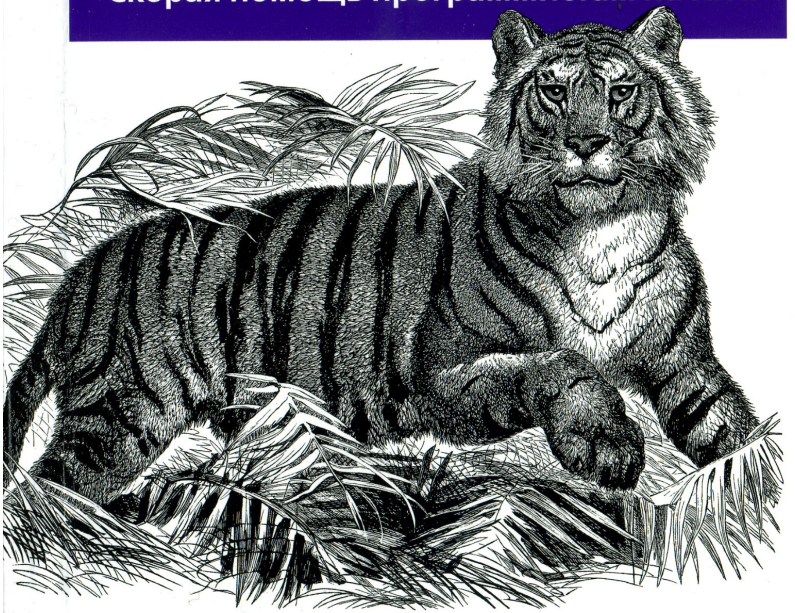
O'REILLY®

Java 8

Карманный справочник

Скорая помощь программистам на Java



ВИЛЬЯМС

www.williamspublishing.com

Роберт Лигуори

Патрисия Лигуори

Java 8

Карманный справочник

Java 8

Карманный справочник

Роберт Лигуори, Патрисия Лигуори

Москва · Санкт-Петербург · Киев
2016

ББК 32.973.26-018.2.75
Л55
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского *О.Л. Пелявского*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Лигуори, Роберт, **Лигуори**, Патрисия.

Л55 Java 8. Карманный справочник. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2016. — 256 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-2050-8 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Роберт Лигуори, Патрисия Лигуори

Java 8. Карманный справочник

Литературный редактор *И.А. Попова*
Верстка *М.А. Удалов*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

Подписано в печать 13.10.2015. Формат 84x108/32
Гарнитура Times
Усл. печ. л. 13,52. Уч.-изд. л. 7,0
Тираж 400 экз. Заказ № 5924

Отпечатано способом ролевой струйной печати
в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-2050-8 (рус.)
ISBN 978-1-49-190086-4 (англ.)

Оглавление

Введение 15

Часть I. Язык 19

Глава 1. Соглашение об именах 21
Глава 2. Лексические элементы 25
Глава 3. Основные типы 39
Глава 4. Ссылочные типы 51
Глава 5. Объектно-ориентированное программирование 63
Глава 6. Операторы и блоки 81
Глава 7. Обработка исключений 91
Глава 8. Модификаторы Java 105

Часть II. Платформа 109

Глава 9. Платформа Java, стандартный выпуск 111
Глава 10. Основы разработки 129
Глава 11. Управление памятью 143
Глава 12. Основы ввода-вывода 153
Глава 13. Новое API ввода-вывода NIO 2.0 165
Глава 14. Параллелизм 169
Глава 15. Коллекции Java 179
Глава 16. Обобщения 187
Глава 17. Языки сценариев Java 195
Глава 18. Дата и время 203
Глава 19. Лямбда-выражения 211

Часть III. Приложения 219

Приложение А. Текучие интерфейсы API 221
Приложение Б. Средства сторонних разработчиков 223
Приложение В. Основы UML 233
Предметный указатель 243

Содержание

Введение 15
Структура книги 16
Соглашения, используемые в книге 16
Об авторах 17
Благодарности 17
Ждем ваших отзывов! 18

Часть I. Язык 19

Глава 1. Соглашение об именах 21
Имена классов 21
Имена интерфейсов 21
Имена методов 21
Имена переменных экземпляра и статических переменных 22
Имена параметров и локальных переменных 22
Имена параметров обобщенного типа 23
Имена констант 23
Имена перечислений 23
Имена пакетов 24
Имена аннотаций 24
Аббревиатуры 24

Глава 2. Лексические элементы 25
Символы Unicode и ASCII 25
Отображаемые ASCII-символы 26
Неотображаемые ASCII-символы 26
Комментарии 27
Ключевые слова 28
Идентификаторы 29
Разделители 30
Операторы 30
Литералы 32
Булевы литералы 32
Символьные литералы 32

Целочисленные литералы 32
Литералы с плавающей точкой 34
Строковые литералы 34
Нулевые литералы 35
Управляющие последовательности 35
Символы валют в Unicode 36

Глава 3. Основные типы 39
Простые типы 39
Литералы простого типа 40
Сущности с плавающей точкой 42
Операции с особыми сущностями 44
Числовое продвижение простых типов 44
Унарное числовое продвижение 45
Бинарное числовое продвижение 45
Особые случаи для условных операторов 46
Интерфейсные классы 46
Автоупаковка и распаковка 47
Автоупаковка 47
Распаковка 48

Глава 4. Ссылочные типы 51
Сравнение ссылочных и простых типов 51
Стандартные значения 52
Переменные экземпляра и локальные переменные 52
Массивы 53
Преобразование ссылочных типов 55
Расширяющие преобразования 55
Сужающие преобразования 55
Преобразования между простыми и ссылочными типами 56
Передача ссылочных типов в методы 56
Сравнение переменных ссылочного типа 57
Использование операторов равенства 57
Использование метода `equals()` 58
Сравнение строк 59
Сравнение перечислений 60
Копирование переменных ссылочного типа 60
Копирование ссылки на объект 60
Клонирование объектов 61

Поверхностное и глубокое клонирование 61
Распределение памяти и сборка мусора 62

Глава 5. Объектно-ориентированное программирование 63
Классы и объекты 63
Синтаксис класса 64
Создание экземпляра класса (объекта) 64
Данные-члены и методы 65
Доступ к данным-членам и методам объектов 65
Перегрузка 65
Переопределение 66
Конструкторы 67
Суперклассы и подклассы 68
Ключевое слово `this` 69
Списки с переменным числом параметров 70
Абстрактные классы и методы 72
Абстрактные классы 72
Абстрактные методы 72
Статические данные-члены, методы, константы и инициализаторы 73
Статические данные-члены 73
Статические методы 74
Статические константы 74
Статические инициализаторы 74
Интерфейсы 75
Перечисления 76
Типы аннотаций 76
Встроенные аннотации 77
Аннотации, определяемые разработчиком 77
Функциональные интерфейсы 79

Глава 6. Операторы и блоки 81
Операторы выражений 81
Пустой оператор 82
Блоки 82
Условные операторы 82
Оператор `if` 83
Оператор `if else` 83
Оператор `if else if` 83
Оператор `switch` 84

Итерационные операторы 85
Оператор цикла for 85
Операторы расширенного цикла for 85
Оператор цикла while 86
Оператор цикла do-while 86
Передача управления 87
Оператор break 87
Оператор continue 88
Оператор return 88
Оператор synchronized 89
Оператор assert 89
Операторы обработки исключений 90

Глава 7. Обработка исключений 91
Иерархия исключений 91
Проверяемые и непроверяемые исключения и ошибки 91
Проверяемые исключения 92
Непроверяемые исключения 92
Ошибки 93
Стандартные проверяемые и непроверяемые исключения и ошибки 93
Популярные проверяемые исключения 93
Популярные непроверяемые исключения 94
Популярные ошибки 95
Ключевые слова, используемые для обработки исключений 96
Ключевое слово throw 97
Ключевые слова try/catch/finally 97
Оператор try-catch 97
Оператор try-finally 99
Оператор try-catch-finally 100
Оператор try с ресурсами 100
Конструкция для перехвата нескольких исключений 101
Процесс обработки исключений 101
Определение собственного класса исключений 102
Вывод информации об исключениях 103
Метод getMessage() 103
Метод toString() 103
Метод printStackTrace() 104

Глава 8. Модификаторы Java 105
Модификаторы доступа 106
Остальные модификаторы доступа 106

Часть II. Платформа 109

Глава 9. Платформа Java, стандартный выпуск 111
Общие библиотеки Java SE API 112
Библиотеки поддержки языка и утилит 112
Базовые библиотеки 114
Библиотеки интеграции 116
Различные библиотеки пользовательского интерфейса 117
Библиотеки пользовательского интерфейса: JavaFX 118
Библиотеки пользовательского интерфейса: API AWT (устарело) 120
Библиотеки пользовательского интерфейса: Swing API (устарело) 121
Протокол RMI и библиотеки CORBA 123
Библиотеки обеспечения безопасности 125
Библиотеки XML 126

Глава 10. Основы разработки 129
Среда исполнения Java-программ (JRE) 129
Комплект разработчика программ на языке Java (JDK) 129
Структура программы на языке Java 130
Утилиты командной строки 132
Компилятор Java 133
Интерпретатор Java 134
Упаковщик Java-программ 137
Запуск JAR-файлов на выполнение 138
Документатор Java 139
Опция `classpath` 140

Глава 11. Управление памятью 143
Сборщики мусора 143
Последовательный сборщик мусора 144
Параллельный сборщик мусора 144
Параллельный сборщик мусора с уплотнением 144
Сборщик мусора с одновременной маркировкой и очисткой 145
Сборщик мусора Garbage-First (G1) 145

Средства управления памятью 145
Опции командной строки 147
Изменение размера кучи JVM 150
Метапространство 150
Взаимодействие с GC 151
Сборка мусора в явном виде 151
Финализация 151

Глава 12. Основы ввода-вывода 153
Стандартные потоки `in`, `out` и `err` 153
Иерархия основных классов ввода-вывода 154
Чтение и запись файлов 154
Чтение символьных данных из файла 155
Чтение двоичных данных из файла 156
Запись символьных данных в файл 156
Запись двоичных данных в файл 157
Чтение и запись сокетов 157
Чтение символьных данных из сокета 158
Чтение двоичных данных из сокета 158
Запись символьных данных в сокет 158
Запись двоичных данных в сокет 159
Сериализация 159
Сериализация 160
Десериализация 160
Сжатие и распаковка файлов 160
Работа с ZIP-архивами 161
Архивы в формате GZIP 161
Работа с файлами и каталогами 162
Часто используемые методы класса `File` 162
Доступ к существующим файлам 162
Позиционирование данных в файле 163

Глава 13. Новое API ввода-вывода NIO 2.0 165
Интерфейс Path 165
Класс `Files` 166
Дополнительные возможности 167

Глава 14. Параллелизм 169
Создание потоков 169

Расширение класса Thread 169
Реализация интерфейса Runnable 170
Состояния потока 170
Приоритеты потоков 171
Типичные методы 171
Синхронизация 173
Классы для поддержки параллелизма 174
Исполнители 174
Классы коллекций для параллельного выполнения 176
Синхронизаторы 176
Средства хронометража 177

Глава 15. Коллекции Java 179
Интерфейс Collection 179
Реализации 180
Методы инфраструктуры коллекций 180
Алгоритмы класса Collections 181
Эффективность алгоритмов 182
Функциональный интерфейс Comparator 183

Глава 16. Обобщения 187
Обобщенные классы и интерфейсы 187
Конструкторы с обобщениями 188
Принцип подстановки 189
Параметры типа, символы подстановки и ограничения 190
Принцип "взять и положить" 190
Обобщенная специализация 191
Обобщенные методы в первичных типах 192

Глава 17. Языки сценариев Java 195
Языки сценариев 195
Реализации интерпретаторов сценарных языков 195
Встраивание сценариев в Java-программы 195
Вызов методов сценарных языков 196
Доступ к ресурсам Java из сценариев 197
Установка сценарных языков и интерпретаторов 198
Установка сценарного языка 198
Установка интерпретатора сценарного языка 199
Проверка правильности установки интерпретатора 199

Глава 18. Дата и время 203
Устаревшая функциональная совместимость 204
Региональные календари 204
Календарь ISO 205
Основные классы API 205
Машинный интерфейс 207
Длительности и периоды 208
Соответствие типов JDBC и XSD 209
Форматирование 209

Глава 19. Лямбда-выражения 211
Основы лямбда-выражений 211
Синтаксис и примеры 212
Ссылки на методы и конструкторы 213
Функциональные интерфейсы специального назначения 214
Функциональные интерфейсы общего назначения 215
Дополнительная информация по лямбда-выражениям 216
Руководства 217
Общедоступные ресурсы 217

Часть III. Приложения 219

Приложение А. Текучие интерфейсы API 221

Приложение Б. Средства сторонних разработчиков 223
Средства разработки, конфигурирования и тестирования 223
Библиотеки 226
Интегрированные среды разработки 228
Платформы веб-приложений 229
Интерпретируемые языки (совместимые с JSR-223) 231

Приложение В. Основы UML 233
Диаграммы классов 233
Наименование 234
Атрибуты 234
Операции 234
Видимость 235
Диаграммы объектов 235

Представление в виде графических символов 236
Классы, абстрактные классы и интерфейсы 236
Примечания 236
Пакеты 237
Соединения 237
Индикаторы кратности 237
Имена ролей 238
Отношения между классами 238
Ассоциация 238
Прямая ассоциация 239
Композиция 239
Агрегация 240
Временная ассоциация 240
Генерализация 240
Реализация 240
Диаграммы последовательности 240
Участник (1) 241
Найденное сообщение (2) 241
Синхронное сообщение (3) 241
Ответный вызов (4) 241
Асинхронное сообщение (5) 242
Сообщение, адресованное самому себе (6) 242
Линия жизни (7) 242
Полоса активности (8) 242

Предметный указатель 243

Эта книга посвящается нашей прелестной, потрясающей дочери — Эшли.

Введение

Книги серии *Карманный справочник* призваны быть вашим спутником в офисе, в учебном классе и даже в пути. Эта книга может служить удобным справочником по стандартным возможностям языка программирования Java и его платформы.

Здесь собрана информация, которая понадобится вам при разработке или отладке программ на Java, в том числе полезные примеры программирования, таблицы, рисунки и листинги программ.

В книге содержится также вспомогательная информация по таким темам, как Java Scripting API, средства разработки сторонних фирм и основы унифицированного языка моделирования (Unified Modeling Language — UML).

Материал, представленный в книге, также поможет в подготовке к сдаче экзамена на получение квалификации Oracle Certified Associate Java SE 7 Programmer I. Если у вас есть желание получить такой сертификат, то, возможно, вам следует также приобрести книгу *OCA Java SE 7 Programmer I Study Guide (Exam IZO-803)* Эдварда Файнегана (Edward Finegan) и Роберта Лайгори (Robert Liguori), вышедшую в издательстве McGraw-Hill Osborne Media, 2012.

В настоящей книге представлены сведения о возможностях языка Java (до Java SE 8 включительно), в том числе новое API, предназначенное для работы с датой и временем, а также отдельная глава, посвященная лямбда-выражениям. Здесь описаны также новинки Java SE 7, дана базовая информация о NIO 2.0, описаны сборщик мусора G1 и небольшие улучшения языка Java JSR-334 (Project Coin). Улучшения Project Coin включают усовершенствованные литералы (например, возможность использования символа подчеркивания), новый ромбовидный оператор, связанный с обобщениями, и расширения, предназначенные для обработки исключений, такие, как, например, новые операторы мультиперехвата (multi-catch) и оператор `try` с ресурсами (try-with-resources).

Структура книги

Книга состоит из трех частей: "Язык", "Платформа" и "Приложения". В главах 1–8 подробно рассматривается язык программирования Java с точки зрения спецификации языка Java (Java Language Specification — JLS). В главах 9–19 подробно описаны компоненты платформы Java и связанные с этим темы. В приложениях рассматриваются средства разработки сторонних производителей и основы унифицированного языка моделирования (Unified Modeling Language — UML).

Соглашения, используемые в книге

Для выделения различных элементов используются следующие соглашения об обозначениях.

Текст курсивом

Обозначает новые термины и понятия, а также важные моменты, на которых акцентируется внимание.

`Моноширинный шрифт`

Используется для листингов программ, а также в абзацах для ссылки на такие элементы программ, как названия переменных или функций, типы, базы данных, переменные среды, операторы и ключевые слова. Кроме того, таким шрифтом выделяются URL и адреса электронной почты.

`Полужирный моноширинный шрифт`

Используется для выделения команд или текстовых литералов, которые должны быть введены с клавиатуры непосредственно пользователем.

`Моноширинный курсив`

Обозначает текст, вместо которого следует подставить значение, указанное пользователем, или значение, определяемое по контексту.

НА ЗАМЕТКУ

В таких врезках приведены полезные советы, ценные указания или замечания общего характера.

ВНИМАНИЕ!

Так оформлены места, на которые вы должны обратить особое внимание.

Об авторах

Роберт Джеймс Лигуори (Robert James Liguori) — руководитель компании Gliesian LLC, дипломированный специалист Oracle (Oracle Certified Expert). Он поддерживает работоспособность нескольких приложений управления воздушным движением, созданных на основе Java.

Патриция Лигуори (Patricia Liguori) работает в компании The MITRE Corporation в должности многопланового инженера по информационным системам. Начиная с 1994 года она занимается разработкой систем управления воздушным движением в реальном времени и информационных систем, связанных с авиацией.

Благодарности

Авторы хотят выразить особую благодарность редактору Меган Бланшет (Meghan Blanchette). Постоянный контроль и сотрудничество с ее стороны имели огромное значение для публикации этой книги.

Особое признание мы хотели бы выразить техническому редактору первого издания этой книги Майклу Лукидису (Michael Loukides) и нашему техническому рецензенту Райану Купраку (Ryan Cuprak), а также многим членам коллектива издательства O'Reilly, нашей семье и друзьям.

Кроме того, мы хотели бы еще раз поблагодарить всех, кто принимал участие в публикации первого и второго изданий этой книги.

И, что самое главное, мы хотим выразить признательность и огромное уважение всем тем, кто использует нашу книгу в качестве настольного руководства и разделяет наше увлечение языком Java. Мы будем рады, если вы опубликуете свое фото вместе с этой книге в сервисе микроблогов Тамблер (Tumblr). Нам всегда приятно лицезреть тех, кто использует нашу книгу в самых необычных местах (особенно в отпуске). ☺

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info@williamspublishing.com`

WWW: `http://www.williamspublishing.com`

Наши почтовые адреса:

в России: 127055, Москва, ул. Лесная, д. 43, стр 1

в Украине: 03150, Киев, а/я 152

ЧАСТЬ I

Язык

Соглашение об именах

Соглашение об именах используется для того, чтобы сделать программы на Java более читабельными. Очень важно использовать осмысленные и однозначно интерпретируемые имена, состоящие из символов, допустимых в Java.

Имена классов

Имена классов должны быть существительными, поскольку они представляют "предметы" или "объекты". Имена классов должны состоять из набора строчных и прописных букв латинского алфавита, причем только первая буква каждого слова должна быть прописной, как показано в приведенном ниже примере.

```
public class Fish {...}
```

Имена интерфейсов

Имена интерфейсов должны быть прилагательными. Они должны заканчиваться на `...able` ("-ый") или `...ible` ("-ий") в случаях, когда соответствующий интерфейс обеспечивает какую-либо возможность; в противном случае они должны быть существительными. Имена интерфейсов должны соответствовать такому же соглашению об использовании строчных и прописных букв, как и для имен классов.

```
public interface Serializable {...}
public interface SystemPanel {...}
```

Имена методов

В именах методов должен присутствовать глагол, поскольку они используются, чтобы заставить соответствующий объект

выполнить определенное действие. В именах методов могут также использоваться комбинации строчных и прописных букв, причем первая буква каждого последующего слова должна быть прописной. В имена методов могут включаться как существительные, так и прилагательные.

```
public void locate() {...}              // глагол
public String getWayPoint() {...}       // глагол и имя
                                        // существительное
```

Имена переменных экземпляра и статических переменных

Имена переменных экземпляра и статических переменных должны быть существительными и соответствовать тому же соглашению об использовании строчных и прописных букв, что и для имен методов.

```
private String wayPoint;
```

Имена параметров и локальных переменных

Имена параметров и локальных переменных должны представлять собой отдельно взятые английские слова или аббревиатуры, составленные из строчных букв, и иметь описательный характер. Если в таких именах предпочтительно использовать несколько слов, то они должны соответствовать тому же соглашению об использовании строчных и прописных букв, что и для имен методов.

```
public void printHotSpots(ArrayList spotList) {
   int counter = 0;
   for (String hotSpot : spotList) {
      System.out.println("Hot Spot #"
       + ++counter + ": " + hotSpot);
   }
}
```

Имена временных переменных могут представлять собой отдельно взятые буквы, например `i`, `j`, `k`, `m` и `n` для целых чисел и `c`, `d` и `e` для символов.

Имена параметров обобщенного типа

Имена параметров обобщенного типа должны представлять собой отдельно взятые прописные буквы. Параметр типа, как правило, обозначается буквой `T`.

Обобщения широко используются в инфраструктуре Collections Framework. В ней буква `E` используется для обозначения элементов коллекции, `S` — для обозначения сервисных загрузчиков (service loader), а `K` и `V` используются для обозначения типов ключей и значений отображения (map).

```
public interface Map <K,V> {
   V put(K key, V value);
}
```

Имена констант

Имена констант должны состоять исключительно из прописных букв, а если используется несколько слов, то они должны разделяться символами подчеркивания.

```
public static final int MAX_DEPTH = 200;
```

Имена перечислений

Имена перечислений должны соответствовать соглашениям об именах классов. Объекты, входящие в перечисление, должны обозначаться исключительно прописными буквами.

```
enum Battery {CRITICAL, LOW, CHARGED, FULL}
```

Имена пакетов

Имена пакетов должны быть уникальными и состоять из строчных букв. По мере необходимости можно использовать символы подчеркивания.

```
package com.oreilly.fish_finder;
```

Общедоступные пакеты должны обозначаться реверсивным доменным именем Интернета соответствующей организации, начинающимся с однословного доменного имени верхнего уровня (например, `com`, `net`, `org` или `edu`), за которым следует название соответствующей организации и проект или подразделение. (Внутренние пакеты обычно получают имя по названию соответствующего проекта.)

