в другом блоке catch — с точки зрения компилятора Java это всего лишь еще одна пара фигурных скобок. После выполнения всех необходимых локальных действий исключение возбуждается заново; ведь вызывающий метод не должен считать, что объект был благополучно создан.

В этом примере блок finally определенно *не подходит* для закрытия файла, поскольку в таком варианте закрытие происходило бы каждый раз по завершении работы конструктора. Мы хотим, чтобы файл оставался открытым на протяжении всего жизненного цикла InputFile.

Метод getLine() возвращает объект String со следующей строкой из файла. Он вызывает метод readLine(), способный возбуждать исключения, но они перехватываются; таким образом, сам getLine() исключений не возбуждает. При проектировании обработки исключений вы выбираете между полной обработкой исключения на определенном уровне, его частичной обработкой и передачей далее того же (или другого) исключения и, наконец, простой передачей далее. Там, где это возможно, передача исключения значительно упрощает программирование. В данной ситуации метод getLine() *преобразует* исключение в RuntimeException, чтобы указать на ошибку в программе.

Метод dispose() должен вызываться пользователем при завершении работы с объектом InputFile. Он освобождает системные ресурсы (такие, как открытые файлы), закрепленные за объектами BufferedReader и (или) FileReader. Делать это следует только тогда, когда работа с объектом InputFile действительно будет

завершена. Казалось бы, подобные действия удобно разместить в методе finalize(), но, как упоминалось в главе 5, вызов этого метода не гарантирован (и даже если вы знаете, что он *будет* вызван, то неизвестно, *когда*). Это один из недостатков Java: все завершающие действия, кроме освобождения памяти, не производятся автоматически, так что вам придется информировать пользователя о том, что он ответственен за их выполнение.

Самый безопасный способ использования класса, который способен выдать исключение при конструировании и требует завершающих действий, основан на использовании вложенных блоков try:

```
//: exceptions/Cleanup.java
// Гарантированное освобождение ресурсов.

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)
                                ; // Построчная обработка ..
                       } catch(Exception e) {
                                System.out.println("Перехвачено Exception в main"),
                                e.printStackTrace(System.out);
                       } finally {
                                in.dispose();
                       }
              } catch(Exception e) {
                       System out println("Сбой при конструировании InputFile");
              }
     }
} /* Output:
dispose() успешен
*///:~
```

Присмотритесь к логике происходящего: конструирование объекта InputFile фактически заключено в собственный блок try. Если попытка завершается неудачей, мы входим во внешнюю секцию catch и метод dispose() не вызывается. Но, если конструирование прошло успешно, мы хотим обеспечить гарантированное завершение, поэтому сразу же после конструирования создается новый блок try. Блок finally, выполняющий завершение, связывается с *внутренним* блоком try; таким образом, блок finally не выполняется при неудачном конструировании и *всегда* выполняется, если конструирование прошло удачно.

Эта универсальная идиома применяется и в тех ситуациях, когда конструктор не выдает исключений. Основной принцип: сразу же после создания объекта, требующего завершения, начинается конструкция try-finally:

```
//: exceptions/CleanupIdiom java
// За каждым освобождаемым объектом следует try-finally

class NeedsCleanup { // Конструирование не может завершиться неудачно
     private static long counter = 1,
     private final long id = counter++,
```

продолжение ⇩

```
        public void dispose() {
                System out println("NeedsCleanup " + id + " завершен");
        }
}

class ConstructionException extends Exception {}

class NeedsCleanup2 extends NeedsCleanup {
        // Возможны сбои при конструировании.
        public NeedsCleanup2() throws ConstructionException {}
}

public class CleanupIdiom {
        public static void main(String[] args) {
                // Секция 1·
                NeedsCleanup nc1 = new NeedsCleanup();
                try {
                        // ..
                } finally {
                        nc1.dispose(),
                }

                // Секция 2:
                // Если сбои при конструировании исключены,
                // объекты можно группировать.
                NeedsCleanup nc2 = new NeedsCleanup();
                NeedsCleanup nc3 = new NeedsCleanup();
                try {
                        //  ..
                } finally {
                        nc3 dispose(); // Обратный порядок конструирования
                        nc2.dispose();
                }

                // Секция 3·
                // Если при конструировании возможны сбои, каждый объект
                // защищается отдельно:
                try {
                        NeedsCleanup2 nc4 = new NeedsCleanup2();
                        try {
                                NeedsCleanup2 nc5 = new NeedsCleanup2();
                                try {
                                        // ...
                                } finally {
                                        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 завершен
NeedsCleanup 3 завершен
```

```
NeedsCleanup 2 завершен
NeedsCleanup 5 завершен
NeedsCleanup 4 завершен
*/// ~
```

Секция 1 метода main() весьма прямолинейна: за созданием завершаемого объекта следует try-finally. Если конструирование не может завершиться неудачей, наличие catch не требуется. В секции 2 мы видим, что конструкторы, которые не могут завершиться неудачей, могут группироваться как для конструирования, так и для завершения.

Секция 3 показывает, как поступать с объектами, при конструировании которых *возможны* сбои и которые нуждаются в завершении. Здесь программа усложняется, потому что каждое конструирование должно заключаться в отдельную копию try-catch и за ним должна следовать конструкция try-finally, обеспечивающая завершение.

Неудобства обработки исключения в подобных случаях — веский аргумент в пользу создания конструкторов, выполнение которых заведомо обходится без сбоев (хотя это и не всегда возможно).

Идентификация исключений

Механизм обработки исключений ищет в списке «ближайший» подходящий обработчик в порядке их следования. Когда соответствие обнаруживается, исключение считается найденным и дальнейшего поиска не происходит.

Идентификация исключений не требует обязательного соответствия между исключением и обработчиком. Объект порожденного класса подойдет и для обработчика, изначально написанного для базового класса:

```
//: exceptions/Human.java
// Перехват иерархии 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
```

продолжение ⇩

```
Перехвачено Sneeze
Перехвачено Annoyance
*///·~
```

Исключение Sneeze будет перехвачено в первом блоке catch, который ему соответствует — конечно, это будет первый блок. Но, если удалить первый блок catch, оставив только проверку Annoyance, программа все равно работает, потому что она перехватывает базовый класс Sneeze. Другими словами, блок catch (Annoyance a) поймает Annoyance или *любой другой класс, унаследованный от него*. Если вы добавите новые производные исключения в свой метод, программа пользователя этого метода не потребует изменений, так как клиент перехватывает исключения базового класса.

Если вы попытаетесь «замаскировать» исключения производного класса, поместив сначала блок catch базового класса:

```
try {
        throw new Sneeze();
} catch(Annoyance a) {
        // ..
} catch(Sneeze s) {
        //...
}
```

компилятор выдаст сообщение об ошибке, так как он видит, что блок catch для исключения Sneeze никогда не выполнится.

Альтернативные решения

Система обработки исключений представляет собой «черный ход», позволяющий программе нарушить нормальную последовательность выполнения команд. «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 возникают системы обработки исключений, кроется в том, что программисты не желали иметь дела с громоздкой проверкой всех возможных условий возникновения ошибок каждой функции. В результате ошибки ими просто игнорировались. Стоит отметить, что вопрос удобства программиста при обработке ошибок стоял на первом месте для разработчиков Java.

Основное правило при использовании исключений гласит: «Не обрабатывайте исключение, если 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 сконцентрироваться на том, что вы хотите сделать в одном фрагменте кода, и на том, как вы собираетесь поступить с ошибками в совершенно другом месте программы. В результате основной код не перемежается с логикой обработки ошибок, что упрощает его сопровождение и понимание. Исключения также сокращают объем кода, так как один обработчик может обслуживать несколько потенциальных источников ошибок.

Контролируемые исключения немного усложняют ситуацию, поскольку они заставляют добавлять обрабатывающие исключения предложения там, где вы не всегда еще готовы справиться с ошибкой. В итоге возникает проблема «проглоченных исключений»:

```
try {
     //    делает что-то полезное
} catch(ОбязывающееИсключение e) {} // Проглотили!
```

Программисты (и я в том числе, в первом издании книги), не долго думая, делали самое бросающееся в глаза и «проглатывали» исключение — зачастую непреднамеренно, но, как только дело было сделано, компилятор был удовлетворен, поэтому пока вы не вспоминали о необходимости пересмотреть и исправить код, не вспоминали и об исключении. Исключение происходит, но безвозвратно теряется. Из-за того что компилятор заставляет вас писать код для обработки исключений прямо на месте, это кажется самым простым решением, хотя на самом деле ничего хуже и придумать нельзя.

Ужаснувшись тем, что я так поступил, во втором издании книги я «исправил» проблему, распечатыв в обработчике трассировку стека исключения (и сейчас это можно видеть — в подходящих местах — в некоторых примерах данной главы). Хотя это и полезно при отслеживании поведения исключений, трассировка фактически означает, 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, проблема не только очень сложна, но и к тому же неоднозначна. Существуют твердые приверженцы обеих точек зрения, которые считают, что верный ответ (их) очевиден и просто бросается в глаза. Вероятно, одна из точек зрения основана на несомненных преимуществах перехода от слабо типизированного языка (например, C до выхода стандарта ANSI) к языку с строгой статической проверкой типов (то есть с проверкой во время компиляции), подобному C++ или Java. Преимущества такого перехода настолько очевидны, 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 неформальная граница, за которой 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++, кроме тех аспектов, в которых решения C++ приводили к проблемам.

Обработка исключений была добавлена в C++ на довольно позднем этапе стандартизации. 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).

Следуя подходу CLU при разработке исключений C++, Страуструп считал, что основной целью является сокращение объема кода восстановления после ошибки. Вероятно, он видел немало программистов, которые не писали код обработки ошибок на C, поскольку объем этого кода был устрашающим, а размещение выглядело нелогично. В результате все происходило в стиле C: ошибки в коде игнорировались, а с проблемами справлялись при помощи отладчиков. Чтобы исключения реально заработали, C-программисты должны были писать «лишний» код, без которого они обычно обходились. Таким образом, объем нового кода не должен быть чрезмерным. Важно помнить об этих целях, говоря об эффективности контролируемых исключений в Java.

C++ добавил к идее CLU дополнительную возможность: спецификации исключений, то есть включение в сигнатуру метода информации об исключениях, возникающих при вызове. В действительности спецификация исключения несет двойной смысл. Она означает: «Я возбуждаю это исключение в коде, а вы его обрабатываете». Но она также может означать: «Я игнорирую исключение, которое может возникнуть в моем коде; обеспечьте его обработку». При освещении механизмов исключений мы концентрировались на «обеспечении обработки», но здесь мне хотелось бы поближе рассмотреть тот факт, что зачастую исключения игнорируются, и именно этот факт может быть отражен в спецификации исключения.

В C++ спецификация исключения не входит в информацию о типе функции. Единственная проверка, осуществляемая во время компиляции, относится к согласованному использованию 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, компилятор не проверяет, действительно ли функция или метод возбуждают данное исключение, или полноту спецификации (то есть описывает ли она все исключения, возможные для этого метода). Если возбуждается исключение, не входящее в спецификацию, программа на C++ вызывает функцию `unexpected()` из стандартной библиотеки.

Интересно отметить, что из-за использования шаблонов (templates) спецификации исключений отсутствуют в стандартной библиотеке C++. В Java существуют ограничения на использование параметризованных типов со спецификациями исключений.

Перспективы

Во-первых, 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 внима-

ния). Это был эксперимент, повторить который с тех пор пока не решился еще ни один язык.

Во-вторых, контролируемые исключения однозначно хороши при рассмотрении вводных примеров и в небольших программах. Оказывается, что трудноуловимые проблемы начинают проявляться при разрастании программы. Конечно, программы не разрастаются 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 изменилась с управляемой на затруднительную в управлении. Именно это, как я полагаю, может произойти, когда проверок типов слишком много, и особенно в отношении контролируемых исключений.

Одним из важных достижений Java стала унификация модели передачи информации 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 простого игнорирования ошибок. Когда Java изменил модель C++ так, что сообщать об ошибках стало возможно только посредством исключений, необходимость в дополнительных мерах принуждения в виде контролируемых исключений сократилась.

В прошлом я твердо считал, что для разработки надежных программ необходимы и контролируемые исключения, и строгая статическая проверка типов. Однако опыт, полученный лично и со стороны[1], с языками, более динамичными, чем статичными, привел меня к мысли, что на самом деле главные преимущества обусловлены следующими аспектами:

1. Унификация модели сообщения об ошибках посредством исключений (независимо от того, заставляет ли компилятор программиста их обрабатывать).
2. Проверка типов, не привязанная к тому, когда она проводится — на стадии компиляции или во время работы программы.

Вдобавок снижение ограничений времени компиляции весьма положительно отражается на продуктивности программиста. С другой стороны, для компенсации чрезмерной жесткости статической проверки типов необходимы *рефлексия* и *параметризация*, как вы убедитесь в некоторых примерах книги.

Некоторые уверяли меня, что все сказанное является кощунством, безнадежно испортит мою репутацию, приведет к гибели цивилизации и провалу большой доли программных проектов. Вера в то, что выявление ошибок на стадии компиляции спасет ваш проект, весьма сильна, но гораздо важнее сознавать ограничения того, на что способен компьютер. Стоит помнить:

«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 оборудовании Sun, как бы старо оно ни было. Но, если вы чувствуете, что контролируемые исключения становятся для вас препятствием (особенно если вас заставляют обрабатывать исключение, а вы не знаете, как с ним поступить), существует несколько вариантов.

Передача исключений на консоль

В несложных программах, как во многих примерах данной книги, простейшим решением является передача исключения за пределы метода main(), на консоль. К примеру, при открытии файла для чтения (подробности вы вскоре узнаете) необходимо 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() — такой же метод, как и все прочие; он тоже может иметь спецификацию исключений, и здесь типом исключения является Exception, базовый класс всех контролируемых исключений. Передавая его на консоль, вы освобождаетесь от необходимости написания предложений try-catch в теле метода main().

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 при вызове другого метода вы четко сознаете: «Понятия не имею, что делать с исключением дальше, но „съедать" мне его не хочется, так же как и печатать банальное сообщение». Проблема решается при помощи цепочек исключений. Управляемое исключение просто «заворачивается» в класс RuntimeException примерно так:

```
try {
     //  .. делаем что-нибудь полезное
} catch(НеЗнаюЧтоДелатьСЭтимКонтролируемымИсключением e) {
     throw new RuntimeException(e);
}
```

Решение идеально подходит для тех случаев, когда вы хотите «подавить» контролируемое исключение: вы не «съедаете» его, вам не приходится описывать его в своей спецификации исключений, и благодаря цепочке исключений вы не теряете информацию об исходном исключении.

Описанная методика позволяет игнорировать исключение и пустить его «всплывать» вверх по стеку вызова без необходимости писать блоки try-catch и (или) спецификации исключения. Впрочем, при этом вы все равно можете перехватить и обработать конкретное исключение, используя метод getCause(), как показано ниже:

```
//  exceptions/TurnOffChecking.java
// "Подавление" 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();
               // Можно вызвать throwRuntimeException() без блока try,
               // и позволить исключению RuntimeException покинуть метод:
               wce.throwRuntimeException(3);
               // Или перехватить исключение:
               for(int i = 0; i < 4; i++)
                         try {
                                   if(i < 3)
                                             wce.throwRuntimeException(i);
                                   else
                                             throw new SomeOtherException();
                         } catch(SomeOtherException e) {
                                   print("SomeOtherException: " + e);
                         } catch(RuntimeException re) {
                                   try {
```

продолжение ⇩

```
                                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.

При взгляде на класс **TurnOffChecking** нетрудно заметить, что вызвать метод **throwRuntimeException()** можно и без блока **try**, поскольку он не возбуждает никаких контролируемых исключений. Но 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()**, «завернутое» исключение). Так извлекаются изначальные исключения, обрабатываемые в своих предложениях **catch**.

Методика «заворачивания» управляемых исключений в объекты **RuntimeException** встречается в некоторых примерах книги. Другое возможное решение — создание собственного класса, производного от **RuntimeException**. Перехватывать такое исключение не обязательно, но, если вы захотите, такая возможность существует.

Основные правила обработки исключений

Используйте исключения для того, чтобы:

- обработать ошибку на текущем уровне (избегайте перехватывать исключения, если вы не знаете, как с ними поступить);
- исправить проблему и снова вызвать метод, возбудивший исключение;
- предпринять все необходимые действия и продолжить выполнение без повторного вызова метода;
- попытаться найти альтернативный результат вместо того, который должен был бы произвести вызванный метод;
- сделать все возможное в текущем контексте и заново возбудить *это же* исключение, перенаправив его на более высокий уровень;

- сделать все, что можно в текущем контексте, и возбудить *новое* исключение, перенаправив его на более высокий уровень;
- завершить работу программы;
- упростить программу (если используемая вами схема обработки исключений делает все только сложнее, значит, она никуда не годится);
- добавить вашей библиотеке и программе безопасности (сначала это поможет в отладке программы, а в дальнейшем окупится ее надежностью).

Резюме

Исключения являются неотъемлемой частью программирования на Java; существует некий барьер, который невозможно преодолеть без умения работать с ними. По этой причине исключения были представлены именно в этой части книги — многими библиотеками (скажем, библиотекой ввода/вывода) просто невозможно нормально пользоваться без обработки исключений.

Одно из преимуществ обработки исключений состоит в том, что она позволяет сосредоточиться на решаемой проблеме, а затем обработать все ошибки в описанном коде в другом месте. Хотя исключения обычно описываются как средство *передачи информации* и *восстановления* после ошибок на стадии выполнения, я сильно сомневаюсь, что «восстановление» часто реализуется на практике. По моей оценке, это происходит не более чем в 10% случаев, и даже тогда в основном сводится к раскрутке стека к заведомо стабильному состоянию вместо реального выполнения действий по восстановлению. На мой взгляд, ценность исключений в основном обусловлена именно передачей информации. Java фактически настаивает, что программа должна сообщать обо всех ошибках в виде исключений, и именно это обстоятельство обеспечивает Java большое преимущество перед языками вроде C++, где программа может сообщать об ошибках разными способами (а то и не сообщать вовсе).

Информация о типах

13

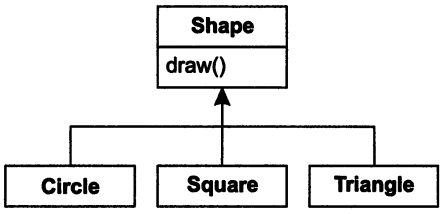
Механизм RTTI (Runtime Type Information) предназначен для получения и использования информации о типах во время выполнения программы.

RTTI освобождает разработчика от необходимости выполнять всю работу с типами на стадии компиляции и открывает немало замечательных возможностей. Потребность в RTTI вскрывает целый ряд интересных (и зачастую сложных) аспектов объектно-ориентированного проектирования.

В этой главе рассматриваются способы получения информации об объектах и классах во время выполнения программы в Java. Существует два механизма получения такой информации: «традиционный» механизм RTTI, подразумевающий, что все типы доступны во время компиляции, а также механизм *рефлексии* (reflection), применяемый исключительно во время выполнения программы.

Необходимость в динамическом определении типов (RTTI)

Рассмотрим хорошо знакомый пример с геометрическими фигурами, основанный на полиморфизме. Обобщенным базовым классом является фигура Shape, а производными классами — окружность Circle, прямоугольник Square и треугольник Triangle.



Это обычная диаграмма наследования — базовый класс расположен вверху, производные классы присоединяются к нему снизу. Обычно при разработке объектно-ориентированных программ код по возможности манипулирует ссылками на базовый класс (в нашем случае это фигура — Shape). Если вдруг в программу будет добавлен 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, пользуясь ссылкой базового типа Shape. Метод draw() переопределяется во всех производных классах, и по природе динамического связывания вызов его по ссылке на базовый класс все равно даст необходимый результат. Это и есть 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 обобщенную фигуру.

Реализация иерархии Shape может выглядеть примерно 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, для чего ссылка this передается методу System.out.println() (обратите внимание: метод toString() объявлен абстрактным, чтобы производные классы были обязаны переопределить его и чтобы предотвратить

создание экземпляров Shape). Когда этот объект встречается в выражении конкатенации строк, автоматически вызывается его метод toString() для получения соответствующего строкового представления. Каждый из производных классов переопределяет метод toString() (из базового класса Object), чтобы метод draw() выводил в каждом случае различную информацию.

В данном примере восходящее преобразование происходит во время помещения объекта-фигуры в контейнер List<Shape>. В процессе восходящего преобразования теряется конкретная информация, в том числе и точный тип фигур. Контейнеру все равно — он хранит просто объекты Shape.

Когда вы извлекаете из контейнера очередной элемент, контейнер, в котором все элементы хранятся в виде Object, автоматически преобразует результат обратно к Shape. Это наиболее основная форма RTTI, поскольку все подобные преобразования в языке Java проверяются на правильность на стадии исполнения. 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 нам *известно* только то, что контейнер List<Shape> заполнен фигурами Shape. Во время компиляции соблюдение этого требования обеспечивается контейнером и системой параметризованных типов Java, а при выполнении оно подтверждается успешным преобразованием типов.

Теперь в действие вступает полиморфизм — для каждой фигуры Shape вызывается свой метод draw(), в зависимости от того, окружность это (Circle), прямоугольник (Square) или треугольник (Triangle). И в основном именно так и должно быть; основная часть кода не должна зависеть от точного типа объекта, она оперирует с универсальным представлением целого семейства объектов (в нашем случае это фигура (Shape)). Такой подход упрощает написание программы, а впоследствии ее чтение и сопровождение. По этой причине полиморфизм часто используется при написании объектно-ориентированных программ.

Но что, если у вас имеется не совсем обычная задача, для успешного решения которой необходимо узнать точный тип объекта, располагая только ссылкой на базовый тип? Допустим, пользователи программы с фигурами хотят выделить определенные фигуры (скажем, все треугольники) на экране фиолетовым цветом. При помощи RTTI можно узнать точный тип объекта, на который указывает ссылка базового типа Shape.

Объект Class

Чтобы понять, как работает RTTI в Java, необходимо знать, каким образом хранится информация о типе во время выполнения программы. Для этой цели используется специальный *объект* типа Class, который и содержит описание класса. Объект Class используется при создании всех «обыкновенных» объектов любой программы.

Каждый класс, задействованный в программе, представлен своим объектом Class. Иначе говоря, при написании и последующей компиляции нового класса для него создается объект Class (который затем сохраняется в одноименном файле с расширением .class). Для создания объекта этого класса виртуальная машина Java (JVM), исполняющая программу, использует подсистему, называемую *загрузчиком классов*.

Подсистема загрузчиков классов в действительности состоит из цепочки загрузчиков классов, но только *основной загрузчик* является частью реализации JVM. Основной загрузчик классов загружает так называемые *доверенные классы*, в том числе классы Java API, с локального диска. Обычно включать дополнительные загрузчики классов в цепочку не требуется, но в особых ситуациях (например, при загрузке классов для поддержки приложений веб-сервера или при загрузке классов по сети) существует возможность подключения дополнительных загрузчиков.

Все классы загружаются в JVM динамически, при первом использовании класса. Таким образом, программа на Java никогда не бывает полностью загружена до начала своего выполнения, и в этом отношении Java отличается от многих традиционных языков. Динамическая загрузка открывает возможности, недоступные или трудно реализуемые в языках со статической загрузкой вроде C++.

Сначала JVM проверяет, загружен ли объект Class для этого нового класса. При отрицательном результате JVM ищет подходящий файл .class и подгружает его (а дополнительный загрузчик, например, может подгружать байт-код из базы данных). Загруженный код класса проходит проверку на целостность и на отсутствие некорректного байт-кода Java (одна из «линий защиты» в Java).

После того как объект Class для определенного типа будет помещен в память, в дальнейшем он используется при создании всех объектов этого типа. Следующая программа проясняет сказанное:

```
//· typeinfo/SweetShop.java
// Проверка процесса загрузки классов.
import static net.mindview.util.Print *.

class Candy {
     static { print("Загрузка класса Candy"). }
}

class Gum {
     static { print("Загрузка класса Gum"). }
}

class Cookie {
     static { System.out.println("Загрузка класса Cookie"). }
}

public class SweetShop {
     public static void main(String[] args) {
               print("в методе main()").
               new Candy();
               print("После создания объекта Candy");
               try {
```

продолжение ⇩

```
                Class forName("Gum");
            } catch(ClassNotFoundException e) {
                print("Класс Gum не найден");
            }
            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 из классов Candy, Gum и Cookie присутствует статический блок, который отрабатывает 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, по которым можно оценить, в какой момент загружается тот или иной класс.

Из результата работы программы мы видим, что объект Class загружается только при непосредственной необходимости, а статическая инициализация производится при загрузке этого объекта.

Особенно интересно выглядит строка программы

```
Class forName("Gum");
```

Все объекты Class принадлежат классу Class. Объект Class ничем принципиально не отличается от других объектов, поэтому вы можете выполнять операции со ссылкой на него (именно так и поступает загрузчик классов). Один из способов получения ссылки на объект Class заключается в вызове метода forName(), которому передается строка (String) с именем класса (следите за правильностью написания и регистром символов!). Метод возвращает ссылку на объект Class, которая в данном примере нам не нужна; метод Class.forName() вызывался ради побочного эффекта, то есть загрузки класса Gum, если он еще не в памяти. В процессе загрузки выполняется static-инициализатор класса Gum.

Если бы в рассмотренном примере метод Class.forName() сработал неудачно (не смог бы найти класс, который вы хотели загрузить), он возбудил бы исключение ClassNotFoundException. Здесь мы просто сообщаем о проблеме и двигаемся дальше, однако в более совершенной программе можно было бы попытаться исправить ошибку в обработчике исключения.

Чтобы использовать информацию RTTI на стадии исполнения, прежде всего необходимо получить ссылку на подходящий объект Class. Один из способов — вызов метода Class.forName() — удобен тем, что вам не потребуется уже существующий объект нужного типа. Впрочем, если такой объект уже существует, для получения ссылки на его объект Class можно вызвать метод getClass(), определенный в корневом классе Object. Метод возвращает объект Class, представляю-

щий фактический тип объекта. Класс Class содержит немало интересных методов, продемонстрированных в следующем примере:

```
//· typeinfo/ToyTest.java
// Тестирование класса Class.
package typeinfo.toys;
import static net.mindview.util Print.*;

interface HasBatteries {}
interface Waterproof {}
interface Shoots {}

class Toy {
     // Закомментируйте следующий далее конструктор по
     // умолчанию, тогда в строке с пометкой (*1*)
     // возникнет ошибка NoSuchMethodError.
     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() +
               " это интерфейс? [" + cc.isInterface() + "]");
          print("Простое имя: " + cc.getSimpleName());
          print("Каноническое имя· " + cc.getCanonicalName());
     }
     public static void main(String[] args) {
          Class c = null;
          try {
               c = Class.forName("typeinfo.toys FancyToy");
          } catch(ClassNotFoundException e) {
               print("Не найден класс 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
               o = up.newInstance();
          } catch(InstantiationException e) {
               print("Не удалось создать объект");
               System.exit(1);
          } catch(IllegalAccessException e) {
                    print("Нет доступа");
                    System.exit(1);
          }
          printInfo(obj.getClass());
     }
```

продолжение ⇩

```
} /* Output
Имя класса  typeinfo toys FancyToy это интерфейс? [false]
Простое имя  FancyToy
Каноническое имя· typeinfo.toys FancyToy
Имя класса  typeinfo toys HasBatteries это интерфейс? [true]
Простое имя  HasBatteries
Каноническое имя: typeinfo toys HasBatteries
Имя класса  typeinfo toys Waterproof это интерфейс? [true]
Простое имя. Waterproof
Каноническое имя  typeinfo toys Waterproof
Имя класса. typeinfo.toys Shoots это интерфейс? [true]
Простое имя: Shoots
Каноническое имя: typeinfo.toys Shoots
Имя класса: typeinfo.toys Toy это интерфейс? [false]
Простое имя: Toy
Каноническое имя  typeinfo.toys.Toy
*///.~
```

Класс FancyToy, производный от Toy, реализует несколько интерфейсов: HasBatteries, Waterproof и Shoots. В методе main() создается ссылка на объект Class для класса FancyToy, для этого в подходящем блоке try вызывается метод forName(). Обратите внимание на необходимость использования полного имени (с именем пакета) в строке, передаваемой forName().

Метод printInfo() использует getName() для получения полного имени класса и методы getSimpleName() и getCanonicalName() (появившиеся в Java SE5), возвращающие имя без пакета и полное имя соответственно. Метод isInterface() проверяет, представляет ли объект Class интерфейс. Таким образом, по объекту Class можно узнать практически все, что может потребоваться узнать о типе.

Метод Class.getInterfaces() возвращает массив объектов Class, представляющих интерфейсы, реализованные объектом Class. Метод getSuperclass() возвращает непосредственный (то есть ближайший) базовый класс для объекта Class.

Метод newInstance() фактически реализует «виртуальный конструктор». Вы как бы говорите: «Я не знаю ваш точный тип, так что создайте себя самостоятельно». В рассмотренном примере ссылка up просто указывает на объект Class, больше никакой информации о типе у вас нет. Поэтому при создании нового экземпляра методом newInstance() вы получаете ссылку на обобщенный объект Object. Однако полученная ссылка на самом деле указывает на объект Toy. Следовательно, перед посылкой сообщений, характерных для класса Toy, придется провести нисходящее преобразование. Вдобавок объект, созданный с помощью метода newInstance(), обязан определить конструктор по умолчанию. Позднее в этой главе будет показано, как динамически создать объект класса любым конструктором с использованием механизма *рефлексии* Java.

Литералы class

В Java существует еще один способ получения ссылок на объект Class — посредством *литерала* class. В предыдущей программе получение ссылки выглядело бы так:

```
FancyToy.class;
```

Такой способ не только проще, но еще и безопасней, поскольку проверка осуществляется еще во время компиляции. К тому же он не требует вызова forName(), а значит, является более эффективным.

Литералы class работают со всеми обычными классами, так же как и с интерфейсами, массивами и даже с примитивами. Вдобавок во всех классах-обертках для примитивных типов имеется поле с именем TYPE. Это поле содержит ссылку на объект Class для ассоциированного с ним простейшего типа, как показано в табл. 13.1.

Таблица 13.1. Альтернативное обозначение объекта Class с помощью литералов

Литерал	Ссылка на объект Class
boolean.class	Boolean.TYPE
char.class	Character.TYPE
byte.class	Byte.TYPE
short.class	Short.TYPE
int.class	Integer.TYPE
long.class	Long.TYPE
float.class	Float.TYPE
double.class	Double.TYPE
void.class	Void.TYPE

Хотя эти версии эквивалентны, я предпочитаю использовать синтасис .class, так как он лучше сочетается с обычными классами.

Интересно заметить, что создание ссылки на объект Class с использованием записи .class не приводит к автоматической инициализации объекта Class. Подготовка класса к использованию состоит из трех этапов:

- *Загрузка* — выполняется загрузчиком классов. Последний находит байт-код и создает на его основе объект Class.
- *Компоновка* — в фазе компоновки проверяется байт-код класса, выделяется память для статических полей, и при необходимости разрешаются все ссылки на классы, созданные этим классом.
- *Инициализация* — если у класса имеется суперкласс, происходит его инициализация, выполняются статические инициализаторы и блоки статической инициализации.

Инициализация откладывается до первой ссылки на статический метод (конструкторы являются статическими методами) или на неконстантное статическое поле:

```
//: 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");
```

продолжение ⇨

```
        }
}

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 Initable");
                // Не приводит к инициализации
                System.out.println(Initable.staticFinal);
                // Приводит к инициализации:
                System.out.println(Initable.staticFinal2);
                // Приводит к инициализации
                System.out.println(Initable2.staticNonFinal);
                Class initable3 = Class.forName("Initable3");
                System.out.println("После создания ссылки Initable3");
                System.out.println(Initable3.staticNonFinal);
        }
} /* Output
После создания ссылки Initable
47
Инициализация Initable
258
Инициализация Initable2
147
Инициализация Initable3
После создания ссылки Initable3
74
*///:~
```

По сути, инициализация откладывается настолько, насколько это возможно. Из результатов видно, что простое использование синтаксиса .class для получения ссылки на класс не приводит к выполнению инициализации. С другой стороны, вызов Class.forNames() немедленно инициализирует класс для получения ссылки на Class, как мы видим на примере initable3.

Параметризованные ссылки

Объект Class используется для создания экземпляров класса и содержит полный код методов этих экземпляров. Кроме того, в нем содержатся статические

члены класса. Таким образом, ссылка на Class подразумевает точный тип того, на что она указывает — на объект класса Class.

Однако проектировщики Java SE5 решили предоставить возможность уточнения записи посредством ограничения типа объекта Class, на который может указывать ссылка; для этой цели применяется синтаксис параметризации. В следующем примере верны оба варианта синтаксиса:

```
// 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 самое
              intClass = double.class;
              // genericIntClass = double.class; // Недопустимо
     }
} ///:~
```

Если обычная ссылка на класс может быть связана с любым объектом Class, параметризованная ссылка может связываться только с объектами типа, указанного при ее объявлении. Синтаксис параметризации позволяет компилятору выполнить дополнительную проверку типов.

Новый синтаксис преобразования

В Java SE5 также появился новый синтаксис преобразования ссылок на Class, основанный на методе 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; // ... А можно и так.
     }
} /// ~
```

Метод cast() получает объект-аргумент и преобразует его к типу ссылки на Class. Конечно, при взгляде на приведенный код может показаться, что он занимает слишком много места по сравнению с последней строкой main(), которая делает то же самое. Новый синтаксис преобразования полезен в тех ситуациях, когда обычное преобразование *невозможно*. Обычно это происходит тогда, когда при написании параметризованного кода (см. далее) ссылка на Class сохраняется для преобразования в будущем. Подобные ситуации встречаются крайне редко — во всей библиотеке Java SE5 cast() используется всего один раз (в com.sun.mirror.util.DeclarationFilter).

Другая новая возможность — Class.asSubclass() — вообще не встречается в библиотеке Java SE5. Этот метод позволяет преобразовать объект класса к более конкретному типу.

Проверка перед приведением типов

Итак, мы рассмотрели следующие формы RTTI:

- Классическое преобразование; аналог выражения «(Shape)», которое проверяет, «законно» ли приведение типов в данной ситуации, и в случае неверного преобразования возбуждает исключение ClassCastException.
- Объект Class, представляющий тип вашего объекта. К объекту Class можно обращаться для получения полезной информации во время выполнения программы.

В языке C++ классическая форма типа «(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. Поскольку компилятор знает, что Circle является частным случаем Shape, он позволяет использовать «восходящее» присваивание без явного преобразования типа. Тем не менее, получив некий объект Shape, компилятор *не может* быть уверен в том, что он получил: то ли действительно Shape, то ли один из производных типов (Circle, Square или Triangle). На стадии компиляции он видит только Shape и поэтому не позволит использовать «нисходящее» присваивание без явного преобразования типа.

Существует и третья форма RTTI в Java — ключевое слово instanceof, которое проверяет, является ли объект экземпляром заданного типа. Результат возвращается в логическом (boolean) формате, поэтому вы просто «задаете» вопрос в следующей форме:

```
if(x instanceof Dog)
    ((Dog)x).bark();
```

Команда if сначала проверяет, принадлежит ли объект x классу Dog, и *только после этого* выполняет приведение объекта к типу Dog. Настоятельно рекомендуется использовать ключевое слово instanceof перед проведением нисходящего преобразования, особенно при недостатке информации о точном типе объекта; иначе возникает опасность исключения ClassCastException.

Обычно проводится поиск одного определенного типа (например, поиск треугольников среди прочих фигур), но с помощью ключевого слова instanceof легко можно идентифицировать все типы объекта. Предположим, что у нас есть иерархия классов для описания домашних животных Pet (и их владельцев — эта особенность пригодится нам в более позднем примере). Каждое существо (Individual) в этой иерархии обладает идентификатором id и необязательным

именем. В данный момент код Individual нас не интересует — достаточно знать, что объект можно создавать с именем или без, и у каждого объекта Individual имеется метод id(), возвращающий уникальный идентификатор. Также имеется метод toString(); если имя не указано, toString() выдает имя типа.

Иерархия классов, производных 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 {
```

продолжение ⇩

```
    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(); }
} ///:~
```

Затем нам понадобятся средства для создания случайных типов **Pet**, а для удобства — массивов и списков (**List**) с элементами **Pet**. Чтобы этот инструментарий мог «пережить» несколько разных реализаций, мы определим его в виде абстрактного класса:

```
// typeinfo/pets/PetCreator.java
// Создание случайных последовательностей Pet
package typeinfo.pets;
import java.util.*;

public abstract class PetCreator {
    private Random rand = new Random(47);
    // Список создаваемых типов, производных от 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() поручает производному классу получение списка объектов Class. В качестве типа класса указан «любой производный от Pet», поэтому newInstance() создает Pet без необходимости преобразования типа. Метод randomPet() осуществляет случайную выборку из List и использует полученные объекты Class для создания нового экземпляра данного класса вызовом Class.newInstance(). Метод createArray() использует randomPet() для заполнения массива, а arrayList(), в свою очередь, использует createArray().

При вызове newInstance() возможны два вида исключений, обрабатываемые в секциях catch за блоком try. Имена исключений достаточно хорошо объясняют суть проблемы (IllegalAccessException — нарушение механизма безопасности Java, в данном случае если конструктор по умолчанию объявлен private).

Определяя субкласс PetCreator, достаточно предоставить список типов Pet, которые должны создаваться с использованием randomPet() и других методов. Метод getTypes() возвращает ссылку на статический объект List. Реализация с использованием forName() выглядит так:

```
//: 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>>();
```

продолжение ⇩

```
        // Типы, создаваемые 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 список List объектов Class с использованием метода Class.forName(). При этом может произойти исключение ClassNotFoundException, что вполне понятно — ведь ему передается строка, содержимое которой невозможно проверить на стадии компиляции. При ссылке на эти классы необходимо указывать имя пакета, которому они принадлежат (typeinfo).

Для получения типизованного списка объектов Class требуется преобразование типа, что приводит к выдаче предупреждения на стадии компиляции. Метод loader() определяется отдельно и размещается в секции статической инициализации, потому что директива @SuppressWarnings не может располагаться прямо в секции статической инициализации.

Для подсчета объектов Pet нам понадобится механизм подсчета их разных видов. Для этой цели идеально подойдет карта (Map), в которой ключами являются имена типов Pet, а значениями — переменные Integer с количеством Pet. Например, это позволит получать ответы на вопросы типа «сколько существует объектов Hamster?». При подсчете Pet будет использоваться ключевое слово instanceof:

```
//: typeinfo/PetCount.java
// Использование 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 всех 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 результатов подсчета.
        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 **countPets** массив случайным образом заполняется объектами **Pet** с использованием **PetCreator**. Затем каждый объект **Pet** в массиве тестируется и подсчитывается при помощи **instanceof**.

У ключевого слова **instanceof** имеется одно серьезное ограничение: объект можно сравнивать только с именованным типом, но не с объектом **Class**. Возможно, вам показалось, что в предыдущем примере перебор всех выражений **instanceof** выглядит неудобно и громоздко, и вы правы. Тем не менее автоматизировать этот

процесс невозможно — создать из объектов Class список ArrayList и сравнивать объекты по очереди с каждым его элементом не получится (к счастью, существует альтернативное решение, но это попозже). Впрочем, особенно горевать по этому поводу не стоит — если вам приходится записывать множество проверок instanceof, скорее всего, изъян кроется в архитектуре программы.

Использование литералов class

Если записать пример PetCreator.java с использованием литералов class, программа во многих отношениях становится более понятной:

```
//· typeinfo/pets/LiteralPetCreator.java
// Using class literals
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 для случайного создания:
     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 будущем примере PetCount3.java контейнер Map заполняется всеми типами Pet (не только генерируемыми случайным образом), поэтому нам понадобился список allTypes. Список types представляет собой часть allTypes (создается вызовом List.subList()) со всеми типами Pet, поэтому он используется для случайного генерирования Pet.

На этот раз при создании types блок try не нужен, так как необходимые проверки типов проводятся еще во время компиляции и исключения не возбуждаются, в отличие от метода Class.forName().

Теперь библиотека typeinfo.pets содержит две реализации PetCreator. Чтобы вторая реализация использовалась по умолчанию, мы можем создать *фасад* (façade), использующий LiteralPetCreator:

```
//· typeinfo/pets/Pets java
// Фасад для получения 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);
    }
} ///:~
```

При этом также обеспечиваются косвенные вызовы randomPet(), createArray() и arrayList().

Поскольку PetCount.countPets() получает аргумент PetCreator, мы можем легко проверить работу LiteralPetCreator (через представленный фасад):

```
//: typeinfo/PetCount2.java
import typeinfo.pets.*;

public class PetCount2 {
    public static void main(String[] args) {
            PetCount.countPets(Pets.creator);
    }
} /* (Выполните, чтобы увидеть результат) *///:~
```

Результат будет таким же, как у PetCount.java.

Динамический вызов instanceof

Метод Class.isInstance() позволяет выполнить динамическую проверку типа объекта. Благодаря ему в примере PetCount.java наконец-то можно будет избавиться от нагромождения 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 множественных instanceof:
                    for(Map.Entry<Class<? extends Pet>,Integer> pair
                                    entrySet())
                            if(pair.getKey().isInstance(pet))
                                    put(pair.getKey(), pair.getValue() + 1);
```

продолжение ⇩

```
                }
                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)) {
                        printnb(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 контейнер PetCounter Map заполняется типами из LiteralPetCreator.allTypes. При этом используется класс net.mindview.util.MapData, который получает Iterable (allTypesList) и константу (0 в данном случае) и заполняет Map ключами из allTypes со значениями 0. Без предварительного заполнения Map будут подсчитаны только случайно сгенерированные типы, но не базовые типы (такие, как Pet и Cat).

Как видите, метод isInstance() избавил нас от необходимости нагромождать конструкции с instanceof. Вдобавок теперь в программу можно легко добавить новые типы Pet — для этого следует просто изменить массив LiteralPet Creator.types; остальная часть программы не потребует правки (которая была бы неизбежна с операторами instanceof).

Метод toString() был перегружен для получения удобочитаемого вывода.

Рекурсивный подсчет

Контейнер Map в PetCount3.PetCounter был заполнен всеми классами Pet. Вместо предварительного заполнения карты мы также можем воспользоваться методом Class.isAssignableFrom() и создать обобщенный инструмент подсчета, не ограниченный подсчетом Pet:

```
//: net/mindview/util/TypeCounter.java
// Подсчет экземпляров в семействе 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 + " incorrect type: "
                                + type + ", should be type or subtype of "
                                + baseType);
              countClass(type);
     }
     private void countClass(Class<?> type) {
              Integer quantity = get(type);
              put(type, quantity == null ? 1 : quantity + 1);
              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 **Class** для своего аргумента, а затем использует **isAssignableFrom()** для проверки принадлежности объекта к интересующей вас иерархии. Метод **countClass()** сначала производит подсчет для точного типа класса, а затем, если **baseType** допускает присваивание из суперкласса, рекурсивно вызывает **countClass()** 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);
```

продолжение ⇩

```
    }
} /* Output: (Пример)
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 результатов, подсчитываются как базовые, так и конкретные типы.

Регистрация фабрик

У построения объектов иерархии Pet есть один недостаток: каждый раз, когда в иерархию включается новый тип Pet, вы должны добавить его в LiteralPetCreator.java. В системах с регулярным добавлением новых классов это может создать проблемы.

Первое, что приходит в голову, — добавить в каждый класс статический инициализатор, который добавлял бы свой класс в некий список. К сожалению, статические инициализаторы вызываются только при первой загрузке класса, поэтому возникает «порочный круг»: класс отсутствует в списке генератора, поэтому генератор не может создать объект этого класса, соответственно, класс не загрузится и не будет помещен в список.

По сути, вы вынуждены создать список вручную (разве что вы напишете утилиту, которая будет анализировать исходный код, а затем создавать и компилировать список). Вероятно, лучшее, что можно сделать, — это разместить список в одном централизованном, очевидном месте. Вероятно, лучшим местом для него будет базовый класс иерархии.

В этом разделе мы также внесем другое изменение: создание объекта будет передано самому классу с использованием паттерна *«метод-фабрика»*. Метод-фабрика может вызываться полиморфно и создает объект соответствующего типа. В следующей упрощенной версии методом-фабрикой является метод create() интерфейса Factory:

```
//: typeinfo/factory/Factory.java
package typeinfo.factory;
public interface Factory<T> { T create(); } ///:~
```

Обобщенный параметр T позволяет create() возвращать разные типы для разных реализаций Factory. Также при этом используется ковариантность возвращаемых типов.

В следующем примере базовый класс Part содержит список объектов-фабрик. Фабрики типов, которые должны создаваться методом createRandom(), «регистрируются» в базовом классе включением в список partFactories:

```
//: typeinfo/RegisteredFactories.java
// Регистрация фабрик 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 {
            // При вызове Collections addAll() выдается предупреждение
            // "unchecked generic array creation     for varargs parameter"
            partFactories.add(new FuelFilter Factory());
            partFactories add(new AirFilter Factory());
            partFactories add(new CabinAirFilter.Factory());
            partFactories add(new OilFilter Factory());
            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 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 {}
```

продолжение ⇩

```
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
*///:~
```

Не все классы иерархии рассчитаны на создание экземпляров; в нашем примере классы **Filter** и **Belt** существуют исключительно в целях классификации. Экземпляры этих классов не создаются — только одного из их субклассов. Если класс *должен* создаваться посредством **createRandom()**, он содержит внутренний класс **Factory**.

Хотя для включения всех фабрик в список можно воспользоваться вызовом **Collections.addAll()**, компилятор выдает предупреждение, поэтому я вернулся к вызовам **add()**. Метод **createRandom()** случайным образом выбирает объект фабрики из **partFactories** и вызывает его метод **create()** для получения нового объекта **Part**.

instanceof и сравнение 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. Вот пример, который показывает, в чем их различия:

```
//. typeinfo/FamilyVsExactType.java
// Различия между instanceof и 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 эквивалентность. Пример доказывает справедливость утверждения о том, что действие оператора instanceof и метода isInstance() одинаково. Совпадают и результаты работы операции сравнения == и метода equals(). Но сами тесты приводят к разным заключениям. В соответствии с концепцией типа instanceof дает ответ на вопрос: «Объект принадлежит этому классу или производному от него?» С другой стороны, сравнение объектов Class оператором == не затрагивает наследования — либо тип точно совпадает, либо нет.