Имена пакетов, которые начинаются с `java` или `javax`, носят ограниченный характер и могут использоваться только для создания реализаций, удовлетворяющих требованиям библиотек классов Java.

Имена аннотаций

В JavaSE API имена аннотаций могут выбираться по-разному в зависимости от предопределенного типа аннотации (в виде прилагательного, глагола или имени существительного), как показано ниже.

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public@interfaceFunctionalInterface{}"
```

Аббревиатуры

При использовании аббревиатур в именах только первая буква должна быть прописной — и только тогда, когда это представляется уместным.

```
public String getGpsVersion() {...}
```

Лексические элементы

Исходный код на Java состоит из слов или символов, называемых лексическими элементами, или лексемами (tokens). В лексические элементы языка Java входят указатели конца строк, пробельный символ, комментарии, ключевые слова, идентификаторы, разделители, операторы и литералы. В словах или символах языка Java используется набор символов Unicode.

Символы Unicode и ASCII

Стандарт Unicode, поддерживаемый организацией стандартов Unicode Consortium, представляет собой универсальный набор символов, первые 128 символов которого совпадают с набором символов ASCII (American Standard Code for Information Interchange — Американский стандартный код для обмена информацией). В Unicode предусмотрен уникальный номер для каждого символа, используемый на всех платформах, во всех программах и языках. В Java SE 8 используется Unicode 6.2.0 (подробнее о нем вы можете прочитать в соответствующем электронном справочном руководстве). В Java SE 7 используется стандарт Unicode 6.2.0, а в Java SE 6 и J2SE 5.0 — Unicode 4.0.

НА ЗАМЕТКУ

В комментариях, идентификаторах и строковых литералах языка Java допускается использовать не только ASCII-символы. Во всех других конструкциях языка Java используются только ASCII-символы.

Версия набора символов Unicode, используемая в конкретной версии платформы Java, указана в классе `Character` Java API.

С таблицей кодов символов Unicode, используемых в сценариях, и соответствующими им символами и знаками пунктуации можно ознакомиться на сайте `http://unicode.org/charts/`.

Отображаемые ASCII-символы

Для отображаемых ASCII-символов зарезервированы код 32 (пробел) и коды 33–126, которым соответствуют буквы, цифры, знаки пунктуации и несколько служебных символов. В табл. 2.1 приведены десятичные значения кодов и соответствующие им ASCII-символы.

Таблица 2.1. Отображаемые ASCII-символы

32 SP	48 0	64 @	80 P	96 `	112 p
33 !	49 1	65 A	81 Q	97 a	113 q
34 "	50 2	66 B'	82 R	98 b	114 r
35 #	51 3	67 C	83 S	99 c	115 s
36 $	52 4	68 D	84 T	100 d	116 t
37 %	53 5	69 E	85 U	101 e	117 u
38 &	54 6	70 F	86 V	102 f	118 v
39 '	55 7	71 G	87 W	103 g	119 w
40 (	56 8	72 H	88 X	104 h	120 x
41)	57 9	73 I	89 Y	105 i	121 y
42 *	58 :	74 J	90 Z	106 j	122 z
43 +	59 ;	75 K	91 [	107 k	123 {
44 ,	60 <	76 L	92 \	108 l	124 |
45 -	61 =	77 M	93]	109 m	125 }
46 .	62 >	78 N	94 ^	110 n	126 ~
47 /	63 ?	79 O	95 _	111 o	

Неотображаемые ASCII-символы

В коде ASCII десятичные числа 0–31 и 127 зарезервированы для *управляющих символов*. В табл. 2.2 приведены десятичные значения кодов и соответствующие им ASCII-символы.

Таблица 2.2. Неотображаемые ASCII-символы

00 NUL	07 BEL	14 SO	21 NAK	28 FS
01 SOH	08 BS	15 SI	22 SYN	29 GS
02 STX	09 HT	16 DLE	23 ETB	30 RS
03 ETX	10 NL	17 DC1	24 CAN	31 US
04 EOT	11 VT	18 DC2	25 EM	127 DEL
05 ENQ	12 NP	19 DC3	26 SUB	
06 ACK	13 CR	20 DC4	27 ESC	

НА ЗАМЕТКУ

ASCII-коду 10 соответствует символ новой строки (NL, или newline) или перевода строки (LF, или linefeed). ASCII-коду 13 соответствует символ возврата каретки (CR, или carriage return).

Комментарии

Однострочный комментарий начинается с двух символов косой черты (//) и заканчивается непосредственно перед символом конца строки.

```
// Комментарий в отдельной строке
```

Многострочный комментарий начинается с символа косой черты, сразу за которым следует символ звездочки (/*), и заканчивается символом звездочки, сразу за которым следует символ косой черты (*/). Одиночные звездочки перед каждой строкой многострочного комментария облегчают чтение такого комментария; такие звездочки, хоть и необязательны, на практике используются очень часто.

```
/*
 * Комментарий, который может состоять из нескольких
 * строк, как, например, этот
 *
 */
```

Комментарий в стиле Javadoc обрабатывается утилитой javadoc для автоматического создания документации в формате

HTML. Комментарий в стиле Javadoc начинается с символа косой черты, за которым следуют два символа звездочки, и заканчивается символом звездочки, за которым следует символ косой черты (подробнее об утилите `javadoc` можно прочитать на соответствующей странице документации сайта Oracle).

```
/** Это мой комментарий в стиле Javadoc */
```

Комментарии в Java не могут быть вложенными.

```
/** Это /* недопустимый комментарий */ в Java */
```

Ключевые слова

В табл. 2.3 указаны ключевые слова языка Java. Два из этих ключевых слов, `const` и `goto`, зарезервированы, но не используются в языке Java. Ключевое слово `enum` впервые появилось в Java 5.0.

НА ЗАМЕТКУ

Ключевые слова Java не могут использоваться в программах на Java в качестве идентификаторов.

Таблица 2.3. Ключевые слова Java

abstract	double	int	super
assert	else	interface	switch
boolean	enum	long	synchronized
break	extends	native	this
byte	final	new	throw
case	finally	package	throws
catch	float	private	transient
char	for	protected	try
class	if	public	void
const	goto	return	volatile
continue	implements	short	while
default	import	static	
do	instanceof	strictfp	

НА ЗАМЕТКУ

Иногда литералы `true`, `false` и `null` ошибочно принимают за ключевые слова. Они не являются ключевыми словами — это зарезервированные литералы.

Идентификаторы

Идентификатор в Java представляет собой имя, которое программист присваивает тому или иному классу, методу, переменной и т.п.

Последовательность символов Unicode, формирующая идентификатор в Java, не должна совпадать с каким-либо из ключевых слов, включая `boolean`, или литералом `null`. Другими словами, в качестве идентификаторов в Java нельзя выбирать ключевые слова или зарезервированные литералы.

Идентификаторы должны состоять из букв Java (Java letter). Буква Java представляет собой символ, для которого метод `Character.isJavaIdentifierStart(int)` возвращает значение `true`. Буквами Java из набора символов ASCII являются лишь знак доллара ($), символ подчеркивания (_), а также строчные и прописные буквы английского алфавита.

В идентификаторах можно также использовать цифры, но только *после* какого-либо символа.

```
// Примеры допустимых идентификаторов
class TestDriver {...}
String testVariable;
int _testVariable;
Long $testVariable;
startTest(testVariable1);
```

Правила выбора имен изложены в главе 1.

Разделители

Некоторые из ASCII-символов используются для разграничения частей программы. Символы `()`, `{}` и `[]` используются попарно.

```
( ) { } [ ] < > :: : ; , . ->
```

В табл. 2.4 приведен перечень разных типов скобочных разделителей и их краткое описание. Первым указано название скобок, которое обычно используется в языке Java.

Таблица 2.4. Скобочные разделители в языке Java

Обозначение	Название	Описание
`()`	Скобки, круглые скобки, овальные скобки, закругленные скобки	Определяют порядок выполнения арифметической операции; в них указывается тип, к которому приводится выражение; с их помощью выделяются аргументы метода
`{}`	Фигурные скобки, кривые скобки, волнистые скобки	В них заключается блок кода, а также элементы массивов
`[]`	Квадратные скобки, замкнутые скобки	Используются при инициализации массивов
`<>`	Угловые скобки, ромбовидные скобки, шевроны	Используются для выделения обобщений

Двойные угловые скобки << >> используются для указания стереотипов в UML.

Операторы

Операторы (operators) выполняют операции над одним, двумя или тремя операндами и возвращают результат этой операции. В Java возможны следующие типы операций: присваивание, арифметические операции, сравнение, поразрядные (побитовые) операции, увеличение/уменьшение значения операнда на 1 (инкремент/декремент) и класс/объект. В табл. 2.5 перечислены операторы Java в порядке их старшинства (т.е. приоритета выполнения

операции); операторы с самым высоким приоритетом расположены в верхней части таблицы. В табл. 2.5 приведены также краткие описания операторов и указана их ассоциативность (слева направо или справа налево).

Таблица 2.5. Операторы Java

Приоритет	Оператор	Описание	Ассоциативность
1	`++, --`	Постинкремент, постдекремент	П→Л
2	`++, --`	Преинкремент, предекремент	П→Л
	`+, -`	Унарный плюс, унарный минус	П→Л
	`~`	Поразрядное дополнение	П→Л
	`!`	Булево НЕ	П→Л
3	`new`	Создать объект	П→Л
	`(type)`	Приведение типов данных	П→Л
4	`*, /, %`	Умножение, деление, взятие остатка	Л→П
5	`+, -`	Сложение, вычитание	Л→П
	`+`	Конкатенация строк	Л→П
6	`<<, >>, >>>`	Сдвиг влево, сдвиг вправо, беззнаковый сдвиг вправо	Л→П
7	`<, <=, >, >=`	Меньше чем, не больше, больше чем, не меньше	Л→П
	`instanceof`	Сравнение типов	Л→П
8	`==, !=`	Равенство и неравенство значения	Л→П
	`==, !=`	Равенство и неравенство ссылки	Л→П
9	`&`	Булево И	Л→П
	`&`	Поразрядное И	Л→П
10	`^`	Булево исключающее ИЛИ (XOR)	Л→П
	`^`	Поразрядное исключающее ИЛИ (XOR)	Л→П
11	`|`	Булево включающее ИЛИ	Л→П
	`|`	Поразрядное включающее ИЛИ	Л→П
12	`&&`	Логическое И (оно же условное И)	Л→П
13	`||`	Логическое ИЛИ (оно же условное ИЛИ)	Л→П
14	`?:`	Тернарный условный оператор	Л→П
15	`=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=`	Операторы присваивания	П→Л

Литералы

Литералы представляют в исходном коде различные значения. Для улучшения читабельности кода начиная с Java SE 7 в численных литералах можно использовать символы подчеркивания. Их разрешается помещать только между отдельными числами; при выполнении кода символы подчеркивания игнорируются.

Подробнее о литералах простого типа можно прочитать в разделе "Литералы простого типа" главы 3 (с. 23).

Булевы литералы

Булевы литералы могут принимать либо значение `true`, либо `false`.

```
boolean isReady = true;
boolean isSet = new Boolean(false); // распакованный
boolean isGoing = false;
```

Символьные литералы

Символьный литерал представляет собой либо отдельно взятый символ, либо последовательность управляющих символов, заключенную в одинарные кавычки. При этом использовать символы конца строки не разрешается.

```
char charValue1 = 'a';
// Апостроф
Character charValue2 = new Character ('\'');
```

Целочисленные литералы

Литералы целочисленного типа (`byte`, `short`, `int` и `long`) могут быть заданы в десятичном, шестнадцатеричном, восьмеричном или двоичном представлении. По умолчанию целочисленные литералы имеют тип `int`.

```
int intValue1 = 34567, intValue2 = 1_000_000;
```

Десятичные целые литералы могут содержать произвольное количество цифр в кодах ASCII от нуля до девяти и представляют положительные числа.

```
Integer integerValue1 = new Integer(100);
```

Если добавить к десятичному числу префикс в виде унарного оператора отрицания, то можно получить отрицательное десятичное число.

```
public static final int INT_VALUE = -200;
```

Шестнадцатеричные литералы начинаются с символов `0x` или `0X`, за которыми следуют цифры в кодах ASCII от нуля до девяти и буквы от `a` до `f` (или от `A` до `F`). Когда речь идет о шестнадцатеричных литералах, для языка Java *нет* разницы между строчными и прописными буквами.

Шестнадцатеричные числа могут представлять положительные и отрицательные целые числа и нуль.

```
int intValue3 = 0X64; // десятичное число 100
                      // в шестнадцатеричном
                      // представлении
```

Восьмеричные литералы начинаются с нуля, за которым следует одна или несколько цифр в кодах ASCII от нуля до семи.

```
int intValue4 = 0144; // десятичное число 100
                      // в восьмеричном представлении
```

Двоичные литералы выражаются с использованием префикса `0b` или `0B`, за которым следуют нули и единицы.

```
char msgValue1 = 0b01001111; // 'O'
char msgValue2 = 0B01001011; // 'K'
char msgValue3 = 0B0010_0001; // '!'
```

Чтобы определить целочисленный литерал типа `long`, укажите после числа префикс в виде прописной буквы `L` (предпочтительно и более читабельно) или `l`.

```
long longValue = 100L;
```

Литералы с плавающей точкой

Допустимый литерал с плавающей точкой состоит из целочисленной и/или дробной части, десятичной точки и суффикса, указывающего тип. Показатель степени с префиксом e или E необязателен. Дробные части и десятичные точки не требуются в случае, когда указываются показатели степени или суффиксы типа.

Литерал с плавающей точкой типа `double` представляет собой восьмибайтовое число двойной точности с плавающей точкой. Четырехбайтовое число с плавающей точкой имеет тип `float`. Суффиксами типа для чисел двойной точности с плавающей точкой являются d или D; суффиксами типа для чисел с плавающей точкой одинарной точности являются f или F.

```
[целая-часть].[дробная-часть][e|E экспонента][f|F|d|D]

float floatValue1 = 9.15f, floatValue2 = 1_168f;
Float floatValue3 = new Float(20F);
double doubleValue1 = 3.12;
Double doubleValue2 = new Double(1e058);
float expValue1 = 10.0e2f, expValue2=10.0E3f;
```

Строковые литералы

Строковые литералы могут содержать нуль или большее число символов, включая управляющие последовательности, заключенные в двойные кавычки. В строковых литералах нельзя использовать символы Unicode \u000a и \u000d в качестве символов конца строки; вместо этого следует использовать управляющие символы \r и \n. Строки являются неизменяемыми.

```
String stringValue1 = new String("Допустимый строковый
литерал.");
String stringValue2 = "Допустимо.\nНа новой строке.";
String stringValue3 = "Объединение стр" + "ок";
String stringValue4 = "\"Управляющая
последовательность\"\r";
```

Существует специальный пул строк, связанных с классом `String`. Поначалу этот пул является пустым. Литеральные строки

и выражения-константы, имеющие строковые значения, размещаются в пуле и добавляются к нему лишь однократно.

В приведенном ниже примере показано, как литералы добавляются в пул и используются в нем.

```
// Добавляет в пул строку "thisString"
String stringValue5 = "thisString";

// Использует строку "thisString" из пула
String stringValue6 = "thisString";
```

Строку можно добавить в пул (если ее еще там нет), вызвав метод `intern()` объекта данной строки. Метод `intern()` возвращает строку; возвращаемое значение представляет собой либо ссылку на новую строку, которая была добавлена к данному пулу, либо на уже существующую.

```
String stringValue7 = new String("thatString");
String stringValue8 = stringValue7.intern();
```

Нулевые литералы

Нулевой литерал имеет тип `null`, может быть применен к ссылочным типам и не применяется к простым типам.

```
String n = null;
```

Управляющие последовательности

В табл. 2.6 приведена совокупность управляющих последовательностей в языке Java.

Таблица 2.6. Символьные и строковые литеральные управляющие последовательности

Название	Последовательность	Десятичное представление	Unicode
Символ возврата на одну позицию	\b	8	\u0008
Горизонтальная табуляция	\t	9	\u0009

Окончание табл. 2.6

Название	Последовательность	Десятичное представление	Unicode
Перевод строки	\n	10	\u000A
Прогон страницы	\f	12	\u000C
Возврат каретки	\r	13	\u000D
Двойная кавычка	\"	34	\u0022
Одинарная кавычка	\'	39	\u0027
Обратная косая черта	\\	92	\u005C

Для перехода на новую строку в разных компьютерных платформах используются разные управляющие последовательности, указывающие на конец строки (табл. 2.7). Лучше всего при выводе текстового сообщения с новой строки пользоваться методом `println()`, а не жестко кодировать символы управляющих последовательностей `\n` и `\r` прямо в строке.

Таблица 2.7. Варианты перехода на новую строку

Операционная система	Символ конца строки
Совместимая с POSIX (например, Solaris, Linux) и Mac OS X	LF(\n)
Mac OS (вплоть до версии 9)	CR(\r)
Microsoft Windows	CR+LF(\r\n)

Символы валют в Unicode

Символы валют в Unicode представлены в диапазоне `\u20A0-\u20CF` (8352–8399). Соответствующие примеры приведены в табл. 2.8.

Таблица 2.8. Символы валют в кодировке Unicode

Название	Символ	Десятичное	Unicode
Знак франка	₣	8355	\u20A3
Знак лиры	£	8356	\u20A4

Название	Символ	Десятичное	Unicode
Знак тысячной части доллара (используется при вычислениях)	₥	8357	\u20A5
Знак рупии	₨	8360	\u20A8
Знак донга	₫	8363	\u20AB
Знак евро	€	8364	\u20AC
Знак драхмы	₯	8367	\u20AF
Знак немецкого пенни	₰	8368	\u20B0

Существует ряд символов валют за пределами указанного выше диапазона Unicode. Соответствующие примеры приведены в табл. 2.9.

Таблица 2.9. Символы валют за пределами диапазона Unicode

Название	Символ	Десятичное	Unicode
Знак доллара	$	36	\u0024
Знак цента	¢	162	\u00A2
Знак фунта стерлингов	£	163	\u00A3
Знак валюты	¤	164	\u00A4
Знак иены	¥	165	\u00A5
Маленькая латинская f с крючком	ƒ	402	\u0192
Знак бенгальской рупии	৲	2546	\u09F2
Символ бенгальской рупии	৳	2547	\u09F3
Символ гуджаратской рупии	૱	2801	\u0AF1
Символ тамильской рупии	௹	3065	\u0BF9
Символ тайского бата	฿	3647	\u0E3F
Прописная рукописная M	ℳ	8499	\u2133
Унифицированный знак иероглифа 1	元	20803	\u5143
Унифицированный знак иероглифа 2	円	20870	\u5186
Унифицированный знак иероглифа 3	圓	22278	\u5706
Унифицированный знак иероглифа 4	圓	22291	\u5713

Основные типы

К основным типам относятся простые типы Java и соответствующие им типы интерфейсных классов и ссылочных типов. В Java 5.0 и более поздних версиях предусмотрено автоматическое преобразование между простыми и ссылочными типами посредством автоупаковки и распаковки. По мере необходимости к простым типам применяется числовое продвижение.

Простые типы

В языке Java существует восемь простых (primitive) типов, и для обозначения каждого из них предусмотрено определенное зарезервированное ключевое слово. С помощью этих ключевых слов описываются переменные, содержащие одиночные значения соответствующего формата и размера (табл. 3.1). Простые типы всегда имеют указанную в таблице точность, независимо от точности используемой аппаратной платформы (32- или 64-разрядная).

Таблица 3.1. Простые типы

Тип	Описание	Занимаемая память	Диапазон значений
`boolean`	Принимает значение `true` или `false`	1 бит	—
`char`	Представляет один символ Unicode	2 байта	От `\u0000` до `\uFFFF`
`byte`	Однобайтовое целое число	1 байт	От `-128` до `127`
`short`	Короткое целое число	2 байта	От `-32768` до `32767`
`int`	Целое число одинарной точности	4 байта	От `-2147483648` до `2147483647`
`long`	Целое двойной точности	8 байт	От -2^{63} до $2^{63}-1$

Окончание табл. 3.1

Тип	Описание	Занимаемая память	Диапазон значений
`float`	Число с плавающей точкой одинарной точности	4 байта	От $1.4e^{-45}$ до $3.4e^{+38}$
`double`	Число с плавающей точкой двойной точности	8 байт	От $5e^{-324}$ до $1.8e^{+308}$

НА ЗАМЕТКУ

Простые типы `byte`, `short`, `int`, `long`, `float` и `double` могут быть положительными или отрицательными. Тип `char` является беззнаковым.

Литералы простого типа

Все простые типы, за исключением `boolean`, могут задаваться в виде литералов символьного, десятичного, шестнадцатеричного и восьмеричного форматов, а также в виде символьных управляющих последовательностей и Unicode. При первом же удобном случае значение литерала автоматически приводится к соответствующему типу или преобразовывается в нужную форму. Не забывайте, что во время округления или усечения часть значащих битов теряется. Ниже приведен ряд примеров использования литералов в операторах присваивания значений простых типов.

```
boolean isTitleFight = true;
```

Для простого типа `boolean` допускаются только два значения литералов: `true` или `false`.

```
char[] cArray = {'\u004B', 'O', '\'', 0x0064,
 041,(char) 131105, 0b00100001}; // KO'd!!!
```

С помощью простого типа `char` задается один символ Unicode. Литеральные значения для типа `char`, большие двух байтов, необходимо явно привести к соответствующему типу.

```
byte rounds = 12, fighters = (byte) 2;
```

Для простого типа byte допускается использование целочисленных четырехбайтовых литералов со знаком. Если приведение типов не выполнено явно, то во время операции присваивания оно выполняется автоматически (в неявном виде) и число усекается до одного байта.

```
short seatingCapacity = 17157,
  vipSeats = (short) 500;
```

Для простого типа short допускается использование целочисленных четырехбайтовых литералов со знаком. Если приведение типов не выполнено явно, то во время операции присваивания оно выполняется автоматически (в неявном виде) и число усекается до двух байтов.

```
int ppvRecord = 19800000, vs = vipSeats,
  venues = (int) 20000.50D;
```

Для простого типа int допускается использование целочисленных четырехбайтовых литералов со знаком. Когда во время операции присваивания используются литералы простого типа char, byte и short, они автоматически приводятся к четырехбайтовым целым числам, как в случае использования переменной vipSeats типа short. Литералы с плавающей точкой и длинные литералы должны явно приводиться к соответствующему типу.

```
long wins = 38L, losses = 41, draws = 0,
  knockouts = (long) 30;
```

Для простого типа long допускается использование целочисленных восьмибайтовых литералов со знаком. Они обозначаются суффиксом L или l. Когда суффикс или операция явного приведения типов не указаны, соответствующее значение автоматически преобразуется из четырех байтов в восемь.

```
float payPerView = 54.95F, balcony = 200.00f,
      ringside = (float) 2000, cheapSeats = 50;
```

Для простого типа float допускается использование четырехбайтовых литералов с плавающей точкой со знаком.

Они обозначаются суффиксом F или f или задаются как результат операции явного приведения типов. Несмотря на то что для литерала типа int явное приведение типов не требуется, значение типа int не будет помещаться в переменную типа float, если оно превышает 2^{23}.

```
double champsPay = 20000000.00D,
       challengersPay = 12000000.00d,
       chlTrainerPay = (double) 1300000,
       refereesPay = 3000, soda = 4.50;
```

Для простого типа double допускается использование восьмибайтовых литералов с плавающей точкой со знаком. Они обозначаются суффиксом D или d; в случае использования результата явного приведения типов суффикс не указывается. Если литерал является целым числом, приведение выполняется неявно.

Подробнее о литералах см. в главе 2.

Сущности с плавающей точкой

Чтобы удовлетворить требования стандарта IEEE 754-1985, в язык Java были введены положительные и отрицательные бесконечно большие величины с плавающей точкой, отрицательный нуль и признак не числа NaN (Not-a-Number), которые являются особыми сущностями (табл. 3.2).

Если в результате выполнения какой-либо операции с плавающей точкой получается слишком большое либо слишком малое значение, которое нельзя представить традиционным способом, то вместо него будут возвращены сущности Infinity, -Infinity и -0.0.

Таблица 3.2. Сущности с плавающей точкой

Сущность	Описание	Примеры
Infinity	Представляет концепцию положительной бесконечности	1.0/0.0, 1e300/1e-300, Math.abs(-1.0/0.0)

Окончание табл. 3.2

Сущность	Описание	Примеры
`-Infinity`	Представляет концепцию отрицательной бесконечности	`-1.0/0.0,1.0/(-0.0), 1e300/-1e-300`
`-0.0`	Представляет отрицательное число, близкое к нулю	`-1.0/(1.0/0.0), -1e-300/1e300`
`NaN`	Представляет неопределенные результаты	`0.0/0.0,1e300*Float.NaN, Math.sqrt(-9.0)`

Сущности `Infinity`, `-Infinity` и `NaN` могут быть представлены в программах в виде констант с плавающей точкой одинарной и двойной точности, как показано ниже

```
Double.POSITIVE_INFINITY; // Infinity
Float.POSITIVE_INFINITY; // Infinity

Double.NEGATIVE_INFINITY; // -Infinity
Float.NEGATIVE_INFINITY; // -Infinity

Double.NaN; // NaN
Float.NaN; // NaN
```

Для интерфейсных классов `Double` и `Float` предусмотрены методы, которые позволяют определить, является ли рассматриваемое число конечным, бесконечно большим или `NaN`.

```
Double.isFinite(Double.POSITIVE_INFINITY); // false
Double.isFinite(Double.NEGATIVE_INFINITY); // false
Double.isFinite(Double.NaN); // false
Double.isFinite(1); // true

Double.isInfinite(Double.POSITIVE_INFINITY); // true
Double.isInfinite(Double.NEGATIVE_INFINITY); // true
Double.isInfinite(Double.NaN); // false
Double.isInfinite(1); // false

Double.isNaN(Double.NaN); // true
Double.isNaN(1); // false
```

Операции с особыми сущностями

В табл. 3.3 представлены результаты операций, в которых задействованы особые сущности. В этой таблице аббревиатура операндов INF обозначает Double.POSITIVE_INFINITY, -INF обозначает Double.NEGATIVE_INFINITY, а NAN — Double.NaN.

Например, результатом выполнения операции, которая находится на пересечении 4-го столбца, озаглавленного (-0.0), и 12-й строки, озаглавленной (*NAN), является NaN. Это можно представить в виде следующего оператора Java:

```
// Будет выведено 'NaN'
System.out.print((-0.0) * Double.NaN);
```

Таблица 3.3. Операции с особыми сущностями

	INF	(-INF)	(-0.0)
*INF	Infinity	-Infinity	NaN
+INF	Infinity	NaN	Infinity
-INF	NaN	-Infinity	-Infinity
/INF	NaN	NaN	-0.0
*0.0	NaN	NaN	-0.0
+0.0	Infinity	-Infinity	0.0
+0.5	Infinity	-Infinity	0.5
*0.5	Infinity	-Infinity	-0.0
+(-0.5)	Infinity	-Infinity	-0.5
*(-0.5)	-Infinity	Infinity	0.0
+NAN	NaN	NaN	NaN
*NAN	NaN	NaN	NaN

НА ЗАМЕТКУ

Результатом любой операции по отношению к NaN является NaN. Такого понятия, как -NaN, вообще не существует.

Числовое продвижение простых типов

Числовое продвижение состоит из правил, которые применяются к операндам какой-либо арифметической операции при

определенных условиях. Правила числового продвижения включают как унарные, так и бинарные правила продвижения.

Унарное числовое продвижение

Когда какой-либо операнд простого типа является частью того или иного выражения, как указано в табл. 3.4, применяются правила продвижения, перечисленные ниже.

- Если операнд имеет тип `byte`, `short` или `char`, то он продвигается к типу `int`.
- В противном случае тип операнда остается неизменным.

Таблица 3.4. Выражения для унарных правил продвижения

Выражение
Унарный плюс +
Унарный минус –
Побитовое дополнение ~
Все операторы сдвига >>, >>> или <<
Индексное выражение при доступе к массиву
Выражение, определяющее размер при создании массива

Бинарное числовое продвижение

Когда два числовых операнда разных простых типов сравниваются между собой с помощью операций, перечисленных в табл. 3.5, один тип продвигается на основе перечисленных ниже бинарных правил продвижения.

- Если один из операндов относится к типу `double`, то операнд простого типа, не относящийся к `double`, преобразовывается в тип `double`.
- Если один из операндов относится к типу `float`, то операнд простого типа, не относящийся к `float`, преобразовывается в тип `float`.

- Если один из операндов относится к типу `long`, то операнд простого типа, не относящийся к `long`, преобразовывается в тип `long`.
- В противном случае оба операнда преобразовываются в тип `int`.

Таблица 3.5. Операторы для бинарных правил продвижения

Операторы	Описание
`+` и `-`	Аддитивные операторы
`*`, `/` и `%`	Мультипликативные операторы
`<`, `<=`, `>` и `>=`	Операторы сравнения
`==` и `!=`	Операторы равенства
`&`, `^` и `|`	Побитовые операторы
`?:`	Условный оператор (см. следующий раздел)

Особые случаи для условных операторов

- Если один операнд относится к типу `byte`, а другой — к типу `short`, то условное выражение будет относиться к типу `short`.

```
short = true ? byte : short
```

- Если один операнд *R* относится к типу `byte`, `short` или `char`, а другой является константным выражением типа `int`, значение которого находится в диапазоне *R*, то условное выражение будет относиться к типу *R*.

```
short = (true ? short : 1967)
```

- В противном случае применяется бинарное числовое продвижение, а тип условного выражения будет соответствовать продвинутому типу второго и третьего операндов.

Интерфейсные классы

Каждому из простых типов соответствует определенный тип интерфейсного класса и ссылочного типа, который определен в пакете `java.lang`. У каждого интерфейсного класса есть

определенная совокупность методов, в том числе метод, возвращающий значение типа, как показано в табл. 3.6. Методы этих интерфейсных классов, как целочисленные, так и с плавающей точкой, могут возвращать значения нескольких простых типов.

Таблица 3.6. Интерфейсные классы

Простой тип	Ссылочный тип	Методы, возвращающие простой тип
`boolean`	`Boolean`	`booleanValue()`
`char`	`Character`	`charValue()`
`byte`	`Byte`	`byteValue()`, `shortValue()`, `intValue()`, `longValue()`, `floatValue()`, `doubleValue()`
`short`	`Short`	Аналогичны типу `byte`
`int`	`Integer`	Аналогичны типу `byte`
`long`	`Long`	Аналогичны типу `byte`
`float`	`Float`	Аналогичны типу `byte`
`double`	`Double`	Аналогичны типу `byte`

Автоупаковка и распаковка

Автоупаковка и распаковка обычно используются при создании коллекций простых типов. Автоупаковка предполагает динамическое распределение памяти и инициализацию того или иного объекта для каждого простого типа. Обратите внимание на то, что непроизводительные издержки могут зачастую превосходить время выполнения желаемой операции. Распаковка предполагает создание значения простого типа для каждого объекта.

Задачи, требующие значительного объема вычислений, в которых используются простые типы (например, такие, как перебор элементов в каком-либо контейнере), должны выполняться с использованием массивов значений простых типов, а не коллекций объектов интерфейсных классов.

Автоупаковка

Автоупаковка представляет собой автоматическое преобразование значений простых типов в соответствующие им объекты

интерфейсных классов. В приведенном ниже примере вес боксера автоматически преобразуется в соответствующий ему интерфейсный класс, поскольку в коллекциях хранятся ссылки на объекты, а не сами значения простых типов.

```
// Создать хеш-массив весовых категорий
HashMap<String, Integer> weightGroups =
                         new HashMap<String, Integer> ();
weightGroups.put("Второй полусредний вес", 67);
weightGroups.put("Средний вес", 73);
weightGroups.put("Полутяжелый вес ", 80);
```

В следующем примере проиллюстрировано приемлемое, но не рекомендуемое использование автоупаковки:

```
// Задать весовое ограничение
Integer weightAllowanceW = 5; //не рекомендуется
```

НА ЗАМЕТКУ

В приведенном выше примере для удобства имя переменной интерфейсного (wrapper) класса заканчивается прописной буквой W. Это не является соглашением.

Поскольку в данном случае нет настоятельной потребности в использовании автоупаковки, предыдущий оператор следовало бы записать так:

```
Integer weightAllowanceW = new Integer (5);
```

Распаковка

Распаковка представляет собой автоматическое преобразование объектов интерфейсных классов в соответствующие им значения простого типа. В приведенном ниже примере из хеш-массива весовых категорий, созданного в предыдущем разделе, по заданному ключу извлекается объект ссылочного типа. Он автоматически распаковывается так, чтобы полученное значение соответствовало требуемому простому типу.

```
// Получить значение весового ограничения
int weightLimitP = weightGroups.get("Средний вес");
```

НА ЗАМЕТКУ

В приведенном выше примере для удобства имя переменной, содержащей значение распакованного класса, заканчиваются прописной буквой P. Это не является соглашением.

В приведенном ниже примере проиллюстрировано приемлемое, но не рекомендуемое использование распаковки.

```
// Задать весовое ограничение
weightLimitP = weightLimitP + weightAllowanceW;
```

Лучше было бы записать данное выражение с помощью метода `intValue()`, как показано ниже.

```
weightLimitP = weightLimitP +
               weightAllowanceW.intValue();
```

Ссылочные типы

В переменных ссылочного типа хранятся ссылки на объекты. По сути они служат средством доступа к этим объектам, хранящимся где-то в оперативной памяти. Точное расположение объектов в оперативной памяти для программистов не имеет особого значения. Все ссылочные типы представляют собой производный класс от класса `java.lang.Object`.

В табл. 4.1 перечислены пять ссылочных типов Java.

Таблица 4.1. Ссылочные типы

Ссылочный тип	Описание
`Annotation`	Позволяет связать метаданные (данные о данных) с элементами программы
`Array`	Обеспечивает структуру данных фиксированного размера, в которой хранятся элементы одного типа
`Class`	Предназначен для обеспечения наследования, полиморфизма и инкапсуляции. Обычно моделирует что-то в реальном мире и состоит из совокупности свойств (переменных экземпляра, или объектных переменных), содержащих данные, и методов, выполняющих операции над этими данными
`Enumeration`	Ссылка на набор объектов, представляющих набор связанных между собой значений
`Interface`	Формирует открытый программный интерфейс приложения (API), который реализуется с помощью других классов Java

Сравнение ссылочных и простых типов

В языке Java есть две категории типов: ссылочные и простые. В табл. 4.2 приведены их основные отличия. Более подробную информацию см. в главе 3.

Таблица 4.2. Сравнение ссылочных и простых типов

Ссылочный тип	Простой тип
Неограниченное количество ссылочных типов, поскольку они определяются пользователем	Состоит из булевого и числового типов: `char`, `byte`, `short`, `int`, `long`, `float` и `double`
В переменной данного типа хранится ссылка на данные	В переменной данного типа хранятся фактические данные соответствующего простого типа
При присвоении значения переменной ссылочного типа другой переменной ссылочного типа обе переменные будут содержать ссылку на один и тот же объект в памяти	При присвоении значения переменной простого типа другой переменной простого типа создается копия данных
При передаче объекта ссылочного типа в тот или иной метод вызванный метод может изменить содержимое этого объекта, но ссылка на этот объект останется неизменной	При передаче значения переменной простого типа в тот или иной метод будет передана только его копия. Вызванный метод не имеет доступа к исходному значению переменной и, следовательно, не может изменить его. Вызванный метод может изменить только скопированное значение, которое по завершении метода теряется

Стандартные значения

Стандартные значения — это значения, автоматически присваиваемые переменным экземпляра в Java, когда для них не было явно указано какое-либо первоначальное значение.

Переменные экземпляра и локальные переменные

Стандартным значением переменных экземпляра (т.е. переменных, объявленных на уровне класса) является `null`. Нулевая ссылка ни на что не ссылается.

Для локальных переменных (т.е. переменных, объявленных в каком-либо методе) стандартное значение не присваивается (даже `null`). Поэтому локальные переменные нужно инициализировать явно. Попытка использования неинициализированной локальной переменной приведет к ошибке во время компиляции (даже если ей неявно присвоено значение `null`).

Если переменной ссылочного типа присвоено значение null, то она не будет ссылаться ни на какой объект, расположенный в динамической памяти программы (так называемой *куче*). Тем не менее такие переменные можно свободно использовать в коде программы, поскольку это не вызовет ошибок во время компиляции или выполнения, как показано ниже.

```
Date dateOfParty = null;
// Это будет скомпилировано
if (dateOfParty == null) {
...
}
```

Если значение переменной ссылочного типа равно null, то при ее использовании для вызова метода объекта или доступа к данным-членам с помощью оператора-точки возникает исключение java.lang.NullPointerException:

```
private static int MAX_LENGTH = 20;
...
String theme = null;

// При выполнении оператора if генерируется исключение,
// поскольку значение переменной theme равно null
if (theme.length() > MAX_LENGTH) {
     ...
}
```

Массивы

Переменным, содержащим ссылки на массив, всегда присваивается стандартное значение: и в случаях, когда они объявлены как переменные экземпляра, и в случаях, когда они объявлены как локальные переменные. Если массив объявлен, но не проинициализирован, то соответствующей переменной ссылочного типа присваивается стандартное значение null.

В приведенном ниже коде объявлены массивы gameList1 и gameList2, элементы которых должны содержать ссылки на объекты типа Game. Поэтому сразу после объявления переменным gameList1 и gameList2 присваиваются стандартные значения

null. После размещения в памяти массива gameList1 с помощью оператора new переменной gameList1 присваивается ссылка на этот массив. Сразу после размещения в памяти все элементы этого массива пусты, т.е. не содержат никаких ссылок на объекты типа Game. Поэтому им также были присвоены стандартные значения null. После добавления объектов к массиву ситуация меняется, как показано ниже.

```
// Массивы gameList1 и gameList2 объявлены, но не
// размещены в памяти. Поэтому переменным gameList1
// и gameList2 присвоены стандартные значения null
Game[] gameList1;
Game gameList2[];

// В следующем операторе массив gameList1 из 10 элементов
// размещается в памяти, но значения всех его элементов
// по-прежнему равны null, поскольку этот массив
// не содержит ссылок на какие-либо объекты типа Game
gameList1 = new Game[10];

// Объект типа Game добавляется в массив. Теперь
// значение первого элемента массива не равно null.
gameList1[0] = new Game();
```

Многомерные массивы в Java являются, по сути, массивами массивов. Их можно инициализировать с помощью оператора new или оператора присваивания, в котором индексы массивов указаны в квадратных скобках. Многомерные массивы могут быть по своей форме однородными или неоднородными.

```
// Анонимный массив
int twoDimensionalArray[][] = new int[6][6];
twoDimensionalArray[0][0] = 100;

int threeDimensionalArray[][][] = new int[2][2][2];
threeDimensionalArray[0][0][0] = 200;

int varDimensionArray[][] = {{0,0},{1,1,1},{2,2,2,2}};
varDimensionArray[0][0] = 300;
```

Анонимные массивы позволяют создать новый массив значений в любом месте кода программы.

```
// Примеры использования анонимных массивов
int[] luckyNumbers = new int[] {7, 13, 21};
int totalWinnings = sum(new int[] {3000, 4500, 5000});
```

Преобразование ссылочных типов

Любой объект может быть преобразован в тип его суперкласса (расширение) или какого-либо из его подклассов (сужение).

Компилятор проверяет такие преобразования в ходе компиляции, а *виртуальная машина Java* (Java Virtual Machine — JVM) проверяет преобразования в ходе выполнения программы.

Расширяющие преобразования

- При расширении подкласс неявно преобразуется в его родительский класс (суперкласс).
- Расширяющие преобразования не генерируют исключения во время выполнения программы.
- При этом не требуется явное приведение к нужному типу.

```
String s = new String();
Object o = s; // Расширение
```

Сужающие преобразования

- При сужении более общий тип преобразуется в более конкретный тип.
- Сужение представляет собой преобразование суперкласса в подкласс.
- При этом требуется явное приведение к нужному типу. Для этого заключите в круглые скобки тип, к которому нужно привести, и после него укажите объект приведения.
- В случае некорректного преобразования генерируется исключение `ClassCastException`.
- Сужение может привести к потере данных или точности.

Объект нельзя преобразовывать в неродственный тип — т.е. тип, отличный от типа его подклассов или суперклассов.

Попытка преобразовать объект в неродственный тип приведет к выдаче сообщения об ошибке о не конвертируемости типов (inconvertible types) во время компиляции. Ниже приведен пример преобразования, которое приведет к ошибке inconvertible types во время компиляции.

```
Object c = new Object();
String d = (Integer) c; // Ошибка во время компиляции
```

Преобразования между простыми и ссылочными типами

Автоматическое преобразование простых типов в ссылочные называется *автоупаковкой*, а обратное преобразование — *распаковкой*. Более подробно об этом см. в главе 3.

Передача ссылочных типов в методы

Когда какой-либо объект передается в тот или иной метод в виде переменной, происходит описанная ниже последовательность действий.

- Методу передается копия ссылочной переменной, а не сам объект.
- И в вызывающем, и в вызываемом методе оперируют с идентичными копиями ссылок.
- Любые изменения, вносимые вызываемым методом в соответствующий объект, будут "видны" в вызывающем методе. Чтобы предотвратить внесение каких-либо изменений в исходный объект, в вызываемом методе ему следует передать копию этого объекта.
- В вызываемом методе нельзя изменить ссылку на переданный ему объект, можно изменить только содержимое этого объекта.

В приведенном ниже примере проиллюстрирована передача переменных ссылочного и простого типов в методы, а также

значение этих переменных после вызова методов, изменяющих внутри эти значения.

```
void roomSetup() {

   // Передача ссылки на таблицу
   Table table = new Table();
   table.setLength(72);

   // Длина будет изменена
   modTableLength(table);

   // Передача значения
   // Значение переменной chairs не изменится
   int chairs = 8;
   modChairCount(chairs);
}

void modTableLength(Table t) {
      t.setLength(36);
}

void modChairCount(int i) {
      i = 10;
}
```

Сравнение переменных ссылочного типа

Переменные ссылочного типа в Java можно сравнивать. Это выполняется с помощью операторов равенства и метода `equals()`.

Использование операторов равенства

Операторы равенства `!=` и `==` используются для сравнения указателей (т.е. адресов в оперативной памяти) двух объектов. Если адреса в оперативной памяти сравниваемых объектов совпадают, то такие объекты считаются идентичными друг другу. Эти операторы равенства не используются для сравнения содержимого двух объектов.

В приведенном ниже примере объекты guest1 и guest2 имеют один и тот же адрес в оперативной памяти, поэтому в результате сравнения будет выведена строка "Равны между собой!".