Рефлексия: динамическая информация о классе

Если вы не знаете точный тип объекта, 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, к которым вы затем хотели бы применить динамическое определение типов (RTTI).

Сначала кажется, что это ограничение не столь существенно, но предположим, что у вас появилась ссылка на объект, который не находится в пространстве вашей программы. Более того, 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 представляет некоторый класс. Но компилятор ничего не знал об этом классе, когда обрабатывал вашу программу, как же его можно использовать?

В традиционных средах программирования такая задача показалась бы далекой от реальности. Однако границы мира программирования все больше расширяются и мы все чаще встречаемся с такими ситуациями. Во-первых, такие возможности требуются для компонентного программирования, которое служит основой для систем *быстрой разработки приложений* (Rapid Application Development, RAD). Это визуальный подход для создания программ (экран представлен в виде «формы»), где значки, представляющие визуальные компоненты, перетаскиваются на форму. Затем происходит настройка этих компонентов, они устанавливаются в некоторое состояние во время работы программы. Чтобы изменить состояние компонентов, необходимо некоторым образом создавать их экземпляры, просматривать их содержимое, считывать и записывать внутренние значения. Вдобавок компоненты с поддержкой событий графического интерфейса должны как-то рассказать о них, чтобы система быстрой разработки приложений помогла программисту реализовать поддержку этих событий. Механизм рефлексии предоставляет средства для получения информации о доступных методах и их именах. Такое компонентное программирование поддерживается и в Java, с помощью технологии JavaBeans.

Другая важная предпосылка поддержки динамической информации о классе — предоставление возможности создавать и использовать объекты на удаленных платформах. Этот механизм, называемый *удаленным вызовом методов* (Remote Method Invocation, RMI), позволяет программе на Java распределять свои объекты по нескольким машинам. Необходимость в удаленном вызове методов возникает по разным причинам: например, при выполнении задачи с интенсивными вычислениями можно сбалансировать нагрузку по доступным компьютерам. Иногда код, выполняющий определенные операции, размещается на одной машине, чтобы она стала общим хранилищем этих операций и любые изменения кода на такой машине автоматически распространялись на всех клиентов этого кода. (Интересный поворот — компьютер существует исключительно для того, чтобы упростить внесение изменений в программное обеспечение!) 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) поддерживает концепцию *рефлексии* (reflection), для которой 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. Объекты Constructor используются 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. (За подробной информацией обращайтесь к описанию класса Class в электронной 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 рефлексии нет ничего сверхъестественного. Когда вы используете рефлексию для работы 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 объект Class. Таким образом, файл .class для класса этого объекта должен быть доступен JVM либо в сети, либо в локальной системе. Таким образом, истинное различие между традиционным RTTI и рефлексией состоит в том, что при использовании RTTI файл .class открывается и анализируется компилятором. Другими словами, вы можете вызывать методы объекта «нормальным» способом. При использовании рефлексии файл .class во время компиляции недоступен; он открывается и обрабатывается системой выполнения.

Извлечение информации о методах класса

Рефлексия редко используется напрямую; она существует в языке в основном для поддержки других возможностей, таких как сериализация объектов и компоненты JavaBeans. Однако существуют ситуации, в которых динамическая информация о классе просто незаменима.

Для примера возьмем программу, выводящую на экран список методов некоторого класса. При просмотре исходного кода класса или его документации будут 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. Искать их и сложно, и долго[1]. К счастью, рефлексия позволяет написать простой инструмент, выводящий полную информацию о полном интерфейсе класса. Вот как он работает:

```
//: typeinfo/ShowMethods.java
// Использование рефлексии для вывода полного списка методов
// класс, в том числе и определенных в базовом классе.
// {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()).replaceAl
l(""));

                                 for(Constructor ctor · ctors)
                                          print(p.matcher(ctor.toString()).replaceAll("")
);
                                 lines = methods.length + ctors.length,
                        } else {
                                 for(Method method : methods)
```

[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.

```
                                        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);
                }
        }
} /* 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 для получения и анализа имен, аргументов и возвращаемых значений представляемых методов и конструкторов. Впрочем, также 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.

Результат, полученный от **Class.forName()**, не может быть известен во время компиляции, поэтому вся информация о сигнатуре методов становится доступной во время выполнения. Если вы тщательно изучите документацию по рефлексии из JDK, то увидите, что рефлексия позволяет установить необходимые аргументы и вызвать метод объекта, «абсолютно неизвестного» во время компиляции программы (чуть позже будут приведены соответствующие примеры). Скорее всего, вам эти возможности никогда не понадобятся, но сам факт их существования интересен.

Приведенный выше результат был получен из командной строки

```
java ShowMethods ShowMethods
```

На экран выводится открытый (public) конструктор по умолчанию, хотя в тексте программы такой конструктор не определяется. Тот конструктор, что имеется теперь в классе, автоматически сгенерирован 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), сгенерированный компилятором конструктор исчезнет из списка методов. Сгенерированный конструктор имеет тот же уровень доступа, что и его класс.

Также интересно запустить программу в виде

```
java ShowMethods java.lang.String
```

с передачей дополнительного параметра char, int, String и т. п.

Эта программа сэкономит вам немало времени при программировании, когда вы будете мучительно вспоминать, есть ли у этого класса определенный метод, если вам потребуется узнать, имеются ли у некоторого класса методы, возвращающие объекты Color, и т. д.

Динамические посредники

«Посредник» (proxy) принадлежит к числу основных паттернов проектирования. Он представляет собой объект, который подставляется на место «настоящего» объекта для расширения или модификации его операций. Приведу тривиальный пример, показывающий структуру посредника:

```
//. 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
*/// ~
```

Поскольку consumer() получает Interface, он не знает, что ему передается — «настоящий» объект (RealObject) или посредник (Proxy), потому что оба типа реализуют Interface. Объект Proxy, находящийся между клиентом и «настоящим» объектом, выполняет операции, а затем вызывает идентичные методы RealObject.

Посредник пригодится в любой ситуации, когда требуется отделить дополнительные операции от «настоящего» объекта, и особенно когда нужно легко переключаться из режима использования дополнительных операций в режим отказа от них (и наоборот — главной целью паттернов является инкапсуляция изменений, поэтому для оправдания их применения что-то должно изменяться). Допустим, вы хотите отслеживать вызовы методов RealObject, измерять затраты на эти вызовы, и т. д. Такой код не должен встраиваться в приложение, а посредник позволит легко добавить или убрать его по мере необходимости.

Динамические посредники Java развивают концепцию посредника — и объект посредника создается динамически, и обработка вызовов опосредованных методов тоже осуществляется динамически. Все вызовы, обращенные к динамическому посреднику, перенаправляются одному *обработчику*, который определяет, что это за вызов и как с ним следует поступить. Вот как выглядит пример SimpleProxyDemo.java, переписанный для динамического посредника:

```
//  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)
```

продолжение ⇩

```
                    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);
            // Вставляем посредника и вызываем снова:
            Interface proxy = (Interface)Proxy.newProxyInstance(
                    Interface.class.getClassLoader(),
                    new Class[]{ Interface.class },
                    new DynamicProxyHandler(real));
            consumer(proxy);
        }
    } /* Output.
    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 посредник создается вызовом статического метода **Proxy.newProxyInstance()**, которому должен передаваться загрузчик класса, список интерфейсов, которые должны реализовываться посредником (а не классов или абстрактных классов!), а также реализация интерфейса **InvocationHandler**. Динамический посредник перенаправляет все вызовы обработчику, поэтому конструктор обработчика обычно получает ссылку на «настоящий» объект для перенаправления ему запросов.

Метод **invoke()** получает объект посредника на случай, если ему понадобится определить, откуда поступил запрос — впрочем, обычно это несущественно. Будьте внимательны при вызове методов посредника из **invoke()**, потому что вызовы через интерфейс перенаправляются через посредника.

В общем случае вы выполняете опосредованную операцию, а затем используете **Method.invoke()** для перенаправления запроса опосредованному объекту с передачей необходимых аргументов. При этом некоторые вызовы методов могут отфильтровываться, а другие — проходить:

```
//: typeinfo/SelectingMethods.java
// Looking for particular methods in a dynamic proxy.
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("Посредник обнаружил интересный метод");
                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
Посредник обнаружил интересный метод
interesting bonobo
boring3
*///:~
```

В данном случае мы просто проверяем имена методов, но с таким же успехом можно анализировать другие аспекты сигнатуры и даже значения аргументов.

Вряд ли вам придется каждый день пользоваться динамическими посредниками, но они хорошо подходят для решения многих разновидностей задач.

Объекты с неопределенным состоянием

Если использовать для обозначения неопределенного состояния (то есть отсутствия) объекта встроенное значение null, то при каждом использовании ссылки придется проверять, не равна ли она null. Это быстро утомляет, а код получается излишне громоздким. Проблема заключается в том, что null не имеет собственного поведения, кроме выдачи NullPointerException при попытке выполнения с ним какой-либо операции. Иногда бывает полезно ввести понятие *объекта с неопределенным состоянием*[1], который принимает сообщения, но возвращает значение, свидетельствующее об отсутствии «настоящего» объекта. Таким образом, вы можете считать, что все объекты действительны, и вам не придется тратить время на проверки null (и читать полученный код).

Было бы интересно представить себе язык программирования, автоматически создающий объекты с неопределенным состоянием, но на практике они применяются не так уж часто — иногда проверки null оказывается достаточно, иногда можно уверенно считать, что значение null вам не попадется, а иногда даже обработка аномальных ситуаций через NullPointerException является допустимой. Наибольшую пользу объекты с неопределенным состоянием приносят «вблизи от данных», представляя сущности в пространстве задачи. Простой пример: во многих системах имеется класс Person, а в коде возникают ситуации, когда объект не представляет конкретную личность (или, по крайней мере, информация о ней недоступна); при традиционном подходе вам следовало бы проверить ссылку null. Также можно воспользоваться объектом с неопределенным состоянием, но, даже несмотря на то, что такой объект будет отвечать на все сообщения, на которые отвечает «настоящий» объект, все равно потребуется способ проверки его на «определенность». Проще всего определить для этого специальный интерфейс:

```
//. net/mindview/util/Null java
package net.mindview util;
public interface Null {} ///·~
```

Это позволяет instanceof обнаруживать объекты с неопределенным состоянием и, что еще важнее, не требует включения метода isNull() во все классы (в конце концов, это фактически будет другим способом выполнения RTTI — так почему бы сразу не воспользоваться встроенными средствами?):

```
//  typeinfo/Person java
// Класс с неопределенным состоянием объекта
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;
```

[1] Идея принадлежит Бобби Вульфу (Bobby Woolf) и Брюсу Андерсону (Bruce Anderson).

```
            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 объект с неопределенным состоянием является синглетным, поэтому он создается как экземпляр static final. Это возможно благодаря тому, что объект Person *неизменяем* — значения задаются в конструкторе, а затем читаются, но не могут изменяться (поскольку поля String по своей природе неизменяемы). Если вы захотите изменить NullPerson, его придется заменить новым объектом Person. Обратите внимание: для обнаружения обобщенной поддержки Null или более конкретного типа NullPerson можно использовать instanceof, но при синглетной архитектуре можно воспользоваться просто equals() или даже == для сравнения с Person.NULL.

Представьте, что вы собираетесь открыть новое предприятие, но, пока вакансии еще не заполнены, в каждой должности Position можно временно хранить «заполнитель» — объект Person с неопределенным состоянием:

```
//· 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;
```

продолжение ⇩

```
    }
} ///:~
```

Превращать Position в объект с неопределенным состоянием не обязательно, потому что существование Person.NULL подразумевает неопределенность Position (возможно, позднее выяснится, что явная поддержка неопределенного состояния для Position нужна, и вы добавите ее, но в соответствии с одним из канонов экстремального программирования в начальный проект следует включить «простейшее решение, которое будет работать», и включать новые функции лишь по мере возникновения реальной необходимости).

Теперь класс Staff может проверять объекты с неопределенным состоянием 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));
    }
    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 нам по-прежнему приходится проверять объекты на определенное состояние, что принципиально не отличается от проверки null, но в других местах, скажем, при преобразованиях toString()), лишние проверки не нужны; мы просто считаем, что ссылка на объект действительна.

Если вместо конкретных классов используются интерфейсы, для автоматического создания объектов с неопределенным состоянием можно воспользоваться динамическим посредником. Допустим, имеется интерфейс Robot, определяющий имя и модель робота, а также список List<Operation>, определяющий, какие операции выполняет робот. Операция состоит из описания и команды:

```
//: typeinfo/Operation.java

public interface Operation {
    String description();
    void command();
} ///.~
```

Чтобы воспользоваться услугами робота, следует вызвать метод 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("Название: " + r.name());
            System.out.println("Модель  " + r.model());
            for(Operation operation : r.operations()) {
                System.out.println(operation.description());
                operation.command();
            }
        }
    }
} ///.~
```

При этом используется вложенный класс, выполняющий проверку.
Теперь мы можем создать 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 + " может убирать снег";
                    }
                    public void command() {
                         System.out.println(name + " убирает снег");
                    }
               },
               new Operation() {
                    public String description() {
                         return name + " может колоть лед";
                    }
                    public void command() {
                         System.out.println(name + " колет лед");
                    }
               },
               new Operation() {
                    public String description() {
                         return name + " может чистить крышу";
                    }
                    public void command() {
                         System.out.println(name + " чистит крышу");
                    }
               }
          );
     }
     public static void main(String[] args) {
          Robot.Test.test(new SnowRemovalRobot("Slusher"));
     }
} /* Output:
Название: Slusher
Модель: SnowBot Series 11
Slusher может убирать снег
Slusher убирает снег
Slusher может колоть лед
Slusher колет лед
Slusher может чистить крышу
Slusher чистит крышу
*///:~
```

Предполагается, что существуют разные типы роботов, и для каждого типа **Robot** объект с неопределенным состоянием должен делать что-то особенное — в нашем примере выдавать информацию о конкетном типе **Robot**, представленном объектом. Эта информация перехватывается динамическим посредником:

```
//: typeinfo/NullRobot.java
// Использование динамического посредника для создания
```

```
// объекта с неопределенным состоянием
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"),
                       newNullRobot(SnowRemovalRobot.class)
              };
              for(Robot bot : bots)
                       Robot.Test.test(bot),
     }
} /* Output·
Название: SnowBee
Модель: SnowBot Series 11
SnowBee может убирать снег
SnowBee убирает снег
SnowBee может колоть лед
SnowBee колет лед
SnowBee может чистить крышу
SnowBee чистит крышу
[Null Robot]
Название. SnowRemovalRobot NullRobot
Модель: SnowRemovalRobot NullRobot
*///.~
```

Каждый раз, когда вам требуется объект **Robot** с неопределенным состоянием, вы вызываете **newNullRobot()** и передаете тип **Robot**, для которого создается

посредник. Посредник выполняет требования о поддержке интерфейсов Robot и Null, а также предоставляет имя опосредованного типа.

Интерфейсы и информация о типах

Одной из важных целей ключевого слова interface является изоляция компонентов и сокращение привязок. Использование интерфейсов вроде бы позволяет добиться этой цели, однако RTTI позволяет обойти ограничения — интерфейсы не обеспечивают стопроцентной изоляции. Начнем следующий пример с интерфейса:

```
//: typeinfo/interfacea/A.java
package typeinfo.interfacea;

public interface A {
     void f();
} ///:~
```

Затем интерфейс реализуется, и выясняется, что можно «в обход» добраться до фактического типа реализации:

```
//· typeinfo/InterfaceViolation.java
// Интерфейс можно обойти
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 выясняем, что объект a реализован в форме B. Преобразование к типу B позволяет вызвать метод, не входящий в интерфейс A.

Все это абсолютно законно и допустимо, но, скорее всего, вы предпочли бы оградить клиентских программистов от подобных выходок. Казалось бы, ключевое слово interaface должно защищать вас, но на самом деле этого не происходит, а факт использования B для реализации A становится известен любому желающему.

Одно из возможных решений: просто скажите программистам, что если они будут использовать фактический класс вместо интерфейса, то пускай сами разбираются со всеми возникающими проблемами. Вероятно, во многих случаях этого достаточно, но если «вероятно» вас не устраивает — можно применить более жесткие меры.

Проще всего установить для реализации пакетный уровень доступа, чтобы она оставалась невидимой для клиентов за пределами пакета:

```
//. 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 пакета, HiddenC, выдает интерфейс A при вызове. Интересно отметить, что, даже если makeA() будет возвращать C, за пределами пакета все равно удастся использовать только A, потому что имя C недоступно.

Попытка нисходящего преобразования к C тоже завершается неудачей:

```
//: typeinfo/HiddenImplementation.java
// Пакетный доступ тоже можно обойти
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. символическое имя 'C' не найдено
            /* if(a instanceof C) {
                    C c = (C)a;
                    c.g();
            } */
            // Однако рефлексия позволяет вызвать g():
            callHiddenMethod(a. "g");
            // ... И даже еще менее доступные методы!
            callHiddenMethod(a. "u");
            callHiddenMethod(a. "v");
            callHiddenMethod(a. "w");
     }
     static void callHiddenMethod(Object a. String methodName)
     throws Exception {
```

продолжение

```
                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 вызвать *все* методы, даже приватные! Зная имя метода, можно вызвать **setAccessible(true)** для объекта **Method**, чтобы сделать возможным его вызов, как видно из реализации **callHiddenMethod()**.

Можно подумать, что проблема решается распространением только откомпилированного кода, но и это не так. Достаточно запустить **javap** — декомпилятор, входящий в JDK. Командная строка выглядит так:

```
javap -private C
```

Флаг **-private** означает, что при выводе должны отображаться все члены, даже приватные. Таким образом, любой желающий сможет получить имена и сигнатуры приватных методов и вызвать их.

А если реализовать интерфейс в виде приватного внутреннего класса? Вот как это выглядит:

```
//: typeinfo/InnerImplementation.java
// Приватные внутренние классы не скрываются от рефлексии
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());
                // Рефлексия все равно позволяет добраться до приватного класса:
                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()
*/// ~
```

Не помогло. Как насчет анонимного класса?

```
//  typeinfo/AnonymousImplementation.java
// Анонимные внутренние классы тоже не скрыты от рефлексии
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());
          // Рефлексия все равно позволяет добраться до приватного класса.
          HiddenImplementation.callHiddenMethod(a, "g");
          HiddenImplementation.callHiddenMethod(a, "u");
          HiddenImplementation.callHiddenMethod(a, "v");
          HiddenImplementation.callHiddenMethod(a, "w");
     }
} /* Output:
public C.f()
AnonymousA$1
public C.g()
package C.u()
protected C.v()
private C.w()
*///·~
```

Похоже, не существует никакого способа предотвратить обращение и вызов методов с уровнем доступа, отличным от public, посредством рефлексии. Сказанное относится и к полям данных, даже к приватным:

```
//. 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?";
```

продолжение ⇩

```
        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 защищены от изменений. Система времени выполнения спокойно воспринимает любые попытки их изменения, но при этом ничего не происходит.

В действительности все эти нарушения уровня доступа не так уж страшны. Если кто-то захочет вызывать методы, которым вы назначили приватный или пакетный доступ (тем самым ясно показывая, что вызывать их не следует), вряд ли он станет жаловаться на то, что вы изменили некоторые аспекты этих методов. С другой стороны, «черный ход» к внутреннему устройству класса позволяет решить некоторые проблемы, нерешаемые другими средствами, и в общем случае преимущества рефлексии неоспоримы.

Резюме

Динамическое определение типов (RTTI) позволяет вам получить информацию о точном типе объекта тогда, когда у вас для него имеется лишь ссылка базового типа. Таким образом, оно открывает широкие возможности для злоупотреблений со стороны новичков, которые еще не поняли и не успели оценить всю мощь полиморфизма. У многих людей, ранее работавших с процедурными

языками, возникает сильное желание разбить свою программу на множество конструкций switch при помощи RTTI. Однако при этом они лишаются всех преимуществ полиморфизма, относящихя к разработке программы в целом и ее дальнейшей поддержке. В Java рекомендуется использовать именно полиморфные методы, а к услугам RTTI следует прибегать только в крайнем случае.

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 RTTI вас выручит: 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 своем коде в подходящий момент вы обнаруживаете новый тип и вызываете для него нужный метод. Такой подход не противоречит основам полиморфизма и расширяемости программы, так как добавление в программу нового типа не требует изменения бесчисленного множества конструкций switch. Но чтобы извлечь пользу из дополнительной функциональности нового класса, придется использовать RTTI.

Включение некоторого 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». Интерфейс базового класса «размывается» 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. На первый взгляд можно было бы 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 писать программы, ориентируясь только на их эффективность, как бы соблазнительно это ни было. Сначала надо получить *работающую* программу и только после этого решать, достаточно ли быстро она работает, и решать проблемы быстродействия, вооружившись инструментами для проверки скорости исполнения.

Мы также видели, что рефлексия открывает перед программистом множество новых возможностей и делает возможным более динамичный стиль программирования. Пожалуй, динамическая природа рефлексии кому-то покажется пугающей. Для тех, кто привык к безопасной статической проверке типов, сама возможность выполнения действий, правильность которых проверяется только на стадии выполнения, а для выдачи информации используются исключения, выглядит шагом в неверном направлении. Некоторые доходят до утверждений, будто сама возможность исключения на стадии выполнения свидетельствует о том, что такого кода лучше избегать. На мой взгляд, чувство безопасности весьма иллюзорно — неожиданности и исключения возможны всегда, даже если программа не содержит блоков try и спецификации исключений. Предпочитаю думать, что существование логически целостной модели выдачи информации об ошибках *дает возможность* писать динамический код с использованием рефлексии. Конечно, всегда желательно писать код со статической проверкой... когда это возможно. И все же динамический код является одной из важнейших особенностей, отделяющих Java от таких традиционных языков, как C++.

Параметризация 14

Обычные классы и методы работают с конкретными типами: либо, примитивами, либо с классами. Если ваш код должен работать с разными типами, такая жесткость может создавать проблемы.

Одним из механизмов обеспечения универсальности кода в объектно-ориентированных языках является полиморфизм. Например, вы можете написать метод, который получает в аргументе объект базового класса, а затем использует этот метод с любым классом, производным от него. Метод становится чуть более универсальным, а область его применения расширяется. Это относится и к классам — использование базового класса вместо производного обеспечивает дополнительную гибкость. Конечно, наследование возможно только для классов, не являющихся final.

Впрочем, иногда даже рамки одной иерархии оказываются слишком тесными. Если в аргументе метода передается интерфейс вместо класса, то ограничения ослабляются и в них включается все, что реализует данный интерфейс, — в том числе и классы, которые еще не были созданы. Это дает программисту-клиенту возможность реализовать интерфейс, чтобы соответствовать требованиям вашего класса или метода. Таким образом, интерфейсы позволяют выходить за рамки иерархий классов, если только у вас имеется возможность создать новый класс.

Но в некоторых случаях даже интерфейсы оказываются недостаточно гибкими. Интерфейс требует, чтобы ваш код работал в этом конкретном интерфейсе. Если бы было можно указать, что ваш код работает «с некоторым не заданным типом», а не с конкретным интерфейсом или классом, программа приобрела бы еще более общий характер.

В этом и состоит концепция *параметризации* — одного из самых значительных новшеств Java SE5. *Параметризованные типы* позволяют создавать компоненты (прежде всего, контейнеры), которые могут легко использоваться

с разными типами. Если прежде вы еще никогда не встречались с механизмом параметризации в действии, вероятно, параметризованные типы Java покажутся вам довольно удобным дополнением к языку. При создании экземпляра параметризованного типа преобразования типа выполняются автоматически, а правильность типов проверяется на стадии компиляции. С другой стороны, разработчики с опытом использования параметризованных типов в других языках (скажем, в C++) увидят, что в Java они не соответствуют всем ожиданиям. Если использовать готовый параметризованный тип относительно несложно, при попытке написать собственный тип вас ждут сюрпризы. В частности, в этой главе я постараюсь объяснить, почему параметризованные типы Java получились именно такими.

Не стоит думать, что параметризованные типы Java бесполезны — во многих случаях они делают код более четким и элегантным. Но, если вы работали на другом языке, в котором они были реализованы более «чисто», вас могут ждать разочарования. В этой главе мы изучим как достоинства, так и недостатки параметризованных типов Java, чтобы вы могли использовать эту новую возможность более эффективно.

Простая параметризация

Одной из важнейших причин для появления параметризации стало создание классов *контейнеров* (см. главу 11). Контейнер предназначен для хранения объектов, используемых в программе. В принципе это описание подойдет и для массива, но контейнеры обычно обладают большей гибкостью и отличаются по своим характеристикам от простых массивов. Необходимость хранения групп объектов возникает едва ли не в каждой программе, поэтому контейнеры составляют одну из самых часто используемых библиотек классов.

Рассмотрим класс для хранения одного объекта. Конечно, в этом классе можно указать точный тип объекта:

```
//· generics/Holder1 java

class Automobile {}

public class Holder1 {
     private Automobile a;
     public Holder1(Automobile a) { this.a = a; }
     Automobile get() { return a; }
} ///:~
```

Однако такой «контейнер» получается не слишком универсальным — он не может использоваться только для одного типа. Конечно, было бы неудобно создавать новый класс для каждого типа, который нам встретится в программе.

До выхода Java SE5 можно было бы хранить в классе `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("Не Automobile");
            String s = (String)h2.get();
            h2.set(1); // Автоматически упаковывается в Integer
            Integer x = (Integer)h2.get();
    }
} ///:~
```

Теперь класс **Holder2** может хранить все, что угодно, — в приведенном примере один объект **Holder2** используется для хранения трех разных типов данных.

В некоторых случаях бывает нужно, чтобы контейнер мог хранить объекты разных типов, но чаще контейнер предназначается для одного типа объектов. Одна из главных причин для применения параметризованных типов заключается именно в этом: вы можете указать, какой тип должен храниться в контейнере, и заданный тип будет поддерживаться комплиятором.

Итак, вместо **Object** в определении класса было бы удобнее использовать некий условный заменитель, чтобы отложить выбор до более позднего момента. Для этого после имени класса в угловых скобках указывается *параметр типа*, который при использовании заменяется фактическим типом. В нашем примере это будет выглядеть так (**T** — параметр типа):

```
//: 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("Не Automobile"); // Ошибка
            // h3.set(1); // Ошибка
    }
} ///:~
```

При создании **Holder3** необходимо указать тип объектов, хранящихся в контейнере, в угловых скобках, как в **main()**. В дальнейшем в контейнер можно будет помещать объекты только этого типа (или производного, так как принцип заменяемости работает и для параметризованных типов). А при извлечении вы автоматически получаете объект нужного типа.

В этом заключается основная идея параметризованных типов Java: вы указываете, какой тип должен использоваться, а механизм параметризации берет на себя все подробности.

Кортежи

При вызове метода часто требуется, чтобы метод возвращал несколько объектов. Команда **return** позволяет вернуть только один объект, поэтому проблема

решается созданием объекта, содержащего несколько возвращаемых объектов. Конечно, можно создавать специальный класс каждый раз, когда возникает подобная ситуация, но параметризованные типы позволяют решить проблему один раз и избавиться от хлопот в будущем. Заодно решается проблема безопасности типов на стадии компиляции.

Концепция нескольких объектов, «упакованных» в один объект, называется *кортежем* (tuple). Получатель объекта может читать элементы, но не может добавлять их (эта концепция еще называется *объектом передачи данных*).

Обычно кортеж может иметь произвольную длину, а все объекты кортежа могут относиться к разным типам. Однако мы хотим задать тип каждого объекта и при этом гарантировать, что при чтении значения будет получен правильный тип. Для решения проблемы переменной длины мы создадим несколько разных кортежей. Вот один из них, рассчитанный на 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 запоминает сохраняемый объект, а вспомогательная функция `toString()` выводит значения из списка. Обратите внимание: кортеж подразумевает упорядоченное хранение элементов.

При первом чтении может показаться, что такая архитектура нарушает общие принципы безопасности программирования на Java. Разве `first` и `second` не должны быть объявлены приватными, а обращения к ним осуществляться только из методов `getFirst()` и `getSecond()`? Подумайте, какая безопасность реализуется в этом случае: клиент может читать объекты и делать с прочитанными значениями все, что пожелает, но не может изменить `first` и `second`. Фактически объявление `final` делает то же самое, но короче и проще.

Кортежи большей длины создаются посредством наследования. Добавить новый параметр типа 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 воспользоваться этими классами, достаточно определить кортеж нужной длины как возвращаемое значение функции, а затем создать и вернуть его командой 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() {
```

продолжение ⇨

```
                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:
(hi, 47)
(Amphibian@1f6a7b9, hi, 47)
(Vehicle@35ce36, Amphibian@757aef, hi, 47)
(Vehicle@9cab16, Amphibian@1a46e30, hi, 47, 11.1)
*///:~
```

Спецификация final для public-полей предотвращает их изменение после конструирования (поэтому попытка выполнения команды ttsi.first="there" приводит к ошибке).

Конструкции new получаются немного громоздкими. Позднее в этой главе будет показано, как упростить их при помощи *параметризованных методов*.

Класс стека

Давайте рассмотрим менее тривиальный пример: реализацию традиционного стека. В главе 11 была приведена реализация стека на базе LinkedList. В этом примере класс LinkedList уже содержал все методы, необходимые для создания стека. Класс стека строился объединением одного параметризованного класса (Stack<T>) с другим параметризованным классом (LinkedList<T>). Этот пример показывает, что параметризованный тип — такой же тип, как и все остальные (за некоторыми исключениями, о которых речь пойдет позже):

Вместо того, чтобы использовать LinkedList, мы также могли реализовать собственный механизм хранения связанного списка:

```
//: generics/LinkedStack.java
// Стек, реализованный на базе внутренней структуры

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>(); // Предохранитель
        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 параметризованным и имеет собственный параметр типа.

Для определения наличия элементов в стеке в этом примере используется *предохранитель* (end sentinel). Он создается при конструировании LinkedStack, а затем при каждом вызове push() новый объект Node<T> создается и связывается с предыдущим Node<T>. При вызове pop() всегда возвращается top.item, после чего текущий объект Node<T> уничтожается и происходит переход к следующему — если только текущим элементом не является предохранитель; в этом случае переход не выполняется. При повторных вызовах pop() клиент будет получать null, что свидетельствует об отсутствии элементов в стеке.

RandomList

Рассмотрим еще один пример контейнера: допустим, вам понадобилась особая разновидность списка, которая случайным образом выбирает один из своих элементов при вызове select(). Так как класс должен работать для любых объектов, мы воспользуемся параметризацией:

```
//: 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);
```

продолжение ⇩

```
            for(int i = 0; i < 11; i++)
                    System.out.print(rs.select() + " ");
    }
} /* Output:
brown over fox quick quick dog brown The brown lazy brown
*///.~
```

Параметризованные интерфейсы

Параметризация работает и с интерфейсами. Например, класс, создающий объекты, называется *генератором*. В сущности, генератор представляет собой специализированную версию паттерна *«метод-фабрика»*, но при обращении к нему никакие аргументы не передаются, тогда как метод-фабрика обычно получает аргументы. Генератор умеет создавать объекты без дополнительной информации.

Обычно генератор определяет всего один метод — тот, который создает объекты. Назовем его next() и включим в стандартный инструментарий:

```
//· net/mindview/util/Generator.java
// Параметризованный интерфейс
package net.mindview.util;
public interface Generator<T> { T next(); } ///:~
```

Возвращаемое значение метода next() параметризовано по типу T. Как видите, механизм параметризации работает с интерфейсами почти так же, как с классами.

Чтобы продемонстрировать, как работает реализация Generator, мы воспользуемся иерархией классов, представляющих 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 class Americano extends Coffee {} /// ~

//. generics/coffee/Breve java
package generics coffee.
public class Breve extends Coffee {} ///:~
```

Теперь мы можем реализовать интерфейс Generator<Coffee>, который создает случайные типы объектов из иерархии Coffee:

```
//  generics/coffee/CoffeeGenerator.java
// Генератор случайных объектов из иерархии 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().
                       // Сообщение 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
```

продолжение ⇩

```
Mocha 3
Mocha 4
Breve 5
Americano 6
Latte 7
Cappuccino 8
Cappuccino 9
*///:~
```

Параметризованный интерфейс Generator гарантирует, что next() вернет параметр типа. CoffeeGenerator также реализует интерфейс Iterable и поэтому может использоваться в синтаксисе foreach. Аргумент, по которому определяется момент прекращения перебора, передается при вызове второго конструктора.

А вот как выглядит другая реализация Generator<T>, предназначенная для получения чисел Фибоначчи:

```
//. generics/Fibonacci.java
// Построение чисел 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 снаружи класса мы работаем с int, в параметре типа передается Integer. В этом проявляется одно из ограничений параметризации в языке Java: примитивные типы не могут использоваться в качестве параметров типа. Впрочем, в Java SE5 была добавлена удобная автоматическая упаковка (распаковка) для перехода от примитивных типов к объектным «оберткам», и наоборот.

Можно сделать следующий шаг вперед и создать генератор чисел Фибоначчи с реализацией Iterable. Конечно, можно изменить реализацию класса и добавить интерфейс Iterable, но исходные коды не всегда находятся в вашем распоряжении, и вообще там, где это возможно, лучше обойтись без их модификации. Вместо этого мы воспользуемся *«адаптером»* для получения нужного интерфейса (этот паттерн уже упоминался ранее в книге).

Существует несколько вариантов реализации адаптеров. Например, для получения адаптируемого класса можно воспользоваться наследованием:

```
//: generics/IterableFibonacci.java
// Adapt the Fibonacci class to make it Iterable.
import java.util.*;
```

```
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
*///.~
```

Для использования IterableFibonacci в синтаксисе foreach мы передаем конструктору границу, чтобы метод hasNext() знал, когда следует возвращать false.

Параметризованные методы

До настоящего момента мы рассматривали параметризацию целых классов, однако параметризация может применяться и к отдельным методам классов. Сам класс при этом может быть параметризованным, а может и не быть — это не зависит от наличия параметризованных методов.

Параметризованный метод может изменяться независимо от класса. В общем случае параметризованные методы следует использовать «по мере возможности». Иначе говоря, если возможно параметризовать метод вместо целого класса, вероятно, стоит выбрать именно этот вариант. Кроме того, статические методы не имеют доступа к параметрам типа параметризованных классов; если такие методы должны использовать параметризацию, это должно происходить на уровне метода, а не на уровне класса.

Чтобы определить параметризованный метод, следует указать список параметров 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("");
```

продолжение ⇩

```
            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 параметризованными одновременно. Но в данном случае только метод f() имеет параметр типа, обозначаемый списком параметров перед возвращаемым значением метода.

Учтите, что при использовании параметризованного класса параметры типов должны указываться при создании экземпляра. Но при использовании параметризованного метода указывать параметры типа не обязательно, потому что компилятор способен «вычислить» их за вас. Таким образом, вызов f() выглядит как обычный вызов метода; создается впечатление, что метод f() существует в бесконечном количестве перегруженных версий. При вызове ему даже может передаваться аргумент типа GenericMethods.

Для вызовов f(), использующих примитивные типы, в действие вступает механизм автоматической упаковки — примитивные типы автоматически преобразуются в соответствующие объекты. Это позволяет исключить некоторые фрагменты кода, которые были необходимы прежде из-за явного выполнения преобразований.

Вычисление типа аргумента

Параметризацию иногда упрекают в том, что она увеличивает объем кода. Для наглядности возьмем пример holding/MapOfList.java из главы 11. Создание контейнера Map с List выглядит так:

```
Map<Person, List<? extends Pet>> petPeople =
  new HashMap<Person, List<? extends Pet>>();
```

(Ключевое слово extends и вопросительные знаки будут описаны позднее в этой главе.) Казалось бы, эта конструкция избыточна, а компилятор мог бы вычислить один из списков аргументов по-другому. В действительности сделать это он не может, но вычисление аргументов типов все же позволяет немного упростить код. Например, мы можем создать вспомогательную библиотеку с различными статическими методами, содержащими самые распространенные реализации различных контейнеров:

```
//: net/mindview/util/New.java
// Utilities to simplify generic container creation
// by using type argument inference.
```

```
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();
    }
} ///:~
```

Примеры использования представлены в `main()` — вычисление аргументов типов устраняет необходимость в повторении списков параметров. Этот прием можно использовать 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();
        // Остальное без изменений...
    }
} ///:~
```

Пример интересный, однако трудно сказать, насколько он эффективен в действительности. Человеку, читающему код, придется просмотреть дополнительную библиотеку и разобраться в ее коде. Возможно, вместо этого стоит оставить исходное (пусть и избыточное) определение — как ни парадоксально, этот вариант проще. Хотя, если в стандартную библиотеку Java будет добавлено некое подобие **New.java**, им можно будет пользоваться.

Вычисление типов не работает ни в каких других ситуациях, кроме присваивания. Если передать результат вызова метода (скажем, `New.map()`) в аргументе другого метода, компилятор *не пытается* выполнить вычисление типа. Вместо

этого вызов метода интерпретируется так, как если бы возвращаемое значение присваивалось переменной типа Object. Пример ошибки такого рода:

```
//: 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
     }
} ///:~
```

Явное указание типа

При вызове параметризованного метода также можно явно задать тип, хотя на практике этот синтаксис используется редко. Тип указывается в угловых скобках за точкой, непосредственно перед именем метода. При вызове метода в пределах класса необходимо ставить this перед точкой, а при вызове статических методов перед точкой указывается имя класса. Проблема, продемонстрированная в LimitsOfInference.java, решается при помощи следующего синтаксиса:

```
//: 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 от использования класса New для уменьшения объема кода, но дополнительный синтаксис необходим только за пределами команд присваивания.

Параметризованные методы и переменные списки аргументов

Параметризованные методы нормально сосуществуют с переменными списками аргументов:

```
//: 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() предоставляет ту же функциональность, что и метод java.util.Arrays.asList() из стандартной библиотеки.

Использование параметризованных методов с Generator

Генераторы хорошо подходят для заполнения Collection, и для выполнения этой операции было бы удобно создать параметризованный метод:

```
//: generics/Generators.java
// Обобщенный метод заполнения коллекции
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 то, как параметризованный метод fill() применяется к контейнерам и генераторам как для типа Coffee, так и для Integer.

Обобщенный генератор

Следующий класс создает генератор для любого класса, обладающего конструктором по умолчанию. Для уменьшения объема кода в него также включен параметризованный метод для получения 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 является public-классом.
                       return type.newInstance();
              } catch(Exception e) {
                       throw new RuntimeException(e);
              }
     }
     // Получение генератора по умолчанию для заданного type:
     public static <T> Generator<T> create(Class<T> type) {
              return new BasicGenerator<T>(type);
     }
} ///:~
```

Класс предоставляет базовую реализацию, создающую объекты класса, который (1) является открытым (так как BasicGenerator определяется в отдельном пакете, соответствующий класс должен иметь уровень доступа public, не ограничиваясь пакетным доступом), и (2) обладает конструктором по умолчанию (то есть конструктором без аргументов). Чтобы создать один из таких объектов BasicGenerator, следует вызвать метод create() и передать ему обозначение генерируемого типа. параметризованный метод create() позволяет использовать запись BasicGenerator.create(MyType.class) вместо более громоздкой конструкции new BasicGenerator<MyType>(MyType.class).

Для примера рассмотрим простой класс с конструктором по умолчанию:

```
//: generics/CountedObject.java

public class CountedObject {
     private static long counter = 0;
     private final long id = counter++;
     public long id() { return id; }
     public String toString() { return "CountedObject " + id;}
} ///:~
```

Класс CountedObject отслеживает количество созданных экземпляров и включает его в выходные данные 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
*/// ~
```

Как видите, применение параметризованного метода снижает объем кода, необходимого для получения объекта Generator. Раз уж механизм параметризации Java все равно заставляет вас передавать объект Class, его можно заодно использовать для вычисления типа в методе create().

Упрощение работы с кортежами

Используя вычисление аргументов типов в сочетании со static-импортом, можно оформить приведенную ранее реализацию кортежей в более универсальную библиотеку. В следующем примере кортежи создаются перегруженным статическим методом:

```
//· net/mindview/util/Tuple.java
// Библиотека для работы с кортежами
// с использованием вычисления 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);
     }
} ///:~
```

А вот как выглядит обновленная версия TupleTest.java для тестирования Tuple.java:

```
//: generics/TupleTest2.java
import net.mindview.util.*;
import static net.mindview.util.Tuple.*;
```

продолжение ⇩

```
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
(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, а f2() — непараметризованный объект TwoTuple. Компилятор в данном случае не выдает предупреждения о f2(), потому что возвращаемое значение не используется в «параметризованном» стиле: в каком-то смысле проводится «восходящее преобразование» его до непараметризованного TwoTuple. Но, если попытаться сохранить результат f2() в параметризованном объекте TwoTuple, компилятор выдаст предупреждение.

Вспомогательный класс Set

Рассмотрим еще один пример использования параметризованных методов: математические операции между множествами. Эти операции удобно определить в виде параметризованных методов, используемых с различными 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;
        }
        // Дополнение -- 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, копируя его ссылки в новый объект **HashSet**, поэтому аргументы **Set** не изменяются напрямую. Таким образом, возвращаемое значение представляет собой новый объект **Set**.

Четыре метода представляют математические операции с множествами: **union()** возвращает объект **Set**, полученный объединением множеств-аргументов, **intersection()** возвращает объект **Set** с общими элементами аргументов, **difference()** вычисляет разность множеств, а **complement()** — объект **Set** со всеми элементами, не входящими в пересечение. Чтобы создать простой пример использования этих методов, мы воспользуемся перечислением, содержащим разные названия акварельных 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 избежать уточнения всех имен) в следующем примере это перечисление импортируется статически. Мы используем **EnumSet** — новый инструмент Java SE5 для простого создания **Set** на базе перечисления. Статическому методу **EnumSet.range()** передаются первый и последний элементы диапазона, по которому строится множество:

```
//: generics/WatercolorSets.java
import generics.watercolors.*;
import java.util.*;
import static net.mindview.util.Print.*;
```

продолжение ⇨

```
import static net.mindview.util.Sets *;
import static generics.watercol@rs.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.
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 показаны результаты выполнения каждой операции.

В следующем примере представлены варианты вызова **Sets.difference()** для разных 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);
    }
} ///:~
```

Анонимные внутренние классы

Параметризация также может применяться к внутренним классам и анонимным внутренним классам. Пример реализации интерфейса Generator с использованием анонимных внутренних классов:

```
//: generics/BankTeller.java
// Очень простая имитация банковского обслуживания.
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(); }
```

продолжение ⇩

```
                };
        }
}

class Teller {
        private static long counter = 1;
        private final long id = counter++;
        private Teller() {}
        public String toString() { return "Teller " + id; }
        // Синглетный 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 + " обслуживает " + 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 обслуживает Customer 1
Teller 2 обслуживает Customer 2
Teller 3 обслуживает Customer 3
Teller 1 обслуживает Customer 4
Teller 1 обслуживает Customer 5
Teller 3 обслуживает Customer 6
Teller 1 обслуживает Customer 7
Teller 2 обслуживает Customer 8
Teller 3 обслуживает Customer 9
Teller 3 обслуживает Customer 10
Teller 2 обслуживает Customer 11
Teller 4 обслуживает Customer 12
Teller 2 обслуживает Customer 13
Teller 1 обслуживает Customer 14
Teller 1 обслуживает Customer 15
*///·~
```

И Customer, и Teller содержат приватные конструкторы, поэтому для создания их объектов пользователь вынужден использовать объекты Generator. Customer содержит метод generator(), который при каждом вызове создает новый объект Generator<Customer>. На случай, если множественные объекты Generator вам не понадобятся, в Teller создается синглетный открытый объект generator. Оба подхода продемонстрированы в вызовах fill() внутри main().

Поскольку метод generator() в Customer и объект Generator в Teller являются статическими, они не могут быть частью интерфейса, поэтому «обобщить» эту

конкретную идиому не удастся. Несмотря на это обстоятельство, она достаточно хорошо работает в методе fill().

Построение сложных моделей

К числу важных преимуществ параметризации относится простота и надежность создания сложных моделей. Например, можно легко создать список (List) с элементами-кортежами:

```
//: generics/TupleList.java
// Построение сложных параметризованных типов путем объединения
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:
(Vehicle@11b86e7, Amphibian@35ce36, hi, 47)
(Vehicle@757aef, Amphibian@d9f9c3, hi, 47)
*///.~
```

Запись получается довольно громоздкой (особенно при создании итератора), однако вы получаете довольно сложную структуру данных без излишков программного кода.

А вот другой пример, который показывает, как легко строить сложные модели на основе параметризованных типов. Хотя каждый класс представляет собой автономный «строительный блок», их совокупность имеет сложную структуру. В данном случае моделируется магазин с товарами, полками и стеллажами:

```
//. generics/Store.java
// Построение сложной модели на базе параметризованных 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 + ", цена: $" + price;
```

продолжение ⇩

```
        }
        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. цена: $400.99
861· Test. цена· $160.99
868: Test. цена: $417.99
207· Test. цена· $268.99
551· Test. цена. $114.99
278: Test. цена: $804.99
```

```
520: Test, цена: $554.99
140: Test, цена: $530.99
...
*///:~
```

Как видно из Store.toString(), в результате мы получаем многоуровневую архитектуру контейнеров, не лишаясь преимуществ безопасности типов и управляемости. Впечатляет и то, что построение такой модели не потребует заметных умственных усилий.

Тайна стирания

Когда вы приступаете к более глубокому изучению контейнеров, некоторые обстоятельства на первых порах выглядят довольно странно. Например, запись ArrayList.class возможна, а запись ArrayList<Integer>.class — нет. Или возьмите следующий фрагмент:

```
//: 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
*///.~
```

Было бы логично считать, что ArrayList<String> и ArrayList<Integer> — разные типы, поэтому их поведение должно различаться, и при попытке поместить Integer в ArrayList<String> результат (неудача) должен отличаться от того, который будет получен при помещении Integer в ArrayList<Integer> (успех). Однако эта программа создает впечатление, что эти типы одинаковы.

Следующий пример еще сильнее запутывает ситуацию:

```
//. 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())),
```

продолжение ⇩

```
                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, **Class.getTypeParameters()** «возвращает массив объектов **TypeVariable**, представляющих переменные типов, указанные в параметризованном объявлении...» Казалось бы, по ним можно определить параметры типов — но, как видно из результатов, вы всего лишь узнаете, какие идентификаторы использовались в качестве заполнителей, а эта информация не представляет особого интереса.