```
Guest guest1 = new Guest("имя");
Guest guest2 = guest1;
if (guest1 == guest2)
   System.out.println("Равны между собой!");
```

В приведенном ниже примере объекты имеют разные указатели, поэтому адреса в оперативной памяти не равны между собой. В результате сравнения будет выведена строка "Не равны между собой!":

```
Guest guest3 = new Guest("имя");
Guest guest4 = new Guest("имя");
if (guest3 == guest4)
   System.out.println("Равны между собой!")
else
   System.out.println("Не равны между собой!")
```

Использование метода equals()

Для сравнения между собой содержимого объектов двух классов можно использовать стандартный метод equals() класса Object или любой его переопределенный метод. При переопределении метода equals() в производном классе нужно обязательно также переопределить и метод hashCode(). Это делается для обеспечения совместимости с хеш-коллекциями, такими как HashMap и HashSet.

НА ЗАМЕТКУ

В стандартной реализации метода equals() для сравнения объектов используется только оператор ==. Чтобы этот метод был по-настоящему полезным, его нужно переопределить.

Например, если нужно сравнить значения, содержащиеся в двух экземплярах одного и того же класса, метод equals() должен быть переопределен программистом.

Сравнение строк

Существуют два способа проверки равенства строк в Java, однако понятие равенства в них определяется по-разному. Как правило, если цель заключается в том, чтобы сравнить последовательности символов, содержащиеся в двух строках, следует использовать метод `equals()`.

- Чтобы установить факт равенства или неравенства, в методе `equals()` нужно сравнить две строки, символ за символом. Поэтому стандартная реализации метода `equals()` в классе `Object` здесь не годится. Нужно воспользоваться переопределенной реализацией этого метода в классе `String`.
- С помощью оператора == можно проверить равенство двух ссылок, т.е. убедиться, что две переменные ссылочного типа указывают на один и тот же экземпляр объекта.

Ниже приведен пример программы, в котором показано, как с помощью метода `equals()` и оператора == оцениваются значения двух строк (подробнее о том, как хранятся значения строк, см. в разделе "Строковые литералы" главы 2).

```
class MyComparisons {

   // Добавить строку в пул
   String first = "Стулья";

   // Использовать строку из пула
   String second = "Стулья";

   // Создать новую строку
   String third = new String ("Стулья");

   void myMethod() {

      // Вопреки широко распространенному мнению, это
      // выражение истинно. Убедитесь сами!
      if (first == second) {
         System.out.println("first == second");
      }

      // И это выражение истинно.
```

```
        if (first.equals(second)) {
            System.out.println("first равен second");
        }
        // А это выражение будет ложно,
        // несмотря на то, что строки равны!
        if (first == third) {
            System.out.println("first == third");
        }
        // Зато это выражение истинно
        if (first.equals(third)) {
            System.out.println("first равен third");
        }
    } // Конец myMethod()
} //конец class
```

НА ЗАМЕТКУ

Объекты класса `String` нельзя изменить. Для этого существуют объекты классов `StringBuffer` и `StringBuilder`.

Сравнение перечислений

Значения класса `enum` можно сравнивать с помощью оператора == или метода `equals()`, поскольку они возвращают один и тот же результат. Для сравнения перечисляемых типов оператор == используется чаще.

Копирование переменных ссылочного типа

При копировании переменных ссылочного типа создается либо копия ссылки на один и тот же объект, либо новый объект. Последний вариант называется в Java *клонированием*.

Копирование ссылки на объект

При копировании ссылки на объект получается один объект с двумя ссылками. В приведенном ниже примере переменной

closingSong присваивается ссылка на объект, хранящаяся в переменной lastSong. Любые изменения, касающиеся объекта lastSong, отразятся в closingSong, и наоборот.

```
Song lastSong = new Song();
Song closingSong = lastSong;
```

Клонирование объектов

Клонирование приводит к созданию еще одной копии соответствующего объекта, а не просто копии ссылки на этот объект. В стандартной реализации классов операция клонирования недоступна. Обратите внимание: клонирование обычно является очень сложной операцией, поэтому вместо него следует воспользоваться конструктором копирования.

- Чтобы тот или иной класс поддерживал клонирование, в нем должен быть реализован интерфейс Cloneable.
- С помощью защищенного метода clone() объекты могут клонировать самих себя.
- Чтобы какой-либо объект мог клонировать какой-то другой объект (не самого себя), в клонируемом объекте нужно переопределить метод clone() и сделать его открытым (public).
- При выполнении клонирования необходимо использовать явное приведение типов, поскольку метод clone() возвращает объект типа object.
- При клонировании может генерироваться исключение CloneNotSupportedException.

Поверхностное и глубокое клонирование

В языке Java поддерживаются два типа клонирования — поверхностное и глубокое.

При поверхностном клонировании выполняется обычное копирование значений переменных простого и ссылочного типа в клонируемом объекте. Копирование объектов, на которые указывают эти ссылки, не выполняется.

В приведенном ниже примере в объект leadingSong будут скопированы значения переменных простого типа length и year и ссылочных перемененных title и artist.

```
Class Song {
   String title;
   Artist artist;
   float length;
   int year;
   void setData() {...}
}

Song firstSong = new Song();

try {
   // Получить новую копию объекта путем клонирования
   Song leadingSong = (Song) firstSong.clone();
}
catch (CloneNotSupportedException cnse) {
   cnse.printStackTrace();
}   // конец
```

При глубоком клонировании в новом объекте создается копия каждого из объектов, на которые ссылается исходный объект. Метод глубокого клонирования должен быть определен программистом, поскольку в API Java такой не предусмотрен. Альтернативами глубокого клонирования являются сериализация и конструкторы копирования, причем чаще всего предпочтение отдается последним.

Распределение памяти и сборка мусора

Когда создается какой-либо новый объект, для него выделяется определенная память. Когда на объект нет ссылок, память, используемая этим объектом, может быть "утилизирована" в процессе сборки мусора. Подробнее об этом речь пойдет в главе 11.

Объектно-ориентированное программирование

Основными элементами объектно-ориентированного программирования (ООП) в Java являются классы, объекты и интерфейсы.

Классы и объекты

Классы определяют логические категории, которые обычно представляют что-то в реальном мире. Они состоят из совокупности характеристик, которые представлены в виде данных, и определенного набора методов, оперирующих с этими данными.

Экземпляр класса называется *объектом*, и для каждого объекта выделяется память. Может существовать несколько экземпляров одного класса.

Классы могут наследовать данные и методы из других классов. Каждый класс может непосредственно унаследовать данные и методы только из одного класса, называемого *суперклассом*. У класса может быть только один прямой предок (суперкласс). Этот процесс называется *наследованием*.

При реализации класса все внутренние детали этой реализации должны быть скрыты (иметь атрибут `private`), а доступ к ним должен осуществляться лишь посредством открытых интерфейсов. Этот процесс называется *инкапсуляцией*. Существует специальное соглашение, принятое в JavaBean, которое заключается в том, что к закрытым членам класса можно получить только косвенный доступ через специальные методы чтения/записи (так называемые get-/set-методы, или методы доступа), например `getFirstName()` и `setFirstName("Иван")`. Это позволяет гарантировать, что никакой другой класс не сможет случайно

изменить закрытые члены класса. Существует и другой способ защиты данных от непреднамеренного изменения со стороны других объектов — использование неизменяемых значений, таких как строки, значения простых типов данных, а также объекты, доступные только для чтения.

Синтаксис класса

У каждого класса есть идентифицирующая его сигнатура, один или несколько необязательных конструкторов, данные-члены и методы.

```
[Модификаторы]    class ИмяКласса
                  [extends ИмяСуперКласса]
                  [implements ИменаИнтерфейсов через запятую]
{
                  // Данные-члены
                  // Конструктор(ы)
                  // Метод(ы)
}
```

Создание экземпляра класса (объекта)

Объект является экземпляром класса. После создания у объекта появляется собственный набор данных-членов и методов.

```
// Примеры определений класса
public class Candidate {...}
class Stats extends ToolSet {...}

public class Report extends ToolSet
    implements Runnable {...}
```

Отдельные объекты (экземпляры) класса `Candidate` создаются с помощью ключевого слова `new`.

```
Candidate candidate1 = new Candidate();
Candidate candidate2 = new Candidate();
```

Данные-члены и методы

В данных-членах, которые иногда называют также полями (*fields*) или свойствами (*properties*), содержится вся информация о классе. Нестатические данные-члены называются также *переменными экземпляра*.

```
[Модификаторы] type ИмяПеременной
```

Методы оперируют с данными класса.

```
[Модификаторы] type ИмяМетода (СписокПараметров)
        [throws СписокИсключений через запятую] {
        // Тело метода
}
```

Ниже приведен пример класса `Candidate`, а также его данных-членов и методов.

```
public class Candidate {
   // Данные-члены, или поля
   private String firstName;
   private String lastName;
   private int year;

   // Методы
   public void setYear (int y) { year = y; }
   public String getLastName() {return lastName;}
} // Конец класса Candidate
```

Доступ к данным-членам и методам объектов

Для доступа к данным-членам и методам объектов используется оператор точка (`.`). При доступе к данным-членам или методам внутри объекта использовать точку необязательно.

```
candidate1.setYear(2016);
String name = getFirstName() + getLastName();
```

Перегрузка

Методы, включая конструкторы, могут быть перегружены. *Перегрузка* означает, что два метода или более имеют одинаковые

имена, но разные *сигнатуры* (список передаваемых и возвращаемых значений). Обратите внимание на то, что перегруженные методы должны иметь разный список входных параметров и *могут* возвращать разные типы значений. Однако если метод лишь возвращает разные типы значений, то это не является перегрузкой. У перегруженных методов могут быть разные модификаторы доступа.

```
public class VotingMachine {
   ...
    public void startUp() {...}
   private void startUp(int delay) {...}
}
```

При перегрузке какого-либо метода в его сигнатуре можно указать, что метод генерирует разные виды проверяемых исключений.

```
private String startUp(District d) throws new
                                    IOException {...}
```

Переопределение

В подклассе можно переопределить методы, унаследованные из родительского класса (суперкласса). В результате метод будет иметь ту же сигнатуру (имена и типы передаваемых и возвращаемых значений), что и его аналог в суперклассе, однако детали их реализации будут отличаться.

Например, вот как можно переопределить метод `startUp()` из суперкласса `Display` в классе `TouchScreenDisplay`:

```
public class Display {
   void startUp(){
    System.out.println("Используется основной монитор.");
   }
}

public class TouchScreenDisplay extends Display {
   void startUp() {
     System.out.println("используется сенсорный экран.");
   }
}
```

Ниже перечислены правила переопределения методов.

- Переопределять можно только методы, не являющиеся завершенными (`final`), закрытыми (`private`) или статическими (`static`).
- Защищенные (`protected`) методы могут переопределять только методы, у которых не указаны модификаторы доступа.
- У переопределяющего метода не может быть более жесткого модификатора доступа (т.е. `package`, `public`, `private`, `protected`), чем у исходного метода.
- В переопределяющем методе не могут возбуждаться какие-либо новые проверяемые исключения.

Конструкторы

Конструкторы вызываются после создания объекта и используются для инициализации данных во вновь созданном объекте. Конструкторы являются необязательным средством, имеют в точности такое же имя, что и соответствующий класс, а в их теле, в отличие от методов, не указывается оператор `return`.

У класса может быть несколько конструкторов. При этом при создании нового объекта будет вызван конструктор, имеющий подходящую сигнатуру.

```
public class Candidate {
   ...
   Candidate(int id) {
      this.identification = id;
   }

   Candidate(int id, int age) {
      this.identification = id;
      this.age = age;
   }
}

// Создаем новый экземпляр класса Candidate и
//  вызываем его конструктор
class ElectionManager {
```

```
    int id = getIdFromConsole();
    Candidate candidate = new Candidate(id);
}
```

Если при определении класса конструктор не задан явно, то предполагается, что у класса неявно существует стандартный конструктор без аргументов. Обратите внимание: если был явно задан конструктор с аргументами, то предполагается, что неявного конструктора без аргументов не существует, хотя вы всегда можете добавить его вручную.

Суперклассы и подклассы

В Java каждый класс (называемый *подклассом*) может непосредственно унаследовать данные и методы только из одного класса, называемого *суперклассом*. Для этого используется ключевое слово `extends`. При определении класса оно указывает, что некий класс наследует данные-члены и методы из указанного за ним класса. У подклассов нет непосредственного доступа к закрытым (`private`) членам суперкласса, но есть доступ к открытым (`public`) и защищенным (`protected`) его членам. У подкласса есть также доступ к закрытым на уровне пакета или защищенным членам суперкласса, в которых совместно используется общий пакет. Как указывалось ранее, доступ к закрытым (`private`) членам любого класса, в том числе и суперкласса, возможен только косвенно через специальные методы чтения/записи (get-/set-методы).

```
public class Machine {
    boolean state;
    void setState(boolean s) {state = s;}
    boolean getState() {return state;}
}

public class VotingMachine extends Machine {
    ...
}
```

Для доступа к тем методам суперкласса, которые переопределены в текущем подклассе, используется ключевое слово `super`. Например, оно используется в переопределенном методе

printSpecs() класса Curtain для вызова исходного метода printSpecs() класса PrivacyWall, как показано ниже.

```
public class PrivacyWall {
   public void printSpecs() {...}
}

public class Curtain extends PrivacyWall {
   public void printSpecs() {
      ...
      super.printSpecs();
   }
}
```

Ключевое слово super обычно используется для вызова конструктора суперкласса и передачи ему параметров. Но этот вызов должен быть первым в том конструкторе, где используется ключевое слово super.

```
public class PrivacyWall {
   public PrivacyWall(int l, int w) {
      int length = l;
      int width = w;
      }
}

public class Curtain extends PrivacyWall {
   // Установить стандартные длину и ширину
   public Curtain() {super(15, 25);}
}
```

Если в конструкторе подкласса отсутствует явный вызов конструктора суперкласса, то выполняется автоматический вызов конструктора суперкласса без параметров.

Ключевое слово this

Ключевое слово this чаще всего используется для ссылки на текущий объект, для вызова конструктора из другого конструктора в том же классе и для передачи ссылки на текущий объект другому объекту.

Ниже показано, как можно присвоить значение параметра, переданного методу, переменной экземпляра текущего объекта.

```
public class Curtain extends PrivacyWall {
   String color;
   public void setColor(String color) {
      this.color = color;
   }
}
```

Для вызова конструктора в другом конструкторе в том же классе используйте следующий фрагмент кода:

```
public class Curtain extends PrivacyWall {
   public Curtain(int length, int width) {}
   public Curtain() {this(10, 9);}
}
```

Ниже показано, как можно передать ссылку на текущий объект другому объекту.

```
public class Curtain {
   Builder builder = new Builder();
   builder.setWallType(this);

   //Выведем на печать содержимое класса Curtain
     System.out.println(this);
}
public class Builder {
   public void setWallType(Curtain c) {...}
}
```

Списки с переменным числом параметров

Начиная с Java 5.0 у метода может быть переменное число параметров. При реализации метода *с переменным числом параметров* (*varargs*) нужно учитывать, что при вызове этого метода только последний параметр может повторяться несколько раз либо отсутствовать вовсе. Этот параметр может относиться к одному из простых типов либо к объекту. При объявлении метода с переменным числом параметров в списке аргументов его сигнатуры используется многоточие (...):

Ниже приведен пример сигнатуры для метода с переменным числом параметров.

```
public setDisplayButtons(int row,
        String... names) {...}
```

Компилятор Java модифицирует методы с переменным числом параметров так, чтобы они были похожи на обычные методы. Приведенный выше пример в процессе компиляции был бы модифицирован так:

```
public setDisplayButtons(int row,
   String [] names) {...}
```

У метода с переменным числом параметров может не быть других параметров, кроме списка с переменным числом параметров.

```
// Строк может быть несколько или их вообще может не быть
public void setDisplayButtons (String... names)
{...}
```

Метод с переменным числом параметров вызывается так же, как и обычный метод, за исключением того, что ему можно передать произвольное количество аргументов, причем повторяться может только последний из них.

```
setDisplayButtons("Иван");
setDisplayButtons("Федор", "Мария", "Петр");
setDisplayButtons("Светлана", "Дмитрий", "Владимир",
"Иван");
```

Для вывода форматированного значения переменных часто используется метод `printf()`, которому передается список с переменным числом параметров. Он определяется в Java API следующим образом:

```
public PrintStream printf(String format,
   Object... args)
```

При вызове метода `printf()` ему передается строка определения формата и переменное число объектов.

```
System.out.printf("Привет, избиратель %s%n
   Это автомат %d%n", "Ирина", 1);
```

За получением подробной информации о строке определения формата, передаваемой в метод `printf()`, обратитесь к документации по классу `java.util.Formatter`.

Для перебора всех аргументов списка с переменным числом параметров часто используется расширенный цикл `for`.

```
void printRows(String... names) {
   for (String name: names)
      System.out.println(name);
}
```

Абстрактные классы и методы

Абстрактные классы и методы объявляются с помощью ключевого слова `abstract`.

Абстрактные классы

Абстрактный класс обычно используется как некий базовый класс, поэтому нельзя создавать экземпляры объектов этого класса. В нем могут находиться абстрактные и неабстрактные методы, и он может являться подклассом какого-либо абстрактного или неабстрактного класса. Все его абстрактные методы должны быть реализованы в классах, которые наследуют (`extend`) его, если только новый подкласс также не является абстрактным.

```
public abstract class Alarm {
   public void reset() {...}
   public abstract void renderAlarm();
}
```

Абстрактные методы

В абстрактном методе содержится только объявление самого метода. Само тело метода должно быть реализовано в каком-либо

неабстрактном классе, который унаследовал класс, содержащий этот абстрактный метод.

```
public class DisplayAlarm extends Alarm {
   public void renderAlarm() {
      System.out.println("Звонит будильник!");
   }
}
```

Статические данные-члены, методы, константы и инициализаторы

Статические данные-члены, методы, константы и инициализаторы относятся к самим классам, а не экземплярам объектов данного класса. Доступ к статическим данным-членам, методам и константам возможен как из самого класса, в котором они определены, так и из какого-либо другого класса с помощью оператора-точки.

Статические данные-члены

Статические данные-члены имеют те же особенности, что и статические методы, кроме того, они хранятся в памяти только в одном экземпляре.

Они используются в тех случаях, когда для всех экземпляров класса нужна всего одна копия данных-членов (например, как при реализации счетчика).

```
// Объявление статического члена данных
public class Voter {
   static int voterCount = 0;

   public Voter() { voterCount++;}
   public static int getVoterCount() {
      return voterCount;
   }
}
...
int numVoters = Voter.voterCount;
```

Статические методы

У статических методов в объявлении метода присутствует ключевое слово `static`.

```
// Объявление статического метода
class Analyzer {
   public static int getVotesByAge() {...}
}

// Использование статического метода
Analyzer.getVotesByAge();
```

Статические методы не могут обращаться к нестатическим методам или переменным, поскольку статические методы ассоциируются с определенным классом, а не объектом.

Статические константы

Статические константы являются статическими членами, объявленными как константа. Они содержат ключевые слова `static` и `final`, и никакая программа не может изменить их значение.

```
// Объявление статической константы
static final int AGE_LIMIT = 18;

// Использование статической константы
if (age == AGE_LIMIT)
   newVoter = "yes";
```

Статические инициализаторы

Статические инициализаторы включают блок кода, которому предшествует ключевое слово `static`. Класс может содержать любое количество блоков статических инициализаторов, причем они будут гарантированно выполняться именно в той последовательности, в которой появляются. Блоки статических инициализаторов выполняются лишь один раз для каждой инициализации класса. Блок запускается в тот момент, когда загрузчик клас-

сов виртуальной машины Java загружает статический класс, код которого вызван первый раз.

```
// Статический инициализатор
static {
   numberOfCandidates = getNumberOfCandidates();
}
```

Интерфейсы

Интерфейсы — это набор объявленных открытых (public) методов без их реализации. Класс, в котором реализуется тот или иной интерфейс, должен содержать реализации для всех методов, определенных данным интерфейсом, или должен быть объявлен абстрактным.

Интерфейс объявляется с помощью ключевого слова interface, за которым указывается имя данного интерфейса и набор объявлений методов.

Имена интерфейсов обычно представляют собой имена прилагательные и заканчиваются на ...able ("-ый") или ...ible ("-ий") в случаях, когда соответствующий интерфейс обеспечивает какую-либо возможность.

```
interface Reportable {
   void genReport(String repType);
   void printReport(String repType);
}
```

Класс, в котором реализован какой-либо интерфейс, должен содержать имя этого интерфейса в своей сигнатуре, которое указывается после ключевого слова implements.

```
class VotingMachine implements Reportable {
   public void genReport (String repType) {
      Report report = new Report(repType);
   }

   public void printReport(String repType) {
      System.out.println(repType);
   }
}
```

НА ЗАМЕТКУ

В классах можно реализовать несколько интерфейсов, а сами интерфейсы могут расширять несколько других интерфейсов.

Перечисления

В самом простом смысле *перечисления* — это набор объектов, представляющих набор связанных между собой значений.

```
enum DisplayButton {ROUND, SQUARE}
DisplayButton round = DisplayButton.ROUND;
```

В более широком смысле перечисление представляет собой класс типа enum, являющийся синглтоном. В классах enum могут содержаться методы, конструкторы и данные-члены.

```
enum DisplayButton {
   // Размер в дюймах
   ROUND  (0.50f),
   SQUARE (0.40f);

   private final float size;

   DisplayButton(float size) {this.size = size;}
   private float size() { return size; }
}
```

Метод values() класса enum возвращает массив упорядоченного списка объектов, определенных для соответствующего перечисления.

```
for (DisplayButton b : DisplayButton.values())
   System.out.println("Кнопка: " + b.size());
```

Типы аннотаций

Аннотации позволяют ассоциировать метаданные (т.е. данные о данных) с элементами программы во время ее компиляции и

выполнения. Можно аннотировать пакеты, классы, методы, поля, параметры, переменные и конструкторы.

Встроенные аннотации

Аннотации в Java позволяют получать метаданные о классе. Как следует из табл. 5.1, в Java предусмотрены три типа встроенных аннотаций, которые содержатся в пакете `java.lang`.

Аннотации нужно размещать непосредственно перед аннотируемым элементом. У аннотаций нет параметров, и они не генерируют исключения. Аннотации возвращают значения простого типа, перечисления, класс `String`, класс `Class`, сами аннотации и массивы (этих типов).

Таблица 5.1. Встроенные аннотации

Тип аннотации	Описание
`@Override`	Указывает, что соответствующий метод предназначен для переопределения какого-либо метода суперкласса
`@Deprecated`	Указывает, что используется или переопределяется не рекомендуемый API
`@SuppressWarnings`	Используется для выборочного подавления предупреждений

Ниже приведен пример использования встроенных аннотаций.