Мы приходим к холодной, бездушной истине:

Информация о параметрах типов недоступна внутри параметризованного кода.

Таким образом, вы можете узнать идентификатор параметра типа и ограничение параметризованного типа, но фактические параметры типов, использованные для создания конкретного экземпляра, остаются неизвестными. Этот факт, особенно раздражающий программистов с опытом работы на C++, является основной проблемой, которую приходится решать при использовании параметризации в Java.

Параметризация в Java реализуется с применением *стирания* (erasure). Это означает, что при использовании параметризации вся конкретная информация о типе утрачивается. Внутри параметризованного кода вы знаете только то, что используется некий объект. Таким образом, **List<String>** и **List<Integer>** действительно *являются* одним типом во время выполнения; обе формы «стираются» до своего *низкоуровневого типа* **List**. Именно стирание и создаваемые им проблемы становятся главной преградой при изучении параметризации в Java; этой теме и будет посвящен настоящий раздел.

Подход C++

В следующем примере, написанном на C++, используются *шаблоны*. Синтаксис параметризованных типов выглядит знакомо, потому что многие идеи C++ были взяты за основу при разработке Java:

```
//: 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 узнает, что у параметра типа T существует метод f()? Компилятор C++ выполняет проверку при создании экземпляра шаблона, поэтому в точке создания Manipulator<HasF> он узнает о том, что HasF содержит метод f(). В противном случае компилятор выдает ошибку, а безопасность типов сохраняется.

Написать такой код на C++ несложно, потому что при создании экземпляра шаблона код шаблона знает тип своих параметров. С параметризацией Java дело обстоит иначе. Вот как выглядит версия HasF, переписанная на Java:

```
//. generics/HasF.java

public class HasF {
     public void f() { System.out.println("HasF.f()"); }
} ///:~
```

Если мы возьмем остальной код примера и перепишем его на Java, он не будет компилироваться:

```
//: generics/Manipulation.java
// {CompileTimeError} (Не компилируется)

class Manipulator<T> {
     private T obj;
     public Manipulator(T x) { obj = x; }
     // Ошибка: не удается найти символическое имя: 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 сопоставить требование о возможности вызова f() для obj из manipulate() с тем фактом, что HasF содержит метод f(). Чтобы вызвать f(), мы должны «помочь» параметризованному классу,

и передать ему *ограничение*; компилятор принимает только те типы, которые соответствуют указанному ограничению. Для задания ограничения используется ключевое слово **extends**. При заданном ограничении следующий фрагмент компилируется нормально:

```
//: 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>** указывает на то, что параметр **T** должен относиться к типу **HasF** или производному от него. Если это условие выполняется, то вызов **f()** для **obj** безопасен.

Можно сказать, что параметр типа *стирается до первого ограничения* (как будет показано позже, ограничений может быть несколько). Мы также рассмотрим понятие *стирания параметра типа*. Компилятор фактически заменяет параметр типа его «стертой» версией, так что в предыдущем случае **T** стирается до **HasF**, а результат получается таким, как при замене **T** на **HasF** в теле класса.

Справедливости ради нужно заметить, что в **Manipulation2.java** параметризация никакой реальной пользы не дает. С таким же успехом можно выполнить стирание самостоятельно, создав непараметризованный класс:

```
//· generics/Manipulator3.java

class Manipulator3 {
     private HasF obj.
     public Manipulator3(HasF x) { obj = x; }
     public void manipulate() { obj f(). }
} /// ~
```

Мы приходим к важному заключению: параметризация полезна только тогда, когда вы хотите использовать параметры типов, более «общие», нежели конкретный тип (и производные от него), то есть когда код должен работать для разных классов. В результате параметры типов и их применение в параметризованном коде сложнее простой замены классов. Впрочем, это не означает, что форма **<T extends HasF>** чем-то ущербна. Например, если класс содержит метод, возвращающий **T**, то параметризация будет полезной, потому что метод вернет точный тип:

```
//  generics/ReturnGenericType.java

class ReturnGenericType<T extends HasF> {
     private T obj.
     public ReturnGenericType(T x) { obj = x; }
     public T get() { return obj; }
} ///:~
```

Просмотрите код и подумайте, достаточно ли он «сложен» для применения параметризации.

Ограничения будут более подробно рассмотрены далее в этой главе.

Миграционная совместимость

Чтобы избежать всех потенциальных недоразумений со стиранием, необходимо четко понимать, что этот механизм *не является* особенностью языка. Скорее это компромисс, использованный при реализации параметризации в Java, потому что параметризация не являлась частью языка в его исходном виде. Этот компромисс создает определенные неудобства, поэтому вы должны поскорее привыкнуть к нему и понять, почему он существует.

Если бы параметризация была частью Java 1.0, то для ее реализации стирание не потребовалось бы — параметры типов сохранили бы свой статус равноправных компонентов языка, и с ними можно было бы выполнять типизованные языковые и рефлексивные операции. Позднее в этой главе будет показано, что стирание снижает «обобщенность» параметризованных типов. Параметризация в Java все равно приносит пользу, но не такую, какую могла бы приносить, и причиной тому является стирание.

В реализации, основанной на стирании, параметризованные типы рассматриваются как второстепенные компоненты языка, которые не могут использоваться в некоторых важных контекстах. Параметризованные типы присутствуют только при статической проверке типов, после чего каждый параметризованный тип в программе заменяется непараметризованным верхним ограничением. Например, обозначения типов вида `List<T>` стирается до `List`, а обычные переменные типа — до `Object`, если ограничение не задано.

Главная причина для применения стирания заключается в том, что оно позволяет параметризованным клиентам использовать непараметризованные библиотеки, и наоборот. Эта концепция часто называется *миграционной совместимостью*. Наверное, в идеальном мире параметризация была бы внедрена везде и повсюду одновременно. На практике программисту, даже если он пишет только параметризованный код, приходится иметь дело с непараметризованными библиотеками, написанными до Java SE5. Возможно, авторы этих библиотек вообще не намерены параметризовать свой код или собираются сделать это в будущем.

Из-за этого механизму параметризации Java приходится поддерживать не только *обратную совместимость* (существующий код и файлы классов остаются абсолютно законными и сохраняют свой прежний смысл), но и миграционную совместимость — чтобы библиотеки могли переводиться в параметризованную форму в собственном темпе, причем их параметризация не влияла бы на работу зависящего от него кода и приложений. Выбрав эту цель, проектировщики Java и различные группы, работавшие над проблемой, решили, что единственным приемлемым решением является стирание, позволяющее непараметризованному коду нормально сосуществовать с параметризованным.

Проблемы стирания

Итак, главным аргументом для применения стирания является процесс перехода с непараметризованного кода на параметризованный и интеграция параметризации в язык без нарушения работы существующих библиотек. Стирание

позволяет использовать существующий непараметризованный код без изменений, пока клиент не будет готов переписать свой код с использованием параметризации.

Однако за стирание приходится расплачиваться. Параметризованные типы не могут использоваться в операциях, в которых явно задействованы типы времени выполнения — преобразования типов, instanceof и выражения new. Вся информация о типах параметров теряется, и при написании параметризованного кода вам придется постоянно напоминать себе об этом. Допустим, вы пишете фрагмент кода

```
class Foo<T> {
  T var;
}
```

Может показаться, что при создании экземпляра Foo:

```
Foo<Cat> f = new Foo<Cat>();
```

код class Foo должен знать, что он работает с Cat. Синтаксис создает впечатление, что тип T подставляется повсюду внутри класса. Но на самом деле это не так, и при написании кода для класса вы должны постоянно напоминать себе: «Нет, это всего лишь Object».

Кроме того, стирание и миграционная совместимость означают, что контроль за использованием параметризации не настолько жесткий, как хотелось бы:

```
//: 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.
//   Обнаружен 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 параметризации, и компилятор не выдает при этом никаких предупреждений. Предупреждение выводится позже, при вызове set().

Для подавления этого предупреждения в Java существует *директива*, приведенная в листинге (до выхода Java SE5 она не поддерживалась):

```
@SuppressWarnings("unchecked")
```

Обратите внимание: директива применяется к методу, генерирующему предупреждение, а не ко всему классу. При подавлении предупреждений желательно действовать в самых узких рамках, чтобы случайно не скрыть настоящую проблему.

Ошибка, выдаваемая в **Derived3**, означает, что компилятор рассчитывает увидеть «обычный» базовый класс.

Добавьте к этому дополнительные усилия на управление ограничениями, если вы не желаете интерпретировать параметр типа как простой **Object**, — и что мы получаем в остатке? Гораздо больше хлопот при гораздо меньше, пользе по сравнению с параметризованными типами в языках вроде C++, Ada или Eiffel. Конечно, это вовсе не означает, что эти языки в целом эффективнее Java в большинстве задач программирования, а говорит лишь о том, что их механизмы параметризации типов отличаются большей гибкостью и мощью, чем в Java.

Проблемы на границах

Пожалуй, самый странный аспект параметризации, обусловленный стиранием, заключается в возможности представления заведомо бессмысленных вещей. 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]
*///:~
```

Несмотря на то что объект **kind** хранится в виде **Class<T>**, стирание означает, что фактически он хранится в виде **Class** без параметра. Следовательно, при выполнении с ним каких-либо операций (например, при создании массива) **Array.newInstance()** не обладает информацией о типе, подразумеваемой **kind**. Метод не сможет выдать нужный результат, не требующий преобразования типа, а это приводит к выдаче предупреждения, с которым вам не удастся справиться.

Обратите внимание: для создания массивов в параметризованном коде рекомендуется использовать Array.newInstance().

Если вместо массива создается другой контейнер, ситуация меняется:

```
//: 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, что <T> в new ArrayList<T>() внутри create() удаляется — во время выполнения <T> внутри класса нет, поэтому здесь его присутствие выглядит бессмысленным. Однако если вы попробуете применить эту идею на практике и преобразуете выражение в new ArrayList(), компилятор выдаст предупреждение.

Но действительно ли этот элемент не имеет смысла? Что произойдет, если мы поместим в список несколько объектов, прежде чем возватим его?

```
//: 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 ничего не может знать о T в create(), он все равно способен проверить — на стадии компиляции — что заносимые в result объекты имеют тип T и согласуются с ArrayList<T>. Таким образом, несмотря на то что стирание удаляет информацию о фактическом типе внутри метода или класса, компилятор все равно может проверить корректность использования типа в методе или классе.

Так как стирание удаляет информацию о типе внутри тела метода, на стадии выполнения особую роль приобретают *границы* — точки, в которых объект входит и выходит из метода. Именно в этих точках компилятор выполняет проверку типов и вставляет код преобразования. Рассмотрим следующий непараметризованный 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();
     }
} ///:~
```

Декомпилировав результат командой javap -c SimpleHolder, мы получим (после редактирования):

```
public  void set(java.lang.Object);
   0      aload_0
   1:     aload_1
   2:     putfield #2; // Поле obj:Object;
   5.     return

public  java.lang.Object get();
   0:     aload_0
   1·     getfield #2; // Поле obj·Object,
   4      areturn

public  static void main(java.lang.String[]);
   0:     new #3, // Класс SimpleHolder
   3·     dup
   4:     invokespecial #4; // Метод "<init>".()V
   7.     astore_1
   8·     aload_1
   9.     ldc #5; // String Item
   11     invokevirtual #6; // Метод set (Object;)V
   14:    aload_1
   15.    invokevirtual #7, // Метод get:()Object;
   18:    checkcast #8, //·Класс java/lang/String
   21:    astore_2
   22.    return
```

Методы set() и get() просто записывают и читают значение, а преобразование проверяется в точке вызова get().

Теперь включим параметризацию в приведенный фрагмент:

```
//: 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 преобразования выходного значения get() отпала, но мы также знаем, что тип значения, передаваемого set(), проверяется во время компиляции. Соответствующий байт-код:

```
public  void set(java.lang.Object);
   0:     aload_0
   1:     aload_1
   2:     putfield #2; // Поле obj:Object;
   5:     return

public  java.lang.Object get();
   0:     aload_0
   1:     getfield #2; // Поле obj:Object;
   4:     areturn

public  static void main(java.lang.String[]);
   0.     new #3; // Класс GenericHolder
   3:     dup
   4:     invokespecial #4; // Метод "<init>"·()V
   7:     astore_1
   8:     aload_1
   9:     ldc #5; // String Item
   11:    invokevirtual #6; // Метод set:(Object;)V
   14:    aload_1
   15:    invokevirtual #7; // Метод get:()Object;
   18:    checkcast #8; // Класс java/lang/String
   21:    astore_2
   22:    return
```

Как видите, байт-код идентичен. Дополнительная работа по проверке входного типа set() выполняется компилятором «бесплатно». Преобразование выходного значения get() по-прежнему сохранилось, но, по крайней мере, вам не приходится выполнять его самостоятельно — оно автоматически вставляется компилятором.

Компенсация за стирание

Как мы видели, в результате стирания становится невозможным выполнение некоторых операций в параметризованном коде. Все, для чего необходима точная информация о типе во время выполнения, работать не будет:

```
//: 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]; // Предупреждение
     }
} ///:~
```

Иногда такие проблемы удается обойти на программном уровне, но в отдельных случаях стирание приходится компенсировать посредством введения *метки типа*. Другими словами, вы явно передаете объект Class для своего типа.

Например, попытка использования instanceof в предыдущем примере завершилась неудачей из-за того, что информация о типе была стерта. При введении метки типа вместо instanceof можно использовать динамический метод 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.

Создание экземпляров типов

Попытка создания new T() в Erased.java не работает отчасти из-за стирания, а отчасти из-за того, что компилятор не может убедиться в наличии у T конструктора по умолчанию (без аргументов). Но в C++ эта операция естественна, прямолинейна и безопасна (проверка выполняется во время компиляции):

```
//: generics/InstantiateGenericType.java
import static net.mindview.util.Print.*;

class ClassAsFactory<T> {
     T x;
     public ClassAsFactory(Class<T> kind) {
              try {
```

продолжение

```
                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> успех");
        try {
            ClassAsFactory<Integer> fi =
                    new ClassAsFactory<Integer>(Integer.class);
        } catch(Exception e) {
            print("ClassAsFactory<Integer> неудача");
        }
    }
} /* Output:
ClassAsFactory<Employee> успех
ClassAsFactory<Integer> неудача
*///:~
```

Программа компилируется, но с **ClassAsFactory<Integer>** происходит сбой, так как **Integer** не имеет конструктора по умолчанию. Ошибка не обнаруживается во время компиляции, поэтому специалисты из Sun считают такие решения нежелательными. Вместо этого рекомендуется 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());
        }
} ///:~
```

В сущности, это всего лишь разновидность передачи Class<T>. В обоих вариантах передаются объекты фабрик; просто в случае с Class<T> объект фабрики оказывается встроенным, а при предыдущем решении он создается явно. Тем не менее в обоих случаях реализуется проверка времени компиляции.

Другое решение основано на использовании паттерна *«шаблонный метод»*. В следующем примере get() — шаблонный метод, а create() определяется в субклассе для получения объекта этого 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() {
                System.out.println(element.getClass().getSimpleName());
        }
}

public class CreatorGeneric {
        public static void main(String[] args) {
                Creator c = new Creator();
                c.f();
        }
} /* Output:
X
*///:~
```

Массивы параметризованных типов

Как мы видели в Erased.java, создавать массивы параметризованных типов нельзя. Везде, где возникает необходимость в создании таких массивов, следует применять ArrayList:

```
//: generics/ListOfGenerics.java
import java.util.*;

public class ListOfGenerics<T> {
        private List<T> array = new ArrayList<T>();
```

продолжение ⇩

```
    public void add(T item) { array.add(item); }
    public T get(int index) { return array.get(index); }
} ///:~
```

При этом вы получаете поведение массивов с безопасностью типов на стадии компиляции, возможной для параметризации.

Впрочем, иногда бывает нужно создать именно массив параметризованных типов (скажем, во внутренней реализации ArrayList используются массивы). Оказывается, можно переопределить *ссылку* так, чтобы предотвратить протесты компилятора. Пример:

```
//: generics/ArrayOfGenericReference.java

class Generic<T> {}

public class ArrayOfGenericReference {
    static Generic<Integer>[] gia;
} ///:~
```

Компилятор принимает эту запись без каких-либо предупреждений. С другой стороны, вы не сможете создать массив указанного типа (включая параметры типа), поэтому все это сбивает с толку. Поскольку все массивы обладают одинаковой структурой (размер каждого элемента и способ размещения в памяти) независимо от типа хранящихся данных, создается впечатление, что вы сможете создать массив Object и преобразовать его к нужному типу. Код откомпилируется, но работать не будет — он выдает 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, но приводит к ClassCastException:
        //! gia = (Generic<Integer>[])new Object[SIZE];
        // Тип времени выполнения является "стертым" type:
        gia = (Generic<Integer>[])new Generic[SIZE];
        System.out.println(gia.getClass().getSimpleName());
        gia[0] = new Generic<Integer>();
        //! gia[1] = new Object(); // Ошибка компиляции
        // Обнаруживается несоответствие типов во время компиляции:
        //! gia[2] = new Generic<Double>();
    }
} /* Output:
Generic[]
*///:~
```

Проблема в том, что массивы отслеживают свой фактический тип, который задается в точке создания массива. Таким образом, даже несмотря на то, что gia преобразуется в Generic<Integer>[], эта информация существует только на стадии компиляции (а без директивы @SuppressWarnings вы получите предупреждение). Во время выполнения мы по-прежнему имеем дело с массивом Object, и это создает проблемы. Успешно создать массив параметризованного типа

можно только одним способом — создать новый массив «стертого» типа и выполнить преобразование.

Рассмотрим чуть более сложный пример. Допустим, имеется простая параметризованная «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]; }
     // Метод, предоставляющий доступ к базовому представлению:
     public T[] rep() { return array; }
     public static void main(String[] args) {
             GenericArray<Integer> gai =
                     new GenericArray<Integer>(10);
             // Приводит к ClassCastException:
             //! Integer[] ia = gai.rep();
             // А так можно.
             Object[] oa = gai.rep();
     }
} ///:~
```

Как и прежде, мы не можем использовать запись `T[] array = new T[sz]`, поэтому мы создаем массив объектов и преобразуем его.

Метод `rep()` возвращает `T[]`; в методе `main()` для `gai` это должен быть тип `Integer[]`, но при попытке вызова и сохранения результата по ссылке на `Integer[]` будет получено исключение `ClassCastException` — это снова происходит из-за того, что фактическим типом объекта времени выполнения является `Object[]`. Если мы немедленно проводим преобразование к `T[]`, то на стадии компиляции фактический тип массива теряется и компилятор может упустить некоторые потенциальные ошибки. Из-за этого лучше использовать в коллекции `Object[]`, а затем добавить преобразование к `T` при использовании элемента массива. Вот как это будет выглядеть 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() {
```

продолжение ⇩

```
        return (T[])array; // Предупреждение: непроверенное преобразование
    }
    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 преобразование типа было перемещено. Без директив **@SuppressWarnings** вы по-прежнему будете получать предупреждения, но теперь во внутренней реализации используется **Object[]** вместо **T[]**. При вызове **get()** объект преобразуется к T; это правильный тип, поэтому преобразование безопасно. Но при вызове **rep()** снова делается попытка преобразования **Object[]** в **T[]**, которое остается неверным; в результате вы получите предупреждение во время компиляции и исключение во время выполнения. Не существует способа обойти тип базового массива, которым может быть только **Object[]**. У внутренней интерпретации **array** как **Object[]** вместо **T[]** есть свои преимущества: например, вы с меньшей вероятностью забудете тип массива, что приведет к случайному появлению ошибок (впрочем, подавляющее большинство таких ошибок будет быстро выявлено на стадии выполнения).

В новом коде следует передавать метку типа. В обновленной версии **GenericArray** выглядит так:

```
//: 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]; }
    // Expose the underlying representation:
    public T[] rep() { return array; }
    public static void main(String[] args) {
        GenericArrayWithTypeToken<Integer> gai =
                new GenericArrayWithTypeToken<Integer>(
                        Integer.class, 10);
        // This now works:
```

```
            Integer[] ia = gai.rep();
    }
} ///:~
```

Метка типа Class<T> передается конструктору для восстановления информации после стирания, чтобы мы могли создать фактический тип нужного массива (предупреждения при преобразовании по-прежнему приходится подавлять @SuppressWarnings). Получив фактический тип, мы возвращаем его для получения желаемых результатов, как видно из main().

К сожалению, просмотрев исходный код стандартных библиотек Java SE5, вы увидите, что преобразования массивов Object в параметризованные типы происходят повсеместно. Например, вот как выглядит копирующий конструктор для создания ArrayList из Collection после некоторой правки и упрощения:

```
public ArrayList(Collection c) {
  size = c.size();
  elementData = (E[])new Object[size];
  c.toArray(elementData);
}
```

В ArrayList.java подобные преобразования встречаются неоднократно. И конечно, при их компиляции выдается множество предупреждений.

Ограничения

Ограничения, уже упоминавшиеся ранее в этой главе, сужают круг параметров типов, используемых при параметризации. Хотя это позволяет предъявлять требования к типам, к которым применяется ваш параметризованный код, у ограничений имеется и другой, потенциально более важный эффект: возможность вызова методов, определенных в ограничивающих типах.

Поскольку стирание уничтожает информацию о типе, при отсутствии ограничений для параметров типов могут вызываться только методы Object. Но, если ограничить параметр подмножеством типов, вы сможете вызвать методы из этого подмножества. Для установления ограничений в Java используется ключевое слово extends. Важно понимать, что в контексте параметризации extends имеет совершенно иной смысл, нежели в обычной ситуации. Следующий пример демонстрирует основы установления 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 вызвать метод:
     java.awt.Color color() { return item.getColor(); }
}

class Dimension { public int x, y, z; }
```

продолжение ⇩

```
// Не работает -- сначала класс, потом интерфейсы:
// class ColoredDimension<T extends HasColor & Dimension> {

// Несколько ограничений:
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 при наследовании, конкретный класс может быть только один,
// а интерфейсов может быть несколько:
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 =
               new Solid<Bounded>(new Bounded());
          solid.color();
          solid.getY();
          solid.weight();
     }
} ///:~
```

Вероятно, вы заметили, что пример **BasicBounds.java** содержат некоторую избыточность, которая может быть устранена посредством наследования. С каждым уровнем наследования добавляются новые 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 реализовывал HasColor. ColoredDimension2 и Solid2 продолжают расширение иерархии и добавляют на каждом уровне новые ограничения. Теперь методы наследуются, и их не нужно повторять в каждом классе.

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>
```

продолжение ⇩

```
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 параметризованных 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 можно:
              List<? extends SuperHearing> audioBoys;
              // А так нельзя:
              // List<? extends SuperHearing & SuperSmell> dogBoys;
     }
} ///:~
```

Метасимволы

Мы уже встречали простые примеры использования *метасимволов* — вопросительных знаков в выражениях аргументов параметризации — в главах 11 и 13. В этом разделе тема будет рассмотрена более подробно.

Начнем с примера, демонстрирующего одну особенность 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[] или Orange[]:
             try {
                     // Компилятор позволяет добавлять объекты Fruit:
                     fruit[0] = new Fruit(); // ArrayStoreException
             } catch(Exception e) { System.out.println(e); }
             try {
                     // Компилятор позволяет добавлять объекты 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**. Выглядит логично — **Apple** является разновидностью **Fruit**, поэтому массив **Apple** также одновременно должен быть массивом **Fruit**.

С другой стороны, если фактическим типом массива является **Apple[]**, в массиве можно разместить только **Apple** или субтип **Apple**, причем это правило должно соблюдаться как во время компиляции, так и во время выполнения. Но обратите внимание на то, что компилятор также позволит разместить в массиве ссылку на объект **Fruit**. Для компилятора это вполне логично, потому что он имеет дело со ссылкой **Fruit[]** — так почему бы не разрешить занести в массив объект **Fruit** или любого типа, производного от **Fruit**, — скажем, **Orange**? Во время компиляции это разрешено. Однако механизм времени выполнения знает, что он имеет дело с **Apple[]**, и при попытке занесения постороннего типа происходит исключение.

Впрочем, для массивов это не создает особых проблем, потому что при вставке объекта неверного типа вы об этом очень быстро узнаете во время выполнения. Но одна из основных целей параметризации как раз и состоит в том, чтобы по возможности переместить выявление подобных ошибок на стадию выполнения. Итак, что же произойдет при использовании параметризованных контейнеров вместо массивов?

```
//: generics/NonCovariantGenerics.java
// {CompileTimeError} (Won't compile)
import java.util.*;

public class NonCovariantGenerics {
     // Ошибка компиляции: несовместимые типы
     List<Fruit> flist = new ArrayList<Apple>();
} ///:~
```

На первый взгляд это выглядит как утверждение «Контейнер с элементами Apple нельзя присвоить контейнеру с элементами Fruit», но следует вспомнить, что параметризация — это не только контейнеры. В действительности утверждение следует трактовать шире: «Параметризованный тип, *в котором задействован* тип Apple, нельзя присвоить параметризованному типу, *в котором задействован* тип Fruit». Если бы, как в случае с массивами, компилятор располагал достаточной информацией и мог понять, что речь идет о контейнерах, он мог бы проявить некоторую снисходительность. Но компилятор такой информацией не располагает, поэтому он отказывается выполнить «восходящее преобразование». Впрочем, это и не является восходящим преобразованием — List с элементами Apple не является «частным случаем» List с элементами Fruit. Первый может хранить Apple и подтипы Apple, а второй — любые разновидности Fruit... да, в том числе и Apple, но от этого он не становится List с элементами Apple, а по-прежнему остается List с элементами Fruit.

Проблема в том, что речь идет о типе контейнера, а не о типе элементов, которые в этом контейнере хранятся. В отличие от массивов, параметризованные типы не обладают встроенной ковариантностью. Это связано с тем, что массивы полностью определяются в языке и для них могут быть реализованы встроенные проверки как во время компиляции, так и во время выполнения, но с параметризованными типами компилятор и система времени выполнения не знают, что вы собираетесь делать с типами и какие правила при этом должны действовать.

Но иногда между двумя разновидностями параметризованных типов все же требуется установить некоторую связь, аналогичную восходящему преобразованию. Именно это и позволяют сделать метасимволы.

```
//: generics/GenericsAndCovariance.java
import java.util.*;

public class GenericsAndCovariance {
     public static void main(String[] args) {
               // Метасимволы обеспечивают ковариантность:
               List<? extends Fruit> flist = new ArrayList<Apple>();
               // Ошибка компиляции: добавление объекта
               // произвольного типа невозможно
               // flist.add(new Apple());
               // flist.add(new Fruit());
               // flist.add(new Object());
               flist.add(null); // Можно, но неинтересно
               // Мы знаем, что возвращается по крайней мере Fruit:
               Fruit f = flist.get(0);
     }
} ///:~
```

Теперь flist относится к типу List<? extends Fruit>, что можно прочитать как «список с элементами любого типа, производного от Fruit». Однако в действительности это не означает, что List будет содержать именно типы из семейства Fruit. Метасимвол обозначает «некоторый конкретный тип, не указанный в ссылке flist». Таким образом, присваиваемый List должен содержать некий

конкретный тип (например, Fruit или Apple), но для восходящего преобразования к flist этот тип несущественен.

Если единственное ограничение состоит в том, что List содержит Fruit или один из его подтипов, но вас не интересует, какой именно, что же с ним можно сделать? Если вы не знаете, какие типы хранятся в List, возможно ли безопасное добавление объекта? Нет, как и в случае с CovariantArrays.java, но на этот раз ошибка выявляется компилятором, а не системой времени выполнения.

Может показаться, что такой подход не совсем логичен — вам не удастся даже добавить Apple в List, в котором, как вы только что указали, должны храниться Apple. Да, конечно, но компилятор-то этого не знает! List<? extends Fruit> вполне может указывать на List<Orange>.

С другой стороны, вызов метода, возвращающего Fruit, безопасен; мы знаем, что все элементы List должны по меньшей мере относиться к Fruit, поэтому компилятор это позволит.

Насколько умен компилятор?

Казалось бы, из всего сказанного следует, что вызов любых методов с аргументами невозможен, но рассмотрим 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'
     }
} ///:~
```

Как видите, вызовы contains() и indexOf() с аргументами Apple воспринимаются нормально. Означает ли это, что компилятор действительно анализирует код, чтобы узнать, модифицирует ли некоторый метод свой объект?

Просмотр документации ArrayList показывает, что компилятор не настолько умен. Если add() получает аргумент параметризующего типа, contains() и indexOf() получают аргумент типа Object. Таким образом, когда вы указываете ArrayList<? extends Fruit>, аргумент add() превращается в «? extends Fruit». По этому описанию компилятор не может определить, какой именно подтип Fruit требуется в данном случае, поэтому не принимает никакие типы Fruit. Даже если вы предварительно преобразуете Apple в Fruit, компилятор все равно откажется вызывать метод (например, add()), если в списке аргументов присутствует метасимвол.

У методов contains() и indexOf() аргументы относятся к типу Object, метасимволы в них отсутствуют, поэтому компилятор разрешает вызов. Это означает, что проектировщик параметризованного класса должен сам решить, какие вызовы «безопасны», и использовать типы Object для их аргументов. Чтобы

сделать невозможным вызов при использовании типа с метасимволами, включите параметр типа в список аргументов.

В качестве 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; // Повышение невозможно
              Holder<? extends Fruit> fruit = Apple; // OK
              Fruit p = fruit.get();
              d = (Apple)fruit.get(); // Возвращает 'Object'
              try {
                       Orange c = (Orange)fruit.get(); // Предупреждения нет
              } 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
*///:~
```

Holder содержит метод set(), получающий T; метод get(), возвращающий T; и метод equals(), получающий Object. Как вы уже видели, Holder<Apple> невозможно преобразовать в Holder<Fruit>, но зато можно в Holder<? extends Fruit>. При вызове get() будет возвращен только тип Fruit — то, что известно компилятору по ограничению «все, что расширяет Fruit». Если вы располагаете дополнительной информацией, то сможете выполнить преобразование к конкретному типу Fruit и обойтись без предупреждений, но с риском исключения ClassCastException. Метод set() не работает ни с Apple, ни с Fruit, потому что аргумент set() тоже содержит «? extends Fruit»; по сути, он может быть чем угодно, а компилятор не может проверить безопасность типов для «чего угодно».

Впрочем, метод equals() работает нормально, потому что он получает Object вместо T. Таким образом, компилятор обращает внимание только на типы передаваемых и возвращаемых объектов. Он не анализирует код, проверяя, выполняются ли реальные операции чтения или записи.

Контравариантность

Также можно пойти другим путем и использовать *метасимволы супертипов*. В этом случае вы сообщаете, что метасимвол ограничивается базовым классом

некоторого класса; при этом используется запись <? super MyClass>, и даже с параметром типа <? super T>. Это позволяет безопасно передавать типизованный объект параметризованному типу. Таким образом, с использованием метасимволов супертипов становится возможной запись в коллекцию:

```
//· 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 является контейнером List для некоторого типа, являющегося базовым для Apple; из этого следует, что Apple и производные от Apple типы могут безопасно включаться в контейнер. Но, поскольку *нижним ограничением* является Apple, мы не знаем, безопасно ли включать Fruit в такой List, так как это откроет List для добавления типов, отличных от Apple, с нарушением статической безопасности типов.

Ограничения супертипов расширяют возможности по передаче аргументов 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 параметр типа «как есть», без метасимволов. На примере f1() мы видим, что этот способ отлично работает — при условии, что в List<Apple> помещаются только объекты Apple. Однако writeExact() не позволяет поместить Apple в List<Fruit>, хотя мы знаем, что это должно быть возможно.

В writeWithWildcard() используется аргумент List<? super T>, поэтому List содержит конкретный тип, производный от T; следовательно, T или производные от него типы могут безопасно передаваться в аргументе методов List. Пример встречается в f2: как и прежде, Apple можно поместить в List<Apple>, но, как и предполагалось, также стало можно поместить Apple в List<Fruit>.

Неограниченные метасимволы

Казалось бы, неограниченный метасимвол <?<@062> должен означать «все, что угодно», а его использование эквивалентно использованию низкоуровневого типа. В самом деле, на первый взгляд компилятор подтверждает эту оценку:

```
//: 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: непроверенное 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·
              // Непроверенное преобразование. Обнаружен· ArrayList
              // Требуется: List<? extends Object>
              assign1(new ArrayList<String>());
              assign2(new ArrayList<String>());
              assign3(new ArrayList<String>());
              // Приемлемы обе формы·
              List<?> wildList = new ArrayList();
              wildList = new ArrayList<String>();
              assign1(wildList);
              assign2(wildList);
              assign3(wildList);
     }
} ///:~
```

Во многих ситуациях, подобных рассмотренной, для компилятора совершенно не существенно, используется низкоуровневый тип или <?>. Конструкцию <?>

можно считать обычным украшением; впрочем, она обладает некоторой практической ценностью, потому что фактически означает: «Код написан с учетом параметризации Java, и здесь эта конструкция означает не то, что я использую низкоуровневый тип, а то, что параметр параметризации может содержать произвольный тип».

Второй пример демонстрирует важное практическое использование неограниченных метасимволов. Когда вы имеете дело с несколькими параметрами, иногда важно указать, что один параметр может относиться к произвольному типу, а другой ограничить определенным типом:

```
//: 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:
               // Непроверенное преобразование. Обнаружен: HashMap
               // Требуется: Map<String,?>
               assign1(new HashMap<String,Integer>());
               assign2(new HashMap<String,Integer>());
               assign3(new HashMap<String,Integer>());
     }
} ///:~
```

Когда в записи используются *только* неограниченные метасимволы, как в примере `Map<?,?>`, компилятор не отличает такой тип от `Map`. Кроме того, пример `UnboundedWildcards1.java` показывает, что компилятор по-разному интерпретирует `List<?>` и `List<? extends Object>`.

Ситуация осложняется тем, что компилятор не всегда интересуется различиями между `List` и `List<?>` (например), поэтому может показаться, что это одно и то же. В самом деле, поскольку параметризованный аргумент стирается до первого ограничения, `List<?>` кажется эквивалентным `List<Object>`, а `List`, по сути, тоже является `List<Object>` — однако ни одно из этих утверждений не является в полной мере истинным. `List` в действительности означает «низкоуровневый `List`, содержащий любой тип `Object`», тогда как `List<?>` означает «не-низкоуровневый `List`, содержащий *какой-то конкретный тип*, хотя мы не знаем, какой именно».

Когда же компилятор различает низкоуровневые типы и типы с неограниченными метасимволами? В следующем примере используется класс `Holder<T>`, определение которого приводилось ранее. Класс содержит методы, получающие аргумент `Holder`, но в разных формах: в виде низкоуровневого типа, с конкретным параметром типа, с неограниченным метасимволом:

```
//: generics/Wildcards.java
// Exploring the meaning of wildcards.

public class Wildcards {
     // Низкоуровневый аргумент:
     static void rawArgs(Holder holder, Object arg) {
               // holder set(arg); // Предупреждение·
               //   Непроверенный вызов set(T) как члена
               //   низкоуровневого типа Holder
               // holder.set(new Wildcards()); // То же предупреждение

               // Невозможно; нет информации о 'T'
               // T t = holder.get(),

               // Допустимо, но информация типа теряется
               Object obj = holder get(),
     }
     // По аналогии с rawArgs(), но ошибки вместо предупреждений.
     static void unboundedArg(Holder<?> holder, Object arg) {
               // holder.set(arg); // Ошибка:
               //   set(capture of ?) in Holder<capture of ?>
               //   не может применяться к (Object)
               // holder.set(new Wildcards()), // Та же ошибка

               // Невозможно; нет информации о 'T':
               // 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) in
               //   Holder<capture of ? extends T>
               //   cannot be applied to (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); // Предупреждение
//   Непроверенное преобразование Holder в Holder<T>
//   Непроверенный вызов метода: exact1(Holder<T>)
//   применяется к (Holder)
Long r2 = exact1(qualified);
Object r3 = exact1(unbounded); // Должен возвращать Object
Long r4 = exact1(bounded);

// Long r5 = exact2(raw, lng); // Предупреждения:
//   Непроверенное преобразование Holder в Holder<Long>
//   Непроверенный 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
//   Непроверенное преобразование Holder
//   к Holder<? extends Long>
//   Непроверенный вызов метода:
//   wildSubtype(Holder<? extends T>,T)
//   применяется к (Holder,Long)
Long r10 = wildSubtype(qualified, lng);
// Допустимо, но возвращать может только Object:
Object r11 = wildSubtype(unbounded, lng);
Long r12 = wildSubtype(bounded, lng);

// wildSupertype(raw, lng); // Предупреждения:
//   Непроверенное преобразование Holder
//   к Holder<? super Long>
//   Непроверенный 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
```

продолжение ⇩

```
        //   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 методе rawArgs() компилятор знает, что Holder является параметризованным типом, поэтому несмотря на то, что здесь он выражен как низкоуровневый тип, компилятору известно, что передача Object методу set() небезопасна. Так как в данном случае используется низкоуровневый тип, методу set() можно передать объект произвольного типа, и он будет преобразован в Object. Таким образом, при использовании низкоуровневого типа вы лишаетесь проверки на стадии компиляции. Вызов get() демонстрирует ту же проблему: никакого T нет, поэтому результатом может быть только Object.

Может создаться впечатление, что низкоуровневый Holder и Holder<?> — приблизительно одно и то же. Однако метод unboundedArgs() демонстрирует различия между ними — в нем выявляются те же проблемы, но информация о них выдается в виде ошибок, а не предупреждений, поскольку низкоуровневый Holder может содержать разнородные комбинации типов, тогда как Holder<?> содержит однородную коллекцию *одного конкретного типа*.

В exact1() и exact2() используются точные параметры типов (то есть без метасимволов). Мы видим, что exact2() обладает иными ограничениями, нежели exact1(), из-за дополнительного аргумента.

В wildSubtype() ограничения на тип Holder опускаются до Holder с элементами любого типа, удовлетворяющими условию extends T. И снова это означает, что T может быть типом Fruit, а holder сможет вполне законно стать Holder <Apple>. Чтобы предотвратить возможное размещение Orange в Holder<Apple>, вызовы set() (и любых других методов, получающих в аргументах параметр типа) запрещены. Однако мы знаем, что все объекты, полученные из Holder<? extends Fruit>, по меньшей мере, являются Fruit, поэтому вызов get() (или любого метода с возвращаемым значением параметра типа) допустим.

Реализация параметризованных интерфейсов

Класс не может реализовать две разновидности одного параметризованного интерфейса — вследствие стирания они будут считаться одним и тем же интерфейсом. Пример конфликта такого рода:

```
//: generics/MultipleInterfaceVariants.java
// {CompileTimeError} (Не компилируется)

interface Payable<T> {}

class Employee implements Payable<Employee> {}
class Hourly extends Employee
    implements Payable<Hourly> {} ///:~
```

Класс Hourly компилироваться не будет, потому что стирание сокращает Payable<Employee> и Payable<Hourly> до Payable, а в приведенном примере это означало бы двукратную реализацию одного интерфейса. Интересная подробность:

если удалить параметризованные аргументы из обоих упоминаний Payable, как это делает компилятор при стирании, программа откомпилируется.

Преобразования типов и предупреждения

Преобразование типа или instanceof с параметром типа не приводит ни к какому эффекту. В следующем контейнере данные хранятся во внутреннем представлении в форме Object и преобразуются к T при выборке:

```
//. generics/GenericCast.java

class FixedSizeStack<T> {
     private int index = 0;
     private Object[] storage;
     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 директивы @SuppressWarnings компилятор выдает для pop() предупреждение о «непроверенном преобразовании». Вследствие стирания он не знает, безопасно преобразование или нет, поэтому метод pop() никакого преобразования не выполняет. T стирается до первого ограничения, которым по умолчанию является Object, так что pop() на самом деле преобразует Object в Object.

Перегрузка

Следующий пример не компилируется, хотя на первый взгляд выглядит вполне разумно:

```
//  generics/UseList.java
// {CompileTimeError} (Не компилируется)
import java.util.*;

public class UseList<W,T> {
     void f(List<T> v) {}
```

продолжение ⇨

```
    void f(List<W> v) {}
} /// ~
```

Перегрузка метода создает идентичную сигнатуру типа вследствие стирания. В таких случаях следует определять методы с различающимися именами:

```
//. generics/UseList2 java
import java util.*:

public class UseList2<W,T> {
    void f1(List<T> v) {}
    void f2(List<W> v) {}
} ///.~
```

К счастью, проблемы такого рода обнаруживаются компилятором.

Резюме

Мне довелось работать с шаблонами C++ с момента их появления. Скорее всего, приведенный далее аргумент я выдвигал в спорах чаще, чем большинство моих единомышленников. Лишь недавно я задумался над тем, насколько в действительности справедлив этот аргумент, — сколько раз проблема, которую я сейчас опишу, проникала в рабочий код?

Аргумент такой: одним из самых логичных мест для использования механизма параметризации являются контейнерные классы: **List**, **Set**, **Map** и т. д. До выхода Java SE5 объект, помещаемый в контейнер, преобразовывался в **Object**, и информация типа терялась. Если же вы хотели снова извлечь объект из контейнера, его приходилось преобразовывать к нужному типу. Я пояснял происходящее на примере **List** с элементами **Cat** (разновидность этого примера с **Apple** и **Orange** приведена в начале главы 11). Без параметризованной версии контейнера из Java SE5 вы помещаете и извлекаете из контейнера **Object**, поэтому в **List** с элементами **Cat** легко поместить объект **Dog**.

Однако версии Java, существовавшие до появления параметризации, не допускали *злоупотреблений* объектами, помещаемыми в контейнер. Если вы помещали **Dog** в контейнер **Cat**, а затем пытались интерпретировать все элементы контейнера как **Cat**, то при извлечении ссылки на **Dog** и ее преобразовании к **Cat** происходило преобразование **RuntimeException**. Проблема обнаруживалась, пусть и на стадии выполнения, а не во время компиляции.

В предыдущих изданиях книги я писал:

«Это не просто мелкая неприятность, а потенциальный источник трудноуловимых ошибок. Если одна часть (или несколько частей) программы вставляет объекты в контейнер, а в другой части программы обнаруживается, что в контейнер был помещен недопустимый объект, вам придется искать, где именно была выполнена неверная операция вставки».

Но позже я задумался над этим аргументом, и у меня появились сомнения. Во-первых, насколько часто это происходит? Не помню, чтобы такая ошибка встретилась в моей программе. Когда я спрашивал людей на конференциях, мне тоже не удалось найти никого, с кем бы это случилось. В другой книге использовался пример списка с именем **files**, содержащего объекты **String**, — в этом

примере казалось абсолютно логичным добавить в список объект типа File, так что объекту, вероятно, стоило присвоить имя fileNames. Какую бы проверку типов ни обеспечивал язык Java, программист все равно может написать малопонятную программу — а плохо написанная программа, даже если она компилируется, все равно остается плохо написанной. Вероятно, нормальный разработчик присвоит контейнеру понятное имя вроде cats, которое послужит предупреждением для программиста, пытающегося занести в контейнер другой объект, отличный от Cat. Но, даже если это и произойдет, как долго такая ошибка останется скрытой? Здравый смысл подсказывает, что исключение произойдет вскоре после начала тестирования с реальными данными.

Один автор даже предположил, что такая ошибка может «оставаться скрытой несколько лет». Но я что-то не помню потока сообщений от людей, у которых возникали проблемы с поиском ошибок «Dog в списке Cat», или хотя бы с их частым появлением. Так неужели такая заметная и довольно сложная возможность, как параметризация, была включена в Java из-за проблем такого рода?

Я считаю, что *побудительной причиной* для включения параметризации в язык (не обязательно конкретной реализации ее в Java!) является *выразительность*, а не создание типизованных контейнеров. Типизованные контейнеры — всего лишь побочный эффект возможности создания универсального кода. Таким образом, хотя аргумент «Dog в списке Cat» часто используется для оправдания параметризации, этот аргумент спорен.

Из-за того, что параметризация была «встроена» в Java (а не проектировалась как составная часть языка с самого начала), некоторые контейнеры получились не такими мощными, как хотелось бы. Для примера взгляните на Map, особенно на методы containsKey(Object key) и get(Object key). Если бы эти классы проектировались в расчете на параметризацию, в этих методах вместо Object использовались бы параметризованные типы; тем самым обеспечивались бы необходимые проверки стадии компиляции. Скажем, в аналогичных контейнерах C++ тип ключа всегда проверяется во время компиляции.

Бесспорно, введение любого механизма параметризации в более позднюю версию языка, получившего широкое распространение, — крайне хлопотная затея. В C++ шаблоны были включены в исходную ISO-версию языка, так что они фактически *всегда* являлись его составной частью. В Java параметризация была введена лишь спустя 10 лет после выхода первой версии. Этот факт породил немало проблем с миграцией кода, а также оказал значительное влияние на архитектуру. В результате программисты страдают из-за близорукости, проявленной проектировщиками языка при создании версии 1.0. Конечно, при создании исходной версии они знали о шаблонах C++ и даже рассматривали возможность включения их в язык, но по тем или иным причинам решили этого не делать (скорее всего, просто торопились). В результате пострадал как язык, так и работающие на нем программисты. Только время покажет, как подход к параметризации в Java отразится на самом языке.

Массивы 15

В конце главы 5 было показано, как определить и инициализировать массив.

Программист создает и инициализирует массивы, извлекает из них элементы по целочисленным индексам, а размер массива остается неизменным. Как правило, при работе с массивами этого вполне достаточно, но иногда приходится выполнять более сложные операции, а также оценивать эффективность массива по сравнению с другими контейнерами. В этой главе массивы рассматриваются на более глубоком уровне.

Особенности массивов

В Java существует немало разных способов хранения объектов, почему же массивы занимают особое место?

Массивы отличаются от других контейнеров по трем показателям: эффективность, типизация и возможность хранения примитивов. Массивы Java являются самым эффективным средством хранения и произвольного доступа к последовательности ссылок на объекты. Массив представляет собой простую линейную последовательность, благодаря чему обращения к элементам осуществляются чрезвычайно быстро. За скорость приходится расплачиваться тем, что размер объекта массива фиксируется и не может изменяться на протяжении его жизненного цикла. Как говорилось в главе 11, контейнер `ArrayList` способен автоматически выделять дополнительное пространство, выделяя новый блок памяти и перемещая в него все ссылки из старого. Хотя обычно `ArrayList` отдается предпочтение перед массивами, за гибкость приходится расплачиваться, и `ArrayList` значительно уступает по эффективности обычному массиву.

И массивы, и контейнеры защищены от возможных злоупотреблений. При выходе за границу массива или контейнера происходит исключение `RuntimeException`, свидетельствующее об ошибке программиста.

До появления параметризации другим контейнерным классам приходилось работать с объектами так, как если бы они не имели определенного типа. Иначе говоря, объекты рассматривались как принадлежащие к типу Object, корневому для всей иерархии классов в Java. Массивы удобнее «старых» контейнеров тем, что массив создается для хранения конкретного типа. Проверка типов на стадии компиляции не позволит использовать неверный тип или неверно интерпретировать извлекаемый тип. Конечно, Java так или иначе запретит отправить неподходящее сообщение объекту на стадии компиляции или выполнения, так что ни один из способов не является более рискованным по сравнению с другим. Просто будет удобнее, если компилятор укажет на существующую проблему, и снижается вероятность того, что пользователь программы получит неожиданное исключение.

Массив может содержать примитивные типы, а «старые» контейнеры — нет. С другой стороны, параметризованные контейнеры могут проверять тип хранимых объектов, а благодаря автоматической упаковке они работают так, как если бы поддерживали хранение примитивов, поскольку преобразование выполняется автоматически. В следующем примере массивы сравниваются с параметризованными 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));
```

продолжение

```
    }
} /* 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, а единственное очевидное различие заключается в том, что массивы используют для обращения к элементам конструкцию [], а List — методы вроде **add()** или **get()**. Разработчики языка намеренно сделали массивы и **ArrayList** настолько похожими, чтобы программисту было концептуально проще переключаться между ними. Но, как было показано в главе 11, контейнеры обладают более широкими возможностями, чем массивы.

С появлением механизма автоматической упаковки контейнеры по удобству работы с примитивами почти не уступают массивам. Единственным реальным преимуществом массивов остается их эффективность. Впрочем, при решении более общих проблем может оказаться, что возможности массивов слишком ограничены, и тогда приходится пользоваться контейнерными классами.

Массив как объект

С каким бы типом массива вы ни работали, идентификатор массива в действительности представляет собой ссылку на объект, созданный в динамической памяти. Этот объект содержит ссылки на другие объекты и создается либо неявно (в синтаксисе инициализации массива), либо явно конструкцией **new**. Одной из частей объекта массива (а по сути, единственным доступным полем) является доступная только для чтения переменная **length**, которая указывает, сколько элементов может храниться в объекте массива. Весь доступ к объекту массива ограничивается синтаксисом [ ].

Следующий пример демонстрирует различные способы инициализации массивов и присваивания ссылок на массивы. Он также наглядно показывает, что массивы объектов и массивы примитивных типов практически идентичны. Единственное различие заключается в том, что массивы объектов содержат ссылки, а массивы примитивов содержат сами примитивные 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) {
        // Массивы 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) // Проверка ссылки на действительность
                        c[i] = new BerylliumSphere();
        // Агрегатная инициализация.
        BerylliumSphere[] d = { new BerylliumSphere(),
                new BerylliumSphere(), new BerylliumSphere()
        };
        // Динамическая агрегатная инициализация
        a = new BerylliumSphere[]{
                new BerylliumSphere(), new BerylliumSphere(),
        };
        // (Завершающая запятая не обязательна в обоих случаях)
        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; // Ссылка null
        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, 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 — неинициализированная локальная переменная, и компилятор не позволяет что-либо делать с этой ссылкой до тех пор, пока она не будет соответствующим образом инициализирована. Массив b инициализируется массивом ссылок на объекты BerylliumSpere, хотя ни один такой объект в массив не заносится. Несмотря на это, мы можем запросить размер массива, потому что b указывает на действительный объект. В этом проявляется некоторый недостаток массивов: поле length сообщает, сколько элементов *может быть* помещено в массив, то есть размер объекта массива, а не количество хранящихся в нем элементов. Тем не менее при создании объекта массива все ссылки автоматически инициализируются значением null, и, чтобы узнать, связан ли некоторый элемент массива с объектом, достаточно проверить ссылку на равенство null. Аналогично, массивы примитивных типов автоматически инициализируются нулями для числовых типов: (char)0 для char и false для boolean.

Массив c демонстрирует создание массива с последующим присваиванием объектов BerylliumSphere всем элементам массива. Массив d демонстрирует синтаксис «агрегатной инициализации», при котором объект массива создается (с ключевым словом new, как массив c) *и* инициализируется объектами BerylliumSphere, причем все это происходит в одной команде.

Следующую конструкцию инициализации массива можно назвать «динамической агрегатной инициализацией». Агрегатная инициализация, используемая d, должна использоваться в точке определения d, но при втором синтаксисе объект массива может создаваться и использоваться в любой точке. Предположим, методу hide() передается массив объектов BerylliumSphere. Его вызов может выглядеть так:

```
hide(d);
```

однако массив, передаваемый в аргументе, также можно создать динамически:

```
hide(new BerylliumSphere[]{ new BerylliumSphere(),
  new BerylliumSphere() });
```

Во многих ситуациях такой синтаксис оказывается более удобным.

Выражение

```
a=d;
```

показывает, как взять ссылку, связанную с одним объектом массива, и присвоить ее другому объекту массива, как это делается с любым другим типом ссылки на объект. В результате a и d указывают на один объект массива в куче.

Вторая часть ArrayOptions.java показывает, что примитивные массивы работают точно так же, как массивы объектов, *за исключением* того, что примитивные значения сохраняются в них напрямую.

Возврат массива

Предположим, вы пишете метод, который должен возвращать не отдельное значение, а целый набор значений. В таких языках, как C и C++, это сделать нелегко, потому что возвращается из метода не массив, а только указатель на массив.

При этом возникают проблемы, поскольку сложности с управлением жизненным циклом массива могут привести к утечке памяти.

В Java вы просто возвращаете массив. Вам не нужно беспокоиться о нем — массив будет существовать до тех пор, пока он вам нужен, а когда надобность в нем отпадет, массив будет уничтожен уборщиком мусора.

В качестве примера рассмотрим возвращение массива 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 results с элементами String. Размер массива равен n; он определяется аргументом, передаваемым при вызове метода. Далее метод случайным образом выбирает элементы из массива FLAVORS и помещает их в массив results, возвращаемый методом. Массив возвращается точно так же, как любой другой объект, — по ссылке. При этом не важно, был ли массив создан методом flavorSet(), или он был создан в другом месте. Массив останется с вами все время, пока он будет нужен, а потом уборщик мусора позаботится о его уничтожении.

Из выходных данных видно, что метод flavorSet() действительно выбирает случайное подмножество элементов при каждом вызове.

Многомерные массивы

Создание многомерных массивов в Java не вызывает особых сложностей. Для многомерных массивов примитивных типов 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 описывает новую размерность массива.

В этом примере используется метод Java SE5 Arrays.deepToString(). Как видно из выходных данных, он преобразует многомерные массивы в String.

Массив также может создаваться ключевым словом new. Пример создания трехмерного массива выражением new:

```
//: arrays/ThreeDWithNew.java
import java.util.*;

public class ThreeDWithNew {
     public static void main(String[] args) {
               // Трехмерный массив фиксированной длины:
               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 массиву примитивных типов не заданы явные значения, он автоматически инициализируется значениями по умолчанию. Массивы объектов инициализируются ссылками null.

Векторы массивов, образующих матрицу, могут иметь разную длину (это называется *ступенчатым массивом*):

```
//: 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 остаются неопределенными. Вторая конструкция new в цикле for заполняет элементы, но оставляет третий индекс неопределенным вплоть до выполнения третьего new.

Массивы с не-примитивными элементами заполняются аналогичным образом. Пример объединения нескольких 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(),
                                 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]]
*///:~
```

Массив spheres также является ступенчатым, то есть длины вложенных списков объектов различаются.

Механизм автоматической упаковки работает с инициализаторами массивов:

```
//: arrays/AutoboxingArrays.java
import java.util.*;

public class AutoboxingArrays {
     public static void main(String[] args) {
               Integer[][] a = { // Автоматическая упаковка
                        { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 },
```

продолжение ⇨

```
                        { 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 поэлементное построение массива не-примитивных 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 присутствует только для того, чтобы поместить менее тривиальное значение в Integer.

Метод Arrays.deepToString() работает как с массивами примитивных типов, так и с массивами объектов:

```
//: arrays/MultiDimWrapperArray.java
// Multidimensional arrays of "wrapper" objects.
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 в массивах **Integer** и **Double** механизм автоматической упаковки Java SE5 создает необходимые объекты-«обертки».

Массивы и параметризация

В общем случае массивы и параметризация плохо сочетаются друг с другом. Например, массивы не могут инициализироваться параметризованными типами:

```
Peel<Banana>[] peels = new Peel<Banana>[10]; // Не разрешено
```

Стирание удаляет информацию о параметре типа, а массив должен знать точный тип хранящихся в нем объектов для обеспечения безопасности типов.

Впрочем, параметризовать можно сам тип массива:

```
//: 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, как удобно использовать параметризованный метод вместо параметризованного класса: вам не придется создавать очередную «версию» класса с параметром для каждого типа, к которому он применяется, и его можно сделать **static**. Конечно, параметризованный класс не всегда можно заменить

параметризованным методом, но такое решение может оказаться предпочтительным.

Как выясняется, не совсем правильно говорить, что вы не можете создавать массивы параметризованных типов. Действительно, компилятор не позволит *создать экземпляр* массива параметризованного типа, но вы можете создать ссылку на такой массив. Пример:

```
List<String>[] ls.
```

Такая конструкция проходит проверку без малейших возражений со стороны компилятора. И хотя вы не можете создать объект массива с параметризацией, можно создать объект непараметризованного типа и преобразовать его:

```
//· arrays/ArrayOfGenerics.java
// Возможность создания массивов параметризованных типов
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; // Предупреждение о
                                      // непроверенном преобразовании
             ls[0] = new ArrayList<String>();
             // Приводит к ошибке на стадии компиляции
             //! ls[1] = new ArrayList<Integer>();

             // Проблема: List<String> является подтипом Object
             Object[] objects = ls; // Поэтому присваивание возможно
             // Компилируется и выполняется без ошибок и предупреждений
             objects[1] = new ArrayList<Integer>();

             // Но если ваши потребности достаточно элементарны:
             // создать массив параметризованных типов можно, хотя
             // и с предупреждением о "непроверенном" преобразовании·
             List<BerylliumSphere>[] spheres =
                     (List<BerylliumSphere>[])new List[10];
             for(int i = 0; i < spheres.length; i++)
                     spheres[i] = new ArrayList<BerylliumSphere>();
     }
} ///·~
```

Мы видим, что при при получении ссылки на **List<String>[]** выполняется некоторая проверка на стадии компиляции. Проблема в том, что массивы ковариантны, поэтому **List<String>[]** также является **Object[]**, поэтому вашему массиву можно присвоить **ArrayList<Integer>** без выдачи ошибок на стадии компиляции или выполнения.

Если вы уверены в том, что восходящее преобразование выполняться не будет, а ваши потребности относительно просты, можно создать массив параметризованных типов, обеспечивающий простейшую проверку типов на стадии компиляции. Тем не менее параметризованный контейнер практически всегда оказывается более удачным решением, чем массив параметризованных типов.

Создание тестовых данных

При экспериментах с массивами (и программами вообще) полезно иметь возможность простого заполнения массивов тестовыми данными. Инструментарий, описанный в этом разделе, заполняет массив объектными значениями.

Arrays.fill()

Класс Arrays из стандартной библиотеки Java содержит весьма тривиальный метод fill(): он всего лишь дублирует одно значение в каждом элементе массива, а в случае объектов копирует одну ссылку в каждый элемент. Пример:

```
//· arrays/FillingArrays.java
// Использование 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)),
               // Интервальные операции:
               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]
```

продолжение ⇩

```
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]
*///:~
```

Метод заполняет либо весь массив, либо, как показывают две последние команды, диапазон его элементов. Но, поскольку вызывать Arrays.fill() можно только для одного значения данных, полученные результаты не слишком полезны.

Генераторы данных

Чтобы создавать менее тривиальные массивы данных с более гибкими возможностями, мы воспользуемся концепцией *генераторов*, представленной в главе 14. Генератор способен выдавать любые данные по вашему выбору (напомню, что он является примером паттерна *«стратегия»* — разные генераторы представляют разные стратегии).

В этом разделе будут представлены некоторые готовые генераторы, но вы также сможете легко определить собственный генератор для своих потребностей.

Для начала рассмотрим простейший набор счетных генераторов для всех примитивных типов и String. Классы генераторов вложены в класс CountingGenerator, чтобы они могли обозначаться именами генерируемых объектов. Например, генератор, создающий объекты Integer, будет создаваться выражением new CountingGenerator.Integer():

```
//: net/mindview/util/CountingGenerator.java
// Простые реализации генераторов.
package net.mindview.util;

public class CountingGenerator {
    public static class
    Boolean implements Generator<java.lang.Boolean> {
            private boolean value = false;
            public java.lang.Boolean next() {
                    value = !value; // Поочередное 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 «счетности». В случае **CountingGenerator.Character** это повторение символов 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 на то,

что CountingGenerator.String использует базовую конструкцию Generator<java.lang.Character> вместо конкретной ссылки на CountingGenerator.Character.

Следующая тестовая программа использует рефлексию с идиомой вложенных генераторов, что позволяет применять ее для любого набора генераторов, построенного по указанному образцу:

```
//: 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
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 серию вложенных объектов Generator, каждый из которых имеет конструктор по умолчанию (то есть без аргументов). Рефлексионный метод getClasses() выдает информацию обо всех вложенных классах. Далее метод test() создает экземпляр каждого генератора и выводит результаты, полученные при десятикратном вызове next().

Следующий набор генераторов основан на случайных числах. Так как конструктор Random инициализируется константой, результаты будут повторяться при каждом запуске программы:

```
//: 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 случайного генератора Character
        { cg = new Character(); } // Инициализатор
        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() {
                // Отсечение до двух разрядов в дробной части.
                int trimmed = Math.round(r nextFloat() * 100);
                return ((float)trimmed) / 100;
        }
}
```

продолжение ⇩

```
        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;
                }
        }
} ///:~
```

Как видите, RandomGenerator.String наследует от CountingGenerator.String, просто подключая новый генератор Character.

Чтобы генерируемые числа были не слишком велики, RandomGenerator.Integer по умолчанию берет остаток от деления на 10 000, но перегруженный конструктор позволяет выбрать меньшее значение. Аналогичный подход используется и для RandomGenerator.Long. Для генераторов Float и Double цифры в дробной части усекаются.

Для тестирования RandomGenerator можно воспользоваться уже готовым классом GeneratorsTest:

```
//: 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
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 true
*///:~
```

Чтобы изменить количество генерируемых значений, воспользуйтесь public-полем GeneratorsTest.size.

Создание массивов с использованием генераторов

Для создания массивов на основе Generator нам потребуются два вспомогательных класса. Первый использует произвольный Generator для получения массива типов, производных от Object. Для решения проблемы с примитивами второй класс получает произвольный массив с объектами-«обертками» и строит для него соответствующий массив примитивов.

Первый вспомогательный класс может работать в двух режимах, представленных перегруженным статическим методом array(). Первая версия метода получает существующий массив и заполняет его с использованием Generator; вто-

рая версия получает объект Class, Generator и количество элементов и создает новый массив, который также заполняется с использованием Generator. Помните, что при этом создаются только массивы субтипов Object, но не массивы примитивных типов:

```
//· 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 создает объект Collection, заполненный элементами, которые были созданы генератором gen. Количество элементов определяется вторым аргументом конструктора. Все субтипы Collection содержат метод toArray(), заполняющий массив-аргумент элементами из Collection.

Второй метод использует рефлексию для динамического создания нового массива соответствующего типа и размера. Затем созданный массив заполняется таким же способом, как в первом методе.

Чтобы протестировать Generated, мы воспользуемся одним из классов CountingGenerator, описанных 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 инициализируется, эти данные перезаписываются при вызове Generated.array(). Инициализация b показывает, как создать заполненный массив «с нуля».

Параметризация не работает с примитивами, поэтому для заполнения примитивных массивов будут использоваться генераторы. Для решения этой проблемы мы создадим преобразователь, который получает произвольный массив объектных «оберток» и преобразует его в массив соответствующих примитивных типов. Без него 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;
     }
     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 примитивный массив правильной длины, а затем копирует элементы из массива in. Обратите внимание на выполнение автоматической распаковки в выражении

```
result[i] = in[i];
```

Пример использования ConvertTo с обеими версиями 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 инструментарий создания массивов с классами RandomGenerator:

```
//: arrays/TestArrayGeneration.java
// Test the tools that use generators to fill arrays.
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));
```

продолжение ⇨

```
            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]
*///:~
```

Как видите, все версии **ConvertTo.primitive()** работают правильно.

Вспомогательный инструментарий Arrays

В библиотеку **java.util** включен класс **Arrays**, содержащий набор вспомогательных статических методов для работы с массивами. Основных методов шесть: **equals()** сравнивает два массива (также существует версия **deepEquals()** для многомерных массивов); **fill()** был описан ранее в этой главе; **sort()** сортирует массив; **binarySearch()** ищет элемент в отсортированном массиве; **toString()** создает представление массива в формате **String**, а **hashCode()** генерирует хеш-код массива. Все эти методы перегружены для всех примитивных типов и **Object**. Кроме того, метод **Arrays.asList()** преобразует любую последовательность или массив в контейнер **List** (см. главу 11).

Прежде чем обсуждать методы **Arrays**, следует рассмотреть еще один полезный метод, не входящий в **Arrays**.

Копирование массива

Стандартная библиотека Java содержит статический метод **System.arraycopy()**, который копирует массивы значительно быстрее, чем при ручном копировании в цикле **for**. Метод **System.arraycopy()** перегружен для работы со всеми типами. Пример для массивов **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));
               // Объекты:
               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, 47, 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, начальная позиция копирования в исходном массиве, приемный массив, начальная позиция копирования в приемном массиве и количество копируемых элементов. Естественно, любое нарушение границ массива приведет к исключению.

Приведенный пример показывает, что копироваться могут как примитивные, так и объектные массивы. Однако при копировании объектных массивов копируются только ссылки, но не сами объекты. Такая процедура называется *поверхностным копированием*.

System.arraycopy() не выполняет ни автоматической упаковки, ни автоматической распаковки — типы двух массивов должны полностью совпадать.

Сравнение массивов

Класс Arrays содержит метод equals() для проверки на равенство целых массивов. Метод перегружен для примитивов и Object. Чтобы два массива считались равными, они должны содержать одинаковое количество элементов, и каждый элемент должен быть эквивалентен соответствующему элементу другого массива (проверка осуществляется вызовом equals() для каждой пары; для примитивов используется метод equals() объектной «обертки» — например, Integer.equals() для int). Пример:

```
//  arrays/ComparingArrays java
// Using 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
*///·~
```

Сначала массивы a1 и a2 полностью совпадают, поэтому результат сравнения равен true, но после изменения одного из элементов будет получен результат false. В последнем случае все элементы s1 указывают на один объект, тогда как s2 содержит пять разных объектов. Однако проверка равенства определяется содержимым (с вызовом Object.equals()), поэтому результат равен true.

Сравнение элементов массивов

Сравнения, выполняемые в ходе сортировки, зависят от фактического типа объектов. Конечно, можно написать разные методы сортировки для всех возможных типов, но такой код придется модифицировать при появлении новых типов.

Главной целью проектирования является «отделение того, что может измениться, от того, что остается неизменным». В данном случае неизменным остается общий алгоритм сортировки, а изменяется способ сравнения объектов. Вместо того, чтобы размещать код сравнения в разных функциях сортировки,

мы воспользуемся паттерном проектирования *«стратегия»*. В этом паттерне переменная часть кода инкапсулируется в отдельном классе. Объект стратегии передается коду, который остается неизменным, и последний использует стратегию для реализации своего алгоритма. При этом разные объекты выражают разные способы сравнения, но передаются универсальному коду сортировки.

В Java функциональность сравнения может выражаться двумя способами. Первый основан на «естественном» методе сравнения, который включается в класс при реализации **java.lang.Comparable** — очень простого интерфейса с единственным методом **compareTo()**. В аргументе метод получает другой объект того же типа. Он выдает отрицательное значение, если текущий объект меньше аргумента, нуль при равенстве и положительное значение, если текущий объект больше аргумента.

В следующем примере класс реализует **Comparable**, а для демонстрации совместимости используется метод стандартной библиотеки Java **Arrays.sort()**:

```
//: arrays/CompType.java
// Реализация классом интерфейса Comparable.
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("перед сортировкой:");
             print(Arrays.toString(a));
             Arrays.sort(a);
             print("после сортировки:");
             print(Arrays.toString(a));
```

продолжение ⇩

```
    }
} /* Output:
перед сортировкой:
[[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]
]
после сортировки:
[[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]
]
*///:~
```

Определяя метод сравнения, вы несете полную ответственность за принятие решения о его результатах. В приведенном примере в сравнении используются только значения i, а значения j игнорируются.

Метод **generator()** производит объект, реализующий интерфейс **Generator**, создавая анонимный внутренний класс. Объект строит объекты **CompType**, инициализируя их случайными значениями. В **main()** генератор заполняет массив **CompType**, который затем сортируется. Если интерфейс **Comparable** не реализован, то при попытке вызова **sort()** произойдет исключение **ClassCastException**. Это объясняется тем, что **sort()** преобразует свой аргумент к типу **Comparable**.

Теперь представьте, что вы получили класс, который не реализует интерфейс **Comparable**... а может быть, реализует, но вам не нравится, как он работает, и вы хотели бы задать для типа другой метод сравнения. Для решения проблемы создается отдельный класс, реализующий интерфейс **Comparator**. Он содержит два метода, **compare()** и **equals()**. Впрочем, вам практически никогда не придется реализовывать **equals()** — разве что при особых требованиях по быстродействию, потому что любой создаваемый класс неявно наследует от класса **Object** метод **equals()**.

Класс **Collections** содержит метод **reverseOrder()**, который создает **Comparator** для порядка сортировки, обратного по отношению к естественному. Он может быть применен к **CompType**:

```
//: arrays/Reverse.java
// Метод 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("перед сортировкой ");
    print(Arrays.toString(a));
    Arrays.sort(a, Collections.reverseOrder());
    print("после сортировки:");
    print(Arrays.toString(a));
  }
```

```
} /* Output:
перед сортировкой:
[[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]
]
после сортировки:
[[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]
]
*///:~
```

Наконец, вы можете написать собственную реализацию **Comparator**. В следующем примере объекты **CompType** сравниваются по значениям **j** вместо **i**:

```
//: arrays/ComparatorTest.java
// Реализация Comparator
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("перед сортировкой:");
             print(Arrays.toString(a));
             Arrays.sort(a, new CompTypeComparator());
             print("после сортировки:");
             print(Arrays.toString(a));
     }
} /* Output:
перед сортировкой:
[[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]
]
после сортировки:
[[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]
]
*///:~
```

Сортировка массива

Встроенные средства сортировки позволяют отсортировать любой массив примитивов, любой массив объектов, реализующих Comparable или ассоциированных с объектом Comparator[1]. Следующий пример генерирует случайные объекты String и сортирует их:

```
//: arrays/StringSorting.java
// Sorting an array of Strings.
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]
*///.~
```

В выходных данных алгоритма сортировки String бросается в глаза то, что алгоритм является лексикографическим, то есть все слова, начинающиеся с прописных букв, предшествуют любым словам, начинающимся со строчных букв. Если вы хотите, чтобы слова группировались независимо от регистра символов, используйте режим String.CASE_INSENSITIVE_ORDER, как показано в последнем вызове sort() из приведенного примера.

Алгоритм сортировки, используемый стандартной библиотекой Java, спроектирован в расчете на оптимальность для сортируемого типа — быстрая сортировка для примитивов, надежная сортировка слиянием для объектов. Обычно вам не приходится беспокоиться о быстродействии, если только профайлер не укажет, что процесс сортировки тормозит работу программы.

Поиск в отсортированном массиве

После того, как массив будет отсортирован, вы сможете быстро найти нужный элемент методом Arrays.binarySearch(). Попытка вызова binarySearch()

[1] Как ни странно, в Java 1.0 и 1.1 отсутствует встроенная поддержка сортировки String.

для несортированного массива приведет к непредсказуемым последствиям. В следующем примере генератор RandomGenerator.Integer заполняет массив, после чего тот же генератор используется для получения искомых значений:

```
//. arrays/ArraySearching java
// Using 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("Отсортированный массив: " + Arrays toString(a)).
               while(true) {
                        int r = gen.next().
                        int location = Arrays binarySearch(a. r).
                        if(location >= 0) {
                                 print("Значение " + r + " находится в позиции " +
location +
                                          ". a[" + location + "] = " + a[location]);
                                 break. // Выход из цикла while
                        }
               }
     }
} /* Output
Отсортированный массив· [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]
Значение 322 находится в позиции 8. a[8] = 322
*/// ~
```

Цикл while генерирует случайные значения как искомые до тех пор, пока одно из них не будет найдено в массиве.

Если искомое значение найдено, метод Arrays.binarySearch() возвращает неотрицательный результат. В противном случае возвращается отрицательное значение, представляющее позицию элемента при вставке (при сохранении сортировки массива). Если массив содержит повторяющиеся значения, алгоритм поиска не дает гарантий относительно того, какой именно из дубликатов будет обнаружен. Алгоритм проектировался не для поддержки дубликатов, а для того, чтобы переносить их присутствие. Если вам нужен отсортированный список без повторений элементов, используйте TreeSet (для сохранения порядка сортировки) или LinkedHashSet (для сохранения порядка вставки). Эти классы автоматически берут на себя все детали. Только в ситуациях, критичных по быстродействию, эти классы заменяются массивами с ручным выполнением операций.

При сортировке объектных массивов с использованием Comparator (примитивные массивы не позволяют выполнять сортировку с Comparator) необходимо включать тот же объект Comparator, что и при использовании binarySearch() (перегруженной версии). Например, программу StringSorting.java можно модифицировать для выполнения поиска:

```
//· arrays/AlphabeticSearch.java
// Поиск с 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("Индекс: "+ 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]
Индекс  10
HxxHv
*///.~
```

Объект **Comparator** передается перегруженному методу **binarySearch()** в третьем аргументе. В приведенном примере успех поиска гарантирован, так как искомое значение выбирается из самого массива.

Резюме

В этой главе вы убедились в том, что язык Java предоставляет неплохую поддержку низкоуровневых массивов фиксированного размера. Такие массивы отдают предпочтение производительности перед гибкостью. В исходной версии Java низкоуровневые массивы фиксированного размера были абсолютно необходимы — не только потому, что проектировщики Java решили включить в язык примитивные типы (также по соображениям быстродействия), но и потому, что поддержка контейнеров в этой версии была крайне ограниченной.

В последующих версиях Java поддержка контейнеров была значительно улучшена. Сейчас контейнеры превосходят массивы во всех отношениях, кроме быстродействия, хотя производительность контейнеров была значительно улучшена. С добавлением автоматической упаковки и параметризации хранение примитивов в контейнерах требует меньших усилий и способствует дальнейшему переходу с низкоуровневых массивов на контейнеры. Так как параметризация открыла доступ к типизованным контейнерам, в этом отношении массивы тоже утратили исходные преимущества.

Как было показано в этой главе, параметризация плохо сочетается с контейнерами. Даже если вам удастся тем или иным способом заставить их работать вместе, во время компиляции вы будете получать предупреждения.

Все эти факторы показывают, что при программировании для последних версий Java следует отдавать предпочтение контейнерам перед массивами. На массивы следует переходить только при критичных требованиях по быстродействию — и только в том случае, если переход принесет ощутимую пользу.

16 Система ввода/вывода Java

Создание хорошей системы ввода/вывода является одной из труднейших задач разработчика языка. Доказательством этого утверждения служит множество 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 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 потеряться в них проще простого (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 посылку и прием *байтов* классам добавились основанные на Юникод классы, работающие с *символами*. В JDK 1.4 классы nio (от сочетания «new I/O», «новый ввод/вывод») призваны улучшить производительность и функциональность. В результате, чтобы понять общую картину ввода/вывода в Java и начать использовать ее, вам придется изучить порядочный ворох классов. Вдобавок не менее важно понять и изучить эволюцию библиотеки ввода/вывода, несмотря на вашу очевидную реакцию: «Избавьте меня от истории! Просто покажите, как работать с библиотекой!» Если не уяснить причины изменений, проведенных в библиотеке ввода/вывода, вскоре мы запутаемся в ней и не сможем твердо аргументировать сделанный нами выбор в пользу того или иного класса.

В этой главе мы познакомимся с различными классами, отвечающими за ввод/вывод в библиотеке Java, а также научимся использовать их.

Класс File

Перед тем как перейти к классам, которые осуществляют реальные запись и чтение данных, мы рассмотрим вспомогательные инструменты библиотеки, предназначенные для работы с файлами и каталогами.

Имя класса File весьма обманчиво: легко подумать, что оно всегда ссылается на файл, но это не так. Класс File может представлять как *имя* определенного файла, так, *имена* группы файлов, находящихся в каталоге. Если класс представляет каталог, его метод list() возвращает массив строк с именами всех файлов. Использовать в данной ситуации массив (а не более гибкий контейнер) очень удобно: количество файлов в каталоге фиксировано, как и размер массива, а если понадобится узнать имена файлов в другом каталоге, достаточно создать еще один объект File. Следующий раздел покажет, как использовать этот класс в совокупности с тесно связанным с ним интерфейсом FilenameFilter.

Список каталогов

Предположим, вы хотите получить содержимое 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 всех файлов с расширением .java, — используйте «фильтр», то есть класс, который описывает критерии отбора объектов File.

Рассмотрим пример. Заметьте, что полученный список без всяких дополнительных усилий сортируется (по алфавиту) с помощью метода java.util.Array.sort() и объекта String.CASE_INSENSITIVE_ORDER:

```
//: io/DirList.java
// Вывод списка каталогов с использованием регулярных выражений
// {Параметры: "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. Посмотрите, как просто выглядит этот интерфейс:

```
public interface FilenameFilter {
    boolean accept(File dir, String name),
}
```

Это показывает, что данный тип объекта должен поддерживать метод с именем accept(), который вызывается методом list() с целью определения того, какие имена файлов должны включаться в выходной список, а какие нет. Перед нами один из примеров паттерна *«стратегия»*: list() реализует базовую функциональность, а FilenameFilter предоставляет алгоритм, необходимый для работы list(). Так как метод list() принимает в качестве аргумента объект FilenameFilter, ему можно передать любой объект любого класса, лишь бы он реализовывал интерфейс FilenameFilter (даже во время выполнения). Таким образом легко изменять результат работы метода list(). Целью данного паттерна является обеспечение гибкости в поведении кода.

Метод accept() получает объект File, представляющий собой каталог, в котором был найден данный файл, и строку с именем файла. Помните, что метод list() вызывает accept() для каждого файла, обнаруженного в каталоге, чтобы определить, какие из них следует включить в выходной список — в зависимости от возвращаемого значения accept() (значение типа boolean).

Метод accept() использует объект регулярного выражения matcher, чтобы посмотреть, соответствует ли имя файла выражению regex. Метод list() возвращает массив.

Безымянные 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/DirList java
// Использование безымянных внутренних классов
// {Параметры  "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,
```

продолжение ⇩

```
            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
*///:~
```

Заметьте, что аргумент метода **filter()** должен быть неизменным (**final**). Это необходимо для того, чтобы внутренний класс смог получить к нему доступ даже за пределами области определения аргумента.

Несомненно, структура программы улучшилась хотя бы потому, что объект **FilenameFilter** теперь неразрывно связан с внешним классом **DirList2**. Впрочем, можно сделать следующий шаг и определить безымянный внутренний класс как аргумент метода **list()**, в результате чего программа станет еще более компактной:

```
//: io/DirList3.java
// Создание безымянного внутреннего класса "на месте".
// {Параметры: "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) объявлен аргумент метода main(), так как безымянный внутренний класс использует параметр командной строки (args[0]) напрямую.

Именно так безымянные внутренние классы позволяют быстро создать «одноразовый» класс, полезный только для решения одной конкретной задачи. Одно из преимуществ такого подхода состоит в том, что весь код, решающий некоторую задачу, находится в одном месте. С другой стороны, полученный код не слишком хорошо читается, поэтому при их использовании необходимо действовать осмотрительно.

Проверка существования и создание каталогов

Класс File не ограничивается представлением существующих файлов или каталогов, он способен на большее. Он также может использоваться для создания нового каталога или даже дерева каталогов, если последние не существуют. Можно также узнать свойства файлов (размер, дату последнего изменения, режим чтения (записи)), определить, файл или каталог представляет объект File, удалить файл. Следующая программа демонстрирует некоторые методы класса File (за полной информацией обращайтесь к документации JDK, доступной для загрузки с сайта java.sun.com):

```
//: io/MakeDirectories.java
// Использование класса File для создания
// каталогов и выполнения операций с файлами.
// {Параметры: MakeDirectoriesTest}
import java.io.*;

public class MakeDirectories {
     private static void usage() {
              System.err.println(
                       "Использование:MakeDirectories путь1 ...\n" +
                       "Создает все пути\n" +
                       "Использование:MakeDirectories -d путь1 ...\n" +
                       "Удаляет все пути\n" +
                       "Использование:MakeDirectories -r путь1 путь2\n" +
                       "Переименовывает путь1 в путь2\n");
              System.exit(1);
     }
     private static void fileData(File f) {
              System.out.println(
                        "Полное имя: " + f.getAbsolutePath() +
                        "\n доступно для чтения: " + f.canRead() +
                        "\n доступно для записи: " + f.canWrite() +
                        "\n имя файла getName(): " + f.getName() +
                        "\n родительский каталог getParent(): " + f.getParent() +
```

продолжение ⇩

```
                        "\n путь getPath()  " + f getPath() +
                        "\n размер  " + f length() +
                        "\n последнее изменение  " + f lastModified()).
                if(f isFile())
                        System out println("Это файл").
                else if(f isDirectory())
                        System out println("Это каталог").
        }
        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 метода 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 + " существует");
                                if(del) {
                                        System out println("удаление   " + f);
                                        f delete();
                                }
                        }
                        else { // Не существует
                                if(!del) {
                                        f.mkdirs().
                                        System.out println("создано " + f);
                                }
                        }
                        fileData(f).
                }
        }
} /* Output:
создано MakeDirectoriesTest
Полное имя: d·\aaa-TIJ4\code\io\MakeDirectoriesTest
 доступно для чтения· true
 доступно для записи. true
 имя файла getName()  MakeDirectoriesTest
 родительский каталог getParent()  null
 путь getPath()  MakeDirectoriesTest
 размер  0
 последнее изменение  1101690308831
Это каталог
*///.~
```

В методе fileData() продемонстрированы различные методы, предназначенные для получения информации о файлах и каталогах.

Сначала 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.

Если вы поэкспериментируете с этой программой, то увидите, что создать пути произвольной сложности очень просто, поскольку всю работу за вас фактически делает метод mkdirs().

Ввод и вывод

В библиотеках ввода/вывода часто используется абстрактное понятие *потока* (stream) — произвольного источника или приемника данных, который способен производить или получать некоторую информацию. Поток скрывает детали низкоуровневых процессов, происходящих 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 документацию JDK. Все классы, производные от базовых классов InputStream или Reader, имеют методы с именами read() для чтения одиночных байтов или массива байтов. Аналогично, все классы, производные 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, предоставляющих более полные возможности. Таким образом, заключение объекта-потока в один класс — занятие довольно неэффективное, обычно несколько объектов «наслаиваются» друг на друга для получения необходимой функциональности. Необходимость построения потока на основе нескольких объектов — главная причина трудностей в освоении библиотеки ввода/вывода Java.

Классы ввода/вывода удобно разделить по категориям, в зависимости от их функций. В Java 1.0 разработчики решили, что все, связанное с вводом данных, должно быть производным от базового класса InputStream, а все, имеющее отношение к выводу данных, — от класса OutputStream.

Как обычно, я постараюсь привести общий обзор этих классов, но за полными описаниями и списками методов каждого 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 (String);
- файл;

- «канал» (pipe): данные помещаются с одного «конца» и извлекаются с другого;
- последовательность различных потоков, которые можно объединить в одном потоке;
- другие источники (например, подключение к Интернету).

С каждым из перечисленных источников связывается некоторый подкласс базового класса InputStream (табл. 16.1). Существует еще класс FilterInputStream, который также является производным классом InputStream и представляет собой основу для классов-«надстроек», наделяющих входные потоки полезными свойствами и интерфейсами. Его мы обсудим чуть позже.

Таблица 16.1. Разновидности входных потоков InputStream

Класс	Назначение	Аргументы конструктора, порядок применения
ByteArrayInputStream	Позволяет использовать буфер в памяти (массив байтов) в качестве источника данных для входного потока	Буфер, из которого читаются байты. Как источник данных. Присоедините поток к классу FilterInputStream, чтобы получить расширенные возможности
StringBufferInputStream	Превращает строку (String) во входной поток данных InputStream	Строка. Лежащая в основе класса реализация на самом деле использует класс StringBuffer. Как источник данных. Присоедините поток к классу FilterInputStream, чтобы получить расширенные возможности
FileInputStream	Для чтения информации из файла	Строка (String) с именем файла или объекты File и FileDescriptor. Как источник данных. Присоедините поток к классу FilterInputStream, чтобы получить расширенные возможности
PipedInputStream	Производит данные, записываемые в соответствующий выходной поток PipedOutputStream. Реализует понятие канала	Объект PipedOutputStream. Как источник данных в многозадачном окружении. Присоедините поток к классу FilterInputStream, чтобы получить расширенные возможности
SequenceInputStream	Сливает два или более потока InputStream в единый поток	Два объекта-потока InputStream или перечисление Enumeration для контейнера, в котором содержатся все потоки. Как источник данных. Присоедините поток к классу FilterInputStream, чтобы получить расширенные возможности
FilterInputStream	Абстрактный класс, предоставляющий интерфейс для классов-надстроек, которые добавляют к существующим потокам полезные свойства	См. табл. 16.3

Типы OutputStream

В данную категорию (табл. 16.2) попадают классы, определяющие, куда направляются ваши данные: в массив байтов (но не напрямую в String; предполагается, что вы сможете создать их из массива байтов), в файл или в канал.

Вдобавок класс FilterOutputStream предоставляет базовый класс для классов-«надстроек», которые способны наделять существующие потоки новыми полезными атрибутами и интерфейсами. Подробности мы отложим на потом.

Таблица 16.2. Разновидности выходных потоков OutputStream

Класс	Назначение	Аргументы конструктора, порядок применения
ByteArrayOutputStream	Создает буфер в памяти. Все данные, посылаемые в этот поток, размещаются в созданном буфере	Начальный размер буфера (не обязательно). Задает приемник данных. Присоедините поток к объекту FilterOutputStream, чтобы получить расширенные возможности
FileOutputStream	Отправка данных в файл на диске	Строка с именем файла или объекты File или FileDescriptor. Задает приемник данных. Присоедините этот поток к объекту FilterOutputStream, чтобы получить расширенные возможности
PipedOutputStream	Все данные, записываемые в поток, автоматически появляются как входные данные в ассоциированном потоке PipedInputStream. Реализует понятие канала	Объект PipedInputStream. Задает приемник данных в многозадачном окружении. Присоедините этот поток к объекту FilterOutputStream, чтобы получить расширенные возможности
FilterOutputStream	Абстрактный класс, предоставляющий интерфейс для классов-надстроек, которые добавляют к существующим потокам полезные свойства	См. табл. 16.4

Добавление атрибутов и интерфейсов

«Наслаивание» дополнительных объектов для получения новых свойств и функций у определенных объектов называется *надстройкой*, или *декоратором*[1]. В библиотеке ввода/вывода Java постоянно требуется совмещение нескольких свойств, поэтому в ней и используются надстройки. Существование

[1] Все употребляемые далее термины: «настраивание», «наслаивание», «фильтрация» или «декорирование» — по сути, означают одно и то же — перегрузку всех методов InputStream для придания дополнительной функциональности при работе с данными потоков. При этом класс FilterInputStream осуществляет перегрузку без посторонней помощи, а данные соответствующим образом трансформируются. Подклассы FilterInputStream могут, в свою очередь, перегрузить эти же методы и добавить дополнительные методы и поля. — *Примеч. ред.*

в библиотеке ввода/вывода Java классов-фильтров объясняется тем, что абстрактный класс фильтра является базовым классом для всех существующих надстроек. (Надстройка должна включать в себя интерфейс «декорируемого» объекта, но расширение этого интерфейса также не запрещено и практикуется некоторыми классами-«фильтрами».)

Однако у такого подхода есть свои недостатки. Надстройки предоставляют дополнительную гибкость при написании программы (можно легко совмещать различные атрибуты), но при этом код получается излишне сложным. Некоторое неудобство библиотеки ввода/вывода Java объясняется тем, что для получения желаемого объекта приходится создавать много дополнительных объектов — «ядро» ввода/вывода и несколько надстроек.

Интерфейс для надстроек предоставляют классы FilterInputStream (для входных потоков) и FilterOutputStream (для выходных потоков). Это абстрактные классы, производные от основных базовых классов библиотеки ввода/вывода InputStream и OutputStream. Наследование от этих классов — основное требование к классу-надстройке (поскольку таким образом обеспечивается общий интерфейс для «наслаиваемых» объектов).

Чтение из InputStream с использованием FilterInputStream

Классы, производные от FilterInputStream (табл. 16.3), выполняют две различные миссии. DataInputStream позволяет читать из потока различные типы простейших данных и строки. (Все методы этого класса начинаются с префикса read — например, readByte(), readFloat() и т. п.) Этот класс, вместе с «парным» классом DataOutputStream, предоставляет возможность направленной передачи простейших данных посредством потоков. Место доставки определяется классами, описанными в табл. 16.1.

Другие классы изменяют внутренние механизмы входного потока: применение буферизации, подсчет количества прочитанных строк (что позволяет запросить или задать номер строки), возможность возврата в поток только что прочитанных символов. Вероятно, последние два класса создавались в основном для построения компиляторов (то есть их добавили в библиотеку в процессе написания компилятора Java), и прока в повседневном программировании от них немного.

Ввод данных почти всегда осуществляется с буферизацией, 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.

Таблица 16.3. Разновидности надстроек FilterInputStream

Класс	Назначение	Аргументы конструктора, порядок применения
DataInputStream	Используется в сочетании с классом DataOutputStream и позволяет читать примитивы	Входной поток InputStream. Содержит все необходимое для чтения всех простейших типов

Класс	Назначение	Аргументы конструктора, порядок применения
	(целые (int, long), символы (char) и т. д.) из потока, без привязки к внутреннему представлению	
BufferedInputStream	Используется для предотвращения физического обращения к устройству при каждом новом запросе данных	Входной поток InputStream, размер буфера (необязательно). По существу, никакого интерфейса этот класс не предоставляет, он просто присоединяет к потоку буфер. Добавьте дополнительно объект, предоставляющий интерфейс
LineNumberInputStream	Следит за количеством считанных из потока строк; вы можете узнать их число, вызвав метод getLineNumber(), а также перейти к определенной строке с помощью метода setLineNumber(int)	Входной поток InputStream. Этот класс просто считывает строки, поэтому к нему стоит добавить еще одну, более полезную, надстройку
PushbackInputStream	Имеет односимвольный буфер, который позволяет вернуть в поток только что прочитанный символ	Входной поток InputStream. Обычно используется при сканировании файлов для компилятора. Скорее всего, вам он не понадобится

Запись в OutputStream с помощью FilterOutputStream

У класса DataInputStream существует «парный» класс DataOutputStream (табл. 16.4), который позволяет форматировать и записывать в поток примитивы и строки таким образом, что на любой машине и на любой платформе их сможет прочитать и правильно обработать DataInputStream. Все его методы начинаются с префикса write (запись): writeByte(), writeFloat() и т. п.

Класс PrintStream изначально был предназначен для вывода значений в формате, понятном для человека. 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. После вывода значения метод println() 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 возбуждаемые исключения. (Приходится явно вызывать метод checkError(), который возвращает true, если во время исполнения какого-либо из методов класса произошла ошибка.) К тому же этот класс как следует не интернационализован и не выполняет перевод строки платформно-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, именно его следует порекомендовать для вывода в поток каких-либо данных.

Таблица 16.4. Разновидности надстроек FilterOutputStream

Класс	Назначение	Аргументы конструктора, порядок применения
DataOutputStream	Используется в сочетании с классом DataInputStream, позволяет записывать в поток примитивы (целые, символы и т. д.) независимо от платформы	Выходной поток OutputStream. Содержит все необходимое для записи простейших типов данных
PrintStream	При записи форматирует данные. Если класс DataOutputStream отвечает за хранение данных, то этот класс отвечает за отображение данных	Выходной поток OutputStream, а также необязательный логический (boolean) аргумент, показывающий, нужно ли очищать буфер записи при переходе на новую строку. Должен быть последним в «наслоении» объектов для выходного потока OutputStream. Вероятно, вы будете часто его использовать
BufferedOutputStream	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 flush()	Выходной поток OutputStream, а также размер буфера записи (необязательно). Не предоставляет интерфейса как такового, просто указывает, что при выводе необходимо использовать буферизацию. К классу следует присоединить более содержательный класс-«фильтр»

Классы Reader и Writer

При выпуске Java версии 1.1 в библиотеке ввода/вывода были сделаны значительные изменения. Когда в первый раз видишь новые классы, основанные на базовых классах Reader и Writer, возникает мысль, что они пришли на замену классам InputStream и OutputStream. Тем не менее это не так. Хотя некоторые аспекты изначальной библиотеки ввода/вывода, основанной на потоках, объявлены устаревшими (и при их использовании компилятор выдаст соответствующее предупреждение), классы InputStream и OutputStream все еще предоставляют достаточно полезные возможности для проведения байт-ориентированного ввода/ вывода. Одновременно с этим классы Reader и Writer позволяют проводить операции символьно ориентированного ввода/вывода, в кодировке Юникод. Дополнительно:

1. В Java 1.1 в иерархию потоков, основанную на классах InputStream и OutputStream, были добавлены новые классы. Очевидно, эта иерархия не должна была уйти в прошлое.