```
@Override
   public String toString() {
      return super.toString() + " далее...";
   }
```

Поскольку `@Override` является маркерной аннотацией, в результате компиляции выводится предупреждение, если переопределяемый метод не будет найден.

Аннотации, определяемые разработчиком

Разработчики могут определять собственные аннотации, используя три типа аннотаций. У *маркерной* аннотации нет параметров, *аннотация с одним значением* имеет только один параметр, а *аннотация со многими значениями* — несколько параметров.

Определением аннотации является символ @, за которым следует слово interface, за которым, в свою очередь, следует имя аннотации.

Можно использовать повторяемые аннотации.

Мета-аннотация Retention указывает, что заданная аннотация должна быть сохранена виртуальной машиной Java, чтобы ее значение можно было прочитать во время выполнения программы. Определение аннотации Retention находится в пакете java.lang.annotation.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Feedback {} // Маркер

public @interface Feedback {
   String reportName();
} // Одно значение

public @interface Feedback {
   String reportName();
   String comment() default "None";
} // Несколько значений
```

Аннотация, определенная пользователем, помещается непосредственно перед аннотируемым элементом.

```
@Feedback(reportName="Report 1")
public void myMethod() {...}
```

При выполнении программы можно проверить наличие аннотаций и получить их значения, вызвав метод getAnnotation() для нужного метода.

```
Feedback fb =
   myMethod.getAnnotation(Feedback.class);
```

В формальной спецификации типов аннотаций "JSR 308" допускается написание аннотаций среди элементов массива и аргументов обобщенного типа. Аннотации также могут быть записаны для суперклассов, реализованных интерфейсов, приведения типов, проверок instanceof, спецификаций исключений, символов подстановки, ссылок на методы и конструкторы. Подробную информацию об использовании аннотаций в перечисленных

выше контекстах можно получить в книге Кея Хорстмана *Java SE 8. Вводный курс* (пер. с англ., ИД "Вильямс", 2014).

Функциональные интерфейсы

Функциональный интерфейс еще называют SAM-интерфейсом (Single Abstract Method — единственный абстрактный метод). Это такой тип интерфейса, в котором может быть определен один и только один абстрактный метод. Чтобы обозначить некий интерфейс как функциональный, перед его объявлением следует поместить аннотацию `@FunctionalInterface`. В интерфейсе может быть определено произвольное количество стандартных (default) методов.

```
@FunctionalInterface
  public interface InterfaceName {

    // Допускается только один абстрактный метод
       public void doAbstractTask();

    // Допускается несколько стандартных методов
       default public void performTask1(){
          System.out.println("Сообщение от задачи 1.");
       }

       default public void performTask2(){
          System.out.println("Сообщение от задачи 2.");
       }
}
```

Экземпляры функциональных интерфейсов могут быть созданы вместе с лямбда-выражениями, ссылками на методы или конструкторы.

Операторы и блоки

Оператор представляет собой отдельно взятую команду, которая осуществляет то или иное действие интерпретатора Java при выполнении программы.

```
GigSim simulator = new GigSim(
                                "Давай, сыграем на гитаре!");
```

К числу операторов Java относятся: выражение, пустой оператор, блок, условный оператор, итерация, передача управления, обработка исключения, переменная, метка, утверждение и синхронизированные операторы.

Зарезервированными словами, которые используются в операторах Java, являются: `if`, `else`, `switch`, `case`, `while`, `do`, `for`, `break`, `continue`, `return`, `synchronized`, `throw`, `try`, `catch`, `finally` и `assert`.

Операторы выражений

Оператор выражения — это оператор, который изменяет состояние программы; любое выражение в Java заканчивается точкой с запятой. К числу операторов выражения относятся: присвоения, префиксные и постфиксные приращения (инкременты), префиксные и постфиксные вычитания (декременты), создание объекта и вызовы методов. Ниже приведены примеры операторов выражений.

```
isWithinOperatingHours = true;
++fret; patron++; --glassOfWater; pick--;
Guitarist guitarist = new Guitarist();
guitarist.placeCapo(guitar, capo, fret);
```

Пустой оператор

Пустой оператор не выполняет какого-либо действия и записывается в виде символа точки с запятой (`;`) или как пустой блок `{}`.

Блоки

Группа операторов называется блоком или блоком операторов. Блок операторов заключается в фигурные скобки. Переменные и классы, объявленные в блоке, называются локальными переменными и локальными классами соответственно. Областью действия локальных переменных и локальных классов является блок, в котором они объявлены.

В блоках операторы интерпретируются друг за другом, в той последовательности, в которой они записаны, или в порядке управления выполнением программы. Ниже приведен пример блока.

```
static {
   GigSimProperties.setFirstFestivalActive(true);
   System.out.println("Первый фестиваль открыт!");
   gigsimLogger.info("Начался первый фестиваль.");
}
```

Условные операторы

Операторы `if`, `if else` и `if else if` являются операторами управления выполнением программы (операторами принятия решений). Они используются для выполнения операторов в зависимости от наступления тех или иных условий. Выражение для любого из этих операторов должно иметь тип `Boolean` или `boolean`. При использовании типа `Boolean` будет выполнена распаковка, т.е. автоматическое преобразование `Boolean` в `boolean`.

Оператор `if`

Оператор `if` состоит из выражения и оператора или блока операторов, которые выполняются, если значение соответствующего выражения равно `true`.

```
Guitar guitar = new Guitar();
guitar.addProblemItem("Треснувший гриф");
if (guitar.isBroken()) {
   Luthier luthier = new Luthier();
   luthier.repairGuitar(guitar);
}
```

Оператор `if else`

Если оператор `if` используется в сочетании с `else`, то первый блок операторов выполняется, если значение соответствующего выражения равно `true`; в противном случае выполняется блок кода в `else`.

```
CoffeeShop coffeeshop = new CoffeeShop();
if (coffeeshop.getPatronCount() > 5) {
   System.out.println("Отобразить приветствие.");
} else {
   System.out.println("Автомат неисправен!");
}
```

Оператор `if else if`

Оператор `if else if` обычно используется, когда нужно сделать выбор между несколькими блоками кода. Когда используемый критерий выбора не позволяет выполнить ни один из этих блоков, выполняется код в последнем блоке `else`.

```
ArrayList<Song> playList = new ArrayList<>();
Song song1 = new Song("Mister Sandman");
Song song2 = new Song("Amazing Grace");
playList.add(song1);
playList.add(song2);
...
int numOfSongs = playList.size();
```

```
if (numOfSongs <= 24) {
   System.out.println("Мало песен.");
} else if ((numOfSongs >  24) & (numOfSongs < 50)){
   System.out.println("Хватит на один вечер");
} else if ((numOfSongs >= 50) & (numOfSongs < 100)) {
   System.out.println("Хватит на два вечера.");
} else {
   System.out.println("Хватит на неделю.");
}
```

Оператор `switch`

Оператор `switch` является оператором управления выполнением программы. В зависимости от значения его выражения управление передается одному из следующих за ним операторов `case`. Оператор `switch` работает с типами `char`, `byte`, `short`, `int`, а также с интерфейсными типами `Character`, `Byte`, `Short` и `Integer`; с типами перечислений и типом `String`. Поддержка объектов `String` была обеспечена только в Java SE 7. Оператор `break` используется для завершения работы оператора `switch`. Если оператор `case` не содержит оператора `break`, то будет выполняться следующая за ним строка кода (обычно следующий оператор `case`).

Это продолжается до тех пор, пока либо не будет достигнут оператор `break`, либо конец оператора `switch`. Допускается использование одной метки `default`; как правило, она используется в конце оператора `switch` с целью улучшения читабельности кода.

```
String style;
String guitarist = "Эрик Клэптон";
...
switch (guitarist) {
   case "Чет Аткинс":
      style = "Фингерстайл";
      break;

   case "Томми Эммануэль":
      style = "Сложная импровизация";
      break;
```

```
    default:
        style = "Неизвестен";
        break;
}
```

Итерационные операторы

Операторы простого и расширенного цикла `for`, а также операторы `while` и `do-while` являются итерационными. Они используются для циклического выполнения определенных фрагментов кода.

Оператор цикла `for`

Оператор цикла `for` состоит из трех частей: инициализации, условного выражения и обновления. Как показано ниже, перед использованием этого оператора должна быть инициализирована переменная цикла (т.е. `i`). Условное выражение (т.е. `i < bArray.length()`) всегда вычисляется до начала выполнения цикла. Итерация (т.е. выполнение цикла) начинается лишь в случае, если значение условного выражения истинно, а переменная цикла обновляется (т.е. `i++`) после каждой очередной итерации.

```
Banjo [] bArray = new Banjo[2];

bArray[0] = new Banjo();
bArray[0].setManufacturer("Windsor");

bArray[1] = new Banjo();
bArray[1].setManufacturer("Gibson");

for (int i=0; i < bArray.length; i++){
    System.out.println(bArray[i].getManufacturer());
}
```

Операторы расширенного цикла `for`

Операторы расширенного цикла `for`, т.е. циклы типа "for in" или "for each", используются для итерационной обработки того

или иного объекта или массива, допускающего перебор своих элементов. Цикл выполняется однократно для каждого элемента массива или коллекции и не использует счетчик, поскольку количество итераций известно заранее.

```
ElectricGuitar eGuitar1 = new ElectricGuitar();
eGuitar1.setName("Blackie");

ElectricGuitar eGuitar2 = new ElectricGuitar();
eGuitar2.setName("Lucille");

ArrayList <ElectricGuitar> eList = new ArrayList<>();
eList.add(eGuitar1);
eList.add(eGuitar2);

for (ElectricGuitar e : eList) {
   System.out.println("Name:" + e.getName());
}
```

Оператор цикла `while`

В операторе цикла `while` сначала вычисляется выражение, и цикл выполняется лишь в том случае, если значение выражения истинно. Выражение может быть типа `boolean` или `Boolean`.

```
int bandMembers = 5;
while (bandMembers > 3) {
   CoffeeShop c = new CoffeeShop();
   c.performGig(bandMembers);
   Random generator = new Random();
   bandMembers = generator.nextInt(7) + 1; // 1-7
}
```

Оператор цикла `do-while`

В операторе `do-while` цикл всегда выполняется по меньшей мере один раз и будет выполняться до тех пор, пока значение выражения истинно. Само выражение может быть типа `boolean` или `Boolean`.

```
int bandMembers = 1;
do {
```

```
    CoffeeShop c = new CoffeeShop();
    c.performGig(bandMembers);
    Random generator = new Random();
    bandMembers = generator.nextInt(7) + 1; // 1-7
} while (bandMembers > 3);
```

Передача управления

Операторы передачи управления используются для изменения потока команд в программе. К числу таких операторов относятся: `break`, `continue` и `return`.

Оператор `break`

Оператор `break` без метки используется для выхода из текущей ветки оператора `switch` или немедленного выхода из цикла. Этот оператор может прекращать работу простого и расширенного цикла `for`, а также циклов типа `while` и `do-while`.

```
Song song = new Song("Pink Panther");
Guitar guitar = new Guitar();
int measure = 1;
int lastMeasure = 10;

while (measure <= lastMeasure) {
    if (guitar.checkForBrokenStrings()) {
        break;
    }
    song.playMeasure(measure);
    measure++;
}
```

Оператор `break` с меткой приводит к завершению цикла с указанной меткой и передаче управления следующему за этим циклом оператору. Метки обычно используются с циклами `for` и `while`, когда есть вложенные циклы и требуется точно определить, из какого именно цикла нужно выйти. Чтобы пометить какой-либо цикл или оператор, поместите оператор метки непосредственно перед помечаемым циклом или оператором, как показано в приведенном ниже примере.

```
...
playMeasures:
while (isWithinOperatingHours()) {
   while (measure <= lastMeasure) {
      if (guitar.checkForBrokenStrings()) {
         break playMeasures;
      }
      song.playMeasure(measure);
      measure++;
   }
}
// Выход осуществляется в эту точку
```

Оператор `continue`

Выполнение оператора `continue` без метки приводит к прекращению выполнения текущей итерации простого или расширенного цикла `for`, а также цикла `while` или `do-while` и началу следующей итерации соответствующего цикла. При этом будет выполнена проверка условий цикла. Оператор `continue` с меткой вызывает следующую итерацию указанного цикла:

```
for (int i=0; i<25; i++) {
   if (playList.get(i).isPlayed()) {
      continue;
   } else {
      song.playAllMeasures();
   }
}
```

Оператор `return`

Оператор `return` используется для выхода из метода и возвращения того или иного значения, если в описании метода указано, что данный метод должен вернуть значение определенного типа.

```
private int numberOfFrets = 18; // Стандартное значение
...
public int getNumberOfFrets() {
   return numberOfFrets;
}
```

Оператор return использовать необязательно, если он является последним оператором в методе и этот метод ничего не возвращает.

Оператор synchronized

Ключевое слово synchronized в Java может использоваться для предоставления доступа к определенным участкам кода (например, к определенным методам) только одному потоку команд. Оператор synchronized позволяет управлять доступом к ресурсам, которые совместно используются в нескольких потоках. Подробнее об этом рассказывается в главе 14.

Оператор assert

Утверждения (assertions) представляют собой булевы выражения, используемые в режиме отладки для проверки, ведет ли себя код именно так, как ожидалось (т.е. при запуске приложения с использованием параметра командной строки -enableassertions или -ea интерпретатора Java). Утверждения записываются следующим образом:

```
assert булево_выражение;
```

Утверждения облегчают выявление ошибок, в том числе обнаружение неожиданных значений. Они предназначены для подтверждения предположений, которые всегда должны быть истинны. При выполнении в режиме отладки, если значение утверждения ложно, генерируется исключение java.lang.AssertionError и осуществляется выход из программы; в противном случае ничего не происходит. Утверждения нужно активизировать в явном виде. Аргументы командной строки, используемые для активизации утверждений, описаны в главе 10.

```
// Значение переменной 'strings' должно
// равняться 4, 5, 6, 7, 8 или 12
```

```
assert (strings == 12 ||
        (strings >= 4 & strings <= 8));
```

В утверждении можно также указать необязательный код ошибки. Несмотря на такое название ("код ошибки"), в действительности он представляет собой обычный текст или значение, используемые исключительно для информационных целей.

```
assert булево_выражение : код_ошибки;
```

Когда утверждение, в котором содержится код ошибки, истинно, значение кода ошибки преобразуется в строку и отображается для пользователя непосредственно перед завершением работы. Ниже приведен пример утверждения, в котором используется код ошибки.

```
// Показать недопустимое количество струн
// для музыкального инструмента'
assert (strings == 12 ||
        (strings >= 4 && strings <= 8))
        : "Недопустимое число струн: " + strings;
```

Операторы обработки исключений

Операторы обработки исключений используются для указания кода, который должен быть выполнен при возникновении необычных обстоятельств. Для обработки исключений используются ключевые слова `throw` и `try/catch/finally`. Подробнее об обработке исключений можно прочитать в главе 7.

Обработка исключений

Исключение — это аномальное условие, которое изменяет или прерывает поток выполнения команд программы. Чтобы находить правильный выход из таких ситуаций, в Java предусмотрена встроенная обработка исключений. Обработка исключений не должна быть частью нормального хода выполнения программы.

Иерархия исключений

Как показано на рис. 7.1, все классы исключений и ошибок наследуются из класса `Throwable`, который, в свою очередь, наследуется из класса `Object`.

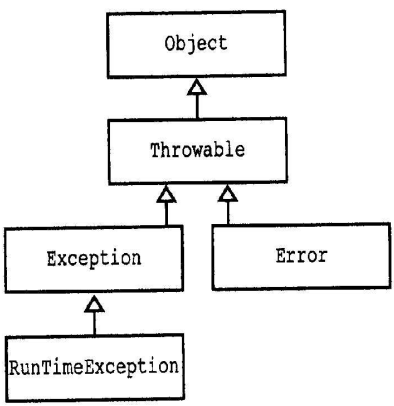


Рис. 7.1. Схема иерархии исключений

Проверяемые и непроверяемые исключения и ошибки

Исключения и ошибки бывают трех категорий: проверяемые и непроверяемые исключения и ошибки.

Проверяемые исключения

- Проверяемые исключения проверяются компилятором на этапе компиляции.
- Если в методах может генерироваться проверяемое исключение, то этот факт должен быть указан в объявлении методов с помощью оператора `throws`. Оператор `throws` нужно указывать во всех вложенных методах вызывающей программы до тех пор, пока соответствующее исключение не будет обработано.
- Все проверяемые исключения должны быть перехвачены в явном виде блоком `catch`.
- К проверяемым исключениям относятся исключения типа `Exception` и все классы, которые являются подтипами класса `Exception`, кроме класса `RuntimeException` и его подтипов.

Ниже приведен пример объявления метода, в котором генерируется проверяемое исключение.

```
// Объявление метода, в котором генерируется
// исключение IOException
void readFile(String filename)
   throws IOException {
   ...
}
```

Непроверяемые исключения

- Компилятор не контролирует непроверяемые исключения во время компиляции.
- Непроверяемые исключения возникают во время выполнения программы вследствие ошибок в ее коде (например, выход индекса массива за допустимые пределы, деление на нуль и исключение при использовании нулевого указателя) или исчерпания системных ресурсов.
- Непроверяемые исключения не обязательно должны обрабатываться.

- Если в методе может генерироваться непроверяемое исключение, то этот факт можно указать (а можно и не указывать) в его объявлении.
- К непроверяемым исключениям относятся исключения типа `RuntimeException` и всех его подтипов.

Ошибки

- Ошибки, как правило, являются необратимыми и отражают серьезные ситуации.
- Ошибки не проверяются во время компиляции программы и не обязательно должны (но могут) перехватываться/обрабатываться.

НА ЗАМЕТКУ

Любые проверяемые исключения, непроверяемые исключения или ошибки могут перехватываться.

Стандартные проверяемые и непроверяемые исключения и ошибки

Существуют разные типы проверяемых и непроверяемых исключений, а также ошибок, которые являются частью стандартной платформы Java. Вероятность появления каких-то из них больше, чем вероятность появления других.

Популярные проверяемые исключения

`ClassNotFoundException`

Генерируется, когда какой-либо класс не может быть загружен из-за того, что его определение не найдено.

`IOException`

Генерируется, когда операция ввода-вывода завершается аварийно или досрочно прекращается по какой-либо

причине. Чаще всего встречаются два подтипа исключения: `IOException`: `EOFException` и `FileNotFoundException`.

`EOFException`

Генерируется при попытке прочитать очередную порцию данных из файла, если ранее был достигнут конец файла.

`FileNotFoundException`

Генерируется при попытке открыть файл, который невозможно найти.

`SQLException`

Генерируется, когда происходит ошибка при работе с базой данных.

`InterruptedException`

Генерируется при прерывании потока команд.

`NotSuchMethodException`

Генерируется в случае, если невозможно найти вызываемый метод.

`CloneNotSupportedException`

Генерируется при вызове метода `clone()` для объекта, который не является клонируемым.

Популярные непроверяемые исключения

`ArithmeticException`

Генерируется, если во время выполнения арифметических операций происходит исключительная ситуация (например, переполнение).

`ArrayIndexOutOfBoundsException`

Генерируется, если значение индекса массива выходит за допустимые пределы.

`ClassCastException`

Генерируется при попытке преобразования объекта в подкласс, экземпляром которого этот объект не является.

`DateTimeException`

Генерируется при возникновении проблем при создании, запросе или работе с объектами даты/времени.

`IllegalArgumentException`

Генерируется во время вызова метода, которому передан аргумент недопустимого типа.

`IllegalStateException`

Генерируется, если некий метод был вызван в неподходящий момент.

`IndexOutOfBoundsException`

Генерируется, если значение индекса выходит за допустимые пределы.

`NullPointerException`

Генерируется, когда для ссылки на некий объект используется нулевая ссылка.

`NumberFormatException`

Генерируется при попытке некорректного преобразования строки в числовой тип.

`UncheckedIOException`

Класс-оболочка для `IOException`, используемый для непроверяемых исключений.

Популярные ошибки

`AssertionError`

Генерируется, если проверка утверждения завершилась неудачно.

`ExceptionInInitializeError`

Генерируется при возникновении неожиданного исключения в статическом инициализаторе.

`VirtualMachineError`

Генерируется при возникновении проблемы с JVM.

`OutOfMemoryError`

Генерируется в случае нехватки памяти при ее выделении какому-либо объекту или выполнении сборки мусора.

`NoClassDefFoundError`

Генерируется, когда JVM не может найти определение какого-либо класса, которое было обнаружено во время компиляции.

`StackOverflowError`

Генерируется в случае переполнения программного стека.