2. В некоторых ситуациях для решения задачи используются как «байтовые», *так и* «символьные» классы. Для этого в библиотеке появились классы-адаптеры: 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 символы 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 спроектированы 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, во многих ситуациях 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 будет такая тактика: пытаться использовать классы Reader и Writer где только возможно. Обнаружить место, где эти классы неприменимы, будет нетрудно — компилятор выдаст вам сообщение об ошибке.

В табл. 16.5 показано соответствие между источниками и получателями информации двух иерархий библиотеки ввода/вывода Java.

Таблица 16.5. Соответствие между источниками и получателями информации двух иерархий библиотеки ввода/вывода Java

Источники и приемники: классы Java 1.0	Соответствующие классы Java 1.1
InputStream	Reader адаптер: InputStreamReader
OutputStream	Writer адаптер: OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream	StringReader
(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 взяла на вооружение данный подход, но по-другому.

В табл. 16.6 соответствие классов уже не такое точное, как это было в предыдущей таблице. Причина — организация классов: в то время как BufferedOutputStream является подклассом FilterOutputStream, класс BufferedWriter не наследует 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, интерфейсы классов очень похожи.

Таблица 16.6. Соответствие между фильтрами двух иерархий библиотеки ввода/вывода Java

Фильтры: классы Java 1.0	Соответствующие классы Java 1.1
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter (абстрактный класс без подклассов)
BufferedInputStream	BufferedReader (также есть метод для чтения строк readLine())
BufferedOutputStream	BufferedWriter
DataInputStream	Используйте класс DataInputStream (за исключением чтения строк методом readLine() — для их чтения предпочтителен класс BuffredReader)
PrintStream	PrintWriter
LineNumberInputStream (устарел)	LineNumberReader
StreamTokenizer	StreamTokenizer (используйте конструктор с аргументом Reader)
PushBackInputStream	PushBackReader

Один совет очевиден: 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 выбором «номер один» из всего многообразия 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). Интерфейс форматирования PrintWriter практически идентичен интерфейсу PrintStream.

В Java SE5 были добавлены конструкторы PrintWriter, упрощающие создание файлов при выводе (см. далее).

Кроме того, в конструкторе класса PrintWriter можно указать дополнительный флаг, чтобы содержимое буфера каждый раз сбрасывалось при записи новой строки (методом println()).

Классы, оставленные без изменений

Некоторые классы избежали перемен и остались в версии Java 1.1 в том же виде, что и в версии 1.0:

- DataOutputStream;
- File;
- RandomAccessFile;
- SequenceInputStream.

Обращает на себя внимание тот факт, что изменения не коснулись класса DataOutputStream, используемого для пересылки данных независимым от платформы и машины способом, поэтому для передачи данных между компьютерами по-прежнему остаются актуальными иерархии InputStream и OutputStream.

RandomAccessFile: сам по себе

Класс RandomAccessFile предназначен для работы с файлами, содержащими записи известного размера, между которыми можно перемещаться методом seek(), а также выполнять операции чтения и модификации. Записи не обязаны иметь фиксированную длину; вы просто должны уметь определить их размер и то, где они располагаются в файле.

Поначалу с трудом верится, что класс RandomAccessFile не является полноценным представителем иерархии потоков ввода/вывода на основе классов 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 (также реализуемые классами DataInputStream и DataOutputStream). Он не использует функциональность существующих классов из иерархии InputStream и OutputStream — это полностью независимый класс, написанный «с чистого листа», со своими собственными методами. Причина кроется, скорее всего, в том, что класс RandomAccessFile позволяет свободно перемещаться по файлу как в прямом, так и в обратном направлении, что для других типов ввода/вывода невозможно. Так или иначе, он стоит особняком и напрямую наследует от корневого класса Object.

По сути, класс RandomAccessFile похож на пару совмещенных в одном классе потоков DataInputStream и DataOutputStream, к которым на всем «протяжении» применимы: метод getFilePointer(), показывающий, где вы «находитесь» в данный момент; метод seek(), позволяющий перемещаться на заданную позицию файла; и метод length(), определяющий максимальный размер файла. Вдобавок, конструктор этого класса требует второй аргумент (схоже с методом fopen() в C), устанавливающий режим использования файла: только для чтения (строка «r») или 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 вам

пометить некоторую позицию потока методом mark(), а затем вернуться к ней методом 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. Следующие примеры можно рассматривать как простое руководство по созданию типичных сочетаний классов для организации ввода/вывода и координации их взаимодействия.

В этих примерах используется упрощенная обработка исключений с передачей их на консоль, но такой способ подойдет только для небольших программ и утилит. В реальном коде следует использовать более совершенные средства обработки ошибок.

Буферизованное чтение из файла

Чтобы открыть файл для посимвольного чтения, используется класс FileInputReader; имя файла задается в виде строки (String) или объекта File. Ускорить процесс чтения помогает буферизация ввода, для этого полученная ссылка передается в конструктор класса BufferedReader. Так как в интерфейсе класса имеется метод readLine(), все необходимое для чтения имеется в вашем распоряжении. При достижении конца файла метод readLine() возвращает ссылку null.

```
//: io/BufferedInputFile.java
import java.io.*;

public class BufferedInputFile {
     // Исключения направляются на консоль·
     public static String
     read(String filename) throws IOException {
              // Чтение входных данных по строкам
              BufferedReader in = new BufferedReader(
                       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"));
```

```
    }
} /// ~
```

Объект StringBuilder sb служит для объединения всего прочитанного текста (включая переводы строк, поскольку метод readLine() их отбрасывает). В завершение файл закрывается методом close().

Чтение из памяти

В этой секции результат String файла BufferedInputFile.read() используется для создания StringReader. Затем символы последовательно 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),
     }
} /// ~
```

Обратите внимание: метод read() возвращает следующий символ в формате int, и для правильного вывода его необходимо предварительно преобразовать в char.

Форматированное чтение из памяти

Для чтения «форматированных» данных применяется класс DataInputStream, ориентированный на ввод/вывод байтов, а не символов. В данном случае необходимо использовать классы иерархии InputStream, а не их аналоги на основе класса Reader. Конечно, можно прочитать все, что угодно (например, файл), 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");
```

продолжение ⇩

```
            }
        }
} /// ~
```

Для преобразования строки в массив байтов, пригодный для помещения в поток ByteArrayInputStream, в классе String предусмотрен метод getBytes(). Полученный ByteArrayInputStream представляет собой поток InputStream, подходящий для передачи DataInputStream.

При побайтовом чтении символов из форматированного потока DataInputStream методом readByte() любое полученное значение будет считаться действительным, поэтому возвращаемое значение неприменимо для идентификации конца потока. Вместо этого можно использовать метод available(), который сообщает, сколько еще осталось символов. В следующем примере показано, как читать файл побайтно:

```
//: io/TestEOF.java
// Проверка достижения конца файла одновременно
// с чтением из него по байту.
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());
     }
} ///:~
```

Заметьте, что метод available() работает по-разному в зависимости от источника данных; дословно его функция описывается следующим образом: «количество байтов, которые можно прочитать *без блокировки*». При чтении из файла это означает весь файл, но для другого рода потоков это не обязательно верно, поэтому используйте этот метод разумно.

Определить конец входного потока можно и с помощью перехвата исключения. Впрочем, применение исключений в таких целях считается злоупотреблением.

Вывод в файл

Объект FileWriter записывает данные в файл. При вводе/выводе практически всегда применяется буферизация (попробуйте прочитать файл без нее, и вы увидите, насколько ее отсутствие влияет на производительность — скорость чтения уменьшится в несколько раз), поэтому мы присоединяем надстройку BufferedWriter. После этого подключается PrintWriter, чтобы выполнять форматированный вывод. Файл данных, 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 содержимого файла
              System out println(BufferedInputFile read(file)),
  }
} /// ~
```

При записи строк в файл к ним добавляются их номера. Заметьте, что надстройка **LineNumberInputStream** для этого *не применяется*, поскольку этот класс тривиален, да и вообще не нужен. Как и показано в рассматриваемом примере, своя собственная нумерация ничуть не сложнее.

Когда данные входного потока исчерпываются, метод **readLine()** возвращает **null**. Для потока **out1** явно вызывается метод **close()**; если не вызвать его для всех выходных файловых потоков, в буферах могут остаться данные, и файл получится неполным.

Сокращенная форма вывода текстового файла

В Java SE5 у **PrintWriter** появился вспомогательный конструктор. Благодаря ему вам не придется вручную выполнять всю работу каждый раз, когда вам потребуется создать текстовый файл и записать в него данные. Вот как выглядит пример **BasicFileOutput.java** в обновленном виде:

```
//· 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(
                                BufferedInputFile 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 содержимого файла·
              System out.println(BufferedInputFile read(file));
```

продолжение ⇩

```
  }
} ///:~
```

Буферизация по-прежнему обеспечена, но вам не приходится включать ее самостоятельно. К сожалению, для других распространенных операций сокращенной записи не предусмотрено, поэтому типичный код ввода/вывода по-прежнему содержит немало избыточного текста.

Сохранение и восстановление данных

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 посылку байтов, поэтому для них требуются 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() нормально читает
              // строки 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 данные записываются в выходной поток DataOutputStream, язык Java гарантирует, что эти 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. Если Java поддерживается на обеих платформах, проблема исчезает сама собой.

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 — это разновидность кодировки Юникод, в которой каждый символ хранится в двух байтах. Если вы работаете только с кодировкой ASCII, «удвоение» данных в Юникоде приводит к неоправданным затратам дискового 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 Java[1] (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, позволяющий верно ее считать.

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 просто воспроизвести 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; при наличии такой информации прочитать число double как какую-то последовательность байтов или символов не представляет сложности. Поэтому данные в файле должны иметь определенный формат, или вам придется использовать дополнительную информацию, показывающую, какие именно данные находятся в определенных местах. Заметьте, что сериализация объектов (описанная в этой главе чуть позже) часто предоставляет простейший способ записи и восстановления сложных структур данных.

Чтение/запись файлов с произвольным доступом

Как уже было замечено, работа с классом RandomAccessFile напоминает использование совмещенных в одном классе потоков DataInputStream и DataOutputStream (они реализуют те же интерфейсы DataInput и DataOutput). Кроме того, метод seek() позволяет переместиться к определенной позиции и изменить хранящееся там значение.

При использовании RandomAccessFile необходимо знать структуру файла, чтобы правильно работать с ним. Класс RandomAccessFile содержит методы для чтения и записи примитивов и строк UTF-8. Пример:

---

[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.*

```
//: 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(
                                   "Значение " + 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(47.0001);
               rf.close();
               display();
     }
} /* Output:
Значение 0: 0.0
Значение 1: 1.414
Значение 2: 2.828
Значение 3: 4.242
Значение 4: 5.656
Значение 5: 7.069999999999999
Значение 6: 8.484
The end of the file
Значение 0: 0.0
Значение 1: 1.414
Значение 2: 2.828
Значение 3: 4.242
Значение 4: 5.656
Значение 5: 47.0001
Значение 6: 8.484
The end of the file
*///:~
```

Метод **display()** открывает файл и выводит семь значений в формате **double**. Метод **main()** создает файл, открывает и модифицирует его. Поскольку значение **double** всегда занимает 8 байт, для перехода к пятому числу методу **seek()** следует передать смещение 5*8.

Как упоминалось ранее, класс **RandomAccessFile** отделен от остальных классов иерархии ввода/вывода, если не считать того факта, что он реализует интерфейсы **DataInput** и **DataOutput**. Приходится предполагать, что для этого **RandomAccessFile** правильно организована буферизация, потому что включить ее в программе не удастся.

Некоторая свобода выбора предоставляется только со вторым аргументом конструктора: RandomAccessFile может открываться в режиме чтения ("r") или чтения/записи ("rw").

Также стоит рассмотреть возможность употребления вместо класса RandomAccessFile механизма отображаемых в память файлов.

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 *потоками программы*. Они будут описаны позднее.

Средства чтения и записи файлов

Очень часто в программировании производится такая цепочка действий: файл считывается в память, там он изменяется, а потом снова записывается на диск. Одна из проблем при работе с библиотекой ввода/вывода Java состоит в том, что для выполнения таких достаточно типичных операций вам придется написать некоторое количество кода — не существует вспомогательных функций, на которые можно переложить такую деятельность. Что еще хуже, с надстройками вообще трудно запомнить, как открываются файлы. Поэтому имеет смысл добавить в вашу библиотеку вспомогательные классы, которые легко сделают нужное за вас. В Java SE5 у PrintWriter появился вспомогательный конструктор, позволяющий легко открыть текстовый файл для чтения. Тем не менее существует много других типичных задач, часто выполняемых в повседневной работе, и было бы разумно избавиться от лишнего кода, связанного с их выполнением.

Ниже показан такой класс TextFile с набором статических методов, построчно считывающих и записывающих текстовые файлы. Вдобавок можно создать экземпляр класса TextFile, который будет хранить содержимое файла в списке ArrayList (и функциональность списка ArrayList станет доступной при работе с содержимым файла):

```
//: net/mindview/util/TextFile.java
// Статические функции для построчного считывания и записи
// текстовых файлов, а также манипуляции файлом как списком ArrayList
package net.mindview.util;
import java.io.*;
import java.util.*;

public class TextFile extends ArrayList<String> {
    // Чтение файла как одной строки:
    public static String read(String fileName) {
        StringBuilder sb = new StringBuilder();
        try {
            BufferedReader in= new BufferedReader(new FileReader(
```

продолжение ⇩

```
                        new File(fileName).getAbsoluteFile()));
                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 за один вызов метода:
public static void write(String fileName, String text) {
        try {
                PrintWriter out = new PrintWriter(
                        new File(fileName).getAbsoluteFile());
                try {
                        out.print(text);
                } finally {
                        out.close();
                }
        } catch(IOException e) {
                throw new RuntimeException(e);
        }
}
// Чтение файла с разбиением по регулярному выражению:
public TextFile(String fileName, String splitter) {
        super(Arrays.asList(read(fileName).split(splitter)));
        // Вызов split() часто оставляет пустой объект
        // String в начальной позиции:
        if(get(0).equals("")) remove(0);
}
// Обычное построчное чтение:
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);
        }
}
// Простая проверка :
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");
            // Разбиение на уникальный отсортированный список слов:
            TreeSet<String> words = new TreeSet<String>(
                    new TextFile("TextFile.java", "\\W+"));
            // Вывод слов, начинающихся с прописной буквы·
            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 каждую строку к StringBuilder, а за ней присоединяется перевод строки, удаленный при чтении. Затем возвращается объект String, содержащий весь файл. Метод write() открывает файл и записывает в него текст.

Обратите внимание: к каждой операции открытия файла добавляется парный вызов close() в секции finally. Тем самым обеспечивается гарантированное закрытие файла после завершения работы.

Конструктор использует метод read() для превращения файла в String, после чего он вызывает метод String.split(), чтобы разбить результат на строки. В качестве разделителя используются символы новой строки (если вы будете часто использовать этот класс, то, возможно, захотите переписать этот конструктор, чтобы он работал эффективнее). К сожалению, аналогичного метода для соединения строк нет, так что для записи строк придется обойтись нестатическим методом write().

Так как класс должен упростить процесс чтения и записи файлов, все исключения IOException преобразуются в RuntimeException, чтобы пользователю не пришлось создавать блоки try/catch. Возможно, вы предпочтете создать другую версию, которая возвращает IOException вызывающей стороне.

В методе main() выполняется небольшой 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.

Стандартный ввод/вывод

Термин *«стандартный ввод/вывод»* возник еще в эпоху UNIX (и в некоторой форме имеется и в Windows, и во многих других операционных системах). Он означает единственный поток информации, используемый программой. Вся информация программы приходит из *стандартного ввода* (standard input), все данные записываются в *стандартный вывод* (standard output), а все ошибки программы передаются 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 вводом другой программы. Это мощный инструмент.

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 дополнительных надстроек. Это значит, что потоки System. out и System.err можно использовать напрямую, в то время как стандартный ввод System.in желательно надстраивать.

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 считывающее устройство Reader посредством класса-преобразователя InputStreamReader. Следующий пример просто отображает на экране последнюю строку, введенную пользователем (эхо-вывод):

```
//· 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 нажатие Ctrl+Z завершает программу
     }
} ///·~
```

Присутствие спецификации исключений объясняется тем, что метод readLine() может возбуждать исключение IOException. Снова обратите внимание, что поток System.in обычно буферизуется, впрочем, как и большинство потоков.

Замена System.in на PrintWriter

Стандартный вывод System.out является объектом PrintStream, который, в свою очередь, наследует от базового класса OutputStream. В классе PrintWriter имеется конструктор, который принимает в качестве аргумента выходной поток OutputStream. Таким образом, вы можете 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 true, чтобы обеспечить автоматический сброс буфера на печать, иначе можно вовсе не увидеть никакого вывода.

Перенаправление стандартного ввода/вывода

Класс System позволяет вам перенаправить стандартный ввод, вывод и поток ошибок. Для этого предусмотрены 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 сообщениями. Перенаправление ввода удобно для программ, работающих с командной строкой, в которых необходимо поддержать некоторую последовательность введенных пользователем данных. Вот простой пример, показывающий, как использовать эти методы:

```
//: io/Redirecting.java
// Перенаправление стандартного ввода/вывода.
import java.io.*;

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 забывайте!
            System.setOut(console);
    }
} ///.~
```

Программа присоединяет стандартный ввод к файлу и перенаправляет стандартный ввод и поток для ошибок в другие файлы. Обратите внимание

на сохранение ссылки на исходный объект System.out в начале программы и его восстановление в конце.

Перенаправление основано на байтовом, а не на символьном вводе/выводе, поэтому в примере используются InputStream и OutputStream, а не их символьно-ориентированные эквиваленты Reader и Writer.

Новый ввод/вывод (nio)

При создании библиотеки «нового ввода/вывода» Java, появившейся в JDK 1.4 в пакетах java.nio.*, ставилась единственная цель: скорость. Более того, «старые» пакеты ввода/вывода были переписаны с учетом достижений nio, с намерением использовать преимущества повышенного быстродействия, поэтому улучшения вы получите, даже если не будете писать явный nio-код. Подъем производительности просматривается как в файловом вводе/выводе, который мы здесь рассматриваем, так и в сетевом вводе/выводе.

Увеличения скорости удалось достичь с помощью структур, близких к средствам самой операционной системы: *каналов*[1] (channels) и *буферов* (buffers). Канал можно сравнить с угольной шахтой, вырытой на угольном пласте (данные), а буфер — с вагонеткой, 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» его в канал. Канал либо извлекает данные из буфера, либо помещает их в него.

Напрямую взаимодействует с каналом только буфер ByteBuffer, то есть буфер, хранящий простые байты. Если вы просмотрите документацию JDK для класса java.nio.ByteBuffer, то увидите что он достаточно прост: вы создаете его, указывая, сколько места надо выделить под данные. Класс содержит набор 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. Буфер работает на достаточно низком уровне, поскольку обеспечивается более эффективная совместимость с большинством операционных систем.

Три класса из «старой» библиотеки ввода/вывода были изменены так, чтобы они позволяли получить канал FileChannel: это FileInputStream, FileOutputStream и 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 получить объекты Reader и Writer для каналов.

Простой пример использования всех трех типов потоков. Создаваемые каналы поддерживают запись, чтение/запись и только чтение:

---

[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.*

```
//: 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()); // Переходим 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()** выдает канал **FileChannel**. Канал довольно прост: ему передается байтовый буфер **ByteBuffer** для чтения и записи, и вы можете заблокировать некоторые участки файла для монопольного доступа (этот процесс будет описан чуть позже).

Для помещения байтов в буфер **ByteBuffer** используется один из нескольких методов для записи данных (**put**); данные записываются в виде одного или нескольких байтов или значений примитивов. Впрочем, как было показано в примере, можно «заворачивать» уже существующий байтовый массив в буфер **ByteBuffer**, используя метод **wrap()**. Когда вы так делаете, байтовый массив не копируется, а используется как хранилище для полученного буфера **ByteBuffer**. В таких случаях говорят, что буфер **ByteBuffer** создается на базе массива.

Файл **data.txt** заново открывается с помощью класса **RandomAccessFile**. Заметьте, что канал **FileChannel** может перемещаться внутри файла; в нашем примере он сдвигается в конец файла так, чтобы дополнительные записи присоединялись за существующим содержимым.

Чтобы доступ к файлу ограничивался только чтением, следует явно получить байтовый буфер **ByteBuffer** статическим методом **allocate()**. Предназначение **nio** — быстрое перемещение большого количества данных, поэтому размер буфера имеет значение: на самом деле установленный в примере размер в 1 килобайт меньше, чем обычно требуется (поэкспериментируйте с работающим приложением, чтобы найти оптимальное решение).

Можно получить еще большее быстродействие, используя вместо метода allocate() метод allocateDirect(). Он производит буфер «прямого доступа», еще теснее привязанный к низкоуровневой работе операционной системы. Однако такой буфер требует больше ресурсов, а реализация его различается в различных операционных системах. Опять же, поэкспериментируйте со своим приложением и выясните, дадут ли буферы прямого доступа лучшую производительность.

После вызова метода read() буфера FileChannel для сохранения байтов в буфере ByteBuffer также необходимо вызвать для буфера метод flip(), позволяющий впоследствии извлечь из буфера его данные (да, все это выглядит немного неудобно, но помните, что расчет делался на высокое быстродействие, поэтому все делается на низком уровне). И если затем нам снова понадобится буфер для чтения, придется вызывать перед каждым методом read() метод clear(). В этом нетрудно убедиться на примере простой программы копирования файлов:

```
//· io/ChannelCopy.java
// Копирование файла с использованием каналов и буферов
// {Параметры: 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("параметры: Источник Приемник");
                       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
              }
  }
} /// ~
```

В программе 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, пережиток 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() перемещает все его 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
// Использование метода transferTo() для соединения каналов
// {Параметры  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("параметры  источник приемник"),
                       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 от типа **byte** к типу **char**. Такой подход явно примитивен — если вы посмотрите на класс **java.nio.CharBuffer**, то увидите, что в нем есть метод **toString()**, который возвращает строку из символов, находящихся в данном буфере. Байтовый буфер **ByteBuffer** можно рассматривать как символьный буфер **CharBuffer**, как это делается в методе **asCharBuffer()**, почему бы так и не поступить? Как вы увидите уже из первого предложения **expect()**, это не сработает:

```
//: io/BufferToText java
// Получение текста из буфера ByteBuffers 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());
```

продолжение ⇨

```
                // Декодирование с использованием кодировки по умолчанию:
                buff.rewind();
                String encoding = System.getProperty("file.encoding");
                System.out.println("Декодировано в " + encoding + ": "
                        + Charset.forName(encoding).decode(buff));
                // Кодирование в печатной форме:
                fc = new FileOutputStream("data2.txt").getChannel();
                fc.write(ByteBuffer.wrap(
                        "Some text".getBytes("UTF-16BE")));
                fc.close();
                // Повторная попытка чтения:
                fc = new FileInputStream("data2.txt").getChannel();
                buff.clear();
                fc.read(buff);
                buff.flip();
                System.out.println(buff.asCharBuffer());
                // Использование CharBuffer для записи:
                fc = new FileOutputStream("data2.txt").getChannel();
                buff = ByteBuffer.allocate(24); // Больше, чем необходимо
                buff.asCharBuffer().put("Some text");
                fc.write(buff);
                fc.close();
                // Чтение и вывод:
                fc = new FileInputStream("data2.txt").getChannel();
                buff.clear();
                fc.read(buff);
                buff.flip();
                System.out.println(buff.asCharBuffer());
        }
} /* Output:
????
Декодировано в 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, 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
// Перечисление кодировок и их символических имен
import java.nio.charset.*;
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
...
*///:~
```

Вернемся 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. Чтобы узнать кодировку по умолчанию вызовите метод System.getProperty("file.encoding"), который возвращает строку с названием кодировки. Передавая эту строку методу Charset.forName(), вы получите объект Charset, с помощью которого и декодируете строку.

Другой подход — кодировать данные методом encode() так, чтобы при чтении файла выводились данные, пригодные для вывода на печать (пример представлен в программе BufferToText.java). Здесь для записи текста в файл используется кодировка UTF-16BE, и при последующем чтении вам остается лишь преобразовать данные в буфер CharBuffer и вывести его содержимое.

Наконец, мы видим, что происходит, когда вы записываете 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, что и показывают результаты.

Извлечение примитивов

Несмотря на то что в буфере ByteBuffer хранятся только байты, он поддерживает методы для выборки любых значений примитивных типов из этих байтов. Следующий пример демонстрирует вставку и выборку из буфера разнообразных значений примитивных типов:

```
//. io/GetData java
// Получение различных данных из буфера 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 заполняется нулями
              int i = 0;
              while(i++ < bb limit())
                       if(bb.get() != 0)
                                print("nonzero");
              print("i = " + i);
              bb rewind();
              // Сохраняем и считываем символьный массив
              bb asCharBuffer() put("Howdy!");
              char c,
              while((c = bb.getChar()) != 0)
                       printnb(c + " ");
              print(),
              bb rewind();
              // Сохраняем и считываем число типа short:
              bb.asShortBuffer().put((short)471142);
              print(bb.getShort()),
              bb.rewind();
              // Сохраняем и считываем число типа int:
              bb.asIntBuffer() put(99471142);
              print(bb getInt()),
              bb.rewind(),
              // Сохраняем и считываем число типа long·
              bb.asLongBuffer() put(99471142);
              print(bb.getLong());
              bb.rewind(),
              // Сохраняем и считываем число типа float.
              bb.asFloatBuffer() put(99471142);
              print(bb getFloat());
              bb.rewind(),
              // Сохраняем и считываем 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 убеждаемся в том, что его содержимое действительно заполнено нулями. Проверяются 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 основан на получении подходящего «представления» этого буфера методами asCharBuffer(), asShortBuffer() и т. п., и последующем занесении в это представление значения методом put(). В примере мы так поступаем для каждого из простейших типов. Единственным исключением из этого ряда является 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 взглянуть на соответствующий байтовый буфер «через призму» некоторого 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 было показано в предыдущем примере, 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();
               // Сохранение массива int:
               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);
               // Назначение нового предела перед смещением буфера
               ib.flip();
               while(ib.hasRemaining()) {
                        int i = ib.get();
                        System.out.println(i);
               }
     }
} /* Output:
99
11
42
47
1811
```

продолжение ⇩

```
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 простейшим 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. Настолько же просто считать данные из канала и использовать представление для преобразования данных к конкретному простейшему типу. Вот пример, который трактует 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, 'a' });
               bb.rewind();
               printnb("Буфер Byte ");
               while(bb.hasRemaining())
                         printnb(bb.position()+ " -> " + bb.get() + ", ");
               print();
               CharBuffer cb =
                         ((ByteBuffer)bb.rewind()).asCharBuffer();
               printnb("Буфер Char ");
               while(cb.hasRemaining())
                         printnb(cb.position() + " -> " + cb.get() + ", ");
               print();
               FloatBuffer fb =
                         ((ByteBuffer)bb.rewind()).asFloatBuffer();
               printnb("Буфер Float ");
               while(fb.hasRemaining())
                         printnb(fb.position()+ " -> " + fb.get() + ", ");
               print();
               IntBuffer ib =
                         ((ByteBuffer)bb.rewind()).asIntBuffer();
               printnb("Буфер Int ");
               while(ib.hasRemaining())
                         printnb(ib.position()+ " -> " + ib.get() + ", ");
               print();
               LongBuffer lb =
                         ((ByteBuffer)bb.rewind()).asLongBuffer();
               printnb("Буфер Long ");
               while(lb.hasRemaining())
                         printnb(lb.position()+ " -> " + lb.get() + ", ");
               print();
```

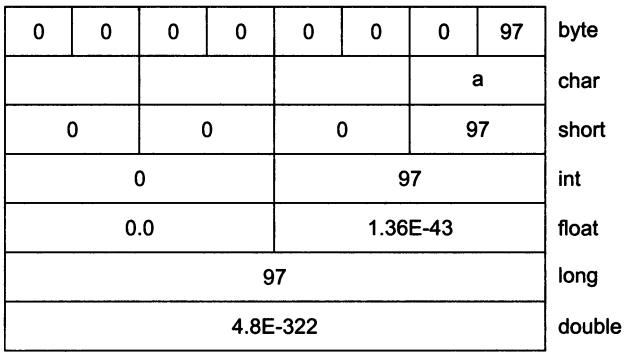
```
                ShortBuffer sb =
                        ((ByteBuffer)bb.rewind()).asShortBuffer();
                printnb("Буфер Short ");
                while(sb.hasRemaining())
                        printnb(sb.position()+ " -> " + sb.get() + ", ");
                print();
                DoubleBuffer db =
                        ((ByteBuffer)bb.rewind()) asDoubleBuffer();
                printnb("Буфер Double ");
                while(db.hasRemaining())
                        printnb(db.position()+ " -> " + db.get() + ", ");
        }
} /* Output
Буфер Byte 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 0, 4 -> 0, 5 -> 0, 6 -> 0, 7 -> 97,
Буфер Char 0 ->  , 1 ->  , 2 ->  , 3 -> a,
Буфер Float 0 -> 0 0, 1 -> 1.36E-43,
Буфер Int 0 -> 0, 1 -> 97,
Буфер Long 0 -> 97,
Буфер Short 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 97,
Буфер Double 0 -> 4 8E-322,
*///.~
```

Байтовый буфер **ByteBuffer** создается как «обертка» для массива из восьми байтов, который затем и просматривается с помощью представлений для различных простейших типов.

О порядке байтов

Различные компьютеры могут хранить данные 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 прочитать эти данные как тип short (ByteBuffer.asShortBuffer()), то получите число 97 (00000000 01100001), но при другом порядке следования байтов будет получено число 24 832 (01100001 00000000).

Следующий пример показывает, как порядок следования байтов отражается на символах в зависимости от настроек буфера:

```
//: io/Endians.java
// Endian differences and data storage.
import java.nio.*;
import java.util *;
import static net.mindview util.Print.*;

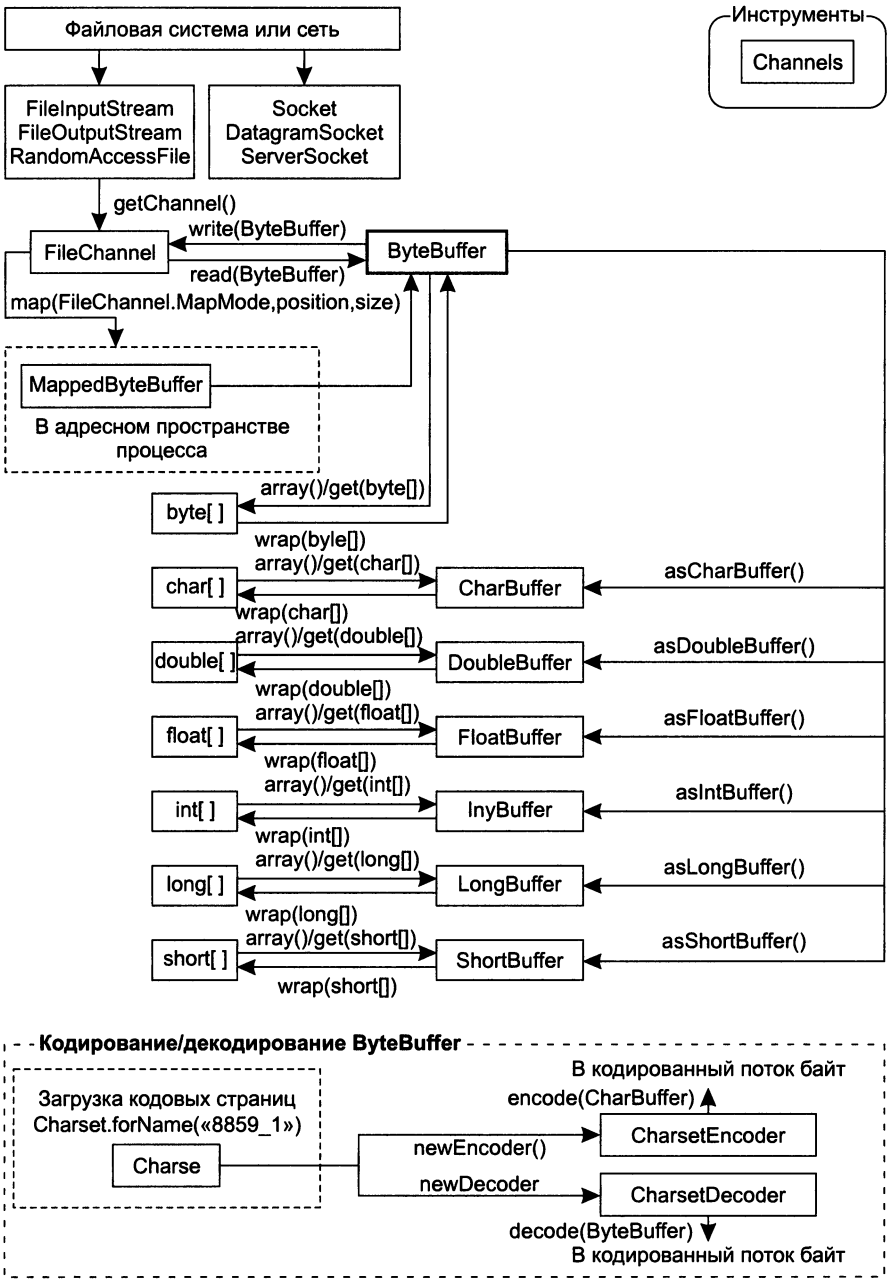
public class Endians {
     public static void main(String[] args) {
               ByteBuffer bb = ByteBuffer.wrap(new byte[12]);
               bb asCharBuffer().put("abcdef");
               print(Arrays toString(bb array()));
               bb rewind();
               bb.order(ByteOrder BIG_ENDIAN);
               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 подходит метод array(). Метод array() является необязательным, и вызывать его следует только для буфера, созданного на базе существующего массива; в противном случае произойдет исключение UnsupportedOperationException.

Символьный массив помещается в буфер ByteBuffer посредством представления CharBuffer. При выводе содержащихся в буфере байтов мы видим, что настройка по умолчанию совпадает с режимом big_endian, в то время как атрибут little_endian переставляет байты в обратном порядке.

Буферы и операции с данными

Следующая диаграмма демонстрирует отношения между классами пакета nio; она поможет вам разобраться, как можно перемещать и преобразовывать данные. 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 запишите данные буфера ByteBuffer в полученный канал FileChannel.



Отметьте, что перемещать данные каналов («из» и «в») допустимо только с помощью байтовых буферов ByteBuffer, а для остальных простейших типов можно либо создать отдельный буфер этого типа, либо получить такой буфер из байтового буфера посредством метода с префиксом as. Таким образом, буфер

с примитивными данными *нельзя* преобразовать к байтовому буферу. Впрочем, вы можете помещать примитивы в байтовый буфер и извлекать их оттуда с помощью представлений, это не такое уж строгое ограничение.

Подробно о буфере

Буфер (Buffer) состоит из данных и четырех индексов, используемых для доступа к данным и эффективного манипулирования ими. К этим индексам относятся *метка* (mark), *позиция* (position), *предельное значение* (limit) и *вместимость* (capacity). Есть методы, предназначенные для установки и сброса значений этих индексов, также можно узнать их значение (табл. 16.7).

Таблица 16.7. Методы буфера

Метод	Описание
capacity()	Возвращает значение емкости буфера
clear()	Очищает буфер, устанавливает позицию в нуль, а предельное значение делает равным вместимости. 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 предельное значение
limit(int lim)	Устанавливает предельное значение
mark()	Приравнивает метке значение позиции
position()	Возвращает значение позиции
position(int pos)	Устанавливает значение позиции
remaining()	Возвращает разницу между предельным значением и позицией
hasRemaining()	Возвращает true, если между позицией и предельным значением еще остались элементы

Методы, вставляющие данные в буфер и считывающие их оттуда, обновляют эти индексы в соответствии с внесенными изменениями.

Следующий пример использует очень простой алгоритм (перестановка смежных символов) для смешивания и восстановления символов в буфере CharBuffer:

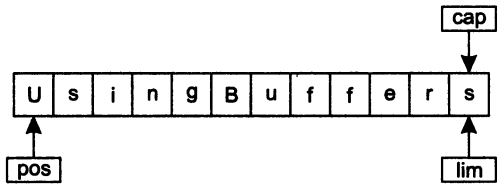
```
//: io/UsingBuffers.java
import java.nio.*;
import static net.mindview.util.Print.*;

public class UsingBuffers {
     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
*///:~
```

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.

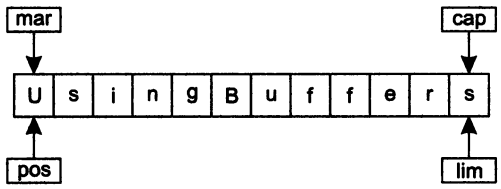
На входе в метод symmetricScramble() буфер выглядит следующим образом:



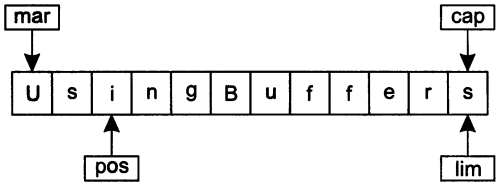
Позиция (pos) указывает на первый элемент буфера, *вместительность* (cap) и *предельное значение* (lim) — на последний.

В методе symmetricScramble() цикл while выполняется до тех пор, пока *позиция* не станет равной *предельному значению*. *Позиция* буфера изменяется при вызове для него «относительных» методов put() или get(). Можно также использовать «абсолютные» версии методов put() и get(), которым передается аргумент-индекс, указывающий, с какого места начнет работу метод put() или метод get(). Эти методы не изменяют значение *позиции* буфера.

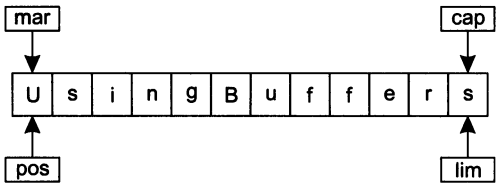
Когда управление переходит в цикл while, вызывается метод mark() для установки значения *метки* (mar). Состояние буфера в этот момент таково:



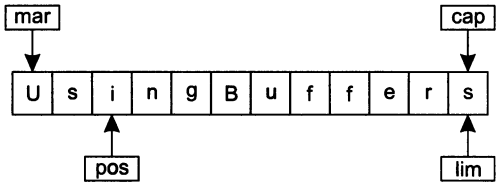
Два вызова «относительных» методов get() сохраняют значение первых двух символов в переменных c1 и c2. После этих вызовов буфер выглядит так:



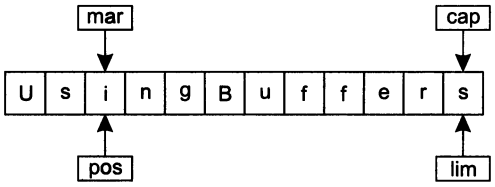
Для смешивания символов нам нужно записать символ c2 в позицию 0, а c1 в позицию 1. Для этого можно обратиться за «абсолютной» версией метода put(), но мы приравняем позицию метке, что и делает метод reset():



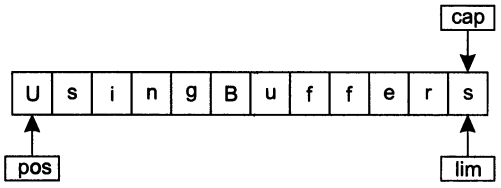
Два вызова метода put() записывают c2, а затем c1:



На следующей итерации значение *метки* приравнивается *позиции*:



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 *предельным значением*. Поэтому, если вы хотите распечатать 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):



При следующем вызове symmetricScramble() процесс повторяется, и буфер CharBuffer возвращается к своему изначальному состоянию.

Отображаемые в память файлы

Механизм отображения файлов в память позволяет 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 изменения файла. Небольшой пример:

```
//. io/LargeMappedFiles.java
// Создание очень большого файла, отображаемого 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 = 0x8FFFFFF, // 128 MB
     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("Запись завершена");
              for(int i = length/2; i < length/2 + 6; i++)
                       printnb((char)out.get(i));
     }
} ///:~
```

Чтобы одновременно выполнять чтение и запись, мы начинаем с создания объекта RandomAccessFile, получаем для этого файла канал, а затем вызываем метод map(), чтобы получить буфер MappedByteBuffer, который представляет 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 он содержит все методы последнего. Здесь представлены только простейшие вызовы методов put() и get(), но вы также можете использовать такие возможности, как метод asCharBuffer() и т. п.

Программа напрямую создает файл размером 128 Мбайт; скорее всего, это превышает ограничения вашей операционной системы на размер блока данных, находящегося в памяти. Однако создается впечатление, 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 (размером до 2 Гбайт) файлами. Заметьте, что для достижения максимальной производительности используются низкоуровневые механизмы отображения файлов используемой операционной системы.

Производительность

Хотя быстродействие «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;
                                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:
Stream Write: 0.56
Mapped Write: 0.12
Stream Read: 0.80
Mapped Read: 0.07
```

продолжение ⇨

```
Stream Read/Write  5.32
Mapped Read/Write  0.02
*/// ~
```

Как уже было видно из предыдущих примеров книги, runTest() — не что иное как *метод шаблона*, предоставляющий тестовую инфраструктуру для различных реализаций метода test(), определенного в безымянных внутренних подклассах. Каждый из этих подклассов 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 так же, как выполняется чтение/запись в рассмотренном примере.

Отметьте, что в методах test() также учитывается инициализация различных объектов для работы с вводом/выводом, и, несмотря на то что настройка отображаемых файлов может быть затратной, общее преимущество по сравнению с потоковым вводом/выводом все равно получается весьма значительным.

Блокировка файлов

Блокировка файлов позволяет синхронизировать доступ к файлу как к совместно используемому ресурсу. Впрочем, потоки, претендующие на один и тот же файл, могут принадлежать различным виртуальным машинам JVM, или один поток может быть Java-потоком, 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.

Вот 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("Файл заблокирован");
                         TimeUnit.MILLISECONDS.sleep(100);
                         fl.release();
                         System out.println("Блокировка снята"),
               }
               fos.close();
     }
} /* Output:
Файл заблокирован
Блокировка снята
*///:~
```

Блокировать файл целиком позволяет объект FileLock, который вы получаете, вызывая метод tryLock() или lock() класса FileChannel. (Сетевые каналы SocketChannel, DatagramChannel и ServerSocketChannel не нуждаются в блокировании, так как они доступны в пределах одного процесса. Вряд ли сокет будет использоваться двумя процессами совместно.) Метод tryLock() не приостанавливает программу. Он пытается овладеть объектом блокировки, но если ему это не удается (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 можно заблокировать часть файла вызовом

```
tryLock(long position, long size, boolean shared)
```

или

```
lock(long position, long size, boolean shared)
```

Блокируется участок файла размером size от позиции position. Третий аргумент указывает, будет ли блокировка совместной.

Методы без аргументов приспосабливаются 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 эксклюзивным или разделяемым доступом должна быть встроена в операционную систему. Если операционная система не поддерживает разделяемые блокировки и был сделан запрос на получение такой блокировки, используется эксклюзивный доступ. Тип блокировки (разделяемая или эксклюзивная) можно узнать при помощи метода FileLock.isShared().

Блокирование части отображаемого файла

Как уже было упомянуто, отображение файлов обычно используется для файлов очень больших размеров. Иногда при работе с таким большим файлом требуется заблокировать некоторые его части, в то время как доступные части будут изменяться другими процессами. В частности, такой подход характерен для баз данных, чтобы несколько пользователей могли работать с базой одновременно.

В следующем примере каждый их 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.*;
```

продолжение ⇨

```
public class LockingMappedFiles {
     static final int LENGTH = 0x8FFFFFF; // 128 MB
     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();
               }
               public void run() {
                        try {
                                 // Монопольная блокировка без перекрытия:
                                 FileLock fl = fc.lock(start, end, false);
                                 System.out.println("Заблокировано: "+ start +" to "+ end);
                                 // Модификация:
                                 while(buff.position() < buff.limit() - 1)
                                 buff.put((byte)(buff.get() + 1));
                                 fl.release();
                                 System.out.println("Освобождено: "+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()** для файлового канала устанавливается блокировка (вы не вправе запросить блокировку для буфера, это позволено только для канала). Вызов **lock()** напоминает механизм синхронизации доступа потоков к объектам, у вас появляется некая «критическая секция» с монопольным доступом к данной части файла.

Блокировки автоматически снимаются при завершении работы JVM, закрытии канала, для которого они были получены, но можно также явно вызвать метод **release()** объекта **FileLock**, что здесь и показано.

Сжатие данных

Библиотека ввода/вывода Java содержит классы, поддерживающие ввод/вывод в сжатом формате (табл. 16.8). Они базируются на уже существующих потоках ввода/вывода.

Эти классы не являются частью иерархии символьно-ориентированных потоков Reader и Writer, они надстроены над байт-ориентированными классами InputStream и OutputStream, так как библиотека сжатия работает не с символами, а с байтами. Впрочем, никто не запрещает смешивать потоки. (Помните, как легко преобразовать потоки из байтовых в символьные — достаточно использовать классы InputStreamReader и OutputStreamWriter.)

Таблица 16.8. Классы для сжатия данных

Название класса	Назначение
CheckedInputStream	Его метод getCheckSum() возвращает контрольную сумму для любого входного потока InputStream (не только для потока распаковки)
CheckedOutputStream	Его метод getCheckSum() возвращает контрольную сумму для любого выходного потока OutputStream (не только для потока сжатия)
DeflaterOutputStream	Базовый класс для классов сжатия данных
ZipOutputStream	Подкласс DeflaterOutputStream, который производит сжатие данных в формате файлов ZIP
GZIPOutputStream	Подкласс DeflaterOutputStream, который производит сжатие данных в формате файлов GZIP
InflaterInputStream	Базовый класс для классов распаковки сжатых данных
ZipInputStream	Подкласс InflaterInputStream, который распаковывает сжатые данные, хранящиеся в формате файлов ZIP
GZIPInputStream	Подкласс InflaterInputStream, распаковывающий сжатые данные, хранящиеся в формате файлов 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 данных в формате GZIP является наиболее простым и идеально подходит для ситуаций, где имеется один поток данных, который необходимо уплотнить (а не разрозненные фрагменты данных). В следующем примере сжимается файл:

```
//  io/GZIPcompress.java
// {Параметры: GZIPcompress.java}
import java.util.zip.*;
import java.io.*;

public class GZIPcompress {
     public static void main(String[] args)
     throws IOException {
```

продолжение ⇨

```
            if(args.length == 0) {
                    System.out.println(
                            "Использование: \nGZIPcompress file\n" +
                            "\tИспользует метод GZIP для сжатия " +
                            "файла в архив 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("Запись файла");
            int c;
            while((c = in.read()) != -1)
                    out.write(c);
            in.close();
            out.close();
            System.out.println("Чтение файла");
            BufferedReader in2 = new BufferedReader(
                    new InputStreamReader(new GZIPInputStream(
                            new FileInputStream("test.gz"))));
            String s;
            while((s = in2.readLine()) != null)
                    System.out.println(s);
    }
} ///:~
```

Работать с классами сжатия данных очень просто: вы просто надстраиваете их для своего потока данных (**GZIPOutputStream** или **ZipOutputStream** для сжатия, **GZIPInputStream** или **ZipInputStream** для распаковки данных). Дальнейшее сводится к элементарным операциям ввода/вывода. В примере продемонстрированы смешанные байтовые и символьные потоки: поток **in** основан на **Reader**, тогда как конструктор класса **GZIPOutputStream** использует только потоки на основе **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 гораздо более широкими возможностями. С ее помощью можно легко упаковывать произвольное количество файлов, а для чтения файлов в формате ZIP даже определен отдельный класс. В библиотеке поддержан стандартный ZIP-формат, поэтому сжатые ею данные будут восприниматься практически любым упаковщиком. Структура следующего примера совпадает со структурой предыдущего, но количество файлов, указываемых в командной строке, не ограничено. Вдобавок демонстрируется применение класса **Checksum** для получения и проверки контрольной суммы. Таких типов контрольных сумм в Java два: один представлен классом **Adler32**´(этот алгоритм быстрее), а другой — классом **CRC32** (медленнее, но точнее).

```
//: io/ZipCompress.java
// Использование формата ZIP для сжатия любого
```

```
// количества файлов, указанных в командной строке.
// {Параметры: 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("Проверка ZIP-сжатия Java"),
              // Однако парного метода для получения комментария
              // getComment() не существует
              for(String arg . args) {
                       print("Запись файла " + 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 становится действительной
              // только после закрытия файла с архивом!
              print("Checksum  " + csum getChecksum() getValue()),
              // Теперь извлекаем файлы·
              print("Чтение файла"),
              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("Контрольная сумма  " + csumi getChecksum() getValue()),
              bis close(),
              // Альтернативный способ открытия и чтения
              // файлов в формате ZIP·
              ZipFile zf = new ZipFile("test zip"),
              Enumeration e = zf entries(),
              while(e hasMoreElements()) {
                       ZipEntry ze2 = (ZipEntry)e nextElement(),
                       print("File. " + ze2),
```

продолжение ⇩

```
                // ... затем данные извлекаются так же, как прежде
            }
            /* if(args length == 1) */
        }
    } ///:~
```

Для каждого файла, добавляемого в архив, необходимо вызвать метод putNextEntry() с соответствующим объектом ZipEntry. Класс ZipEntry содержит все необходимое для добавления к отдельной записи ZIP-файла дополнительной информации: имени файла, размера в сжатом и обычном виде, контрольной суммы CRC, дополнительных данных, комментариев, метода сжатия, признака каталога. В исходном формате ZIP также можно задать пароли, но библиотека Java не поддерживает эту возможность. 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 вынужденное, поскольку продиктовано требованиями формата ZIP, однако при этом быстрая контрольная сумма Adler32 оказывается в неравных условиях с CRC.

Для извлечения файлов в классе ZipInputStream предусмотрен метод getNextEntry(), который возвращает очередной элемент архива ZipEntry. Для получения более компактной записи можно использовать для архива объект ZipFile, чей метод entries() возвращает итератор Enumeration, с помощью которого можно перемещаться по доступным элементам архивного файла.

Чтобы иметь доступ к контрольной сумме, необходимо каким-либо образом хранить представляющий ее объект Checksum. В нашем случае сохраняются ссылки на потоки CheckedInputStream и CheckedOutputStream, хотя можно было бы просто сохранить ссылки на объекты Checksum.

Неясно, 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 нельзя. Полноценные комментарии поддерживаются только для отдельных вхождений ZIP-архива, объектов ZipEntry.

Конечно, уплотняемые данные не ограничены файлами; пользуясь библиотеками ZIP и GZIP, вы можете сжимать все, что угодно, даже данные сетевых потоков.

Архивы Java ARchives (файлы JAR)

Формат ZIP также применяется в файлах JAR (архивы Java ARchive), предназначенных для упаковки группы файлов в один сжатый файл. Как и все в языке Java, файлы JAR являются кросс-платформенными, поэтому не нужно заботиться о совместимости платформ. Наравне с файлами классов, в них могут содержаться также любые файлы — например, графические и мультимедийные.

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. Объединение всех нужных для запуска апплета файлов в одном сжатом файле сокращает время запроса к серверу, при этом уменьшается и загрузка сервера. Кроме того, каждый элемент архива JAR можно снабдить цифровой подписью.

Файл JAR представляет собой файл, в котором хранится набор сжатых файлов вместе с *манифестом* (manifest), который их описывает. (Вы можете создать манифест самостоятельно или же поручить эту работу программе jar.) За подробной информацией о манифестах JAR обращайтесь к документации JDK.

Инструмент jar, который 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 служебные символы не нужны). Пользователи систем UNIX/Linux сразу заметят сходство с программой tar. Допустимы следующие параметры:

c	Создание нового или пустого архива
t	Вывод содержимого архива
x	Извлечение всех файлов
x файл	Извлечение файла с заданным именем
f	Признак имени файла. Если не использовать этот параметр, jar решит, что входные данные поступают из стандартного ввода, или при создании файла выходные данные будут направляться в стандартный поток вывода
m	Означает, что первый аргумент содержит имя файла, содержащего манифест
v	Выводит краткое описание действий, выполняемых программой jar
o	Сохранение файлов без сжатия (для создания файлов JAR, которые можно указать в переменной окружения CLASSPATH)
M	Отказ от автоматического создания манифеста

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.

Несколько примеров наиболее распространенных вариантов запуска программы jar:

```
jar cf myJarFile jar * class
```

Команда создает файл JAR с именем myJarFile.jar, в котором содержатся все файлы классов из текущего каталога, с автоматически созданным манифестом:

```
jar cmf myJarFile jar myManifestFile mf * class
```

Почти идентична предыдущей команде, за одним исключением — в полученный файл JAR включается пользовательский манифест из файла myManifestFile.mf:

```
jar tf myJarFile jar
```

Вывод содержимого (списка файлов) архива myJarFile.jar:

```
jar tvf myJarFile jar
```

К предыдущей команде добавлен параметр 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, содержимое которых включается в файл myApp.jar. Благодаря параметру v в процессе сжатия выводится дополнительная информация об упаковываемых файлах.

Инструмент jar не обладает возможностями архиватора zip. Например, он 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).

Сериализация объектов

Сериализация (serialization) *объектов* Java позволяет вам взять любой объект, реализующий интерфейс Serializable, и превратить его в последовательность байтов, из 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, а затем 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 деталях.

Сама по себе сериализация объектов интересна потому, что с ее помощью можно осуществить *легковесное долговременное хранение* (lightweight persistence). Вспомните: 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». Причина использования добавки «легковесное» такова: объект нельзя определить как «постоянный» при помощи некоторого ключевого слова, то есть долговременное хранение напрямую не поддерживается языком (хотя вероятно, такая возможность появится в будущем). 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 Java JDO или инструмент, подобный Hibernate (http://hibernate.sourceforge.net).

Механизм сериализации объектов был добавлен в язык для поддержки двух расширенных возможностей. *Удаленный вызов методов* Java (RMI) позволяет работать с объектами, находящимися на других компьютерах, точно так же, как и с теми, что существуют на вашей машине. При посылке сообщений удаленным объектам необходимо транспортировать аргументы и возвращаемые значения, а для этого используется сериализация объектов.

Сериализация объектов также необходима визуальным компонентам JavaBean. Информация о состоянии визуальных компонентов обычно изменяется 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 интерфейс для самоидентификации, в нем нет ни одного метода). Когда в язык был добавлен механизм сериализации, во многие классы стандартной библиотеки внесли изменения так, чтобы они были готовы к сериализации. К таким классам относятся все классы-оболочки для простейших типов, все классы контейнеров и многие другие. Даже объекты Class, представляющие классы, можно сериализовать.

Чтобы сериализовать объект, требуется создать выходной поток OutputStream, который нужно вложить в объект ObjectOutputStream. По сути, вызов метода writeObject() осуществляет сериализацию объекта, и далее вы пересылаете его в выходной поток данных OutputStream. Для восстановления объекта необходимо надстроить объект ObjectInputStream для входного потока InputStream, а затем вызвать метод readObject(). Как обычно, такой метод возвращает ссылку на обобщенный объект Object, поэтому после вызова метода следует провести нисходящее преобразование для получения объекта нужного типа.

Сериализация объектов проводится достаточно разумно и в отношении ссылок, имеющихся в объекте. Сохраняется не только сам образ объекта, но и все связанные с ним объекты, 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 присутствующих в объектах ссылок стало бы весьма нелегкой задачей. Однако в Java никаких трудностей со ссылками нет — судя по всему, в этот язык встроен достаточно эффективный алгоритм создания графов объектов. Следующий пример проверяет механизм сериализации: мы создаем 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
// Тест сериализации 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); }
}

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;
```

продолжение ⇩

```
        // значение i == количество сегментов
        public Worm(int i, char x) {
                print("Конструктор Worm: " + i);
                c = x;
                if(--i > 0)
                        next = new Worm(i, (char)(x + 1));
        }
        public Worm() {
                print("Конструктор по умолчанию");
        }
        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 очистка буфера вывода
                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("Память объекта Worm\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:
Конструктор Worm: 6
Конструктор Worm: 5
Конструктор Worm: 4
Конструктор Worm: 3
Конструктор Worm: 2
Конструктор Worm: 1
w = :a(853) b(119):c(802) d(788) e(199):f(881)
Память объекта Worm
w2 = :a(853):b(119):c(802):d(788):e(199):f(881)
Память объекта Worm
```

```
w3 = :a(853):b(119):c(802):d(788):e(199):f(881)
*///.~
```

Чтобы пример был 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 объект Worm помечается порядковым номером-символом (char), который автоматически генерируется в процессе рекурсивного формирования связанной цепочки объектов Worm. При создании цепочки ее размер указывается 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 указывает на конец цепочки.

Все это делалось лишь по одной причине: для создания более или менее сложной структуры, которая не может быть сериализована тривиальным образом. Впрочем, сам акт сериализации проходит проще простого. После создания потока 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 примитивные 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 производится в файл, а во втором для разнообразия хранилищем служит массив байтов ByteArray. Чтение и запись объектов посредством сериализации возможна в любые потоки, в том числе и в сетевые соединения.

Из выходных данных видно, что восстановленный объект в самом деле содержит все ссылки, которые были 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.

Обнаружение класса

Вам может быть интересно, 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, которые были записаны в файл в процессе сериализации?

Самый надежный способ получить ответ на этот вопрос — провести эксперимент. Следующий файл располагается в подкаталоге данной главы:

```
//: io/Alien java
// Сериализуемый класс
import java.io.*;
public class Alien implements Serializable {} ///:~
```

Файл с программой, создающей и сериализующей объект Alien, находится в том же подкаталоге:

```
//  io/FreezeAlien java
// Создание файла с данными сериализации
import java io *,

public class FreezeAlien {
     public static void main(String[] args) throws Exception {
               ObjectOutput out = new ObjectOutputStream(
                        new FileOutputStream("X file")),
               Alien quellek = new Alien(),
               out writeObject(quellek),
     }
} /// ~
```

Вместо того чтобы перехватывать и обрабатывать исключения, программа идет по простому пути — исключения передаются за пределы main(), поэтому сообщения о них будут выдаваться на консоль.

После того как вы скомпилируете и запустите этот код, в каталоге c12 появится файл с именем X.file. Следующая программа скрыта от чужих глаз в «секретном» 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 открыв файл и прочитав из него данные для восстановления объекта mystery, виртуальная машина Java (JVM) не сможет найти файл Alien.class; объект Class для объекта Alien будет в не досягаемости (в примере сознательно не рассматривается возможность обнаружения через переменную окружения CLASSPATH). Возникнет исключение ClassNotFoundException.

Управление сериализацией

Как вы могли убедиться, стандартный 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 сериализовать какой-либо объект, содержащийся в главном объекте, не имеет смысла, так как немного погодя его все равно потребуется создать заново.

Вы можете управлять процессом сериализации, реализуя 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 для конкретного объекта.

В следующем примере продемонстрирована простая реализация интерфейса Externalizable. Заметьте также, что классы Blip1 и Blip2 практически одинаковы, если не считать одного малозаметного отличия:

```
// io/Blips.java
// Простая реализация интерфейса Externalizable. . с проблемами
import java.io.*;
import static net.mindview.util.Print.*;

class Blip1 implements Externalizable {
    public Blip1() {
        print("Конструктор Blip1");
    }
    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("Конструктор Blip2");
    }
    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("Создание объектов:");
        Blip1 b1 = new Blip1();
        Blip2 b2 = new Blip2();
        ObjectOutputStream o = new ObjectOutputStream(
                new FileOutputStream("Blips.out"));
        print("Сохранение объектов:");
        o.writeObject(b1);
        o.writeObject(b2);
        o.close();
        // Восстановление сохраненных объектов:
```

продолжение

```
                ObjectInputStream in = new ObjectInputStream(
                        new FileInputStream("Blips out"));
                print("Восстановление b1:");
                b1 = (Blip1)in.readObject();
                // Вот те раз! Исключение
//! print("Восстановление b2:");
//! b2 = (Blip2)in readObject();
        }
} /* Output:
Создание объектов·
Конструктор Blip1
Конструктор Blip2
Сохранение объектов:
Blip1 writeExternal
Blip2 writeExternal
Восстановление b1:
Конструктор Blip1
Blip1 readExternal
*///:~
```

Итак, объект **Blip2** в программе не восстанавливается — попытка приводит к возникновению исключения. Заметили ли вы различие между классами **Blip1** и **Blip2**? Конструктор класса **Blip1** объявлен 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 объекта **b1** вызывается конструктор по умолчанию класса **Blip1**. Это отличается от восстановления объекта, реализующего интерфейс **Serializable**, которое проводится на основе данных сериализации, без вызова конструкторов. В случае с объектом **Externalizable** происходит нормальный процесс конструирования (включая инициализацию в точке определения), и далее вызывается метод **readExternal()**. Вам следует иметь это в виду при реализации объектов **Externalizable** — в особенности обратите внимание на то, что вызывается конструктор по умолчанию.

Следующий пример показывает, 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; // Без инициализации
     public Blip3() {
             print("Конструктор Blip3"),
             // s, i не инициализируются
     }
     public Blip3(String x, int a) {
             print("Blip3(String x, int a)"),
             s = x,
             i = a,
```

```
            // s и i инициализируются только вне конструктора по умолчанию
    }
    public String toString() { return s + i; }
    public void writeExternal(ObjectOutput out)
    throws IOException {
            print("Blip3.writeExternal");
            // Необходимо действовать так:
            out.writeObject(s);
            out.writeInt(i);
    }
    public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException {
            print("Blip3.readExternal");
            // Необходимо действовать так:
            s = (String)in.readObject();
            i = in.readInt();
    }
    public static void main(String[] args)
    throws IOException, ClassNotFoundException {
            print("Создание объектов:");
            Blip3 b3 = new Blip3("Строка ", 47);
    print(b3);
    ObjectOutputStream o = new ObjectOutputStream(
      new FileOutputStream("Blip3 out"));
    print("Сохранение объекта:");
    o.writeObject(b3);
    o.close();
    // Now get it back:
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("Blip3.out"));
    print("Восстановление b3:");
    b3 = (Blip3)in.readObject();
    print(b3);
  }
} /* Output:
Создание объектов:
Blip3(String x, int a)
Строка 47
Сохранение объекта:
Blip3.writeExternal
Восстановление b3:
Конструктор Blip3
Blip3.readExternal
Строка 47
*///:~
```

Поля s и i инициализируются только во втором конструкторе, но не в конструкторе по умолчанию. Это значит, что, если переменные s и i не будут инициализированы в методе readExternal(), s останется ссылкой null, а i будет равно нулю (так как при создании объекта его память обнуляется). Если вы закомментируете две строки после фраз **Необходимо действовать так** 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 свой объект.

Итак, чтобы сериализация выполнялась правильно, нужно не просто записать всю значимую информацию в методе writeExternal() (для объектов Externalizable не существует автоматической записи объектов-членов), но и восстановить ее затем в методе readExternal(). Хотя сначала можно запутаться и подумать, что из-за вызова конструктора по умолчанию все необходимые действия по записи и восстановлению объектов Externalizable происходят сами по себе. Но это не так.

Ключевое слово transient

При управлении процессом сериализации может возникнуть ситуация, при которой автоматическое сохранение и восстановление некоторого подобъекта нежелательно — например, если в объекте хранится некоторая конфиденциальная информация (пароль и т. д.). Даже если информация в объекте описана как закрытая (private), это не спасает ее от сериализации, после которой можно 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 управление отдельными элементами находится в ваших руках (внутри writeExternal).

Однако работать с объектами Serializable удобнее, поскольку сериализация для них проходит полностью автоматически. Чтобы запретить запись некоторых полей объекта Serializable, воспользуйтесь ключевым словом transient. Фактически оно означает: «Это не нужно ни сохранять, ни восстанавливать — это мое дело».

Для примера возьмем объект Login, который содержит информацию о некотором сеансе входа в систему, с вводом пароля и имени пользователя. Предположим, что после проверки информации ее необходимо сохранить, выборочно, без пароля. Проще всего удовлетворить заявленным требованиям, реализуя интерфейс Serializable и объявляя поле с паролем password как transient. Вот как это будет выглядеть:

```
//: io/Logon.java
// Ключевое слово "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() {
              return "информация о сеансе: \n   пользователь: " + username +
```

```
                    "\n   дата  " + date + "\n   пароль  " + password;
    }
    public static void main(String[] args) throws Exception {
            Logon a = new Logon("Пользователь", "Пароль");
            print("Сеанс a = " + a);
            ObjectOutputStream o = new ObjectOutputStream(
                    new FileOutputStream("Logon out"));
            o.writeObject(a);
            o.close();
            TimeUnit.SECONDS.sleep(1); // Задержка
            // Восстановление:
            ObjectInputStream in = new ObjectInputStream(
                    new FileInputStream("Logon out"));
            print("Восстановление объекта  Время: " + new Date());
            a = (Logon)in.readObject();
            print("Сеанс a = " + a);
    }
} /* Output
Сеанс a = информация о сеансе
   пользователь: Пользователь
   дата: Sat Nov 19 15:03:26 MST 2005
   пароль  Пароль
Восстановление объекта. Время: Sat Nov 19 15.03 28 MST 2005
Сеанс a = информация о сеансе
   пользователь. Пользователь
   дата  Sat Nov 19 15.03 26 MST 2005
   пароль. null
*/// ~
```

Поля date и username не имеют модификатора transient, поэтому сериализация для них проводится автоматически. Однако поле password описано как transient и поэтому не сохраняется на диске; механизм сериализации его также игнорирует. При восстановлении объекта поле password равно null. Заметьте, что при соединении строки (String) со ссылкой null перегруженным оператором + 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, что я сказал «добавить», а не «переопределить» или «реализовать») методы с именами writeObject() и readObject(). Они автоматически вызываются при сериализации и восстановлении объектов. Иначе говоря, эти два метода заменят собой сериализацию по умолчанию.

Эти методы должны иметь жестко фиксированную сигнатуру:

```
private void writeObject(ObjectOutputStream stream)
throws IOException;
```

продолжение ⇩

```
private void readObject(ObjectInputStream stream)
throws IOException, ClassNotFoundException
```

С точки зрения проектирования программы этот подход вообще непонятен. Во-первых, можно подумать, что, раз уж эти методы не являются частью базового класса и не определены в интерфейсе Serializable, у них должен быть свой собственный интерфейс. Но так как методы объявлены закрытыми (private), вызываться они могут лишь членами их собственного класса. Однако члены обычных классов их не вызывают, вместо этого вызов исходит из методов writeObject() и readObject() классов ObjectInputStream и ObjectOutputStream. (Я не стану разражаться долгой тирадой о выборе тех же имен методов, а скажу лишь одно слово: неразумно.) Интересно, каким образом классы ObjectInputStream и ObjectOutputStream обращаются к закрытым членам другого класса? Можно лишь предположить, что это относится к таинству сериализации.

Так или иначе, все методы, описанные в интерфейсе, реализуются как открытые (public), поэтому, если методы writeObject() и readObject() должны быть закрытыми (private), они не могут быть частью какого-либо интерфейса. Однако вам необходимо строго следовать указанной нотации, и результат очень похож на реализацию интерфейса.

Судя по всему, при вызове метода ObjectOutputStream.writeObject() передаваемый ему объект Serializable тщательно анализируется (вне всяких сомнений, с использованием механизма рефлексии) в поисках его собственного метода writeObject(). Если такой метод существует, процесс стандартной сериализации пропускается, и вызывается метод объекта writeObject(). Аналогичные действия происходят и при восстановлении объекта.

Существует и еще одна хитрость. В вашем собственном методе writeObject() можно вызвать используемый в обычной сериализации метод writeObject(), для этого вызывается метод defaultWriteObject(). Аналогично, в методе readObject() можно вызвать метод стандартного восстановления defaultReadObject(). Следующий пример показывает, как производится пользовательское управление хранением и восстановлением объектов Serializable:

```
//: io/SerialCtl.java
// Управление сериализацией с определением 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 = "Не объявлено transient: " + aa;
              b = "Объявлено transient: " + bb;
     }
     public String toString() { return a + "\n" + b; }
     private void writeObject(ObjectOutputStream stream)
     throws IOException {
              stream.defaultWriteObject();
              stream.writeObject(b);
     }
     private void readObject(ObjectInputStream stream)
```

```
    throws IOException, ClassNotFoundException {
            stream.defaultReadObject();
            b = (String)stream.readObject();
    }
    public static void main(String[] args)
    throws IOException, ClassNotFoundException {
            SerialCtl sc = new SerialCtl("Test1", "Test2");
            System.out.println("Перед записью \n" + sc);
            ByteArrayOutputStream buf= new ByteArrayOutputStream();
            ObjectOutputStream o = new ObjectOutputStream(buf);
            o.writeObject(sc);
            // Now get it back.
            ObjectInputStream in = new ObjectInputStream(
                    new ByteArrayInputStream(buf.toByteArray()));
            SerialCtl sc2 = (SerialCtl)in.readObject();
            System.out.println("После восстановления:\n" + sc2);
    }
} /* Output:
Перед записью
Не объявлено transient: Test1
Объявлено transient: Test2
После восстановления:
Не объявлено transient: Test1
Объявлено transient: Test2
*///.~
```

В данном примере одно из строковых полей класса объявлено с ключевым словом **transient**, чтобы продемонстрировать, что такие поля при вызове метода **defaultWriteObject()** не сохраняются. Строка сохраняется и восстанавливается программой явно. Поля класса инициализируются в конструкторе, а не в точке определения; это демонстрирует, что они не инициализируются каким-либо автоматическим механизмом в процессе восстановления.

Если вы собираетесь использовать встроенный механизм сериализации для записи обычных (не-**transient**) составляющих объекта, нужно при записи объекта в первую очередь вызвать метод **defaultWriteObject()**, а при восстановлении объекта — метод **defaultReadObject()**. Это вообще загадочные методы. Например, если вызвать метод **defaultWriteObject()** для потока **ObjectOutputStream** без передачи 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.

Сохранение и восстановление **transient**-объектов выполняется относительно просто. В методе **main()** создается объект **SerialCtl**, который затем сериализуется потоком **ObjectOutputStream**. (При этом для вывода используется буфер, а не файл — для потока **ObjectOutputStream** это несущественно.) Непосредственно сериализация выполняется в 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 рефлексии.) Если поиск успешен, найденный метод вовлекается в процедуру сериализации. Примерно такой же подход наблюдается и при восстановлении объекта

методом readObject(). Возможно, это единственное решение задачи, но все равно выглядит очень странно.

Долговременное хранение

Было бы замечательно привлечь технологию сериализации, чтобы сохранить состояние вашей программы для его последующего восстановления. Но перед тем как это делать, необходимо ответить на несколько вопросов. Что произойдет при сохранении двух объектов, содержащих ссылку на некоторый общий третий объект? Когда вы восстановите эти объекты, сколько экземпляров третьего объекта появится в программе? А 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();
              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 восстанавливаем записанные объекты
              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@addbf1], 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 стоит обратить внимание на использование механизма сериализации и байтового массива для «глубокого копирования» любого объекта с интерфейсом Serializable. (Глубокое копирование — создание дубликата всего графа объектов, а не просто основного объекта и его ссылок.)

Объекты Animal содержат поля типа House. В методе main() создается список ArrayList с несколькими объектами Animal, его дважды записывают в один поток и еще один раз — в отдельный поток. Когда эти списки восстанавливают и распечатывают, получается приведенный ранее результат (объекты при каждом запуске программы будут располагаться в различных областях памяти).

Конечно, нет ничего удивительного в том, что восстановленные объекты и их оригиналы будут иметь разные адреса. Но заметьте тот факт, что адреса в восстановленных объектах animals1 и animals2 совпадают, вплоть до повторения ссылок на объект House, общий для обоих списков. 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 совершенно иное семейство взаимосвязанных объектов.

Если вы будете проводить сериализацию с использованием единого выходного потока, сохраненная сеть объектов гарантированно восстановится в первоначальном виде, без излишних повторений объектов. Конечно, записать объекты можно тогда, когда они еще не приняли окончательного состояния, 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., у вас не получится безопасного хранилища состояния системы. Вместо этого следует поместить все объекты, являющиеся слагаемыми состояния системы в целом, в контейнер и сохранить этот контейнер *единой* операцией. Затем можно восстановить его вызовом одного метода.

Следующий пример — имитатор воображаемой системы автоматизированного проектирования (CAD), в котором используется такой подход. Вдобавок в нем продемонстрировано сохранение статических (**static**) полей — 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 состояния вымышленной системы CAD
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);
          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>>();
                // Добавление ссылок на объекты класса:
                shapeTypes.add(Circle.class);
                shapeTypes.add(Square.class);
                shapeTypes.add(Line.class);
                List<Shape> shapes = new ArrayList<Shape>();
                // Создание фигур:
                for(int i = 0; i < 10; i++)
                        shapes.add(Shape.randomFactory());
                // Назначение всех статических цветов:
                for(int i = 0; i < 10; i++)
                        ((Shape)shapes.get(i)).setColor(Shape.GREEN);
                // Сохранение вектора состояния:
                ObjectOutputStream out = new ObjectOutputStream(
                        new FileOutputStream("CADState.out"));
                out.writeObject(shapeTypes);
                Line.serializeStaticState(out);
                out.writeObject(shapes);
                // Вывод фигур:
                System.out.println(shapes);
        }
} /* Output:
[class Circlecolor[3] xPos[58] yPos[55] dim[93]
  class Squarecolor[3] xPos[61] yPos[61] dim[29]
```

продолжение ⇘

```
. 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 поддерживают сериализацию и восстановление. 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.) Для задания цвета некоторого типа фигур можно переопределить методы базового класса (статические методы не используют динамическое связывание). Метод randomFactory() создает при каждом вызове новую фигуру, используя для этого случайные значения Shape.

Классы Circle и Square — простые подклассы Shape, различающиеся только способом 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 вполне тривиально:

```
//  io/RecoverCADState java
// Восстановление состояния вымышленной системы CAD
// {RunFirst  StoreCADState}
import java io.*,
import java util *,

public class RecoverCADState {
     @SuppressWarnings("unchecked")
     public static void main(String[] args) throws Exception {
               ObjectInputStream in = new ObjectInputStream(
                         new FileInputStream("CADState.out")),
               // Данные читаются в том порядке, в котором они были записаны·
               List<Class<? extends Shape<@062>> 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]
]
*/// ~
```

Мы видим, 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 (то есть константа RED), а в прямоугольниках поля color вообще равны нулю (помните, в этих объектах инициализация проходит в конструкторе). Похоже, статические поля вообще не сериализовались! Да, это именно так — хотя класс Class и реализует интерфейс Serializable, происходит это не так, как нам хотелось бы. Отсюда, 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 этих программ необходимо сделать следующее:

1. Добавьте методы serializeStaticState() и deserializeStaticState() во все фигуры Shape.
2. Уберите из программы список shapeTypes и весь связанный с ним код.
3. При сериализации и восстановлении вызывайте новые методы для сохранения статической информации.

Также стоит позаботиться о безопасности, ведь сериализация сохраняет и закрытые (private) поля. Если в вашем объекте имеется конфиденциальная информация, 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.

Предпочтения

В пакете JDK 1.4 появился программный интерфейс API для работы с *предпочтениями* (preferences). Предпочтения гораздо более тесно связаны с долговременным хранением, чем механизм сериализации объектов, поскольку они позволяют автоматически сохранять и восстанавливать вашу информацию. Однако они применимы лишь к небольшим, ограниченным наборам данных — хранить в них можно только примитивы и строки, и длина строки не должна превышать 8 Кбайт (не так уж мало, но вряд ли подойдет для решения серьезных задач). Как и предполагает название нового API, *предпочтения* предназначены для хранения и получения информации о предпочтениях пользователя и конфигурации программы.

Предпочтения представляют собой наборы пар «ключ-значение» (как в картах), образующих иерархию узлов. Хотя иерархия узлов и годится для построения сложных структур, чаще всего создают один узел, имя которого совпадает с именем класса, и хранят информацию в нем. 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));
               // Всегда необходимо указывать значение по умолчанию:
               print("How many companions does Dorothy have? " +
               prefs.getInt("Companions", 0));
     }
} /* Output:
Location: Oz
Footwear: Ruby Slippers
Companions: 4
Are there witches?: true
UsageCount: 53
How many companions does Dorothy have? 4
*///:~
```

Здесь используется метод **userNodeForPackage()**, но с тем же успехом можно было бы заменить его методом **systemNodeForPackage()**, это дело вкуса. Предполагается, что префикс **user** используется для хранения индивидуальных предпочтений пользователя, а **system** — для хранения информации общего плана о настройках установки. Так как метод **main()** статический, для идентификации узла применен класс **PreferencesDemo.class**, хотя в нестатических методах обычно вызывается метод **getClass()**. Использовать текущий класс для идентификации узла не обязательно, но чаще всего именно так и поступают.

Созданный узел используется 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 непривычно, если вы привыкли использовать метод **keys()** в коллекциях. 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 мы знаем, что каждому из них сопоставлено значение,

поэтому передача null по умолчанию безопасна, но обычно используется именованный ключ:

```
prefs.getInt("Companions ", 0)).
```

В таких ситуациях стоит обзавестись имеющим смысл значением по умолчанию. В качестве характерного средства выражения часто выступает следующего рода конструкция:

```
int usageCount = prefs.getInt("UsageCount", 0);
usageCount++;
prefs.putInt("UsageCount", usageCount);
```

В этом случае при первом запуске программы значение переменной usageCount будет нулем, а при последующих запусках оно должно измениться.

Запустив программу PreferencesDemo.java, вы увидите, что значение usageCount действительно увеличивается при каждом запуске программы, но где же хранятся данные? Никакие локальные файлы после запуска программы не создаются. Система предпочтений привлекает для хранения данных системные ресурсы, а конкретная реализация зависит от операционной системы. Например, в Windows используется реестр (поскольку он и так представляет собой иерархию узлов с набором пар «ключ-значение»). С точки зрения программиста, реализация — это несущественно: информация сохраняется «сама собой», и вам не приходится беспокоиться о том, как это работает на различных системах.

Программный интерфейс предпочтений обладает гораздо большим возможностями, чем было показано в этом разделе. За более подробным описанием обратитесь к документации JDK, этот вопрос там описан достаточно подробно.

Резюме

Библиотека ввода/вывода Java удовлетворяет всем основным потребностям: она позволяет выполнять операции чтения и записи с консолью, файлами, буфером в памяти и даже сетевыми подключениями к Интернету. Наследование позволяет создавать новые типы объектов для ввода и вывода данных. Вы даже можете обеспечить расширяемость для объектов потока, использовав тот факт, что переопределенный метод toString() автоматически вызывается при передаче объекта методу, ожидающему String (ограниченное «автоматическое преобразование типов» Java).

Впрочем, ответы на некоторые вопросы не удается найти ни в документации, ни в архитектуре библиотеки ввода/вывода Java. Например, было бы замечательно, если бы при попытке перезаписи файла возбуждалось исключение — многие программные системы позволяют указать, что файл может быть открыт для вывода только тогда, когда файла с тем же 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 стерт.

При знакомстве с библиотекой ввода/вывода возникают смешанные чувства; с одной стороны, она берёт на себя значительный объем работы и к тому же

обеспечивает переносимость. Но пока вы не вполне поняли суть работы шаблона декоратора, архитектура библиотеки кажется не совсем понятной, и от вас потребуются определенные усилия для ее изучения и освоения. Кроме того, библиотека не совсем полна: например, только отсутствие необходимых средств заставило меня писать инструменты, подобные TextFile (новый класс Java SE5 PrintWriter — шаг в правильном направлении, но это лишь частичное решение). К числу значительных усовершенствований Java SE5 следует отнести и то, что в нем появились возможности форматирования вывода, присутствующие практически в любом другом языке.

Впрочем, как только вы *действительно* проникаетесь идеей шаблона декоратора и начинаете использовать библиотеку в ситуациях, где требуется вся ее гибкость, вы извлекаете из библиотеки все большую выгоду, и даже требующийся для надстроек дополнительный код не мешает этому.

Параллельное выполнение 17

До настоящего момента мы имели дело исключительно с последовательным программированием. Все действия, выполняемые программой, выполнялись друг за другом, то есть последовательно.

Последовательное программирование способно решить многие задачи. Однако для некоторых задач бывает удобно (и даже необходимо) организовать параллельное выполнение нескольких частей программы, чтобы создать у пользователя впечатление одновременного выполнения этих частей, или — если на компьютере установлено несколько процессоров — чтобы они действительно выполнялись одновременно.

Каждая из этих самостоятельных подзадач называется *потоком* (thread)[1]. Программа пишется так, словно каждый поток запускается сам по себе и использует процессор в монопольном режиме. На самом деле существует некоторый системный механизм, который обеспечивает совместное использованием процессора, но в основном думать об этом вам не придется.

Модель потоков (и ее поддержка в языке Java) является программным механизмом, упрощающим одновременное выполнение нескольких операций в одной и той же программе. Процессор периодически вмешивается в происходящие события, выделяя каждому потоку некоторый отрезок времени. Для каждого потока все выглядит так, словно процессор используется в монопольном режиме, но на самом деле время процессора разделяется между всеми существующими в программе потоками (исключение составляет ситуация, когда программа действительно выполняется на многопроцессорном компьютере). Однако при использовании потоков вам не нужно задумываться об этих тонкостях — код не зависит от того, на скольких процессорах вам придется работать. Таким

---

[1] Как уже упоминалось в главе 12, не следует путать два разных понятия — поток данных (stream) в системе ввода/вывода и поток выполнения (thread) в многозадачном окружении. — *Примеч. перев.*

образом, потоки предоставляют механизм масштабирования производительности — если программа работает слишком медленно, вы в силах легко ускорить ее, установив на компьютер дополнительные процессоры. Многозадачность и многопоточность являются, похоже, наиболее вескими причинами использования многопроцессорных систем.

Задачи

Программный поток представляет некоторую задачу или операцию, поэтому нам понадобятся средства для описания этой задачи. Их предоставляет интерфейс Runnable. Чтобы определить задачу, реализуйте Runnable и напишите метод run(), содержащий код выполнения нужных действий.

Например, задача LiftOff выводит обратный отсчет перед стартом:

```
//· concurrency/LiftOff.java
// Реализация интерфейса Runnable

public class LiftOff implements Runnable {
     protected int countDown = 10; // Значение по умолчанию
     private static int taskCount = 0;
     private final int id = taskCount++;
     public LiftOff() {}
     public LiftOff(int countDown) {
              this.countDown = countDown;
     }
     public String status() {
              return "#" + id + "(" +
                       (countDown > 0 ? countDown : "Liftoff!") + "), ";
     }
     public void run() {
              while(countDown-- > 0) {
                       System.out.print(status());
                       Thread.yield();
              }
     }
} ///:~
```

По идентификатору id различаются экземпляры задачи. Поле объявлено с ключевым словом final, поскольку оно не будет изменяться после инициализации.

Метод run() обычно содержит некоторый цикл, который продолжает выполняться до тех пор, пока не будет достигнуто некоторое завершающее условие. Следовательно, вы должны задать условие выхода из цикла (например, просто вернуть управление командой return). Часто run() выполняется в виде бесконечного цикла, а это означает, что при отсутствии завершающего условия выполнение будет продолжаться бесконечно (позднее в этой главе вы узнаете, как организовать безопасное завершение задач).

Вызов статического метода Thread.yield() в run() обращен к *планировщику потоков* (часть потокового механизма Java, обеспечивающая переключение процессора между потоками). Фактически он означает, что очередная важная часть

цикла была выполнена и теперь можно на время переключиться на другую задачу. Вызов yield() не обязателен, но в данном примере он обеспечивает более интересные результаты: вы с большей вероятностью увидите, что программный поток прерывает и возобновляет свою работу.

В следующем примере метод run() не выделяется в отдельный программный поток, а просто вызывается напрямую в main() (впрочем, поток все же 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, реализующий Runnable, должен содержать метод run(), но ничего особенного в этом методе нет — он не обладает никакими особыми потоковыми возможностями. Чтобы использовать многопоточное выполнение, необходимо явно связать задачу с потоком.

Класс Thread

Традиционный способ преобразования объекта Runnable в задачу заключается в передаче его конструктору Thread. Следующий пример показывает, как организовать выполнение LiftOff с использованием Thread:

```
//: concurrency/BasicThreads.java
// Простейший вариант использования 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 передается только объект Runnable. Метод start() выполняет необходимую инициализацию потока, после чего вызов метода run() интерфейса Runnable запускает задачу на выполнение в новом потоке.

Из выходных данных видно, что вызов start() быстро возвращает управление (сообщение «Waiting for LiftOff» появляется до завершения отсчета). В сущности, мы вызываем LiftOff.run(), а этот метод еще не завершил свое выполнение;

но, поскольку LiftOff.run() выполняется в другом потоке, в потоке main() в это время можно выполнять другие операции. (Данная возможность не ограничивается потоком main() — любой поток может запустить другой поток.) Получается, что программа выполняет два метода сразу — main() и LiftOff. run().

В программе можно легко породить дополнительные потоки для выполнения дополнительных задач:

```
//  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: (Пример)
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 задачи выполняются одновременно друг с другом, с поочередной активизацией и выгрузкой потоков. Переключение осуществляется автоматически планировщиком потоков. Если на компьютере установлено несколько процессоров, планировщик потоков автоматически распределяет потоки между разными процессорами.

При разных запусках программы будут получены разные результаты, поскольку работа планировщика потоков недетерминирована. Более того, вы наверняка увидите значительные различия в результатах работы данной программы-примера, запуская ее на различных версиях пакета JDK. К примеру, предыдущие версии JVM не слишком часто выполняли квантование времени, соответственно, поток 1 мог первым закончить свой цикл, затем все свои итерации произвел бы поток 2, и т. д. Фактически то же самое получилось бы, если бы вызывалась процедура, выполняющая все циклы одновременно, за тем исключением, что запуск совокупности потоков требует больших издержек. Более поздние версии JDK обеспечивают более качественное квантование, и каждый поток регулярно получает свою долю внимания. Как правило, Sun не упоминает о подобных изменениях, так что рассчитывать на определенные «правила поведения» потоков не стоит. Лучше всего при написании кода с потоками занять максимально консервативную позицию.

Когда метод main() создает объекты-потоки Thread, он не сохраняет на них ссылки. Обычный объект, «забытый» таким образом, стал бы легкой добычей сборщика мусора, но только не объект-поток Thread. Каждый поток (Thread) самостоятельно «регистрирует» себя, то есть на самом деле ссылка на него где-то существует, и сборщик мусора не вправе удалить его объект.

Исполнители

Исполнители (executors), появившиеся в библиотеке java.util.concurrent в Java SE5, упрощают многозадачное программирование за счет автоматизации управления объектами Thread. Они создают дополнительную логическую прослойку между клиентом и выполнением задачи; задача выполняется не напрямую клиентом, а промежуточным объектом. Исполнители позволяют управлять выполнением асинхронных задач без явного управления жизненным циклом потоков. Именно такой способ запуска задач рекомендуется использовать в Java SE5/6.

Вместо явного создания объектов Thread в MoreBasicThreads.java мы можем воспользоваться исполнителем. Объект LiftOff умеет выполнять определенную операцию и предоставляет единственный метод для выполнения. Объект ExecutorService умеет создавать необходимый контекст для выполнения объектов Runnable. В следующем примере класс CachedThreadPool создает один поток для каждой задачи. Обратите внимание: объект ExecutorService создается статическим методом класса Executors, определяющим разновидность исполнителя:

```
// 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 (Пример)
#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 для создания и управления всеми задачами в системе достаточно одного исполнителя.

Вызов shutdown() предотвращает передачу Executor новых задач. Текущий поток (в данном случае тот, в котором выполняется main()) продолжает выполняться со всеми задачами, переданными до вызова shutdown(). Работа программы прекращается после завершения всех задач в Executor.

CachedThreadPool в этом примере легко заменяется другим типом Executor. Например, в потоковом пуле фиксированного размера (FixedThreadPool) используется ограниченный набор потоков для выполнения переданных задач:

```
// concurrency/FixedThreadPool.java
import java.util.concurrent.*;

public class FixedThreadPool {
    public static void main(String[] args) {
        // В аргументе конструктора передается количество потоков
        ExecutorService exec = Executors.newFixedThreadPool(5);
```

продолжение ⇩

```
            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!),
*///:~
```

С FixedThreadPool дорогостоящая операция создания потоков выполняется только один раз, в самом начале, поэтому количество потоков остается фиксированным. Это способствует экономии времени, поскольку вам не приходится нести затраты, связанные с созданием потока, для каждой отдельной задачи. В системах, управляемых событиями, обеспечивается максимальная скорость выполнения обработчиков событий, так как они могут просто получить поток из пула. Перерасход ресурсов в такой схеме исключен, так как FixedThreadPool использует ограниченное количество объектов Thread.

Исполнитель SingleThreadExecutor представляет собой аналог FixedThreadPool с одним потоком. Например, он может быть полезен для выполнения долгосрочных операций (скажем, прослушивания входящих подключений по сокету) и для коротких операций, выполняемых в отдельных потоках, — скажем, обновления локальных или удаленных журналов.

Если SingleThreadExecutor передается более одной задачи, эти задачи ставятся в очередь, и каждая из них отрабатывает до завершения перед началом следующей задачи, причем все они используют один и тот же поток. В следующем примере мы видим, что задачи последовательно завершаются в порядке их поступления. Таким образом, SingleThreadExecutor организует последовательное выполнение поступающих задач и поддерживает свою (внутреннюю) очередь незавершенных 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++)
                    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!),
*///:~
```

Другой пример: допустим, имеется группа потоков, выполняющих операции с использованием файловой системы. Вы можете запустить эти задачи под управлением SingleThreadExecutor, чтобы в любой момент гарантированно выполнялось не более одной задачи. При таком подходе вам не придется возиться с синхронизацией доступа к общим ресурсам (без риска для целостности файловой системы). Возможно, в окончательной версии кода будет правильнее синхронизировать доступ к ресурсу (см. далее в этой главе), но SingleThreadExecutor позволит быстро организовать координацию доступа при построении рабочего прототипа.

Возврат значений из задач

Интерфейс Runnable представляет отдельную задачу, которая выполняет некоторую работу, но не возвращает значения. Если вы хотите, чтобы задача возвращала значение, реализуйте интерфейс Callable вместо интерфейса Runnable. Параметризованный интерфейс Callable, появившийся в Java SE5, имеет параметр типа, представляющий возвращаемое значение метода call() (вместо run()), а для его вызова 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 "результат TaskWithResult " + id;
     }
}

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 {
                                // Вызов get() блокируется до завершения:
                                System.out.println(fs.get());
                       } catch(InterruptedException e) {
                                System.out.println(e).
                                return;
                       } catch(ExecutionException e) {
                                System out println(e);
                       } finally {
                                exec.shutdown();
                       }
     }
```

продолжение ⇩

```
} /* Output·
результат TaskWithResult 0
результат TaskWithResult 1
результат TaskWithResult 2
результат TaskWithResult 3
результат TaskWithResult 4
результат TaskWithResult 5
результат TaskWithResult 6
результат TaskWithResult 7
результат TaskWithResult 8
результат TaskWithResult 9
*///:~
```

Метод submit() создает объект Future, параметризованный по типу результата, возвращаемому Callable. Вы можете обратиться к Future с запросом isDone(), чтобы узнать, завершена ли операция. После завершения задачи и появления результата производится его выборка методом get(). Если get() вызывается без предварительной проверки isDone(), вызов блокируется до появления результата. Также можно вызвать get() с интервалом тайм-аута.