Ключевые слова, используемые для обработки исключений

В Java код обработки ошибок явно отделяется от кода, в котором эти ошибки могут возникнуть. Если в некотором фрагменте кода может возникнуть исключение, то говорят, что в нем генерируется исключение. Если это исключение обрабатывается во фрагменте кода, то говорят, что в коде перехватывается исключение.

```
// Объявить исключение
public void methodA() throws IOException {
   ...
   throw new IOException();
   ...
}
// Перехватить исключение
public void methodB() {
   ...
  /* Вызов methodA() должен находиться в блоке try/catch,
  ** поскольку данное исключение является проверяемым.
  ** В противном случае это исключение будет
  ** сгенерировано в методе methodB()
  */
   try {
      methodA();
   }catch (IOException ioe) {
      System.err.println(ioe.getMessage());
```

```
        ioe.printStackTrace();
    }
}
```

Ключевое слово `throw`

Чтобы сгенерировать какое-либо исключение, используется ключевое слово `throw`. В коде можно сгенерировать любое проверяемое/непроверяемое исключение и ошибку, как показано ниже.

```
if (n == -1)
    throw new EOFException();
```

Ключевые слова `try/catch/finally`

Для обработки сгенерированных исключений в коде Java используются блоки `try`, `catch` и `finally`. При обработке возникшего исключения интерпретатор Java сначала ищет соответствующий блок кода в текущем методе, а затем, если необходимо, продвигается вверх по стеку вызовов вплоть до метода `main()`. Если соответствующее исключение так и не было обработано в методе `main()` (т.е. поток диспетчирования событий (Event dispatching thread, или EDT) отсутствует), выполняется аварийное завершение программы и выводится информация (дамп) о последовательности вызовов методов в стеке.

```
try {
    method();
} catch (EOFException eofe) {
    eofe.printStackTrace();
} catch (IOException ioe) {
    ioe.printStackTrace();
} finally {
   // очистка
}
```

Оператор `try-catch`

Оператор `try-catch` содержит один блок `try` и один или несколько блоков `catch`.

В блок try помещается код, в котором могут возникать исключения. Для обработки проверяемого исключения, которое может возникнуть в блоке кода try, нужно указать соответствующий блок кода catch. Если никакие исключения не были сгенерированы, блок try завершается обычным образом. Блоку try могут соответствовать несколько блоков catch для обработки исключений указанного типа или ни одного такого блока.

НА ЗАМЕТКУ

У каждого блока try должен быть по меньшей мере один блок catch или блок finally, ассоциированный с ним.

Между блоком try и любым из блоков catch или finally (если таковые имеются) не может быть никакого кода.

В блоках catch содержится код для обработки возникших исключений. В этом коде могут быть операторы вывода дополнительной информации о возникшей проблеме в файл, что даст пользователю возможность разобраться в ситуации и принять правильное решение. Обратите внимание, что блоки catch никогда не должны быть пустыми, поскольку такое "молчание" приводит к тому, что исключения становятся скрытыми, а это затрудняет поиск и исправление ошибок.

Общепринятое соглашение о присвоении имени параметру в выражении catch заключается в том, что это имя должно состоять из начальных букв слов, составляющих имя соответствующего исключения:

```
catch (ArrayIndexOutOfBoundsException aioobe) {
   aioobe.printStackStrace();
}
```

В выражении catch при необходимости можно также сгенерировать новое исключение.

Порядок следования выражений catch в блоке try/catch определяет порядок перехвата исключений. Обработку всегда начинайте с самого определенного исключения, которое может возникнуть, и заканчивайте самым неопределенным.

НА ЗАМЕТКУ

Исключения, генерируемые в блоке `try`, направляются на обработку в первый подходящий блок `catch`, содержащий аргументы того же типа, что и объект исключения, или суперкласс этого же типа. Блок `catch` с параметром типа `Exception` всегда должен быть последним в этом списке.

Если ни один из параметров блоков `catch` не соответствует типу сгенерированного исключения, то система будет отыскивать параметр, тип которого соответствует суперклассу данного исключения.

Оператор `try-finally`

Оператор `try-finally` содержит один блок `try` и один блок `finally`.

Блок `finally` используется, если необходимо освобождение ресурсов системы, как показано ниже.

```
public void testMethod() throws IOException {
    FileWriter fileWriter =
            new FileWriter("\\data.txt");
    try {
            fileWriter.write("Информация...");
    }   finally {
            fileWriter.close();
    }
}
```

Использовать блок `finally` необязательно. Если такой блок присутствует, то он выполняется последним в блоке `try-finally` и будет всегда выполняться, независимо от того, как завершится работа блока `try`. Если в блоке `finally` генерируется какое-либо исключение, то оно должно быть обработано.

Оператор try-catch-finally

Оператор try-catch-finally содержит один блок try, один или несколько блоков catch и один блок finally.

В этом операторе блок finally также используется для очистки и освобождения ресурсов системы, как показано ниже.

```
public void testMethod() {
   FileWriter fileWriter = null;
   try {
      fileWriter = new FileWriter("\\data.txt");
      fileWriter.write("Информация...");
   } catch (IOException ex) {
         ex.printStackTrace();
   } finally {
         try {
           fileWriter.close();
         } catch (Exception e) {
           e.printStackTrace();
         }
   }
}
```

Использовать блок finally необязательно (он используется лишь по мере необходимости). Если такой блок присутствует, то он выполняется последним в блоке try-catch-finally и всегда будет выполняться, независимо от того, как завершится работа блока try и были выполнены блоки catch или нет. Если в блоке finally генерируется какое-либо исключение, то оно должно быть обработано.

Оператор try с ресурсами

Оператор try с ресурсами используется для объявления ресурсов, которые должны быть автоматически закрыты, когда в них уже нет необходимости. Эти ресурсы объявляются в блоке try, как показано ниже.

```
public void testMethod() throws IOException {
   try (FileWriter fw = new FileWriter("\\data.txt"))
   {
```

```
            fw.write("Информация...");
        }
    }
```

В операторе `try` с ресурсами можно использовать любой объект ресурса, в котором реализован интерфейс `Autoclosable`.

Конструкция для перехвата нескольких исключений

Данная конструкция позволяет указать в одном выражении `catch` несколько аргументов для перехвата исключений:

```
boolean isTest = false;
public void testMethod() {
    try {
        if (isTest) {
            throw new IOException();
        } else {
            throw new SQLException();
        }
    } catch (IOException | SQLException e) {
        e.printStackTrace();
    }
}
```

Процесс обработки исключений

Ниже описаны этапы процесса обработки исключений.

1. При возникновении исключения создается объект исключения соответствующего типа.
2. На основе этого нового объекта исключения генерируется исключение.
3. Система времени выполнения пытается отыскать код для обработки соответствующего исключения, начиная с метода, в котором был создан данный объект исключения. Если код обработки соответствующего исключения не найден,

среда времени выполнения проходит стек вызовов (упорядоченный перечень методов) в обратной последовательности, пытаясь отыскать код для обработки соответствующего исключения. Если данное исключение не будет обработано, осуществляется аварийный выход из программы и автоматически выводится трассировка вызовов в стеке.

4. Система времени выполнения передает данный объект исключения на обработку найденному блоку кода (оператору `catch`).

Определение собственного класса исключений

Исключения, определяемые программистом, должны создаваться в случаях, когда возникает потребность в исключениях, не предусмотренных в Java. Вообще говоря, когда это возможно, следует пользоваться исключениями, предусмотренными в Java.

- Чтобы определить проверяемое исключение, новый класс исключения должен расширить, непосредственно или косвенно, класс `Exception`.
- Чтобы определить непроверяемое исключение, новый класс исключения должен расширить, непосредственно или косвенно, класс `RuntimeException`.
- Чтобы определить непроверяемую ошибку, новый класс ошибки должен расширить класс `Error`.

Исключения, определяемые пользователем, должны содержать по меньшей мере два конструктора: один без аргументов, а другой — с аргументами, как показано ниже.

```
public class ReportException extends Exception {
   public ReportException () {}
   public ReportException (String message, int
      reportId) {
      ...
   }
}
```

Вывод информации об исключениях

В классе Throwable предусмотрено несколько методов, с помощью которых можно вывести информацию о возникшем исключении: getMessage(), toString() и printStackTrace(). Вообще говоря, один из этих методов должен вызываться в блоке catch, в котором обрабатывается соответствующее исключение. Программисты могут также написать код для получения дополнительной полезной информации, когда встречается то или иное исключение (например, вывести имя файла, который не был найден, и т.п.).

Метод getMessage()

Метод getMessage() возвращает строку, содержащую подробное описание причины возникновения соответствующего исключения.

```
try {
   new FileReader("file.js");
}   catch (FileNotFoundException fnfe) {
        System.err.println(fnfe.getMessage());
}
```

Метод toString()

Метод toString() возвращает строку, содержащую подробное описание причины возникновения соответствующего исключения, включая имя его класса.

```
try {
   new FileReader("file.js");
}   catch (FileNotFoundException fnfe) {
        System.err.println(fnfe.toString());
}
```

Метод printStackTrace()

Метод printStackTrace() возвращает строку, содержащую подробное описание причины возникновения соответствующего исключения, включая имя его класса и трассировку вызовов в стеке, откуда была перехвачена данная ошибка, — вплоть до того момента, когда было сгенерировано данное исключение.

```
try {
   new FileReader("file.js");
}   catch (FileNotFoundException fnfe) {
   fnfe.printStackTrace();
}
```

Ниже приведен пример трассировки стека. Первая строка содержит результат, возвращаемый после вызова метода toString() объекта исключения. В остальных строках представлена последовательность вызовов методов, начиная с места, в котором было сгенерировано данное исключение, и заканчивая местом, в котором оно было перехвачено и обработано:

```
java.io.FileNotFoundException: file.js (Нет такого файла
или каталога)
   at java.io.FileInputStream.open(Native Method)
   at java.io.FileInputStream.(init)
   (FileInputSteam.java:106)
   at java.io.FileInputStream.(init)
   (FileInputSteam.java:66)
   at java.io.FileReader(init)(FileReader.java:41)
   at EHExample.openFile(EHExample.java:24)
   at EHExample.main(EHExample.java:15)
```

Модификаторы Java

Модификаторы — это ключевые слова Java, которые могут применяться к классам, интерфейсам, конструкторам, методам и членам данных.

В табл. 8.1 перечислены модификаторы Java и область их применения. Обратите внимание: допускаются закрытые и защищенные классы, но только если они существуют в виде внутренних или вложенных классов.

Таблица 8.1. Модификаторы Java

Модификатор	Класс	Интерфейс	Конструктор	Метод	Член данных
	Модификаторы доступа				
Пакетный закрытый	Да	Да	Да	Да	Да
`private`	Нет	Нет	Да	Да	Да
`protected`	Нет	Нет	Да	Да	Да
`public`	Да	Да	Да	Да	Да
	Другие модификаторы				
`abstract`	Да	Да	Нет	Да	Нет
`final`	Да	Нет	Нет	Да	Да
`native`	Нет	Нет	Нет	Да	Нет
`strictfp`	Да	Да	Нет	Да	Нет
`static`	Нет	Нет	Нет	Да	Да
`synchronized`	Нет	Нет	Нет	Да	Нет
`transient`	Нет	Нет	Нет	Нет	Да
`volatile`	Нет	Нет	Нет	Нет	Да

Во внутренних классах могут также использоваться модификаторы доступа `private` или `protected`. Для локальных переменных можно использовать лишь один модификатор: `final`.

Модификаторы доступа

Модификаторы доступа определяют права доступа к классам, интерфейсам, конструкторам, методам и членам данных. Модификаторы доступа задаются с помощью ключевых слов `public`, `private` и `protected`. Если ни один из модификаторов доступа не указан, то используется доступ, предусмотренный по умолчанию, — *пакетный закрытый (package-private)*.

В табл. 8.2 приведена подробная информация об области действия модификаторов доступа.

Таблица 8.2. Модификаторы доступа и область их действия

Модификатор	Область действия
Пакетный закрытый	Этот, предусмотренный по умолчанию, тип доступа ограничивает доступ из вне к внутренним объектам соответствующего пакета
`private`	Метод с модификатором `private` доступен только из содержащего его класса. Член данных с модификатором `private` доступен только из содержащего его класса. К нему можно также обратиться косвенно, посредством методов доступа (методов `get-` и `set-`)
`protected`	Метод с модификатором `protected` доступен только из содержащего его пакета, а также за границами пакета для всех подклассов того класса, в котором этот метод определен. Член данных с модификатором `protected` доступен из содержащего его пакета, а также за границами пакета для всех подклассов того класса, в котором этот член данных определен
`public`	Модификатор `public` обеспечивает доступ откуда угодно, даже за границами пакета, в котором он был объявлен. Обратите внимание: интерфейсы являются открытыми по умолчанию

Остальные модификаторы доступа

В табл. 8.3 приведена информация об остальных модификаторах Java, не относящихся к модификаторам доступа, и их использовании.

Таблица 8.3. Модификаторы Java, не определяющие тип доступа

Модификатор	Использование
`abstract`	Определяет абстрактный класс. Абстрактный класс не может быть завершенным (т.е. при его определении нельзя использовать ключевое слово `final`). Интерфейсы являются абстрактными по умолчанию (их не требуется объявлять абстрактными). При объявлении абстрактного метода указывают только его сигнатуру. Если по меньшей мере один метод в классе является абстрактным, то сам класс является также абстрактным. При его объявлении нельзя использовать ключевые слова `final`, `native`, `private`, `static` или `synchronized`
`default`	Определяет метод по умолчанию, который еще называют методом-защитником (defender method). Этот тип метода позволяет создавать стандартную реализацию для метода, определенного в интерфейсе
`final`	Объявляет завершенный класс, который нельзя расширить. Завершенный метод нельзя переопределить. Завершенный член данных инициализируется лишь один раз и не подлежит изменению. Член данных, который объявляется как `static final`, инициализируется во время компиляции и не подлежит изменению
`native`	Объявляет машинно-зависимый (иногда его называют собственным) метод, содержащий объектный код, созданный с помощью других языков программирования, таких как C и C++, который можно вызвать из Java-программы. Данный метод содержит только сигнатуру и не имеет тела. При его объявлении нельзя использовать модификатор `strictfp`
`static`	Доступ к статическим методам и переменным осуществляется посредством указания имени соответствующего класса, а не его переменной-экземпляра. Статические методы и переменные являются общими для всего класса и всех экземпляров данного класса. Доступ к статическому члену данных осуществляется посредством имени соответствующего класса. Сколько бы ни было экземпляров соответствующего класса, возможен лишь один статический член данных
`strictfp`	Все операции с плавающей точкой класса, объявленного с помощью этого модификатора, должны соответствовать спецификации IEEE 754-1985. Все выражения в методе, объявленном как `strictfp`, должны в точности отвечать требованиям для операций с плавающей точкой. Методы внутри интерфейсов нельзя объявлять как `strictfp`. Этот модификатор нельзя использовать одновременно с модификатором `native`
`synchronized`	Данный тип метода означает, что в любой отдельно взятый момент времени тело метода может выполняться только в одном потоке команд. В результате метод становится потокобезопасным, т.е. может без проблем выполняться в многопроцессорной и многозадачной среде. Отдельные блоки кода программы также могут быть объявлены как `synchronized`

Окончание табл. 8.3

Модификатор	Использование
`transient`	Член данных, объявленный как `transient`, не сериализуется при выполнении сериализации класса, поскольку он не является частью устойчивого состояния объекта
`volatile`	Член данных, объявленный как `volatile`, сообщает потоку о необходимости получить самое последнее значение соответствующей переменной из памяти, а не использовать копию, сохраненную в кеше, и записывать все значения этой переменной в память по мере их появления

ЧАСТЬ II

Платформа

Платформа Java, стандартный выпуск

Стандартный выпуск (Standard Edition, SE) платформы Java состоит из *среды выполнения* Java-программ (Java Runtime Environment — JRE), которая также входит в *пакет программ* для разработки приложений на языке Java (Java Development Kit — JDK; см. главу 10), собственно *компилятора* языка программирования Java, *виртуальной машины* Java (Java Virtual Machines — JVM), средств разработки и утилит, а также библиотек Java SE API (рис. 9.1). Версии Java SE выпущены для разных компьютерных платформ: Windows (32- и 64-рязрядные), Mac OS X (64-разрядная), Linux (32- и 64-рязрядные), Linux ARMv6/7 VFP — HardFP ABI (32-разрядная), Solaris SPARC (64-разрядная) и Solaris (64-разрядная).

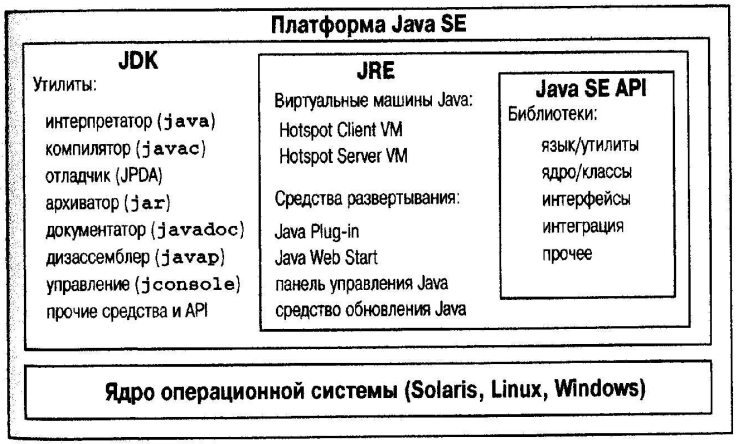


Рис. 9.1. Состав платформы Java SE

Общие библиотеки Java SE API

Стандартные библиотеки Java SE API включены в пакеты, каждый из которых состоит из классов и/или интерфейсов. В этом разделе представлен сокращенный перечень широко используемых пакетов.

В Java SE предусмотрены библиотеки времени выполнения JavaFX из Java SE 7 (обновление 6), а также JavaFX 2.2 и более поздние версии. JavaFX заместил Swing API в качестве нового клиента библиотеки пользовательского интерфейса для Java SE.

Библиотеки поддержки языка и утилит

`java.lang`

Поддержка языка; системные/математические методы, базовые типы, строки, потоки и исключения.

`java.lang.annotation`

Инфраструктура аннотаций; поддержка библиотеки метаданных.

`java.lang.instrument`

Программный инструментарий; агентские сервисы для инструментального оснащения программ JVM.

`java.lang.invoke`

Поддержка динамического языка; поддерживается основными классами и VM.

`java.lang.management`

Java Management Extensions API; мониторинг и управление JVM.

`java.lang.ref`

Классы ссылочных объектов; поддержка взаимодействия с GC.

`java.lang.reflect`

Рефлексивная информация о классах и объектах.

`java.util`

Утилиты; коллекции, модель событий, дата/время и поддержка национальных языков.

`java.util.concurrent`

Средства параллельного выполнения программ; исполнительные программы, очереди, хронометраж и синхронизация.

`java.util.concurrent.atomic`

Средства поддержки неделимости; потоково-безопасное программирование без блокировки на основе одиночных переменных.

`java.util.concurrent.locks`

Инфраструктура блокировок; блокировки и условия.

`java.util.function`

Функциональные интерфейсы; обеспечивает требуемые типы для лямбда-выражений и ссылок на методы.

`java.util.jar`

Поддержка архивного формата файлов Java; чтение и запись.

`java.util.logging`

Ведение журнала; фиксация сбоев, ошибок, проблем с производительностью и дефектов.

`java.util.prefs`

Пользовательские и системные параметры; поиск и хранение.

`java.util.regex`

Регулярные выражения; символьные последовательности, соответствующие шаблону.

`java.util.stream`

Работа с потоками данных; операции с потоками элементов в едином функциональном стиле.

`java.util.zip`

Файловые форматы ZIP и GZIP; чтение и запись.

Базовые библиотеки

`java.applet`

Инфраструктура аплетов; встраиваемые оконные и управляющие методы.

`java.beans`

Компоненты JavaBeans и производные от них; поддержка долговременной сохраняемости компонентов.

`java.beans.beancontext`

Контекст для компонентов JavaBeans; контейнеры для компонентов и среды исполнения.

`java.io`

Поддержка ввода-вывода; потоки данных, файловая система и сериализация.

`java.math`

Математические функции; математические операции с очень большими целыми и десятичными числами.

`java.net`

Работа в сетях; протоколы TCP, UDP, сокеты и адреса.

`java.nio`

Быстродействующий ввод-вывод; буфера, файлы, отображаемые в память.

`java.nio.channels`

Каналы ввода-вывода; селекторы для неблокируемого ввода-вывода.

`java.nio.charset`

Наборы символов; преобразование байтов в Unicode.

`java.nio.file`

Поддержка файлов; файлы, атрибуты файлов, файловые системы.

`java.nio.file.attribute`

Поддержка атрибутов файлов и файловых систем.

`java.text`

Текстовые утилиты; текст, даты, числа и сообщения.

`java.time`

Поддержка времени и дат; работа с длинными и короткими временными интервалами.

`java.time.chrono`

Поддержка времени и дат; различные системы календарей.

`java.time.format`

Поддержка времени и дат; печать и синтаксический анализ дат.

`java.time.temporal`

Поддержка времени и дат; доступ с помощью полей, модулей и сборщиков.

`java.time.zone`

Поддержка времени и дат; работа с временными зонами и их стандартами.

`javax.annotation`

Типы аннотаций; поддержка библиотек.

`javax.management`

JMX API; конфигурация приложений, статистика и изменение состояния.

`javax.net`

Работа в сетях; фабрики сокетов.

`javax.net.ssl`

Уровень защищенных сокетов; обнаружение ошибок, шифрование данных и аутентификация.

`javax.tools`

Интерфейсы запуска утилит; компиляторы, диспетчеры файлов.

Библиотеки интеграции

`java.sql`

Поддержка языка структурированных запросов (Structured Query Language — SQL); доступ и обработка информации из источника данных.

`javax.jws`

Веб-службы Java; поддержка типов аннотаций.

`javax.jws.soap`

Веб-службы Java; SOAP-связывание и параметры сообщений.

`javax.naming`

Службы имен; интерфейс назначения имен и каталогов в Java (Java Naming and Directory Interface — JNDI).

`javax.naming.directory`

Службы каталогов; JNDI-операции для объектов, хранимых в каталогах.

`javax.naming.event`

Службы событий; JNDI-операции уведомления о событиях.

`javax.naming.ldap`

Облегченный протокол доступа к каталогам (Lightweight Directory Access Protocol, LDAP) версии 3; операции и элементы управления.

`javax.script`

Поддержка языка сценариев; интеграция, связывание и вызовы.

`javax.sql`

Поддержка SQL; API баз данных и серверной стороны.