Перегруженный метод Executors.callable() получает Runnable и выдает Callable. ExecutorService содержит методы для выполнения коллекций объектов Callable.

Ожидание

Другим способом управления вашими потоками является вызов метода sleep(), который переводит поток в состояние ожидания на заданное количество миллисекунд. Если в классе LiftOff заменить вызов yield() на вызов метода sleep(), будет получен следующий результат:

```
//: concurrency/SleepingTask.java
// Вызов sleep() для приостановки потока.
import java.util.concurrent.*;

public class SleepingTask extends LiftOff {
     public void run() {
              try {
                       while(countDown-- > 0) {
                                System.out.print(status());
                                // Старый стиль.
                                // Thread.sleep(100);
                                // Стиль 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 метода sleep() может привести к исключению InterruptedException; перехват этого исключения продемонстрирован в run(). Поскольку исключения не распространяются по потокам обратно в main(), вы должны локально обработать любые исключения, возникающие внутри задачи.

В Java SE5 появилась новая версия sleep(), оформленная в виде метода класса TimeUnit; она продемонстрирована в приведенном примере. Она делает программу более наглядной, поскольку вы можете указать единицы измерения продолжительности задержки. Класс TimeUnit также может использоваться для выполнения преобразований, как будет показано далее в этой главе.

На некоторых платформах задачи выполняются в порядке «идеального распределения» — от 0 до 4, затем снова от 4 до 0. Это вполне логично, поскольку после каждой команды вывода задача переходит в состояние ожидания, что позволяет планировщику потоков переключиться на другой поток. Тем не менее такое поведение зависит от базовой реализации потокового механизма, поэтому полагаться на него нельзя. Если вам потребуется управлять порядком выполнения задач, используйте средства синхронизации (см. далее) или же вообще откажитесь от использования потоков и напишите собственные функции синхронизации, которые передают управление друг другу в нужном порядке.

Приоритет

Приоритет (priority) потока сообщает планировщику информацию об относительной важности потока. 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 тупиковых ситуаций из-за приоритетов не возникает). Потоки с более низкими приоритетами просто запускаются чуть реже.

В подавляющем большинстве случаев все потоки должны выполняться со стандартным приоритетом. Любые попытки манипуляций с приоритетами обычно являются ошибкой.

Следующий пример демонстрирует использование приоритетов. Приоритет существующего потока читается методом getPriority() и задается методом setPriority():

```
//· concurrency/SimplePriorities.java
// Использование приоритетов потоков.
import java.util.concurrent.*;

public class SimplePriorities implements Runnable {
     private int countDown = 5;
     private volatile double d; // Без оптимизации
```

продолжение ⇩

```
        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) {
                        // Высокозатратная, 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:
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
...
*///:~
```

В этой версии метод **toString()** переопределяется и использует метод **Thread.toString()**, который выводит имя потока (его можно задать в конструкторе, но здесь имена автоматически генерируются в виде **pool-1-thread-1**, **pool-1-thread-2** и т. д.), приоритет и группу, к которой принадлежит поток. Переопределенная версия **toString()** также выводит обратный отсчет, выполняемый задачей. Обратите внимание: для получения ссылки на объект **Thread**, управляющий задачей, внутри самой задачи, следует вызвать метод **Thread.currentThread()**.

Мы видим, что приоритет последнего потока имеет наивысший уровень, а все остальные потоки находятся на низшем уровне. Учтите, что приоритет задается в начале выполнения **run()**; задавать его в конструкторе бессмысленно, потому что **Executor** в этот момент еще не начал выполнять задачу.

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 применял оптимизацию. Без этих вычислений вы не увидите эффекта установки различных приоритетов (попробуйте закомментировать цикл for с вычислениями). В процессе вычислений мы видим, что планировщик уделяет больше внимания потоку с приоритетом MAX_PRIORITY (по крайней мере, таково было поведение программы на машине под управлением Windows XP). Несмотря даже на то, что 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 несуразицей), 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. Тем не менее для обеспечения переключения контекста в программе периодически выполняются команды yield().

В пакете JDK предусмотрено 10 уровней приоритетов, однако это не слишком хорошо согласуется с большинством операционных систем. К примеру, в Windows имеется 7 классов приоритетов, таким образом, их соотношение неочевидно (хотя в операционной системе Sun Solaris имеется 2^{31} уровней). Переносимость обеспечивается только использованием универсальных констант MAX_PRIORITY, NORM_PRIORITY и MIN_PRIORITY.

Передача управления

Если вы знаете, что в текущей итерации run() сделано все необходимое, вы можете подсказать механизму планирования потоков, что процессором теперь может воспользоваться другой поток. Эта подсказка (*не более чем* рекомендация; нет никакой гарантии, что планировщик потоков «прислушается» к ней) воплощается в форме вызова метода yield(). Вызывая yield(), вы сообщаете системе, что в ней могут выполняться другие потоки *того же приоритета*.

В примере LiftOff метод yield() обеспечивает равномерное распределение вычислительных ресурсов между задачами LiftOff. Попробуйте закомментировать вызов Thread.yield() в LiftOff.run() и проследите за различиями. И все же, в общем случае не стоит полагаться на yield() как на серьезное средство настройки вашего приложения.

Потоки-демоны

Демоном называется поток, предоставляющий некоторый сервис, работая в фоновом режиме во время выполнения программы, но при этом 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("Все демоны запущены");
               TimeUnit.MILLISECONDS.sleep(175);
     }
} /* Output:
Все демоны запущены
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
...
*///:~
```

Чтобы назначить поток демоном, следует перед его запуском вызвать метод **setDaemon()**.

После того как **main()** завершит свою работу, ничто не препятствует завершению программы, поскольку в процессе не работают другие потоки, кроме демонов. Чтобы результаты запуска всех потоков-демонов были более наглядными, поток **main()** на некоторое время погружается в «сон». Без этого вы увидели бы только часть результатов при создании демонов. (Поэкспериментируйте с вызовом **sleep()** для интервалов разной продолжительности.)

В примере **SimpleDaemons.java** используется явное создание объектов **Thread** для установки их «демонского» флага. Вы также можете настроить атрибуты (демон, приоритет, имя) потоков, созданных исполнителем; для этого следует написать пользовательскую 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;
     }
} /// ~
```

Единственное отличие от обычной реализации **ThreadFactory** заключается в том, что в данном случае атрибут демона задается равным **true**. Теперь новый объект **DaemonThreadFactory** передается 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 {
                         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("Все демоны запущены");
               TimeUnit.MILLISECONDS.sleep(500); // Задержка
     }
} /// ~
```

Каждый статический метод создания **ExecutorService** перегружается для получения объекта **ThreadFactory**, который будет использоваться для создания новых потоков.

Сделаем еще один шаг — создадим 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>(),
```

продолжение ⇩

```
                        new DaemonThreadFactory());
        }
} /// ~
```

Чтобы узнать, какие значения должны передаваться при вызове конструктора базового класса, я просто заглянул в исходный код Executors.java.

Чтобы узнать, является ли поток демоном, вызовите метод isDaemon(). Если поток является демоном, то все потоки, которые он производит, также будут демонами, что и демонстрируется следующим примером:

```
//: concurrency/Daemons.java
// Потоки, порождаемые демонами, также являются демонами
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();
        printnb("d.isDaemon() = " + d.isDaemon() + ", ");
        // Даем потокам-демонам завершить процесс запуска:
        TimeUnit.SECONDS.sleep(1);
    }
} /* Output:
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 Daemon переводится в режим демона, а затем порождает группу новых потоков, которые *явно* не назначаются демонами, но при этом все равно оказываются ими. Затем Daemon входит в бесконечный цикл, на каждом шаге которого вызывается метод yield(), передающий управление другими процессам.

Учтите, что потоки-демоны завершают свои методы run() без выполнения секций finally:

```
//: concurrency/DaemonsDontRunFinally.java
// Потоки-демоны не выполняют секцию finally.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class ADaemon implements Runnable {
     public void run() {
               try {
                         print("Запускаем ADaemon");
                         TimeUnit.SECONDS.sleep(1);
               } catch(InterruptedException e) {
                         print("Выход через InterruptedException");
               } finally {
                         print("Должно выполняться всегда?");
               }
     }
}

public class DaemonsDontRunFinally {
     public static void main(String[] args) throws Exception {
               Thread t = new Thread(new ADaemon());
               t.setDaemon(true);
               t.start();
     }
} /* Output:
Запускаем ADaemon
*///:~
```

Запуск программы наглядно показывает, что секция finally не выполняется. С другой стороны, если закомментировать вызов setDaemon(), вы увидите, что секция finally была выполнена.

Такое поведение верно, даже если из предыдущих описаний finally у вас сложилось обратное впечатление. Демоны завершаются «внезапно», при завершении последнего не-демона. Таким образом, сразу же при выходе из main() JVM немедленно прерывает работу всех демонов, не соблюдая никакие формальности. Невозможность корректного завершения демонов ограничивает возможности их применения. Обычно объекты Executor оказываются более удачным решением, потому что все задачи, находящиеся под управлением Executor, могут быть завершены одновременно.

Варианты кодирования

Во всех предшествующих примерах все классы задач реализовали интерфейс Runnable. В очень простых случаях можно использовать альтернативное решение с прямым наследованием 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),
*///.~
```

Чтобы задать объектам **Thread** имена, вы вызываете соответствующий конструктор **Thread**. Имя читается в методе **toString()** при помощи **getName()**.

Также иногда встречается идиома самоуправляемой реализации **Runnable**:

```
//  concurrency/SelfManaged.java
// Реализация Runnable, содержащая собственый объект Thread

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),
*/// ~
```

В целом происходящее не так уж сильно отличается от наследования от Thread, разве что синтаксис получается чуть более громоздким. Однако реализация интерфейса позволяет наследовать от другого класса, тогда как в варианте с Thread это невозможно.

Обратите внимание на вызов start() в конструкторе. Приведенный пример очень прост, поэтому, скорее всего, в нем такое решение безопасно, но вы должны знать, что запуск потоков в конструкторе может создать изрядные проблемы — до завершения конструктора может быть запущена на выполнение другая задача, которая обратится к объекту в нестабильном состоянии. Это еще одна причина, по которой использование Executor предпочтительнее явного создания объектов Thread.

Иногда бывает разумно спрятать потоковый код внутри класса с помощью внутреннего класса, как показано здесь:

```
//  concurrency/ThreadVariations.java
// Создание потоков с использованием внутренних классов.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

// Используем именованный внутренний класс.
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);
     }
}
```

продолжение ⇩

```
// Используем безымянный внутренний класс:
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();
     }
}

// Используем именованную реализацию 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),
     }
}

// Используем анонимную реализацию 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 потоке:
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");
```

продолжение ⇩

```
            new ThreadMethod("ThreadMethod") runTask();
    }
} ///:~
```

InnerThread1 определяет именованный внутренний класс, производный от Thread, и создает экземпляр этого класса в конструкторе. Поступать так стоит в том случае, когда у внутреннего класса есть особые возможности (новые методы), которые могут понадобиться в других методах. Однако в большинстве случаев причина создания потока — использование функциональности класса Thread, поэтому в именованном внутреннем классе особой нужды нет. InnerThread2 показывает другое решение. В конструкторе создается безымянный внутренний субкласс Thread, преобразуемый восходящим преобразованием к ссылке на Thread t. Если другим методам класса понадобится обратиться к t, они смогут сделать это через интерфейс Thread, и им не нужно будет знать точный тип объекта.

Третий и четвертый классы примера повторяют первые два, только вместо класса Thread они используют интерфейс Runnable.

Класс ThreadMethod демонстрирует создание потока в методе. Вы вызываете метод, когда программа готова к запуску потока, а метод возвращает управление после запуска потока. Если поток выполняет только вспомогательные операции и не является фундаментальной частью класса, то этот способ, вероятно, удобнее и практичнее запуска потока из конструктора класса.

Присоединение к потоку

Любой 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 завершит свою работу (когда метод t.isAlive() вернет значение false).

Вызвать метод join() можно также и с аргументом, указывающим продолжительность ожидания (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, поэтому потребуется блок try-catch.

Все эти операции продемонстрированы в следующем примере:

```
//  concurrency/Joining.java
// Демонстрация 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() + " прерван  " +
                                "isInterrupted(): " + isInterrupted());
                        return;
                }
                print(getName() + " активизировался");
        }
}

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("Прерван");
                }
                print(getName() + " join завершен");
        }
}

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 был прерван. isInterrupted(): false
Doc join завершен
Sleepy активизировался
Dopey join завершен
*///:~
```

Класс Sleeper — это тип потока, который приостанавливается на время, указанное в его конструкторе. В методе run() вызов метода sleep() может закончиться по истечении времени задержки, но может и прерваться. В секции catch выводится сообщение о прерывании, вместе со значением, возвращаемым методом isInterrupted(). Когда другой поток вызывает interrupt() для данного потока, устанавливается флаг, показывающий, что поток был прерван. Однако этот флаг сбрасывается при обработке исключения, поэтому внутри секции 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.

Совместное использование ресурсов

Однопоточную программу можно представить в виде одинокого работника, передвигающегося по своему пространству задачи и выполняющего по одной операции за один раз. Раз работник один, вам не приходится принимать во внимание проблему двух сущностей, пытающихся оспорить право на 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.

В условиях многозадачности ситуация меняется: у вас есть сразу два или три потока, которые стремятся получить доступ к одному и тому же ограниченному ресурсу. Если не предотвратить подобные конфликты, 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 одна задача генерирует четные числа, а другие задачи эти числа потребляют. Единственной задачей задач-потребителей является проверка четности этих чисел.

Начнем с определения EvenChecker, задачи-потребителя, поскольку она будет использоваться во всех последующих примерах. Чтобы отделить EvenChecker от различных генераторов, с которыми мы будем экспериментировать, мы определим абстрактный класс IntGenerator, содержащий минимум необходимых методов для EvenChecker: метод для получения следующего значения next() и методы отмены. Класс не реализует интерфейс Generator, потому что он должен выдавать int, а параметризация не поддерживает примитивные параметры.

```
//: 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(), изменяющий состояние флага canceled, и метод isCanceled(), проверяющий, был ли объект отменен. Поскольку флаг canceled относится к типу boolean, простые операции вроде присваивания и возврата значения выполняются *атомарно*, то есть без возможности прерывания,

и вы не увидите поле в промежуточном состоянии между этими простыми операциями. Смысл ключевого слова volatile будет объяснен 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 + " не четно!");
                                generator cancel(); // Отмена всех EvenChecker
                       }
              }
     }
     // Тестирование произвольного типа IntGenerator·
     public static void test(IntGenerator gp. int count) {
              System out.println("Нажмите Control-C. чтобы завершить программу").
              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);
     }
} ///·~
```

Как видно из run(), все задачи EvenChecker, зависящие от объекта IntGenerator, проверяют, не были ли они отменены. При таком подходе задачи, совместно использующие общий ресурс (IntGenerator), наблюдают за ресурсом, ожидая от него сигнала завершения. Тем самым устраняется так называемая «ситуация гонки», когда две и более задачи торопятся отреагировать на некоторое условие; это приводит к возникновению конфликтов или получению других некорректных результатов. Будьте внимательны, постарайтесь продумать все возможные сбои в системах с параллельным выполнением и защититься от них. Например, задача не может зависеть от другой задачи, потому что порядок завершения задач не гарантирован. Зависимость задач от объекта, не являющегося задачей, устраняет потенциальную «ситуацию гонки».

Метод test() настраивает и тестирует произвольный тип IntGenerator, запуская несколько задач EvenChecker с общим IntGenerator. Если IntGenerator приводит к сбою, test() сообщает о происходящем и возвращает управление. В противном случае его следует завершить вручную клавишами Ctrl+C.

Задачи EvenChecker постоянно читают и проверяют значения, полученные от IntGenerator. Если generator.isCanceled() равен true, run() возвращает управление; это сообщает Executor в EvenChecker.test() о том, что задача завершена. Любая задача EvenChecker может вызвать cancel() для связанного с ней IntGenerator, в результате чего все остальные EvenChecker, использующие IntGenerator, будут корректно завершены. Как будет показано далее в этой главе, в Java существуют и более общие механизмы завершения потоков.

В первом варианте IntGenerator, который мы рассмотрим, next() выдает серию четных значений:

```
// concurrency/EvenGenerator.java
// Конфликт потоков

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
Нажмите Control-C, чтобы завершить программу
89476993 не четно!
89476993 не четно!
*///~
```

Одна задача может вызвать next() после того, как другая задача выполнит первый инкремент currentEvenValue, но до второго инкремента (в позиции, помеченной комментарием «Опасная точка!»). При этом значение оказывается в «некорректном» состоянии. Чтобы доказать, что такое возможно, EvenChecker.test() создает группу объектов EvenChecker, которые непрерывно читают результаты EvenGenerator и проверяют их на четность. При обнаружении нечетного числа выводится сообщение об ошибке, и программа завершается.

Сбой рано или поздно произойдет, потому что задачи EvenChecker могут обратиться к информации EvenGenerator в «некорректном» состоянии. Впрочем, проблема может проявиться лишь после многих циклов отработки EvenGenerator; все зависит от особенностей операционной системы и других подробностей реализации. Чтобы ускорить наступление сбоя, попробуйте разместить вызов yield() между инкрементами. В этом и состоит одна из проблем многопоточного программирования: программа, содержащая ошибку, на первый взгляд работает вполне нормально — а все потому, что вероятность сбоя очень мала.

Также стоит учитывать, что сама операция инкремента состоит из нескольких шагов и может быть прервана планировщиком потоков в ходе выполнения — другими словами, инкремент в Java не является атомарной операцией. Даже простое приращение переменной может оказаться небезопасным, если не организовать защиту задачи.

Разрешение конфликтов доступа

Предыдущий 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 на тарелке прямо перед вами. Но, как только вы тянетесь к еде вилкой, она исчезает (как ваш поток был внезапно приостановлен, и другой поток не постеснялся «стянуть» у вас еду). Вот такую проблему нам приходится решать при написании выполняемых одновременно и использующих общие ресурсы программ. Чтобы многопоточность работала, необходим механизм, предотвращающий возможность состязания двух потоков за один ресурс (по крайней мере, во время критичных операций).

Предотвратить такое столкновение интересов несложно — 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 этого другой поток овладеет ресурсом и заблокирует его, и т. д. Если переднее сиденье машины является для детей ограниченным ресурсом, 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 один поток. Чаще всего это выполняется помещением фрагмента кода в секцию блокировки так, что одновременно пройти по этому фрагменту кода может только один поток. Поскольку такое предложение блокировки дает эффект *взаимного исключения*, этот механизм часто называют *мьютексом* (MUTual EXclusion).

Вспомните свою ванную комнату — несколько людей (потоки) могут захотеть эксклюзивно владеть ей (разделяемым ресурсом). Чтобы получить доступ в ванную, человек стучится в дверь, желая проверить, не занята ли она. Если ванная свободна, он входит в нее и запирает дверь. Любой другой поток, желающий оказаться внутри, «блокируется» в этом действии, и ему приходится ждать у двери, пока ванная не освободится.

Аналогия немного нарушается, когда дверь в ванную комнату снова открывается, и приходит время передать доступ другому потоку. 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 потоков недетерминировано. Существует гипотетическая группа блокированных потоков, толкущихся у двери, и, когда поток, который занимал «ванную», разблокирует ее и уйдет, тот поток, что окажется ближе всех к двери, «войдет» в нее. Как уже было замечено, планировщику можно давать подсказки методами `yield()` и `setPriority()`, но эти подсказки необязательно будут иметь эффект, в зависимости от вашей платформы и реализации виртуальной машины JVM.

В Java есть встроенная поддержка для предотвращения конфликтов в виде ключевого слова `synchronized`. Когда поток желает выполнить фрагмент кода,

охраняемый словом synchronized, он проверяет, доступен ли семафор, получает доступ к семафору, выполняет код и освобождает семафор.

Разделяемый ресурс чаще всего является блоком памяти, представляющим объект, но это также может быть файл, порт ввода/вывода или устройство (скажем, принтер). Для управления доступом к разделяемому ресурсу вы сначала 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 потоки не сумеют зайти *ни в какой* synchronized-метод до тех пор, пока первый поток не вернется из своего вызова.

Как известно, в окончательной версии кода поля класса обычно объявляются закрытыми (private), а доступ к их памяти осуществляется только посредством методов. Чтобы предотвратить конфликты, объявите такие методы синхронизированными (с помощью ключевого слова synchronized):

```
synchronized void f() { /* .. */ }
synchronized void g(){ /*..  */ }
```

Каждый объект содержит объект простой блокировки (также называемый *монитором*). При вызове любого синхронизированного (synchronized) метода 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 для объекта вызывается метод f(), метод g() не будет вызван до тех пор, пока метод f() не завершит свою работу и не сбросит блокировку. Таким образом, монитор совместно используется всеми синхронизированными методами определенного объекта и предотвращает использование общей памяти несколькими потоками одновременно.

Один поток 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 блокировался, его счетчик равен нулю. Когда задача захватывает объект в первый раз, счетчик увеличивается до единицы. Каждый раз, когда задача снова овладевает объектом блокировки того же объекта, счетчик увеличивается. Естественно, что все это разрешается только той задаче, которая инициировала первичную блокировку. При выходе задачи из синхронизированного метода счетчик уменьшается на единицу до тех пор, пока не делается равным нулю, после чего объект блокировки данного объекта становится доступен другим потокам.

Также существует отдельный монитор для класса (часть объекта Class), который следит за тем, чтобы статические (static) синхронизированные (synchronized) методы не использовали одновременно общие статические данные класса.

Синхронизация для примера EvenGenerator

Включив в программу EvenGenerator.java поддержку synchronized, мы можем предотвратить нежелательный доступ со стороны потоков:

```
//· concurrency/SynchronizedEvenGenerator.java
// Упрощение работы с мьютексами с использованием
// ключевого слова synchronized
// {RunByHand}

public class
SynchronizedEvenGenerator extends IntGenerator {
     private int currentEvenValue = 0;
     public synchronized int next() {
              ++currentEvenValue;
              Thread.yield(); // Ускоряем сбой
              ++currentEvenValue;
              return currentEvenValue;
     }
     public static void main(String[] args) {
              EvenChecker.test(new SynchronizedEvenGenerator());
     }
} /// ~
```

Вызов Thread.yield() между двумя инкрементами повышает вероятность переключения контекста при нахождении currentEvenValue в нечетном состоянии. Так как мьютекс позволяет выполнять критическую секцию не более чем одной задаче, сбоев не будет.

Первая задача, входящая в next(), устанавливает блокировку, а все остальные задачи, пытающиеся ее установить, блокируются до момента снятия блокировки первой задачей. В этой точке механизм планирования выбирает другую задачу, ожидающую блокировки. Таким образом, в любой момент времени только одна задача может проходить по коду, защищенному мьютексом.

Объекты Lock

Библиотека Java SE5 java.util.concurrent также содержит явный механизм управления мьютексами, определенный в java.util.concurrent.locks. Объект Lock можно явно создать в программе, установить или снять блокировку; правда, полученный код будет менее элегантным, чем при использовании встроенной формы. С другой стороны, он обладает большей гибкостью при решении некоторых типов задач. Вот как выглядит пример SynchronizedEvenGenerator.java с явным использованием объектов Lock:

```
//: concurrency/MutexEvenGenerator.java
// Предотвращение потоковых конфликтов с использованием мьютексов.
// {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(); // Ускоряем сбой
                       ++currentEvenValue;
                       return currentEvenValue;
              } finally {
```

продолжение ⇨

```
                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 следует применять идиому, показанную в примере: сразу же за вызовом lock() необходимо разместить конструкцию try-finally, при этом в секцию finally включается вызов unlock() — только так можно гарантировать снятие блокировки.

Хотя try-finally требует большего объема кода, чем ключевое слово synchronized, явное использование объектов Lock обладает своими преимуществами. При возникновении проблем с ключевым словом synchronized происходит исключение, но вы не получите возможность выполнить завершающие действия, чтобы сохранить корректное состояние системы. При работе с объектами Lock можно сделать все необходимое в секции finally.

В общем случае использование synchronized уменьшает объем кода, а также радикально снижает вероятность ошибки со стороны программиста, поэтому явные операции с объектами Lock обычно выполняются только при решении особых задач. Например, с ключевым словом synchronized нельзя попытаться получить блокировку с неудачным исходом или попытаться получить блокировку в течение некоторого промежутка времени с последующим отказом — в подобных случаях приходится использовать библиотеку concurrent:

```
//: concurrency/AttemptLocking.java
// Объекты Lock из библиотеки concurrent делают возможными
// попытки установить блокировку в течение некоторого времени
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 установления блокировки
                new Thread() {
                        { setDaemon(true); }
                        public void run() {
                                al.lock.lock();
                                System.out.println("acquired");
                        }
                } start();
                Thread.yield(); // Даем 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
*///:~
```

Класс ReentrantLock делает возможной попытку получения блокировки с последующим отказом от нее. Таким образом, если кто-то уже захватил блокировку, вы можете отказаться от своих намерений (вместо того, чтобы дожидаться ее освобождения). В методе timed() делается попытка установления блокировки, которая может завершиться неудачей через 2 секунды (обратите внимание на использование класса Java SE5 TimeUnit для определения единиц времени). В main() отдельный объект Thread создается в виде безымянного класса и устанавливает блокировку, чтобы методы untimed() и timed() могли с чем-то конкурировать.

Атомарные операции и ключевое слово volatile

В дискуссиях, посвященных механизму потоков в Java, часто 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* (переключения выполнения на другой поток). Не полагайтесь на атомарность, она ненадежна и опасна — используйте ее вместо синхронизации только в том случае, если вы являетесь экспертом в области синхронизации или, по крайней мере, можете получить помощь от такого эксперта.

Атомарные операции, упоминаемые в таких дискуссиях, включают в себя «простые операции» с примитивными типами, за исключением long и double.

Чтение и запись примитивных переменных гарантированно выполняются как атомарные (неделимые) операции. С другой стороны, JVM разрешается выполнять чтение и запись 64-разрядных величин (long и double) в виде двух раздельных 32-разрядных операций, с ненулевой вероятностью переключения контекста в ходе чтения или записи. Для достижения атомарности (при простом присваивании и возврате значений) можно определить типы long и double с модификатором volatile (учтите, что до выхода Java SE5 ключевое слово volatile не всегда работало корректно). Некоторые реализации JVM могут предоставлять более сильные гарантии, но вы не должны полагаться на платформенно-специфические возможности.

В многопроцессорных системах (которые в наши дни представлены *многоядерными процессорами*, то есть несколькими процесорами на одном чипе) *видимость* (visibility) играет гораздо более важную роль, чем в однопроцессорных системах. Изменения, вносимые одной задачей, — даже атомарные в смысле невозможности прерывания — могут оставаться невидимыми для других задач (например, если изменения временно хранятся в локальном кэше процессора). Таким образом, разные задачи будут по-разному воспринимать состояние приложения. Механизм синхронизации обеспечивает распространение видимости изменений, вносимых одной задачей в многопроцессорной системе, по всему приложению. Без синхронизации невозможно заранее предсказать, когда именно изменения станут видимыми.

Ключевое слово volatile обеспечивает видимость в рамках приложения. Если поле объявлено как volatile, это означает, что сразу же после записи в поле изменение будет отражено во всех последующих операциях чтения. Утверждение истинно даже при участии локальных кэшей — поля volatile немедленно записываются в основную память, и дальнейшее чтение происходит из основной памяти.

Если слепо следовать концепции атомарности, можно заметить, что метод getValue() в следующем примере вроде бы отвечает этому описанию:

```
//: 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)
               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
191583767
*///:~
```

Однако программа находит нечетные значения и завершается. Хотя return i и является атомарной операцией, отсутствие synchronized позволит читать значение объекта, когда он находится в нестабильном промежуточном состоянии. Вдобавок переменная i не объявлена как volatile, а это приведет к проблемам с видимостью. Оба метода, getValue() и evenIncrement(), должны быть объявлены синхронизируемыми. Только эксперты в области параллельных вычислений могут пытаться применять оптимизацию в подобных случаях.

В качестве второго примера рассмотрим кое-что еще более простое: класс, производящий серийные номера[1]. 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++; // Операция не является потоково-безопасной
    }
} ///:~
```

Представить себе класс тривиальнее SerialNumberGenerator вряд ли можно, 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 в виде одной инструкции микропроцессора. Однако 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.

Поле serialNumber объявлено как volatile потому, что каждый поток обладает локальным стеком и поддерживает в нем копии некоторых локальных переменных. Если вы объявляете переменную как volatile, то тем самым указываете компилятору не проводить оптимизацию, а это приведет к удалению чтения и записи, удерживающих поле в соответствии с локальными данными потока. Операции чтения и записи осуществляются непосредственно с памятью без кэширования. Кроме того, volatile не позволяет компилятору изменять порядок обращений с целью оптимизации. И все же присутствие volatile не влияет на тот факт, что инкремент не является атомарной операцией.

Для тестирования нам понадобится множество, которое не потребует переизбытка памяти в том случае, если обнаружение проблемы отнимет много времени. Приведенный далее класс CircularSet многократно использует память, в которой хранятся целые числа (int); предполагается, что к тому моменту, когда запись в множество начинается по новому кругу, вероятность конфликта

[1] Источником вдохновения послужила книга *Effective Java* Джошуа Блоша, издательство Addison-Wesley, 2001, с. 190.

с перезаписанными значениями минимальна. Методы add() и contains() объявлены как synchronized, чтобы избежать коллизий:

```
//: concurrency/SerialNumberChecker.java
// Кажущиеся безопасными операции с появлением потоков
// перестают быть таковыми.
// {Args: 4}
import java.util.concurrent.*;

// Reuses storage so we don't run out of memory:
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;
    // Возврат индекса к началу с записью поверх старых значений:
    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());
    // Остановиться после 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
Duplicate: 8468656
*///:~
```

В классе SerialNumberChecker содержится статическое поле CircularSet, хранящее все серийные номера, и вложенный поток Thread, который получает эти номера и удостоверяется в их уникальности. Создав несколько потоков, претендующих на серийные номера, вы обнаружите, что какой-нибудь из них довольно быстро получит уже имеющийся номер (заметьте, что на вашей машине программа может и не обнаружить конфликт, но на многопроцессорной системе она успешно их нашла). Для решения проблемы добавьте к методу nextSerialNumber() слово 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, опасно и ненадежно.

Атомарные классы

В Java SE5 появились специальные классы для выполнения атомарных операций с переменными — AtomicInteger, AtomicLong, AtomicReference и т. д. Эти классы содержат атомарную операцию условного обновления в форме

```
boolean compareAndSet(expectedValue, updateValue);
```

Эти классы предназначены для оптимизации с целью использования атомарности на машинном уровне на некоторых современных процессорах, поэтому в общем случае вам они не понадобятся. Иногда они применяются и в повседневном программировании, но только при оптимизации производительности. Например, версия AtomicityTest.java, переписанная для использования AtomicInteger, выглядит так:

```
// 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) {
```

продолжение ⇨

```
                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 вместо ключевого слова synchronized используется AtomicInteger. Так как сбой в программе не происходит, в программу включается таймер, автоматически завершающий ее через 5 секунд.

Вот как выглядит пример MutexEvenGenerator.java, переписанный для использования класса AtomicInteger:

```
//: concurrency/AtomicEvenGenerator.java
// Атомарные классы иногда используются 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());
        }
} ///.~
```

Стоит еще раз подчеркнуть, что классы Atomic проектировались для построения классов из java.util.concurrent. Используйте их в своих программах только в особых случаях и только тогда, когда вы твердо уверены, что это не создаст новых проблем. В общем случае безопаснее использовать блокировки (с ключевым словом synchronized или явным созданием объектов Lock).

Критические секции

Иногда необходимо 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 определяет

объект, блокировка которого должна использоваться для синхронизации последующего фрагмента кода:

```
synchronized(синхронизируемыйОбъект) {
     // К такому коду доступ может получить
     // одновременно только один поток
}
```

Такая конструкция иначе называется *синхронизированной блокировкой* (synchronized block); перед входом в нее необходимо получить блокировку для **syncObject**. Если блокировка уже предоставлена другому потоку, вход в последующий фрагмент кода 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 синхронизации методов. Вдобавок он демонстрирует, как незащищенный класс может «выжить» в многозадачной среде, если он управляется и защищается другим классом:

```
//: concurrency/CriticalSection.java
// Синхронизация блоков вместо целых методов. Также демонстрирует защиту
// неприспособленного к многопоточности класса другим классом
package concurrency;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;

class Pair { // Not thread-safe
     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 - обе переменные должны быть равны:
     public void checkState() {
              if(x != y)
                       throw new PairValuesNotEqualException();
     }
}

// Защита класса Pair внутри приспособленного к потокам класса:
abstract class PairManager {
```

продолжение ⇩

```
        AtomicInteger checkCounter = new AtomicInteger(0).
        protected Pair p = new Pair().
        private List<Pair> storage =
                Collections synchronizedList(new ArrayList<Pair>()).
        public synchronized Pair getPair() {
                // Создаем копию, чтобы сохранить оригинал в безопасност
                return new Pair(p getX(), p getY()).
        }
        // Предполагается, что операция занимает некоторое время
        protected void store(Pair p) {
                storage add(p).
                try {
                        TimeUnit MILLISECONDS sleep(50);
                } catch(InterruptedException ignore) {}
        }
        public abstract void increment().
}

// Синхронизация всего 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 {
  // Сравнение двух 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 + "\npm2: " + pm2);
        System exit(0);
    }
    public static void main(String[] args) {
        PairManager
            pman1 = new PairManager1(),
            pman2 = new PairManager2();
        testApproaches(pman1, pman2);
    }
} /* Output:
pm1: Pair: x: 15, y: 15 checkCounter = 272565
pm2: Pair: x: 16, y: 16 checkCounter = 3956974
*/// ~
```

Как было отмечено, класс Pair не приспособлен к работе с потоками, поскольку его инвариант (предположительно произвольный) требует равенства обоих 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, мы не можем считать, что объект Pair останется неповрежденным в многопоточной программе.

Представьте, что вы получили готовый класс Pair, который должен работать в многопоточных условиях. Класс PairManager хранит объекты Pair и управляет любым доступом к ним. Заметьте, что единственными открытыми (public)

методами являются getPair(), объявленный как synchronized, и абстрактный метод doTask(). 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 проектирования *«шаблонным методом»*. Паттерны проектирования позволяют инкапсулировать изменения в коде — здесь изменяющаяся часть представлена методом increment(). В классе PairManager1 метод increment() полностью синхронизирован, в то время как в классе PairManager2 только часть его была синхронизирована посредством синхронизируемой блокировки. 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, поэтому операция является потоково-безопасной. Следовательно, в защите он не нуждается, поэтому его вызов размещен за пределами синхронизируемого блока.

Класс PairManipulator создается для тестирования двух разновидностей PairManager: метод increment() вызывается в задаче в то время, как в другой задаче работает PairChecker. Метод main() создает два объекта PairManipulator и дает им поработать в течение некоторого времени, после чего выводятся результаты по каждому PairManipulator.

Для создания критических секций также можно воспользоваться явно созданными объектами Lock:

```
//: concurrency/ExplicitCriticalSection.java
// Использование объектов Lock для создания критических секций.
package concurrency;
import java.util.concurrent.locks.*;

// Синхронизация целого 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(),
                        pman2 = new ExplicitPairManager2(),
                CriticalSection.testApproaches(pman1, pman2);
        }
} /* Output
pm1. Pair: x: 15, y· 15 checkCounter = 174035
pm2: Pair· x· 16, y· 16 checkCounter = 2608588
*///.~
```

В программе создаются новые типы **PairManager** с явным использованием объектов **Lock**. **ExplicitPairManager2** демонстрирует создание критической секции с использованием объекта **Lock**; вызов **store()** находится вне критической секции.

Синхронизация по другим объектам

Блоку **synchronized** необходимо передать объект, который будет использоваться для синхронизации. Чаще всего наиболее естественно передавать текущий объект, для которого был вызван метод **synchronized(this)**, и именно такой подход применен в классе **PairManager2**. Таким образом, при входе в синхронизируемый блок другие синхронизированные методы объекта вызвать будет нельзя. Действие синхронизации по **this** фактически заключается в сужении области синхронизации.

Иногда вам нужно что-то иное, и в таких ситуациях вы создаете отдельный объект и выполняете синхронизацию, привлекая его. В таких случаях необходимо позаботиться о том, чтобы все операции синхронизировались по одному и тому же объекту. Следующий пример показывает, как два потока входят в объект, когда методы этого объекта синхронизированы различными блокировками:

```
//  concurrency/SyncObject.java
// Синхронизация по другому объекту.
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() {
```

продолжение ⇩

```
                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:
g()
f()
g()
f()
g()
f()
g()
f()
g()
f()
*///:~
```

Метод f() класса DualSync синхронизируется по объекту this (синхронизируя метод целиком), а метод g() использует синхронизацию посредством объекта syncObject. Таким образом, два варианта синхронизации независимы. Демонстрируется этот факт методом main(), в котором создается поток Thread с вызовом метода f(). Поток main() после этого вызывает метод g(). Из результата работы программы видно, что оба метода работают одновременно и ни один из них не блокируется соседом.

Локальная память потока

Второй механизм предотвращения конфликтов доступа к общим ресурсам основан на исключении их совместного использования. *Локальная память потока* представляет собой механизм автоматического выделения разных областей памяти для одной переменной во всех потоках, использующих объект. Следовательно, если пять потоков используют объект с переменной x, для x будет сгенерировано пять разных областей памяти. Фактически поток связывается с некоторым состоянием.

За выделение локальной памяти потоков и управление ею отвечает класс java.lang.ThreadLocal:

```
//: concurrency/ThreadLocalVariableHolder.java
// Автоматическое выделение собственной памяти каждому потоку.
```

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

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();         // Выход из всех объектов Accessor
     }
} /* Output
#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. Если вы создаете объект ThreadLocal, для обращения к содержимому объекта можно использовать только методы get() и set(). Метод get() возвращает копию объекта, ассоциированного с потоком, а set() сохраняет свой аргумент в объекте потока, возвращая ранее хранившийся объект. Их использование продемонстрировано в методах

increment() и get() класса ThreadLocalVariableHolder. Обратите внимание: методы increment() и get() не синхронизированы, потому что ThreadLocal не гарантирует отсутствия «ситуации гонки».

Взаимодействие между потоками

Итак, мы выяснили, что потоки способны конфликтовать друг с другом, и разобрались с тем, как предотвратить такие конфликты. Следующим шагом должно стать изучение возможностей взаимодействия между потоками. Ключевым моментом в этом процессе является подтверждение связи, безопасно реализуемое методами wait() и notify() класса Object. В многопоточной библиотеке Java SE5 также присутствуют объекты Condition с методами await() и signal(). Мы рассмотрим некоторые возникающие проблемы и их решения.

Методы wait() и notifyAll()

Метод wait() ожидает изменения некоторого условия, неподконтрольного для текущего метода. Довольно часто это условие изменяется в результате выполнения другой задачи. Активное ожидание, то есть проверка условия в цикле, нежелательно из-за неэффективного расходования вычислительных ресурсов. Таким образом, метод wait() обеспечивает механизм синхронизации действий между задачами.

Важно понять, что метод sleep() *не освобождает* объект блокировки. С другой стороны, метод wait() снимает блокировку с объекта, тем самым позволяя остальным потокам вызывать другие синхронизированные методы объекта во время выполнения wait(). Это очень важно, потому что обычно именно «другие» методы приводят к изменению условия и активизации приостановленной задачи.

Существует две формы метода wait(). У первой формы аргумент имеет такой же смысл, как и аргумент метода sleep(): это продолжительность интервала в миллисекундах, на который приостанавливается выполнение потока. 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, более распространенная форма вызывается без аргументов. 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 общему классу Object, а не к классу потоков Thread. Хотя это и кажется немного нелогичным — размещение чего-то, относящегося исключительно к механизму потоков, в общем базовом классе — на самом деле это решение совершенно оправдано, поскольку означенные методы манипулируют блокировками, которые являются частью любого объекта. В результате ожидание

(wait()) может использоваться в любом синхронизированном методе, независимо от того, наследует ли класс от Thread или реализует 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 вызвать метод wait() или notify() в обычном 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.

Вы можете «попросить» объект провести операции с помощью его собственного объекта блокировки. Для этого необходимо сначала захватить блокировку для данного объекта. Например, если вы хотите вызвать notifyAll() для объекта x, то должны сделать это в синхронизируемом блоке, устанавливающем блокировку для x:

```
synchronized(x) {
     x notifyAll();
}
```

Рассмотрим простой пример. В программе WaxOMatic.java задействованы два процесса: один наносит восковую пасту на автомашину (Car), а другой полирует ее. Задача полировки не может приступить к работе до того, как задача нанесения пасты завершит свою операцию, а задача нанесения пасты должна ждать завершения полировки, чтобы наложить следующий слой пасты. Оба класса, WaxOn и WaxOff, используют объект Car, который приостанавливает и возобновляет задачи в ожидании изменения условия:

```
//: concurrency/waxomatic/WaxOMatic.java
// Простейшее взаимодействие задач.
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 обработке
               notifyAll();
     }
     public synchronized void buffed() {
               waxOn = false; // Готово к нанесению очередного слоя
               notifyAll();
     }
     public synchronized void waitForWaxing()
     throws InterruptedException {
               while(waxOn == false)
                         wait();
     }
     public synchronized void waitForBuffing()
     throws InterruptedException {
               while(waxOn == true)
```

продолжение ⇨

```
                        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); // Небольшая задержка. .
                exec.shutdownNow(); // Прерывание всех задач
        }
    } /* Output:
    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
    *///:~
```

Класс Car содержит одну логическую переменную waxOn, которая описывает текущее состояние процесса полировки.

Метод waitForWaxing() проверяет флаг waxOn, и, если он равен false, вызывающая задача приостанавливается вызовом wait(). Очень важно, что это происходит в синхронизированном методе. При вызове wait() поток приостанавливается, а блокировка снимается. Последнее принципиально, потому что для безопасного изменения состояния объекта (например, для присваивания waxOn значения true, без чего приостановленная задача не сможет продолжить работу) блокировка должна быть доступна для другой задачи. В нашем примере при вызове другой задачей метода waxed(), указывающего, что пришло время что-то сделать, для задания истинного значения waxOn необходимо установить блокировку. Затем waxed() вызывает notifyAll(); задача, приостановленная вызовом wait(), активизируется. Для этого нужно сначала заново получить блокировку, освобожденную при входе в wait(). Задача не активизируется, пока блокировка не станет доступной.

Использование каналов для ввода/вывода между потоками

Часто бывает полезно организовать взаимодействие потоков посредством механизмов ввода/вывода. Библиотеки потоков могут предоставлять поддержку ввода/вывода между потоками в форме *каналов* (pipes). Последние представлены в стандартной библиотеке ввода/вывода Java классами PipedWriter (позволяет потоку записывать в канал) и PipedReader (предоставляет возможность другому потоку считывать из того же канала).

Простой пример взаимодействия двух потоков через канал:

```
//· concurrency/PipedIO.java
// Использование каналов для ввода/вывода между потоками
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");
             }
     }
```

продолжение ⇩

```
}

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 следующего символа.
                                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:
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, которые должны взаимодействовать друг с другом. В классе **Sender** создается канал **PipedWriter**, существующий как автономный объект, однако при создании канала **PipedReader** в классе **Receiver** конструктору необходимо передать ссылку на **PipedWriter**. **Sender** записывает данные в канал **Writer** и бездействует в течение случайно выбранного промежутка времени. Класс **Receiver** не содержит вызовов **sleep()** или **wait()**, но при проведении чтения методом **read()** он 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.

Взаимная блокировка

Итак, потоки способны перейти в блокированное состояние, а объекты могут обладать синхронизированными методами, которые запрещают использование

объекта до тех пор, пока не будет снята блокировка. Возможна ситуация, в которой один поток ожидает другой поток, тот, в свою очередь, ждет освобождения еще одного потока и т. д., 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 *взаимной блокировкой* (deadlock) (или «клинчем». — *Примеч. ред.*).

Если вы запускаете программу и в ней незамедлительно возникает взаимная блокировка, проблему удается немедленно отследить. По-настоящему неприятна ситуация, когда ваша программа по всем признакам работает прекрасно, но тем не менее в какой-то момент входит во взаимную блокировку. Такая опасность незаметно присутствует в программе, пока нежданно-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.

Классический пример взаимной блокировки, предложенный Эдгаром Дейкстрой — *задача об обедающих философах*. В стандартной формулировке говорится о пяти философах, но, как будет показано далее, допустимо любое количество. Часть времени философы проводят размышляя, часть времени проводят за едой. Когда они размышляют, то не нуждаются в общих ресурсах, но во время обеда они сидят за круглым столом с ограниченным количеством столовых приборов. В описании оригинальной задачи философы пользуются вилками, и, чтобы набрать спагетти из миски в центре стола, им требуются две вилки. Наверное, задача будет выглядеть более логично, если заменить вилки палочками для еды — очевидно, что каждому философу понадобятся две палочки.

Философы, как это часто бывает, очень бедны, и они смогли позволить себе приобрести лишь пять палочек (или в более общем виде — количество палочек совпадает с количеством философов). Последние разложены кругом по столу, между философами. Когда философу захочется поесть, 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 ее освобождения:

```
//: concurrency/Chopstick.java
// Палочки для обедающих философов.

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();
     }
} ///:~
```

Два философа (Philosopher) ни при каких условиях не смогут успешно взять (take()) одну и ту же палочку (Chopstick) одновременно. Если один философ уже взял палочку, другому философу придется подождать (wait()), пока она не будет освобождена текущим пользователем (drop()).

Когда задача Philosopher вызывает take(), она ожидает, пока флаг taken не перейдет в состояние false (то есть пока палочка не будет освобождена тем философом, который держит ее в данный момент). Далее задача устанавливает флаг taken равным true, показывая тем самым, что палочка занята. Завершив работу с Chopstick, Philosopher вызывает drop(), чтобы изменить флаг и оповестить (notifyAll()) всех остальных философов, ожидающих освобождения палочки:

```
// 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 + " " + "думает");
        pause();
        // Философ проголодался
        print(this + " " + "берет правую");
        right.take();
        print(this + " " + "берет левую");
        left.take();
        print(this + " " + "ест");
        pause();
        right.drop();
        left.drop();
      }
    } catch(InterruptedException e) {
      print(this + " " + "выход через прерывание");
    }
  }
```

```
    public String toString() { return "Философ " + id; }
} /// ~
```

В методе Philosopher.run() все философы непрерывно переходят от размышлений к еде, и наоборот. Метод pause() делает паузу случайной продолжительности, если значение ponderFactor отлично от нуля. Итак, Philosopher думает в течение случайного промежутка времени, затем пытается захватить левую и правую палочки вызовами take(), ест в течение случайного промежутка времени, а затем все повторяется.

В следующей версии программы возникает взаимная блокировка:

```
//  concurrency/DeadlockingDiningPhilosophers.java
// Демонстрация скрытой возможности взаимной блокировки
// {Args  0 5 timeout}
import java.util.concurrent.*;

public class DeadlockingDiningPhilosophers {
    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++)
            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("Нажмите 'Enter', чтобы завершить работу");
            System.in.read();
        }
        exec.shutdownNow();
    }
} ///:~
```

Если философы почти не тратят время на размышления, они будут постоянно конкурировать за палочки при попытках поесть, и взаимные блокировки возникают гораздо чаще.

Первый аргумент командной строки изменяет значение ponder, влияющее на продолжительность размышлений. Если философов очень много или они проводят большую часть времени в размышлениях, взаимная блокировка может и не возникнуть, хотя ее теоретическая вероятность отлична от нуля. С нулевым аргументом взаимная блокировка наступает намного быстрее.

Объектам Chopstick не нужны внутренние идентификаторы; они идентифицируются по своей позиции в массиве sticks. Каждому конструктору Philosopher передаются ссылки на правую и левую палочки Chopstick. Последнему Philosopher в качестве правой палочки передается нулевой объект Chopstick; круг замыкается. Теперь может возникнуть ситуация, когда все философы одновременно

попытаются есть, и каждый из них будет ожидать, пока сосед положит свою палочку. В программе наступает взаимная блокировка.

Если философы тратят на размышления больше времени, чем на еду, вероятность взаимной блокировки значительно снижается. Даже может возникнуть иллюзия, что программа свободна от блокировок (при ненулевом значении ponder или большом количестве объектов Philosopher), хотя на самом деле это не так. Именно этим и интересен настоящий пример: программа вроде бы ведет себя верно, тогда как на самом деле возможна взаимная блокировка.

Для решения проблемы необходимо осознавать, что тупик имеет место при стечении следующих четырех обстоятельств:

1. Взаимное исключение: по крайней мере один ресурс, используемый потоками, не должен быть совместно используемым. В нашем случае одной палочкой для еды не могут одновременно есть два философа.
2. По крайней мере одна задача должна удерживать ресурс и ожидать выделения ресурса, в настоящее время удерживаемого другой задачей. То есть для возникновения тупика философ должен сохранять при себе одну палочку и ожидать другую.
3. Ресурс нельзя принудительно отбирать у задачи. 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 т. д., пока один из процессов не будет ожидать ресурса, занятого первым процессом, что и приведет к порочному кругу. В нашем примере круговое ожидание происходит потому, что каждый философ пытается сначала получить правую палочку, а потом левую.

Так как взаимная блокировка возникает лишь при соблюдении всех перечисленных условий, для упреждения тупика достаточно нарушить всего лишь одно из них. В нашей программе проще всего нарушить четвертое условие: оно выполняется, поскольку каждый философ старается брать палочки в определенном порядке — сначала левую, потом правую. Из-за этого может возникнуть ситуация, когда каждый из них держит свою левую палочку и ждет освобождения правой, что и приводит к циклическому ожиданию. Если инициализировать последнего философа так, чтобы он сначала пытался взять левую палочку, а потом правую, взаимная блокировка станет невозможна. Это всего лишь одно решение проблемы, но вы можете предотвратить ее, нарушив одно из оставшихся условий (за подробностями обращайтесь к специализированной литературе по многозадачному программированию):

```
//. concurrency/FixedDiningPhilosophers.java
// Обедающие философы без взаимной блокировки.
// {Args: 5 5 timeout}
import java.util.concurrent.*;

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();
    }
} ///:~
```

Проследив за тем, чтобы последний философ брал и откладывал левую палочку раньше правой, мы устраняем взаимную блокировку.

В языке Java *нет встроенных средств предупреждения взаимных блокировок*; все зависит только от вас и аккуратности вашего кода. Вряд ли эти слова утешат того, кому придется отлаживать программу с взаимной блокировкой.

Новые библиотечные компоненты

В библиотеке java.util.concurrent из Java SE5 появился целый ряд новых классов, предназначенных для решения проблем многозадачности. Научившись пользоваться ими, вы сможете создавать более простые и надежные многозадачные программы.

В этом разделе приведено немало примеров использования различных компонентов. Другие, относительно редко встречающиеся компоненты, здесь не рассматриваются.

Так как компоненты предназначены для решения разных проблем, простого способа их упорядочения не существует, поэтому мы начнем с более простых примеров и постепенно перейдем к более сложным.

CountDownLatch

Класс синхронизирует задачи, заставляя их ожидать завершения группы операций, выполняемых другими задачами.

Объекту CountDownLatch присваивается начальное значение счетчика, а все задачи, вызвавшие await() для этого объекта, блокируются до момента обнуления счетчика. Другие задачи могут уменьшать счетчик, вызывая метод countDown() для объекта (обычно это делается тогда, когда задача завершает свою работу). Класс CountDownLatch рассчитан на «одноразовое» применение; счетчик не может возвращаться к прежнему состоянию. Если вам нужна версия с возможностью сброса счетчика, воспользуйтесь классом CyclicBarrier.

Задачи, вызывающие countDown(), не блокируются на время вызова. Только вызов await() блокируется до момента обнуления счетчика.

Типичный способ применения — разделение задачи на n независимых подзадач и создание объекта CountDownLatch с начальным значением n. При завершении каждая подзадача вызывает countDown() для объекта синхронизации. Потоки, ожидающие решения общей задачи, блокируются вызовом await(). Описанная методика продемонстрирована в следующем примере:

```
//  concurrency/CountDownLatchDemo.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Часть основной 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) {
               // Приемлемый вариант выхода
          }
     }
     public void doWork() throws InterruptedException {
          TimeUnit.MILLISECONDS.sleep(rand.nextInt(2000));
          print(this + "завершается");
     }
     public String toString() {
          return String.format("%1$-3d ", id);
     }
}

// Ожидание по объекту 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("Барьер пройден для " + 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 подзадачи совместно используют 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("Запущены все задачи");
                exec.shutdown(); // Выход по завершению всех задач
        }
} /// ~
```

TaskPortion некоторое время ожидает, имитируя выполнение части задачи, а класс WaitingTask представляет некую часть системы, которая обязана дождаться завершения всех подзадач. Все задачи используют один и тот же объект CountDownLatch, определяемый в main().

CyclicBarrier

Класс CyclicBarrier используется при создании группы параллельно выполняемых задач, завершения которых необходимо дождаться до перехода к следующей фазе. Все параллельные задачи «приостанавливаются» у барьера, чтобы сделать возможным их согласованное продвижение вперед. Класс очень похож на CountDownLatch, за одним важным исключением: CountDownLatch является «одноразовым», а CyclicBarrier может использоваться снова и снова.

Имитации привлекали меня с первых дней работы с компьютерами, и параллельные вычисления играют в них ключевую роль. Даже самая первая программа, которую я написал на BASIC, имитировала скачки на ипподроме. Вот как выглядит объектно-ориентированная, многопоточная версия этой программы с использованием CyclicBarrier:

```
//: concurrency/HorseRace.java
// Using CyclicBarriers
import java.util concurrent *;
import java.util *;
import static net.mindview util Print.*;
```

продолжение ⇩

```
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); // Produces 0, 1 or 2
                                }
                                barrier.await();
                       }
              } catch(InterruptedException e) {
                       // Приемлемый вариант выхода
              } catch(BrokenBarrierException e) {
                       // Исключение, которое нас интересует
                       throw new RuntimeException(e);
              }
     }
     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("="); // Забор на беговой дорожке
                                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);
                }
        } ///:~
```

Для объекта CyclicBarrier можно задать «барьерное действие» — объект Runnable, автоматически запускаемый при обнулении счетчика (еще одно отличие CyclicBarrier от CountdownLatch). В нашем примере барьерное действие определяется в виде безымянного класса, передаваемого конструктору CyclicBarrier.

Я попытался сделать так, чтобы каждый объект лошади отображал себя, но порядок отображения зависел от диспетчера задач. Благодаря CyclicBarrier каждая лошадь делает то, что ей необходимо для продвижения вперед, а затем ожидает у барьера перемещения всех остальных лошадей. Когда все лошади переместятся, CyclicBarrier автоматически вызывает «барьерную» задачу Runnable, чтобы отобразить всех лошадей по порядку вместе с барьером. Как только все задачи пройдут барьер, последний автоматически становится готовым для следующего захода.

DelayQueue

Класс представляет неограниченную блокирующую очередь объектов, реализующих интерфейс Delayed. Объект может быть извлечен из очереди только после истечения задержки. Очередь сортируется таким образом, что объект в начале очереди обладает наибольшим сроком истечения задержки. Если задержка ни у одного объекта не истекла, начального элемента нет, и вызов poll() возвращает null (из-за этого в очередь не могут помещаться элементы null).

В следующем примере объекты, реализующие Delayed, сами являются задачами, а DelayedTaskContainer берет задачу с наибольшей просроченной задержкой и запускает ее. Таким образом, DelayQueue является разновидностью 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);
    }
    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() { printnb(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) {
                            printnb(pt.summary() + " ");
                    }
                    print();
                    print(this + " вызывает 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(); // Выполнение задачи в текущем потоке
                } catch(InterruptedException e) {
                        // Приемлемый вариант выхода
                }
                print("Завершается DelayedTaskConsumer");
        }
}

public class DelayQueueDemo {
        public static void main(String[] args) {
                Random rand = new Random(47),
                ExecutorService exec = Executors newCachedThreadPool();
                DelayQueue<@060>DelayedTask> queue =
                        new DelayQueue<DelayedTask>();
                // Очередь заполняется задачами со случайной задержкой·
                for(int i = 0; i < 20; i++)
                        queue put(new DelayedTask(rand.nextInt(5000)));
                // Назначение точки остановки
                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 вызывает shutdownNow()
Завершается DelayedTaskConsumer
*///.~
```

DelayedTask содержит контейнер **List<DelayedTask>** с именем **sequence**, в котором сохраняется порядок создания задач, и мы видим, что сортировка действительно выполняется.

Интерфейс **Delayed** содержит единственный метод **getDelay()**, который сообщает, сколько времени осталось до истечения задержки или как давно задержка истекла. Метод заставляет нас использовать класс **TimeUnit**, потому что его аргумент относится именно к этому типу. Впрочем, этот класс очень удобен, поскольку он позволяет легко преобразовывать единицы без каких-либо вычислений. Например, значение **delta** хранится в миллисекундах, а метод Java SE5 **System.nanoTime()** выдает значение в наносекундах. Чтобы преобразовать значение **delta**, достаточно указать исходные и итоговые единицы:

```
NANOSECONDS.convert(delta, MILLISECONDS);
```

В **getDelay()** желаемые единицы передаются в аргументе **unit**. Аргумент используется для преобразования времени задержки во временные единицы, используемые вызывающей стороной.

Для выполнения сортировки интерфейс **Delayed** также наследует интерфейс **Comparable**, поэтому необходимо реализовать метод **compareTo()** для выполнения

осмысленных сравнений. Методы toString() и summary() обеспечивают форматирование вывода.

Из выходных данных видно, что порядок создания задач не влияет на порядок их выполнения — вместо этого задачи, как и предполагалось, выполняются в порядке следования задержек.

PriorityBlockingQueue

Фактически класс PriorityBlockingQueue представляет приоритетную очередь с блокирующими операциями выборки. В следующем примере объектами в приоритетной очереди являются задачи, покидающие очередь в порядке приоритетов. Для определения этого порядка в класс PrioritizedTask включается поле priority:

```
//: 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) {
                       // Приемлемый вариант 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); // Минимальный приоритет в этой 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 без блокировки.
            // Быстрое заполнение случайными приоритетами:
            for(int i = 0; i < 20; i++) {
                    queue.add(new PrioritizedTask(rand.nextInt(10)));
                    Thread.yield();
            }
            // Добавление высокоприоритетных задач:
            try {
                    for(int i = 0; i < 10; i++) {
                            TimeUnit.MILLISECONDS.sleep(250);
                            queue.add(new PrioritizedTask(10));
                    }
                    // Добавление заданий, начиная с наименьших приоритетов:
                    for(int i = 0; i < 10; i++)
                            queue.add(new PrioritizedTask(i));
                    // Предохранитель для остановки всех задач:
                    queue.add(new PrioritizedTask.EndSentinel(exec));
            } catch(InterruptedException e) {
                    // Приемлемый вариант выхода
            }
            print("Завершение PrioritizedTaskProducer");
    }
}

class PrioritizedTaskConsumer implements Runnable {
    private PriorityBlockingQueue<Runnable> q;
    public PrioritizedTaskConsumer(
            PriorityBlockingQueue<Runnable> q) {
            this.q = q;
    }
    public void run() {
            try {
```

продолжение ⇩

```
                while(!Thread.interrupted())
                        // Использование текущего потока для запуска задачи
                        q.take().run();
                } catch(InterruptedException e) {
                        // Приемлемый вариант выхода
                }
                print("Завершение 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));
        }
} ///:~
```

Как и в предыдущем примере, последовательность создания объектов PrioritizedTask сохраняется в контейнере List sequence для сравнения с фактическим порядком выполнения. Метод run() делает небольшую паузу, а затем выводит информацию об объекте, а предохранитель EndSentinel выполняет те же функции, что и прежде.

PrioritizedTaskProducer и PrioritizedTaskConsumer связываются друг с другом через PriorityBlockingQueue. Так как сам блокирующий характер очереди обеспечивает всю необходимую синхронизацию, явная синхронизация не нужна — при чтении вам не нужно думать о том, содержит ли очередь элементы, потому что при отсутствии элементов очередь просто заблокирует читающую сторону.

Управление оранжереей на базе ScheduledExecutor

В главе 10 была представлена система управления гипотетической оранжереей, которая включала (отключала) различные устройства и регулировала их работу. Происходящее можно преобразовать в контекст многозадачности: каждое событие оранжереи представляет собой задачу, запускаемую в заранее заданное время. Класс ScheduledThreadPoolExecutor предоставляет именно тот сервис, который необходим для решения задачи. Используя методы schedule() (однократный запуск задачи) или scheduleAtFixedRate() (повторение задачи с постоянным промежутком), мы создаем объекты Runnable, которые должны запуститься в положенное время. Сравните это решение с тем, что приведено в главе 10, и посмотрите, насколько оно упрощается благодаря готовой функциональности ScheduledThreadPoolExecutor:

```
//: concurrency/GreenhouseScheduler.java
// Новая реализация 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() {
                       // Сюда помещается аппаратный вызов, выполняющий
                       // физическое включение света
                       System out.println("Свет включен");
                       light = true;
              }
     }
     class LightOff implements Runnable {
              public void run() {
                       // Сюда помещается аппаратный вызов, выполняющий
                       // физическое выключение света.
                       System.out.println("Свет выключен"),
                       light = false,
              }
     }
     class WaterOn implements Runnable {
              public void run() {
                       // Здесь размещается код включения
                       // системы полива.
                       System out println("Полив включен");
                       water = true;
              }
     }
     class WaterOff implements Runnable {
              public void run() {
                       // Здесь размещается код выключения
                       // системы полива
                       System out.println("Полив выключен");
                       water = false,
              }
     }
     class ThermostatNight implements Runnable {
              public void run() {
                       // Здесь размещается код управления оборудованием
                       System.out.println("Включение ночного режима");
                       setThermostat("Ночь"),
              }
```

продолжение ⇩

```
}
class ThermostatDay implements Runnable {
        public void run() {
                // Здесь размещается код управления оборудованием
                System out println("Включение дневного режима"),
                setThermostat("День"),
        }
}
class Bell implements Runnable {
        public void run() { System.out.println("Бам!"), }
}
class Terminate implements Runnable {
        public void run() {
                System out println("Завершение");
                scheduler shutdownNow();
                // Для выполнения этой операции необходимо запустить
                // отдельную задачу, так как планировщик был отключен·
                new Thread() {
                        public void run() {
                                for(DataPoint d · data)
                                        System.out.println(d);
                        }
                }.start();
        }
}
// Новая возможность: коллекция 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(
                                " температура: %1$.1f влажность: %2$.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("Сбор данных");
                synchronized(GreenhouseScheduler.this) {
```

```
                                lastTime.set(Calendar.MINUTE,
                                        lastTime.get(Calendar.MINUTE) + 30);
                                // С вероятностью 1/5 происходит смена направления:
                                if(rand.nextInt(5) == 4)
                                        tempDirection = -tempDirection;
                                // Сохранить предыдущее значение:
                                lastTemp = lastTemp +
                                        tempDirection * (1.0f + rand.nextFloat());
                                if(rand.nextInt(5) == 4)
                                        humidityDirection = -humidityDirection;
                                lastHumidity = lastHumidity +
                                        humidityDirection * rand.nextFloat();
                                // Объект Calendar необходимо клонировать, иначе
                                // все DataPoint будут содержать ссылки
                                // на одно и то же lastTime.
                                // Для базового объекта - такого, как Calendar -
                                // вызова clone() вполне достаточно.
                                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);
        }
} ///:~
```

В этой версии, помимо реорганизации кода, добавляется новая возможность: сбор данных о температуре и влажности в оранжерее. Объект `DataPoint` содержит и выводит одну точку данных, а запланированная задача `CollectData` генерирует данные имитации и включает их в `List<DataPoint>` при каждом запуске.

Обратите внимание на ключевые слова `volatile` и `synchronized`; благодаря им задачи не мешают работе друг друга. Все методы контейнера `List` с элементами `DataPoint` синхронизируются с использованием метода `synchronizedList()` библиотеки `java.util.Collections` при создании `List`.

Семафоры

При обычной блокировке доступ к ресурсу в любой момент времени разрешается только одной задаче. *Семафор со счетчиком* позволяет n задачам одновременно обращаться к ресурсу. Можно считать, что семафор «выдает разрешения» на использование ресурса, хотя никаких реальных объектов разрешений в этой схеме нет.

В качестве примера рассмотрим концепцию *пула объектов*: объекты, входящие в пул, «выдаются» для использования, а затем снова возвращаются в пул после того, как пользователь закончит работу с ними. Эта функциональность инкапсулируется в параметризованном классе:

```
//  concurrency/Pool java
// Использование Semaphore в Pool ограничивает количество
// 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>();
     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 наличие конструктора 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 предотвращает переход в эту точку
     }
     private synchronized boolean releaseItem(T item) {
              int index = items indexOf(item),
              if(index == -1) return false; // Отсутствует в списке
              if(checkedOut[index]) {
                       checkedOut[index] = false,
                       return true,
              }
              return false; // Не был освобожден
     }
} ///·~
```

В этой упрощенной форме конструктор использует newInstance() для заполнения пула объектами. Если вам понадобится новый объект, вызовите checkOut(); завершив работу с объектом, передайте его checkIn().

Логический массив checkedOut отслеживает выданные объекты. Для управления его содержимым используются методы getItem() и releaseItem(). В свою очередь, эти методы защищены семафором available, поэтому в checkOut() семафор available блокирует дальнейшее выполнение при отсутствии семафорных разрешений (то есть при отсутствии объектов в пуле). Метод checkIn() проверяет действительность возвращаемого объекта, и, если объект действителен, разрешение возвращается семафору.

Для примера мы воспользуемся классом Fat. Создание объектов этого класса является высокозатратной операцией, а на выполнение конструктора уходит много времени:

```
//: concurrency/Fat.java
// Объекты, создание которых занимает много времени

public class Fat {
     private volatile double d; // Предотвращает оптимизацию
     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; }
} ///:~
```

Мы создадим пул объектов Fat, чтобы свести к минимуму затраты на выполнение конструктора. Для тестирования класса Pool будет создана задача, которая забирает объекты Fat для использования, удерживает их в течение некоторого времени, а затем 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);
```

продолжение ⇨

```
                        TimeUnit.SECONDS.sleep(1);
                        print(this + "checking in " + item);
                        pool.checkIn(item);
                } catch(InterruptedException e) {
                        // Приемлемый 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>();
                for(int i = 0; i < SIZE; i++) {
                        Fat f = pool.checkOut();
                        printnb(i + "  main() thread checked out ");
                        f.operation();
                        list.add(f);
                }
                Future<?> blocked = exec.submit(new Runnable() {
                        public void run() {
                                try {
                                        // Семафор предотвращает лишний вызов checkOut,
                                        // поэтому следующий вызов блокируется:
                                        pool.checkOut();
                                } catch(InterruptedException e) {
                                        print("checkOut() Interrupted");
                                }
                        }
                });
                TimeUnit.SECONDS.sleep(2);
                blocked.cancel(true); // Выход из заблокированного вызова
                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();
        }
} ///:~
```

В коде main() создается объект Pool для хранения объектов Fat, после чего группа задач CheckoutTask начинает использовать Pool. Далее поток main() начинает выдавать объекты Fat, *не возвращая их обратно*. После того как все объекты пула будут выданы, семафор запрещает дальнейшие выдачи. Метод run() блокируется, и через две секунды вызывается метод cancel(). Лишние возвраты Pool игнорирует.

Exchanger

Класс **Exchanger** представляет собой «барьер», который меняет местами объекты двух задач. На подходе к барьеру задачи имеют один объект, а на выходе — объект, ранее удерживавшийся другой задачей. Объекты **Exchanger** обычно используются в тех ситуациях, когда одна задача создает высокозатратные объекты, а другая задача эти объекты потребляет.

Чтобы опробовать на практике класс **Exchanger**, мы создадим задачу-поставщика и задачу-потребителя, которые благодаря параметризации и генераторам могут работать с объектами любого типа. Затем эти параметризованные задачи будут применены к классу **Fat**. **ExchangerProducer** и **ExchangerConsumer** меняют местами **List<T>**; при вызове метода **Exchanger.exchange()** вызов блокируется до тех пор, пока парная задача не вызовет свой метод **exchange()**, после чего оба метода **exchange()** завершаются, а контейнеры **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());
                                // Заполненный контейнер заменяется пустым:
                                holder = exchanger exchange(holder);
                       }
              } catch(InterruptedException e) {
                       // Приемлемый 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 {
```

продолжение

```
                        while(!Thread interrupted()) {
                                holder = exchanger.exchange(holder);
                                for(T x : holder) {
                                        value = x; // Выборка значения
                                        holder remove(x); // Нормально для
CopyOnWriteArrayList
                                }
                        }
                } catch(InterruptedException e) {
                        // Приемлемый способ завершения
                }
                System out.println("Итоговое значение: " + value);
        }
}

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:
Итоговое значение: Fat id: 29999
*///:~
```

В методе **main()** для обеих задач создается один объект **Exchanger**, а для перестановки создаются два контейнера **CopyOnWriteArrayList**. Эта разновидность **List** нормально переносит вызов метода **remove()** при перемещении по списку, не выдавая исключения **ConcurrentModificationException**. **ExchangerProducer** заполняет список, а затем меняет местами заполненный список с пустым, передаваемым от **ExchangerConsumer**. Благодаря **Exchanger** заполнение списка происходит одновременно с использованием уже заполненного списка.

Моделирование

Одна из самых интересных областей применения параллельных вычислений — всевозможные имитации и моделирование. Каждый компонент модели оформляется в виде отдельной задачи, что значительно упрощает его программирование.

Примеры HorseRace.java и GreenhouseScheduler.java, приведенные ранее, тоже можно считать своего рода имитаторами.

Модель кассира

В этой классической модели объекты появляются случайным образом и обслуживаются за случайное время ограниченным количеством серверов. Моделирование позволяет определить идеальное количество серверов. Продолжительность обслуживания в следующей модели зависит от клиента и определяется случайным образом. Вдобавок мы не знаем, сколько новых клиентов будет прибывать за каждый период времени, поэтому эта величина тоже определяется случайным образом.

```
//. concurrency/BankTellerSimulation.java
// Пример использования очередей и многопоточного программирования.
// {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 информацию о своем состоянии:
class CustomerLine extends ArrayBlockingQueue<Customer> {
     public CustomerLine(int maxLineSize) {
              super(maxLineSize).
     }
     public String toString() {
              if(this size() == 0)
                       return "[Пусто]";
              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));
```

продолжение ⇩

```
                        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;
        private final int id = counter++;
        // Счетчик клиентов, обслуженных за текущую 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 + "прерван");
                }
                System out.println(this + "завершается");
        }
        public synchronized void doSomethingElse() {
                customersServed = 0;
                servingCustomerLine = false;
        }
        public synchronized void serveCustomerLine() {
                assert !servingCustomerLine:"уже обслуживает: " + this;
                servingCustomerLine = true;
                notifyAll();
        }
        public String toString() { return "Кассир " + id + " "; }
        public String shortString() { return "К" + 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;
        // Начинаем с одного кассира:
        Teller teller = new Teller(customers);
        exec.execute(teller);
        workingTellers.add(teller);
}
public void adjustTellerNumber() {
        // Фактически это система управления. Регулировка числовых
        // параметров позволяет выявить проблемы стабильности
        // в механизме управления.
        // Если очередь слишком длинна, добавить другого кассира:
        if(customers.size() / workingTellers.size() > 2) {
                // Если кассиры отдыхают или заняты
                // другими делами, вернуть одного из них:
                if(tellersDoingOtherThings.size() > 0) {
                        Teller teller = tellersDoingOtherThings.remove();
                        teller.serveCustomerLine();
                        workingTellers.offer(teller);
                        return;
                }
                // Иначе создаем (нанимаем) нового кассира
                Teller teller = new Teller(customers);
                exec.execute(teller);
                workingTellers.add(teller);
                return;
        }
        // Если очередь достаточно коротка, освободить 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();
}
// Поручаем кассиру другую работу или отправляем его отдыхать:
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("}");
                }
```

продолжение ⇩

```
                } catch(InterruptedException e) {
                        System.out.println(this + "прерван");
                }
                System.out println(this + "завершается");
        }
        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 длинна, клиенты уходят:
                CustomerLine customers =
                        new CustomerLine(MAX_LINE_SIZE);
                exec.execute(new CustomerGenerator(customers));
                // TellerManager добавляет и убирает кассиров
                // по мере 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:
[429][200][207] { K0 K1 }
[861][258][140][322] { K0 K1 }
[575][342][804][826][896][984] { K0 K1 K2 }
[984][810][141][12][689][992][976][368][395][354] { K0 K1 K2 K3 }
Teller 2 прерван
Teller 2 завершается
Teller 1 прерван
Teller 1 завершается
TellerManager прерван
TellerManager завершается
Teller 3 прерван
Teller 3 завершаетсяч
Teller 0 прерван
Teller 0 завершается
CustomerGenerator прерван
CustomerGenerator завершается
*///:~
```

Объекты Customer очень просты; они содержат только поле данных final int. Так как эти объекты никогда не изменяют своего состояния, они являются *объектами, доступными только для чтения*, и поэтому требуют синхронизации или использования volatile. Вдобавок каждая задача Teller удаляет из очереди ввода только один объект Customer и работает с ним до завершения, поэтому задачи все равно будут работать с Customer последовательно.

Класс CustomerLine представляет собой общую очередь, в которой клиенты ожидают обслуживания. Он реализован в виде очереди ArrayBlockingQueue с методом toString(), который выводит результаты в желаемом формате.

Генератор CustomerGenerator присоединяется к CustomerLine и ставит объекты Customer в очередь со случайными интервалами.

Teller извлекает клиентов Customer из CustomerLine и обрабатывает их последовательно, подсчитывая количество клиентов, обслуженных за текущую смену. Если клиентов не хватает, его можно перевести на другую работу (doSomethingElse()), а при появлении большого количества клиентов — снова вернуть на обслуживание очереди методом serveCustomerLine(). Чтобы приказать следующему кассиру вернуться к очереди, метод compareTo() проверяет количество обслуженных клиентов, чтобы приоритетная очередь автоматически ставила в начало кассира, работавшего меньше других.

Вся основная деятельность выполняется в TellerManager. Этот класс следит за всеми кассирами и за тем, что происходит с клиентами. Одна из интересных особенностей данной имитации заключается в том, что она пытается подобрать оптимальное количество кассиров для заданного потока покупателей. Пример встречается в методе adjustTellerNumber() — управляющей системе для надежной, стабильной регулировки количества кассиров. У всех управляющих систем в той или иной мере присутствуют проблемы со стабильностью; слишком быстрая реакция на изменения снижает стабильность, а слишком медленная переводит систему в одно из крайних состояний.

Резюме

В этой главе я постарался изложить основы многопоточного программирования с использованием потоков Java. Прочитав ее, читатель должен понять следующее:

1. Программу можно разделить на несколько независимых задач.
2. Необходимо заранее предусмотреть всевозможные проблемы, возникающие при завершении задач.
3. Задачи, работающие с общими ресурсами, могут мешать друг другу. Основным средством предотвращения конфликтов является блокировка.
4. В неаккуратно спроектированных многозадачных системах возможны взаимные блокировки.

Очень важно понимать, когда рационально использовать параллельное выполнение, а когда этого делать не стоит. Основные причины для его использования:

- управление несколькими подзадачами, одновременное выполнение которых позволяет эффективнее распоряжаться ресурсами компьютера (включая возможность незаметного распределения этих задач по нескольким процессорам);
- улучшенная организация кода;
- удобство для пользователя.

Классический пример распределения ресурсов — использование процессора во время ожидания завершения операций ввода/вывода. Классический пример чуткого пользовательского интерфейса — отслеживание нажатий кнопки «Прервать» во время продолжительного процесса загрузки.

Дополнительным преимуществом потоков является то, что они заменяют «тяжелое» переключение контекста процессов (порядка 1 000 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 блокированных ресурсов.
2. Дополнительная нагрузка на процессор для управления потоками.
3. Совершенно ненужная сложность, являющаяся следствием неудачных решений при проектировании программы.
4. Аномальные ситуации: взаимные блокировки, конфликты доступа, гонки и т. д.
5. Непоследовательное поведение на различных платформах. Например, при разработке некоторых примеров для данной книги я обнаружил ситуации гонки, быстро проявлявшиеся на некоторых компьютерах, но незаметные на других.

Пожалуй, основные трудности с потоками возникают тогда, когда несколько потоков одновременно пытаются использовать один и тот же ресурс — например, память объекта, и вы должны сделать так, чтобы этого ни в коем случае не произошло. Для этого нужно разумно использовать ключевое слово `synchronized` — полезный инструмент языка, который тем не менее необходимо хорошо понимать (без этого в программе может незаметно возникнуть опасность взаимной блокировки).

Вдобавок многозадачное 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 требуется небольшая группа потоков, это ограничение не актуально, но при разработке больших программ оно может создать затруднения.

Алфавитный указатель

A

Arrays, класс, 465

C

CGI, 41
CountDownLatch, класс, 607
CyclicBarrier, класс, 609

F

foreach, синтаксис, 105

H

HTML, 41

J

Java, 48

N

new, ключевое слово, 49

P

Python, 42

S

Simula, 20
SmallTalk, 19
static, ключевое слово, 60

U

UML, 24

А

автоматические переменные, 36
ассоциативный массив, 285
атомарные операции, 585

Б

базовые классы, 25
базовый класс, 172, 201
    инициализация, 174
    конструктор, 210
безымянный внутренний класс, 251, 485
булева алгебра, 84
буферы nio, 510
быстрая разработка приложений (RAD), 376

В

ввод/вывод, 489
    библиотека сжатия, 531
    и многозадачность, 601
    интернационализация, 495
    канал (pipe), 490
    класс File, 484
    классы библиотеки, 483
    переименование файлов, 489
    перенаправление стандартного ввода/вывода, 509
    поток, 489
    свойства файлов, 487
    символы Юникода, 495
    создание каталогов, 487
    список каталогов, 484
    стандартные потоки, 507
    типичное использование, 498
    управление сериализацией, 540
взаимная блокировка, 602
взаимное исключение, mutex, 581
взаимозаменяемость, 19
взаимосвязь
    «имеет», 185
    «является», 185
внутренний класс, 245
    безымянный внутренний класс, 485
    в методах и областях действия, 251
    вложение в область действия, 252
    замыкание, 264

внутренний класс (*продолжение*)
наследование от внутренних классов, 272
обратный вызов, 264
обращение к объекту внешнего класса, 248
переопределение, 273
права на доступ, 246
системы управления, 266
скрытая ссылка на объект объемлющего класса, 248
статические внутренние классы, 258
возобновление, 314
восходящее преобразование, 186, 199
и интерфейс, 227

Г

генераторы, 466
графический интерфейс пользователя, 266

Д

деструктор, 179
динамический вызов instanceof, 369
динамическое определение типов (RTTI), 219
неверное использование, 394
объект Class, 354
пример с фигурами, 352
рефлексия, 376
динамическое связывание, 198, 202
доклеты, 65
долговременное хранение
и предпочтения, 553
объектов, 548
доступ
в пределах пакета, 160
модификатор private, 152

З

завершающие действия, 132
загрузка
классов, 196
файлов .class, 156
замыкание, 264
запуск программы на Java, 63

И

инициализация, 210
Static-инициализация, 196
базового класса, 174
и загрузка классов, 195
и наследование, 195
инициализация для экземпляра, 145
инициализация с помощью конструктора, 116
инициализация членов класса в точке определения, 138
инициализация экземпляра, 255
отложенная, 171
инициализация (*продолжение*)
порядок инициализации, 140, 216
членов класса, 171
инкапсуляция, 164
инструмент
JAR, 154, 157, 534
интерфейс
Externalizable, 541
альтернативный подход, 545
FilenameFilter, 484
Serializable, 536, 541, 544, 545, 552
вложенные интерфейсы, 239
и наследование, 233
инициализация полей интерфейса, 238
интерфейс базового класса, 205
столкновения имен, 235
интерфейс объекта, 21
исключение
ClassCastException, 362
ClassNotFoundException, 540
IllegalMonitorStateException, 599
IOException, 507, 508
NullPointerException, 329
RuntimeException, 329, 348
UnsupportedOperationException, 520
блок try, 312
возбуждение исключения, 311
возобновление и прерывание, 314
защищенная секция, 312
и конструкторы, 339
и наследование, 336, 343
идентификация, 343
контролируемое, 320
неконтролируемое, 329
обработка, 310
обработчик, 313
перехват, 312, 320
повторное возбуждение, 322
потеря, 335
предложение finally, 330
рекомендации по использованию, 350
создание собственных, 314
спецификации, 319
тип Error, 328
тип Exception, 328
исключения, 38
контролируемые
генерация, 350
заворачивание, 350
исключительная ситуация, 310
итератор, 288
методы интерфейса Iterator, 288

К

каналы (pipes), 490, 505, 601
каналы nio (channels), 510
каркас приложения, 266

карта, 280
класс
Adler32, 532
BufferedInputStream, 493
BufferedOutputStream, 494
BufferedReader, 496
BufferedWriter, 496
ByteArrayInputStream, 490, 500
ByteArrayOutputStream, 491
ByteBuffer, 510
Channels, 510
CharArrayReader, 495
CharArrayWriter, 495
CharBuffer, 513
Charset, 514
CheckedInputStream, 531
CheckedOutputStream, 531
Class, 354, 550
CRC32, 532
Daemon, 567
DataInputStream, 492, 496, 499, 502
DataOutputStream, 494, 497, 502
DeflaterOutputStream, 531
File, 484, 497
FileChannel, 510
FileInputReader, 498
FileInputStream, 490
FileOutputStream, 491
FileReader, 495
FileWriter, 495, 500
FilterInputStream, 490
FilterOutputStream, 491
FilterReader, 496
FilterWriter, 496
GZIPInputStream, 531
GZIPOutputStream, 531
InflaterInputStream, 531
InputStream, 489
InputStreamReader, 495
LineNumberInputStream, 493
LineNumberReader, 496
Object, 172, 598
ObjectOutputStream, 537
OutputStream, 489, 491
OutputStreamWriter, 495
PipedInputStream, 490
PipedOutputStream, 491
PipedReader, 495, 601
PipedWriter, 495, 601
PrintStream, 493, 494
PrintWriter, 496, 500
PushbackInputStream, 493
PushBackReader, 496
RandomAccessFile, 497, 503
Reader, 489, 494, 495
RuntimeException, 348
SequenceInputStream, 490, 497
класс (*продолжение*)
StreamTokenizer, 496
StringBufferInputStream, 490
StringReader, 495
StringWriter, 495
System.err, 315
Writer, 489, 494, 495
ZipEntry, 534
ZipInputStream, 531
ZipOutputStream, 531
базовый, 172, 201
вложенный, 258
внутренний, 245
безымянный, 251, 485
переопределение, 273
статический, 258
доступ к классам, 164
загрузка классов, 196
производный, 201
сравнение и оператор instanceof, 375
стиль написания, 164
классы коллекций, 277
клиент/сервер, архитектура, 40
ключевое слово
catch, 313
extends, 163, 172
final, 188
finally, 330
implements, 225
instanceof, 362
interface, 224
private, 162
protected, 163, 185
public, 160
static, 129
super, 175
synchronized, 581, 582
this, 126
throw, 312
transient, 544
try, 312
volatile, 567
кодирование
ASCII, 503
UTF-8, 503
коллекция, 280
команды
break, 108
break и continue с метками, 110
continue, 108
do-while, 103
for, 103
if-else, 102
switch, 113
while, 103
комментарии и встроенная документация, 64
компилируемый модуль, 154
компиляция программы на Java, 63

композиция, 24, 169, 217
выбор между композицией и наследованием, 184
динамическое изменение поведения, 218
совмещение композиции и наследования, 178
константа
времени компиляции, 188
группа постоянных значений, 238
конструктор, 116
аргументы, 117
возвращаемое значение, 118
вызов из другого конструктора, 128
вызов конструктора базового класса с аргументами, 175
и безымянный внутренний класс, 251
и обработка исключений, 339
и полиморфизм, 208
имя конструктора, 117
перегрузка, 119
по умолчанию, 125
поведение полиморфных методов, 214
порядок вызова конструкторов, 208
синтезирование конструктора по умолчанию, 175
контейнеры, 34
классы, 277
контрольная сумма, 532
конфликты имен, 157
конфликты имен при совмещении интерфейсов, 235
копирование
поверхностное, 475
критическая секция, 590
куча, 37

Л

литерал class, 358

М

манифест, 535
массив
передача аргументов в метод, 149
проверка границ, 147
размер, 147
массивы, 454
и параметризация, 463
многомерные, 460
ступенчатые, 460
метод
allocate(), 511
allocateDirect(), 512
array(), 520
Array.sort(), 484
asCharBuffer(), 513
available(), 500
capacity(), 522
Charset.forName(), 515
метод (*продолжение*)
Class.forName(), 356
Class.getInterfaces(), 358
Class.getSuperclass(), 358
Class.isInstance(), 369
Class.newInstance(), 358
clear(), 512, 522
close(), 501
entries(), 534
equals(), 79
final, 202, 216
finalize(), 130
вызов напрямую, 132
условие «готовности», 132
flip(), 512, 522
forName(), 356
getBytes(), 500
getCause(), 349
getChannel(), 511
getCheckSum(), 531
getClass(), 554
getFilePointer(), 497
getNextEntry(), 534
getPriority(), 565
hasRemaining(), 522
interrupt(), 576
isAlive(), 576
isDaemon(), 570
isInterrupted(), 577
isShared(), 529
join(), 576
keys(), 554
length(), 497
limit(), 517, 522
lock(), 529
main(), 173
mark(), 498, 522
mkdirs(), 489
notify(), 598
notifyAll(), 598
order(), 519
position(), 522
print(), 493
println(), 493
read(), 499
readDouble(), 503
readExternal(), 541
readLine(), 501
readObject(), 537, 545
readUTF(), 503
release(), 529, 530
remaining(), 522
renameTo(), 489
reset(), 498
seek(), 497, 503
setDaemon(), 568
setErr(PrintStream), 509
setIn(InputStream), 509

метод (*продолжение*)
setOut(PrintStream), 509
setPriority(), 565
sleep(), 564
отличие от метода wait(), 598
slice(), 530
split(), 507
static, 129
System.getProperty(), 515
systemNodeForPackage(), 554
Thread.toString(), 566
throwRuntimeException(), 350
toString(), 170, 566
transferFrom(), 512
transferTo(), 512
tryLock(), 529
userNodeForPackage(), 554
wait(), 598
writeDouble(), 503
writeExternal(), 541
writeObject(), 537, 545
writeUTF(), 503
yield(), 567
встроенный вызов, 192
закрытый, 216
перегрузка, 118
полиморфный, 198
различение перегруженных методов, 120
связывание «метод-вызов», 201
статический, 582
метод setPriority(), 565
методы, 56
многозадачность, 557
взаимная блокировка, 602
критическая секция, 590
недостатки, 630
простая блокировка, 582
рекомендации по применению, 629
синхронизация потоков, 581
типовые задачи синхронизации, 603
управляющий монитор, 582
уступки, 567
многомерные массивы, 460
множество, 280
монитор (в многозадачности), 582
для класса, 582
мультимножество, 280

Н

наследование, 25, 163, 169, 172, 198
выбор между композицией и наследованием, 184
использование наследования, 217
множественное наследование, 232
от внутренних классов, 272
расширение интерфейсов через наследование, 233
неизменные аргументы, 191
неизменные данные, 188
неизменные классы, 193
неизменные методы, 192
нисходящее преобразование, 218
безопасное нисходящее приведение типов, 362

О

обработчик исключений, 330
обратный вызов, 264
объект
легковесное долговременное хранение, 536
объектно-ориентированное программирование, 353
присвоение и копирование ссылок, 73
процесс создания, 143
равенство, 79
сериализация, 536
создание, 117
условие готовности, 132
объектный подход, 18
объекты, 19
однокоренные иерархии, 33
ООП
диаграммы наследования, 353
протокол, 224
операторы, 72
new, 130
арифметические, 75
выбора, 113
запятая, 105
индексирования, 146
логические, 80
ускоренное вычисление, 81
перегрузка, 89
побочный эффект, 72
поразрядные, 84
приведение
расширяющее приведение, 91
сужающее приведение, 91
приведения, 90
приоритет, 72
сдвига, 85
сравнения, 78
тернарный оператор «если-иначе», 88
типовые ошибки использования, 89

П

пакет, 153
и структура каталогов, 159
имена, 59
по умолчанию, 161
пакеты, 59
параметризованные типы, 35

перегрузка
  и возвращаемые значения, 124
  оператора += для строк, 173
  операторов, 89
переключение контекста, 585
переменная
  автоматическая, 51
  инициализация, 137
переменная окружения CLASSPATH, 156
переполнение, 100
побочный эффект, 125
повторное использование
  кода, 169
повышение, 92
подобъект, 175, 184
позднее связывание, 198, 202
поле TYPE (для примитивов), 359
полиморфизм, 31, 198, 220, 353, 395
  и конструкторы, 208
  разделение типов, 198
потоки, 557
потоки выполнения
  взаимное исключение, 581
  взаимодействие через ввод/вывод, 601
  демоны, 567
  недостатки, 630
  ожидание, 564, 598
  передача управления, 567
  последовательность выполнения, 564
  приоритет, 565
  присоединение, 576
  производительность, 558
  простая блокировка, 582
  рекомендации по применению, 629
  синхронизация, 581
  совместное использование ограниченных ресурсов, 578
  уведомление, 598
предпочтения, 553
преждевременная ссылка, 139
преобразование
  восходящее, 186, 199
    и интерфейс, 227
  нисходящее, 218
  типов к строке, 171
преобразования типов, 32
прерывание, 314
примитивы, 51
приоритет
  операций, 72
  потоков, 565
присвоение, 73
программист-клиент, 152
производные классы, 25
производный класс, инициализация, 174
пространство
  имен, 153
пустые константы, 190

P

разделение
  интерфейса и реализации, 164
раннее связывание, 201
распечатка в двоичном формате, 88
расширяемая программа, 205
реализация
  сокрытие, 163, 249
рефлексия, 376, 468
  отличие от RTTI, 377

C

C++, 48
сборка мусора, 130, 132
  как работает сборщик мусора, 134
  порядок удаления объектов, 182
сборщик мусора, 54
связуемость, 22
связывание
  во время выполнения , 198, 202
  динамическое, 198
  позднее, 198
  позднее (динамическое), 202
  раннее, 202
сервлеты, 47
сериализация
  и ключевое слово transient, 544
  и хранение объектов, 548
  управление процессом сериализации, 540
сигнатура, 57
синонимией, 74
синхронизированная блокировка, 591
система управления, 266
события
  система, управляемая по событиям, 266
совмещение имен, 74
сокрытие реализации, 163
сообщения, 20
спецификатор
  protected, 185
спецификатор доступа, 152, 159
  private, 152, 162
  protected, 152, 163
  public, 152, 160
    интерфейс, 225
список, 280
ссылки, 49
статический блок, 144
стек, 292
стиль написания классов, 164
строка
  перегрузка оператора + =, 173

T

тип, примитивные типы, 51

У

удаленный вызов методов, 536
указатели, отсутствие в Java, 264

Х

хранение данных, 50

Я

язык
    C++, деструктор, 130
    Java, компиляция и запуск программы, 63

Брюс Эккель

Философия Java. Библиотека программиста

4-е издание

Перевел с английского Е. Матвеев

Заведующий редакцией	*А. Сандрыкин*
Руководитель проекта	*П. Маннинен*
Ведущий редактор	*А. Пасечник*
Научные редакторы	*Е. Матвеев, А. Пасечник*
Художественный редактор	*А. Татарко*
Корректоры	*Е. Каюрова, И. Тимофеева*
Верстка	*Л. Харитонов*

Подписано в печать 29.08.08. Формат 70x100/16. Усл. п. л. 51,6. Тираж 2000. Заказ № 829.
ООО «Питер Пресс», 198206, Санкт-Петербург, Петергофское шоссе, д. 73, лит. А29.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93,
том 2;95 3005 — литература учебная.
Отпечатано по технологии CtP в в ГП ПО «Псковская областная типография».
180004, г. Псков, ул. Ротная, 34.