`javax.sql.rowset.serial`

Сериализуемые отображения между типами SQL и типами данных Java.

`javax.sql.rowset`

Стандартные интерфейсы для работы с набором строк, возвращенных через JDBC (Java Database Connectivity, или Java-интерфейс для работы с базами данных).

`javax.transactions.xa`

XA-интерфейс; контракты диспетчера транзакций и ресурсов для JTA.

Различные библиотеки пользовательского интерфейса

`javax.accessibility`

Технология обеспечения доступа; дополнительная поддержка для компонентов пользовательского интерфейса.

`javax.imageio`

Ввод-вывод изображений в Java; описание содержимого файлов изображений (метаданные, миниатюры изображений).

`javax.print`

Службы печати; форматирование и запуск заданий на печать.

`javax.print.attribute`

Службы печати в Java; атрибуты и коллекции атрибутов.

`javax.print.attribute.standard`

Стандартные атрибуты; часто используемые атрибуты и значения.

`javax.print.event`

События печати; службы и мониторинг заданий печати.

`javax.sound.midi`

Поддержка звука; ввод-вывод, формирование последовательности и синтез MIDI типов 0 и 1.

`javax.sound.sampled`

Поддержка звука; дискретизированные аудиоданные (форматы AIFF, AU и WAV).

Библиотеки пользовательского интерфейса: JavaFX

`javafx.animation`
: Поддержка анимации, основанной на переходах.

`javafx.application`
: Поддержка жизненного цикла приложения.

`javafx.beans`
: Общие формы наблюдаемости.

`javafx.beans.binding`
: Характеристики привязки.

`javafx.beans.property`
: Поддержка свойств для чтения/записи.

`javafx.beans.property.adapter`
: Адаптер свойств.

`javafx.beans.value`
: Поддержка чтения и записи.

`javafx.collections`
: Набор утилит JavaFX.

`javafx.concurrent`
: Поддержка параллельного выполнения в JavaFX.

`javafx.embed.swing`
: API интеграции с приложениями Swing.

`javafx.embed.swt`
: API интеграции с приложениями SWT.

`javafx.event`
: Инфраструктура обработки событий (например, доставка и обработка).

`javafx.fxml`
: Языки разметки (например, загрузка иерархии объекта).

`javafx.geometry`
: Поддержка двумерной графики.

`javafx.scene`
: Основные классы; базовое API графа сцены.

`javafx.scene.canvas`
: Классы для поддержки канвы; API для поддержки немедленного режима визуализации.

`javafx.scene.chart`
: Компоненты графиков; визуализация данных.

`javafx.scene.control`
: Элементы пользовательского интерфейса; специализированные узлы в графе сцены.

`javafx.scene.control.cell`
: Классы для поддержки ячеек, т.е. те, которые не относятся к основным.

`javafx.scene.effect`
: Фильтры графических эффектов; поддержка узлов графа сцены.

`javafx.scene.image`
: Загрузка и отображение изображений.

`javafx.scene.input`
: Обработка событий мыши и клавиатуры.

`javafx.scene.layout`
: Классы для макетирования интерфейса.

`javafx.scene.media`
: Классы для работы с аудио и видео.

`javafx.scene.paint`
: Поддержка цветов и градиентов (например, заливка форм и фона).

`javafx.scene.shape`

Поддержка двумерных (плоских) форм.

`javafx.scene.text`

Работа со шрифтами и отрисовка текстовых узлов.

`javafx.scene.transform`

Поддержка трансформаций; поворот, масштабирование, сдвиг и перенос для афинных объектов.

`javafx.scene.web`

Работа с веб-контентом; загрузка и отображение веб-контента.

`javafx.stage`

Поддержка сцены; контейнер верхнего уровня.

`javafx.util`

Утилиты и вспомогательные классы.

`javafx.util.converter`

Преобразователи строк.

Библиотеки пользовательского интерфейса: API AWT (устарело)

`java.awt`

Абстрактно-оконный инструментарий; пользовательские интерфейсы, графика и изображения.

`java.awt.color`

Цветовые пространства в AWT; спецификации формата профиля ICC (International Color Consortium — Международный консорциум по средствам обработки цветных изображений).

`java.awt.datatransfer`

Пересылки данных в AWT между и внутри приложений, поддержка буфера обмена.

`java.awt.dnd`

Поддержка операций "перетащить и опустить" в AWT; поддержка жестов для непосредственного взаимодействия с графическим интерфейсом пользователя.

`java.awt.event`

Слушатели событий/адаптеры AWT для событий, вызванных компонентами AWT.

`java.awt.font`

Поддержка шрифтов в AWT, Type 1 Multiple Master, Open Type и True Type.

`java.awt.geom`

Поддержка геометрических фигур в AWT; поддержка двумерных объектов.

`java.awt.im`

Инфраструктура методов ввода в AWT; ввод текста, языки и поддержка рукописного текста.

`java.awt.image`

Инфраструктура потоковых изображений в AWT; создание и модификация изображений.

`java.awt.image.renderable`

Изображения, независимые от визуализации в AWT; производство.

`java.awt.print`

API печати в AWT; спецификации типов документов, управление параметрами/форматами страницы.

Библиотеки пользовательского интерфейса: Swing API (устарело)

`javax.swing`

Swing API; натуральные компоненты Java (кнопки, разделенные панели, таблицы и т.п.).

`javax.swing.border`

Специализированные рамки Swing; настраиваемые и стандартные рамки, поддерживающие сменные стили и поведение (Look and feel).

`javax.swing.colorchooser`

Поддержка компонента Swing `JColorCooser`; диалоговое окно выбора цвета.

`javax.swing.event`

События Swing; слушатели и адаптеры событий.

`javax.swing.filechooser`

Поддержка компонента Swing `JFileCooser`; диалоговое окно выбора файлов.

`javax.swing.plaf`

Поддержка изменения внешнего вида и поведения компонентов Swing (Pluggable Look-and-Feel, или PLAF); поддержка базового варианта и сменных стилей Metal.

`javax.swing.plaf.basic`

Основные сменные стили интерфейса Swing; стандартное поведение компонентов.

`javax.swing.plaf.metal`

Сменные стили интерфейса Swing типа Metal; переключение внешнего вида Metal/Steel.

`javax.swing.plaf.multi`

Поддержка нескольких сменных стилей интерфейса Swing; позволяет комбинировать несколько сменных стилей и поведений.

`javax.swing.plaf.nimbus`

Сменные стили интерфейса Swing типа Nimbus; кроссплатформенный сменный стиль интерфейса.

`javax.swing.plaf.synth`

Сменные стили и поведение оболочек (Skins) Swing; здесь находятся все методы рисования.

`javax.swing.table`

Поддержка компонентов Swing `Jtable`; табличные структуры данных.

`javax.swing.text`

Поддержка текстовых компонентов Swing; редактируемые и нередактируемые текстовые компоненты.

`javax.swing.text.html`

Текстовые редакторы Swing формата HTML; поддержка создания текстовых редакторов для формата HTML.

`javax.swing.text.html.parser`

Синтаксические анализаторы Swing формата HTML; поддержка стандартного синтаксического анализатора формата HTML.

`javax.swing.text.rtf`

Текстовые редакторы Swing формата RTF (Rich Text Format).

`javax.swing.tree`

Поддержка компонентов Swing `JTree`; построение, управление и визуализация.

`javax.swing.undo`

Операции отмены/восстановления Swing; поддержка текстовых компонентов.

Протокол RMI и библиотеки CORBA

`java.rmi`

Протокол RMI (Remote Method Invocation — дистанционный вызов метода); вызывает объекты на удаленных JVM.

`java.rmi.activation`

Активация объектов RMI; активирует ссылки на сохраняемые (persistent) удаленные объекты.

`java.rmi.dgc`

Распределенная сборка мусора (Distributed Garbage Collection — DGC) для RMI; отслеживание удаленных объектов из клиента.

`java.rmi.registry`

Реестр RMI; удаленный объект, который отображает имена на удаленные объекты.

`java.rmi.server`

Серверная сторона RMI; протокол транспортного уровня RMI, туннелирование протокола передачи гипертекста (Hypertext Transfer Protocol — HTTP), заглушки.

`javax.rmi`

Дистанционный вызов методов; поддержка протоколов RMI, IIOP (Internet InterORB Protocol), RMI-IIOP, JRMP (Java Remote Method Protocol).

`javax.rmi.CORBA`

Поддержка общей архитектуры брокера объектных запросов (Common Object Request Broker Architecture, или CORBA); переносимый API RMI-IIOP и брокеры объектных запросов (Object Request Brokers, или ORBs).

`javax.rmi.ssl`

Протокол защищенных сокетов (Secure Sockets Layer, или SSL); поддержка клиента и сервера RMI.

`org.omg.CORBA`

Поддержка CORBA консорциума OMG; отображение "CORBA-Java", брокеры объектных запросов.

`org.omg.CORBA_2_3`

Дополнения OMG CORBA; дополнительная поддержка теста для JCK (Java Compatibility Kit — комплект обеспечения совместимости с Java).

Библиотеки обеспечения безопасности

`java.security`

Безопасность; алгоритмы, механизмы и протоколы.

`java.security.cert`

Сертификаты; синтаксический анализ, управление списками отзыва сертификатов (Certificate Revocation List, или CRL) и родительскими сертификатами (certification paths).

`java.security.interfaces`

Интерфейсы безопасности: RSA (Rivest, Shamir и Adelman) и DSA (Digital Signature Algorithm — алгоритм цифровой подписи).

`java.security.spec`

Спецификации; ключи безопасности и параметры алгоритмов.

`javax.crypto`

Криптографические операции; шифрование, ключи, генератор MAC.

`javax.crypto.interfaces`

Ключи Диффи–Хеллмана, определенные в стандарте криптографии с открытым ключом №3 (PKCS #3) лаборатории RSA.

`javax.crypto.spec`

Спецификации для ключей безопасности и параметров алгоритмов.

`javax.security.auth`

Безопасная аутентификация и авторизация; спецификации элементов управления доступом.

`javax.security.auth.callback`

Поддержка аутентификации путем функций обратного вызова; взаимодействие служб и приложений.

`javax.security.auth.kerberos`

Протокол сетевой аутентификации Kerberos; соответствующие классы утилит.

`javax.security.auth.login`

Регистрация и конфигурация; инфраструктура подключаемых модулей аутентификации.

`javax.security.auth.x500`

Принципалы и мандаты X500; хранилища сертификатов.

`javax.security.sasl`

Поддержка SASL (Simple Authentication and Security Layer — простой уровень аутентификации и безопасности); SASL-аутентификация.

`org.ietf.jgss`

Поддержка JGSS (Java Generic Security Service — универсальная служба безопасности Java); аутентификация, целостность данных.

Библиотеки XML

`javax.xml`

Поддержка расширяемого языка разметки XML (Extensible Markup Language); константы.

`javax.xml.bind`

Связывание XML-данных во время выполнения программы; маршалинг и демаршалинг, проверка подлинности.

`javax.xml.crypto`

Криптография XML; генерирование подписи и шифрование данных.

`javax.xml.crypto.dom`

XML и объектная модель документов (Document Object Model, или DOM); криптографические реализации.

`javax.xml.crypto.dsig`

Цифровые подписи XML; генерирование и проверка подлинности.

`javax.xml.datatype`

Типы данных XML и Java; отображения.

`javax.xml.namespace`

Пространства имен XML; обработка.

`javax.xml.parsers`

Синтаксические анализаторы XML; простое API для синтаксических анализаторов XML (Simple API for XML, или SAX) и DOM.

`javax.xml.soap`

XML; сообщения SOAP; создание и построение.

`javax.xml.transform`

Процесс преобразования XML-документов; отсутствие зависимости от DOM или SAX.

`javax.xml.transform.dom`

Процесс преобразования XML-документов; API, определяемый DOM.

`javax.xml.transform.sax`

Процесс преобразования XML-документов; API, определяемый SAX.

`javax.xml.transform.stax`

Процесс преобразования XML-документов; API потоковой передачи XML-данных (StAX) API.

`javax.xml.validation`

Процесс преобразования XML-документов; проверка на соответствие схеме XML.

`javax.xml.ws`

Java API для веб-служб XML (JAX-WS); базовый набор API.

`javax.xml.ws.handler`

Обработчики сообщений JAX-WS; интерфейсы контекста сообщений и обработчика.

`javax.xml.ws.handler.soap`

JAX-WS; обработчики сообщений SOAP.

`javax.xml.ws.http`

JAX-WS; HTTP-привязки.

`javax.xml.ws.soap`

JAX-WS; SOAP-привязки.

`javax.xml.xpath`

Поддержка выражений XPath; интерпретация и доступ.

`org.w3c.dom`

Поддержка DOM W3C; доступ к содержимому и структуре файлов и обновления.

`org.xml.sax`

Поддержка SAX от `XML.org`; доступ к содержимому и структуре файлов и обновления.

Основы разработки

Среда исполнения Java-программ (Java Runtime Environment — JRE) служит основой для выполнения приложений Java. В комплект разработчика программ на языке Java (Java Development Kit — JDK) входят все компоненты и необходимые ресурсы для разработки приложений на Java.

Среда исполнения Java-программ (JRE)

JRE представляет собой совокупность программ, которые позволяют выполнять в компьютерной системе любое приложение на языке Java. Эта совокупность программ состоит из виртуальных машин Java (Java Virtual Machines — JVM), которые преобразовывают байт-код Java в машинный код, стандартных библиотек классов, инструментариев пользовательского интерфейса и ряда утилит.

Комплект разработчика программ на языке Java (JDK)

JDK представляет собой среду программирования для компиляции, отладки и выполнения Java-приложений, компонентов Java Beans и Java-аплетов. JDK включает JRE, компилятор языка программирования Java, а также ряд дополнительных средств разработки и инструментальных API. Платформа JDK от Oracle поддерживает Mac OS X, Solaris, Linux (Oracle, SuSe, Red Hat, Ubuntu и Debian (на ARM)) и Microsoft Windows (Server 2008, Server 2012, XP, Vista, Windows 7 и 8). Ознакомиться с поддержкой языка Java в других операционных системах и бесплатно скачать для них

специализированные JVM, JDK и JRE можно с сайта `http://java-virtual-machine.net/other.html`. Поддерживаются такие браузеры, как Internet Explorer (9.x и старше), Firefox, Chrome в системе Windows и Safari 5.x.

В табл. 10.1 перечислены версии JDK, предоставляемые Oracle. Загрузите самую последнюю версию из тех, которые представлены на веб-сайте Oracle (с этого же веб-сайта можно загрузить и более ранние версии).

Таблица 10.1. Комплекты разработчика программ на языке Java (JDK)

JDK	Кодовое название	Выпуск	Количество пакетов	Количество классов
Java SE 8 с JDK 1.8.0	—	2014	217	4240
Java SE 7 с JDK 1.7.0	Dolphin	2011	209	4024
Java SE 6 с JDK 1.6.0	Mustang	2006	203	3793
Java 2 SE 5.0 с JDK 1.5.0	Tiger	2004	166	3279
Java 2 SE с SDK 1.4.0	Merlin	2002	135	2991
Java 2 SE с SDK 1.3	Kestrel	2000	76	1842
Java 2 с SDK 1.2	Playground	1998	59	1520
Development Kit 1.1	—	1997	23	504
Development Kit 1.0	Oak	1996	8	212

В марте 2013 года компания Oracle прекратила выпуск обновлений для Java SE версии 6.

Структура программы на языке Java

Исходные файлы на языке Java создаются с помощью текстовых редакторов, таких как jEdit, Text-Pad, Vim, Notepad++, или текстового редактора, входящего в одну из интегрированных сред разработки программ (Integrated Development Environment — IDE) на языке Java. У исходных файлов на языке Java должно быть расширение `.java`, а имя такого файла должно совпадать с именем открытого класса, содержащегося в этом файле. Если этот класс имеет *пакетный закрытый* метод доступа (*package-private*), то имя класса может отличаться от имени файла.

Следовательно, в исходном файле HelloWorld.java должно находиться определение открытого класса HelloWorld, как показано в приведенном ниже примере (напоминаем, что все имена в Java чувствительны к регистру символов).

```
1 package com.oreilly.tutorial;
2 import java.time.*;
3 // import java.time.ZoneId;
4 // import java.time.Clock;
5
6 public class HelloWorld
7 {
8   public static void main(String[] args)
9   {
10     ZoneId zi = ZoneId.systemDefault();
11     Clock c = Clock.system(zi);
12     System.out.print("From: "
13         + c.getZone().getId());
13     System.out.println(", \"Hello, World!\"");
14   }
15 }
```

В строке 1 указано, что класс HelloWorld содержится в пакете com.oreilly.tutorial. Имя этого пакета указывает на то, что при компиляции и выполнении программы в каталоге com/oreilly/tutorial будет выполняться поиск недостающих классов. Упаковка исходных файлов в пакеты является необязательной, но рекомендуется, чтобы избежать конфликтов в именах классов из других пакетов.

В строке 2 оператор import позволяет JVM выполнять поиск классов из других пакетов. Использование символа "звездочка" делает доступными все классы в пакете java.time. Однако лучше всего включать классы в явном виде, чтобы обеспечить документирование зависимостей. Операторы включения import java.time.ZoneId; и import java.time.Clock;, которые, как видно из приведенного выше кода, закомментированы, являются более разумным выбором, чем просто использование оператора import java.time.*;. Обратите внимание: без операторов import вполне можно обойтись, но тогда программисту придется указывать полное имя пакета перед каждым именем класса.

Очевидно, что все это утомительно и громоздко, поэтому считается плохим стилем кодирования.

НА ЗАМЕТКУ

Пакет `java.lang` является единственным пакетом Java, импортируемым по умолчанию.

В строке 6 должен определяться только один открытый (`public`) класс верхнего уровня, имя которого совпадает с именем исходного файла. Кроме того, в этот файл могут включаться несколько пакетных закрытых (package-private) классов верхнего уровня.

Анализируя строку 8, нетрудно заметить, что в Java-приложениях должен присутствовать открытый статический метод `main()`. Этот метод является точкой входа в программу на языке Java и должен быть всегда определен с использованием модификаторов `public`, `static` и `void`. В качестве параметра методу `main()` передается строковый массив аргументов командной строки.

НА ЗАМЕТКУ

Компоненты приложения, предназначенные для запуска в контейнере (например, таком как Spring или Java EE), не имеют метода `main()`.

Операторы в строках 12 и 13 обеспечивают вызов методов `System.out.print()` и `System.out.println()` для вывода указанного текста в окне консоли.

Утилиты командной строки

В JDK предусмотрено несколько утилит командной строки, которые используются как средства разработки программного обеспечения. Широко используемыми утилитами являются

компилятор, запускающий модуль/интерпретатор, архиватор и документатор. Полный перечень утилит командной строки Java можно найти на сайте `Oracle.com`.

Компилятор Java

Компилятор Java транслирует исходные файлы Java в байт-код Java. Компилятор Java создает файл байт-кода под тем же именем, что и исходный файл, но с расширением `.class`. Ниже приведен перечень часто используемых параметров командной строки компилятора.

`javac [-опции] [исходные файлы]`

Это общий синтаксис, используемый для компиляции исходных файлов на Java.

`javac HelloWorld.java`

Самый простой способ компиляции Java-программы. В результате создается файл `HelloWorld.class`.

`javac -cp /dir/classes/ HelloWorld.java`

Параметры командной строки `-cp` и `-classpath` эквивалентны и указывают имена каталогов, в которых ищутся классы, используемые во время компиляции.

`javac -d /opt/hwapp/classes HelloWorld.java`

Параметр `-d` указывает путь к существующему каталогу, в который помещаются скомпилированные файлы класса. Если в исходном файле имеется определение пакета, то файлы класса помещаются в соответствующую вложенную папку (в данном примере это `/opt/hwapp/classes/com/oreilly/tutorial/`).

`javac -s /opt/hwapp/src HelloWorld.java`

Параметр `-s` указывает существующий каталог, в который будут помещены сгенерированные исходные файлы. Если в них имеется определение пакета, то сгенерированные файлы помещаются в соответствующую вложенную папку

(в данном примере это `/opt/hwapp/src/com/oreilly/tutorial/`).

`javac -source 1.4 HelloWorld.java`

Параметр `-source` обеспечивает совместимость исходного файла с указанным выпуском Java. В результате появляется возможность использования неподдерживаемых ранее ключевых слов в качестве идентификаторов.

`javac -X`

Параметр `-X` позволяет вывести краткую справку по нестандартным опциям командной строки. Например, опция `-Xlint:unchecked` разрешает вывод информационных предупреждений, в которых дается более подробная информация о непроверяемых или небезопасных операциях.

НА ЗАМЕТКУ

Несмотря на то что опция `-Xlint` и прочие опции `-X` широко используются в компиляторах Java, они не стандартизированы, и поэтому их поддержка во всех JDK не гарантируется.

`javac -version`

Параметр `-version` выводит версию утилиты командной строки `javac`.

`javac -help`

Параметр `-help` (без аргументов) позволяет вывести справочную информацию о команде `javac`.

Интерпретатор Java

Интерпретатор Java обеспечивает выполнение программы, в том числе запуск соответствующего приложения. Ниже приведен перечень широко используемых параметров командной строки интерпретатора.

`java [-опции] класс [параметры...]`

Общий синтаксис, используемый для запуска интерпретатора.

`java [-опции] -jar jar-файл [параметры...]`

Общий синтаксис, используемый для запуска JAR-файла на выполнение.

`java HelloWorld`

Простейший способ запуска JRE, загрузки класса `HelloWorld` и выполнения метода `main()` данного класса.

`java com.oreilly.tutorial.HelloWorld`

Простейший способ запуска JRE, загрузки класса `HelloWorld` из пакета, находящегося в каталоге `com/oreilly/tutorial/`, и выполнения метода `main()` данного класса.

`java -cp /tmp/classes/ HelloWorld`

Параметры `-cp` и `-classpath` эквивалентны и указывают каталоги с классами, которые должны использоваться во время выполнения программы.

`java -Dsun.java2d.ddscale=true HelloWorld`

Параметр `-D` позволяет задать значение системного свойства. В данном случае разрешается при масштабировании окон приложения использовать аппаратное ускорение.

`java -ea HelloWorld`

Параметры `-ea` и `-enableassertions` эквиваленты и активизируют проверку утверждений (assertions) в Java-коде. *Утверждения* — это диагностический код, который помещается программистом в приложение, для проверки корректности работоспособности программы. Более подробную информацию об утверждениях см. в разделе "Оператор assert" главы 6.

`java -da HelloWorld`

Параметры `-da` и `-disableassertions` эквиваленты и запрещают проверку утверждений в Java-коде.

`java -client HelloWorld`

Параметр `-client` выбирает клиентскую виртуальную машину (в отличие от серверной виртуальной машины) для повышения эффективности интерактивных приложений с графическим интерфейсом пользователя (GUI).

`java -server HelloWorld`

Параметр `-server` выбирает серверную виртуальную машину (в отличие от клиентской виртуальной машины) для повышения производительности системы в целом.

`java -splash:images/world.gif HelloWorld`

Параметр `-splash` задает имя файла с изображением, которое будет отображаться в качестве начального экрана перед запуском на выполнение соответствующего приложения.

`java -version`

Параметр `-version` выводит на экран версию интерпретатора Java, JRE и виртуальной машины.

`java [-d32 | -d64]`

Параметры `-d32` и `-d64` обусловливают использование 32- или 64-разрядной модели данных (соответственно), если предусмотрена такая возможность.

`java -help`

Параметр `-help` (без аргументов) позволяет вывести на экран справочную информацию о команде `java`.

`javaw` *<имя класса>*

При выполнении в среде ОС Windows команда `javaw` эквивалентна команде `java`, но без открытия окна командной строки. Эквивалентная команда в Linux реализуется путем запуска команды `java` как фонового процесса. Для этого в конце командной строки указывается символ амперсанда: `java` *<имя класса>* `&`.

Упаковщик Java-программ

Утилита `jar` (Java Archive) является инструментом архивирования и сжатия, обычно используемым для объединения нескольких файлов в один архив, называемый JAR-файлом. JAR-файл на самом деле является обычным архивным файлом формата ZIP, в котором содержится файл манифеста (описатель содержимого архива) и необязательные файлы сигнатур (для обеспечения безопасности). Ниже приведен перечень широко используемых опций командной строки утилиты `jar` и примеры их использования.

`jar [`*опции*`] [`*jar-файл*`] [`*файл-манифеста*`] [`*точка-входа*`] [-C` *каталог*`] `*файлы*`...`

Это общий синтаксис, используемый для запуска утилиты `jar`.

```
jar cf files.jar HelloWorld.java com/oreilly/
tutorial/HelloWorld.class
```

Опция `c` позволяет создать JAR-файл. Опция `f` позволяет указать имя этого файла. В данном примере файлы `HelloWorld.java` и `com/oreilly/tutorial/HelloWorld.class` включаются в JAR-файл.

```
jar tfv files.jar
```

Опция `t` используется для вывода оглавления JAR-файла. Опция `f` используется, чтобы указать имя файла. Опция `v` выводит содержимое в понятном для человека формате.

```
jar xf files.jar
```

Опция `x` позволяет извлечь содержимое соответствующего JAR-файла. Опция `f` используется, чтобы указать имя файла.

НА ЗАМЕТКУ

С JAR-файлами могут работать и другие программы-упаковщики, такие как 7-Zip, WinZip и Win-RAR.

Запуск JAR-файлов на выполнение

JAR-файлы можно создавать так, чтобы их можно было запускать на выполнение. Для этого нужно указать, где в архиве находится "главный" класс, чтобы интерпретатор Java знал, какой именно метод `main()` следует использовать. Ниже приведен полный пример того, как сделать JAR-файл исполняемым.

1. Создайте файл `HelloWorld.java`, содержащий класс `HelloWorld`, как было указано в начале этой главы.
2. Создайте вложенные папки `com/oreilly/tutorial/`.
3. Запустите команду `javac -d com/oreilly/tutorial/ HelloWorld.java`.

   Воспользуйтесь этой командой, чтобы скомпилировать программу и поместить файл `HelloWorld.class` в каталог `com/oreilly/tutorial/`.
4. Создайте файл `Manifest.txt` в корневом каталоге пакета (там будет размещен JAR-файл). Включите в этот файл следующую строку, указывающую местонахождение главного класса:

   ```
   Main-Class: com.oreilly.tutorial.HelloWorld
   ```
5. Запустите команду `jar cmf Manifest.txt helloWorld.jar com/oreilly/tutorial`.

   Эта команда создаст JAR-файл и добавит содержимое файла `Manifest.txt` в файл манифеста, `MANIFEST.MF`. Этот файл также используется для определения расширений и всевозможных данных, относящихся к пакету:

   ```
   Manifest-Version: 1.0
   Created-By: 1.7.0 (Oracle Corporation)
   Main-Class: com.oreilly.tutorial.HelloWorld
   ```
6. Запустите `jar tf HelloWorld.jar`.

   Эта команда выведет содержимое JAR-файла:

   ```
   META-INF/
   META-INF/MANIFEST.MF
   com/
   com/oreilly/
   ```

```
com/oreilly/tutorial
com/oreilly/tutorial/HelloWorld.class
```

7. И, наконец, введите команду `java -jar HelloWorld.jar`. Эта команда запустит указанный JAR-файл на выполнение.

Документатор Java

Средство `javadoc` — это утилита командной строки, используемая для генерирования документации из исходных файлов. Такая документация оказывается более подробной, если в исходном коде указаны соответствующие комментарии Javadoc. Подробнее о них говорилось в разделе "Комментарии" главы 2. Ниже приведен перечень часто используемых параметров командной строки утилиты `javadoc` и примеры использования.

`javadoc [`*опции*`] [`*пакеты*`] [`*файлы*`]`

Это общий синтаксис, используемый для запуска утилиты `javadoc` и генерирования документации.

`javadoc HelloWorld.java`

Данная команда генерирует HTML-файлы документации: `HelloWorld.html`, `index.html`, `allclasses-frame.html`, `constant-values.html`, `deprecated-list.html`, `overview-tree.html`, `package-summary.html`, и т.д.

`javadoc -verbose HelloWorld.java`

Параметр `-verbose` позволяет вывести дополнительную информацию во время работы утилиты `javadoc`.

`javadoc -d /tmp/ HelloWorld.java`

Параметр `-d` определяет каталог, в который будут помещаться генерируемые HTML-файлы. Без этой опции генерируемые HTML-файлы будут помещаться в текущий рабочий каталог.

`javadoc -sourcespath /classes/ Test.java`

Параметр `-sourcespath` определяет путь, где находятся исходные файлы `.java` пользователя.

`javadoc -exclude <пакеты> Test.java`

Параметр `-exclude` указывает, для каких именно пакетов не нужно генерировать HTML-файлы документации.

`javadoc -public Test.java`

Параметр `-public` создает документацию только для открытых классов и членов.

`javadoc -protected Test.java`

Параметр `-protected` создает документацию для защищенных и открытых классов и членов. Этот вариант используется по умолчанию.

`javadoc -package Test.java`

Параметр `-package` создает документацию для всего пакета, а также для защищенных и открытых классов и членов.

`javadoc -private Test.java`

Параметр `-private` создает документацию для всех классов и членов.

`javadoc -help`

Опция `-help` (без аргументов) позволяет вывести справочную информацию о команде `javadoc`.

Опция `classpath`

Параметр `-classpath` используется в некоторых утилитах командной строки. Он указывает виртуальной машине Java (JVM) пути к каталогам, в которых следует искать классы и пакеты, определенные пользователем. Соглашения, касающиеся синтаксиса параметра `-classpath`, зависят от типа используемой операционной системы.

В операционных системах Microsoft Windows каталоги в путях разграничиваются символами обратной косой черты, а для разделения путей используется точка с запятой, как показано ниже.

```
-classpath \home\XClasses\;dir\YClasses\;.
```

В операционных системах, совместимых с POSIX (например, Solaris, Linux и Mac OS X), каталоги в путях разграничиваются символами прямой косой черты, а для разделения путей используется двоеточие:

```
-classpath /home/XClasses/:dir/YClasses/:.
```

НА ЗАМЕТКУ

Символ точки представляет текущий рабочий каталог.

Можно также задать переменную среды CLASSPATH, которая указывает компилятору Java, где искать пакеты и файлы класса:

```
rem Windows
set CLASSPATH=classpath1;classpath2

# Linux, Solaris, Mac OS X
# (Точная команда зависит от типа используемой
# командной оболочки)
setenv CLASSPATH classpath1:classpath2
```

Управление памятью

В Java реализовано автоматическое управление памятью, которое также называют *сборкой мусора* (garbage collection — GC). Основными задачами GC являются распределение памяти, учет объектов в памяти, на которые имеются ссылки, а также освобождение памяти от объектов, на которые уже нет ссылок.

Сборщики мусора

С появлением выпуска J2SE 5.0 виртуальная машина Java HotSpot автоматически выполняет собственную тонкую настройку. Этот процесс включает попытку оптимизации параметров GC и соответствующих настроек при запуске, исходя из информации о платформе, а также текущую настройку GC.

Хотя исходные параметры и настройка GC в ходе выполнения в целом не вызывают проблем, в некоторых случаях у пользователя может возникнуть желание изменить параметры своей системы GC, исходя из перечисленных ниже целей.

Максимальная длительность паузы

Это желаемое время, на которое GC приостанавливает выполнение соответствующего приложения, чтобы освободить память.

Производительность

В этом случае целью является желаемое время работы приложения или время, в течение которого GC не выполняется.

В последующих разделах приведен обзор разных сборщиков мусора, их основная задача, а также ситуации, в которых они должны использоваться. Ниже, в разделе "Опции командной строки", объясняется, как установить нужные параметры при запуске Java, чтобы настроить GC вручную.

Последовательный сборщик мусора

Последовательный сборщик мусора выполняется в одном потоке с приложением на одном процессоре. Когда выполняется этот поток GC, выполнение соответствующего приложения приостанавливается до момента завершения сборки мусора.

Такой сборщик мусора лучше всего использовать, когда ваше приложение оперирует относительно небольшим набором данных (не более 100 Мбайт) и при этом не выдвигаются никакие специальные требования по уменьшению длительности пауз.

Параллельный сборщик мусора

Параллельный сборщик мусора, называемый также быстродействующим сборщиком, может выполняться в нескольких потоках на нескольких процессорах. Использование нескольких потоков существенно ускоряет работу GC.

Такой сборщик мусора лучше всего использовать, когда длительность паузы не имеет особого значения, но при этом одним из требований является максимальная скорость работы приложения.

Параллельный сборщик мусора с уплотнением

Параллельный сборщик мусора с уплотнением подобен параллельному сборщику, рассмотренному выше, однако в нем применяются улучшенные алгоритмы, которые позволяют сократить длительность паузы для сборки мусора.

Этот сборщик мусора лучше всего использовать для приложений, в которых наложены специальные требования к длительности паузы.

НА ЗАМЕТКУ

Параллельный сборщик с уплотнением стал доступным для использования в шестом обновлении J2SE 5.0.

Сборщик мусора с одновременной маркировкой и очисткой

В сборщике мусора с одновременной маркировкой и очисткой (Concurrent Mark-Sweep Collector — CMS Collector), называемом также сборщиком с малым временем ожидания, реализованы алгоритмы, позволяющие выполнять очистку крупных блоков памяти, что может вызвать длительную задержку.

Этот сборщик мусора лучше всего использовать, когда время реакции приложения важнее производительности и длительности пауз, вызываемых работой GC.

Сборщик мусора Garbage-First (G1)

Сборщик мусора Garbage-First, называемый также как сборщиком G1, используется для многопроцессорных машин с большими объемами оперативной памяти. Этот серверный GC с высокой степенью вероятности достигает целей, касающихся длительности паузы, обеспечивая при этом высокую производительность. Операции со всей кучей (например, глобальная маркировка) выполняются одновременно с потоками приложений, предотвращая прерывания пропорционально размерам кучи, или используемых в приложении данных.

НА ЗАМЕТКУ

Сборщик мусора Garbage-First стал доступным для использования в четвертом обновлении Java SE 7. Он был создан в качестве замены сборщику CMS.

Средства управления памятью

Несмотря на то что настройка сборщика мусора может оказаться вполне успешной для какой-то конкретной ситуации, важно отметить, что здесь нельзя гарантировать результат, поэтому ставятся только цели. Любое улучшение, достигнутое на одной

платформе, может не сработать на другой. Лучше всего выявить источник проблемы с помощью средств управления памятью, в том числе с помощью программы-профилировщика.

Такие средства перечислены в табл. 11.1. Все они представляют собой приложения командной строки, за исключением HPROF (Heap/CPU Profiling Tool). HPROF динамически загружается при указании соответствующей опции в командной строке. Приведенная ниже команда возвращает полный перечень опций командной строки для утилиты HPROF.

```
java -agentlib:hprof=help
```

Таблица 11.1. Средства управления памятью JDK

Средство	Описание
`jvisualvm`	Универсальное средство поиска и устранения ошибок в программах на языке Java
`jconsole`	Средство мониторинга, совместимое со спецификациями расширенных средств управления Java (Java Management Extensions — JMX)
`jinfo`	Средство получения информации о конфигурации
`jmap`	Средство работы с памятью
`jstack`	Средство работы со стеком
`jstat`	Средство мониторинга статистики JVM
`jhat`	Средство для анализа кучи
HPROF Profiler	Позволяет получить статистику использования процессора, кучи и отследить конкуренцию за ресурсы
`jdb`	Средство отладки Java-программ

НА ЗАМЕТКУ

В качестве средства анализа воспользуйтесь пакетом Oracle Java SE Advanced, в который входят утилиты Java Mission Control (`jmc`) и Java Flight Recorder. Эти утилиты относятся к средствам диагностики и мониторинга корпоративных приложений.

НА ЗАМЕТКУ

Чтобы определить, какой именно сборщик мусора используется, ознакомьтесь с информацией, выводимой `jconsole`.

Опции командной строки

Перечисленные ниже опции командной строки, относящиеся к GC, могут передаваться в интерпретатор Java для обеспечения интерфейса с функциональностью виртуальной машины Java HotSpot (с более полным перечнем опций можно ознакомиться в разделе "Java HotSpot VM Options" электронной документации Oracle).

`-XX:+PrintGC` или `-verbose:gc`

Выводит информацию общего характера о куче и сборке мусора во время каждой сборки.

`-XX:+PrintCommandLineFlags -version`

Распечатывает параметры кучи, применяемые значения `-XX` и информацию о версии.

`-XX:+PrintGCDetails`

Распечатывает подробную информацию о куче и сборке мусора во время каждой сборки.

`-XX:+PrintGCTimeStamps`

Добавляет временны*е* метки к выходной информации от `PrintGC` или `PrintGCDetails`.

`-XX:+UseSerialGC`

Инициализирует работу последовательного сборщика.

`-XX:+UseParallelGC`

Инициализирует работу параллельного сборщика.

`-XX:+UseParallelOldGC`

Инициализирует работу параллельного сборщика с уплотнением. Обратите внимание: слово `Old` указывает на то, что новые алгоритмы используются для "старого" поколения GC.

`-XX:+UseParNewGC`

Инициализирует работу параллельного сборщика "нового" поколения. Может использоваться в сочетании со сборщиком с малым временем ожидания (с одновременной маркировкой и очисткой).

`-XX:+UseConcMarkSweepGC`

Инициализирует работу сборщика CMS с одновременной маркировкой и очисткой (сборщика с малым временем ожидания). Может использоваться в сочетании с параллельным сборщиком "нового" поколения.

`-XX:+UseG1GC`

Инициализирует работу сборщика Garbage-First.

`-XX:+DisableExplicitGC`

Блокирует явные методы GC (`System.gc()`).

`-XX:ParallelGCThreads=[`*потоки*`]`

Определяет количество потоков GC. Вариант, предусмотренный по умолчанию, зависит от количества процессоров. Эта опция применяется к CMS и параллельным сборщикам.

`-XX:MaxGCPauseMillis=[`*миллисекунды*`]`

Служит для GC подсказкой относительно желаемой максимальной длительности паузы (выраженного в миллисекундах). Эта опция применяется к параллельным сборщикам.

`-XX:+GCTimeRatio=[`*значение*`]`

Служит для GC подсказкой относительно желаемого отношения времени работы GC ко времени работы приложения (1 / (1 + [*значение*])) при достижении цели желательной производительности. Значение по умолчанию равно 99. Это означает, что для выполнения соответствующего приложения будет отведено 99% времени и, следовательно, для выполнения GC будет отведен лишь один процент времени. Эта опция применяется к параллельным сборщикам.

`-XX:+CMSIncrementalMode`

Инициирует инкрементальный режим только для сборщика CMS. Используется для машин с одним или двумя процессорами.

`-XX:+CMSIncrementalPacing`

Разрешает автоматическую упаковку только для сборщика CMS.

`-XX:MinHeapFreeRatio=[`*процент*`]`

Устанавливает минимальный требуемый процент для отношения свободного пространства к общему размеру кучи. По умолчанию это соотношение составляет 40%.

`-XX:MaxHeapFreeRatio=[`*процент*`]`

Устанавливает максимальный требуемый процент для отношения свободного пространства к общему размеру кучи. По умолчанию это соотношение составляет 70%.

`-Xms[`*байты*`]`

Переопределяет минимальный размер кучи в байтах. По умолчанию устанавливается $^1/_{64}$-я часть от физического размера памяти системы размером до 1 Гбайт. Начальный размер кучи составляет 4 Мбайт для машин, не относящихся к классу серверов.

`-Xmx[`*байты*`]`

Переопределяет максимальный размер кучи в байтах. По умолчанию берется меньше $^1/_4$-й физического размера памяти системы размером до 1 Гбайт. Максимальный размер кучи составляет 64 Мбайт для машин, не относящихся к классу серверов.

`-Xmn[`*байты*`]`

Размер кучи для нового поколения.

`-XX:OnError=[`*команда [опции]*`]`

Используется для указания сценариев или команд, заданных пользователем, выполняемых, когда происходит фатальная ошибка.

`-XX+AggressiveOpts`

Разрешает оптимизацию производительности, параметры которой, как ожидается, будут использованы по умолчанию в будущих выпусках.

НА ЗАМЕТКУ

При указании значений в байтах можно использовать обозначения [k | K] для килобайтов, [m | M] для мегабайтов и [g | G] для гигабайтов.

Обратите внимание: стабильность опций -XX не гарантируется. Они не являются частью спецификации языка Java (Java Language Specification — JLS) и вряд ли будут присутствовать в указанной выше форме и иметь описанную выше функциональность (если вообще будут присутствовать) в JVM других сторонних разработчиков.

Изменение размера кучи JVM

Куча — это область оперативной памяти, в которой хранятся все объекты, создающиеся при выполнении Java-программы. Размер кучи следует изменять лишь в том случае, если она должна быть больше, чем предусмотрено по умолчанию. Если у вас возникли проблемы с производительностью или вы видите постоянное сообщение PermGen об ошибке от `java.lang.OutOfMemoryError`, это может свидетельствовать о необходимости увеличить размер кучи.

Метапространство

Для области оперативной памяти, которая используется для хранения метаданных класса, создается отдельное пространство памяти, которое называется *метапространством* (Metaspace). Метапространство пришло на смену модели PermGen (Permanent Generation). Поэтому в новой версии виртуальной машины

Java HotSpot, входящей в JDK, 8 вы больше не увидите ошибок PermGen типа `OutOfMemoryError`. В случае возникновения утечек памяти воспользуйтесь средством JVisualVM, которое может анализировать данные метапространства.

Взаимодействие с GC

Взаимодействие со сборщиком мусора может осуществляться путем явного обращения (вызова) или переопределения метода `finalize`.

Сборка мусора в явном виде

К сборщику мусора можно обратиться в явном виде с помощью вызова метода `System.gc()` или `Runtime.getRuntime().gc()`. Однако явного обращения к GC, вообще говоря, следует избегать, поскольку это может инициировать процесс полной сборки (в то время как может оказаться вполне достаточно сокращенной сборки), что приведет к необоснованному увеличению времени пауз. Запрос `System.gc()` выполняется не всегда, поскольку JVM время от времени игнорирует такие запросы.

Финализация

У каждого объекта есть метод `finalize()`, унаследованный от класса `Object`. Сборщик мусора, перед тем как уничтожить объект, может вызвать этот метод, однако такой вызов не гарантирован. В стандартной реализации метода `finalize()` не выполняется никаких действий, и, хотя это не рекомендуется, данный метод можно переопределить следующим образом:

```
public class TempClass extends SuperClass {
   ...
   // Выполняется, когда происходит сборка мусора
   protected void finalize() throws Throwable {
      try {
         /* Желаемые функции выполняются здесь */
      } finally {
```

```
        // Как вариант, вы можете вызвать
        // метод finalize этого суперкласса
        super.finalize(); // Из SuperClass
    }
  }
}
```

В приведенном ниже примере выполняется уничтожение объекта.

```
public class MainClass {
  public static void main(String[] args) {
    TempClass t = new TempClass();
    // Удалены ссылки на объект
    t = null;
    // Обращение к GC
    System.gc();
  }
}
```

Основы ввода-вывода

В Java предусмотрен ряд классов для выполнения основных операций ввода-вывода. Несколько таких классов обсуждаются в настоящей главе. Эти базовые классы могут использоваться для чтения и записи в файлы, сокеты и на консоль. Они также позволяют работать с файлами и каталогами и могут использоваться в процессе сериализации данных. Классы ввода-вывода в Java генерируют исключения, в том числе `IOException`, которые необходимо обрабатывать.

Классы ввода-вывода в Java также поддерживают форматирование данных, сжатие и распаковку потоков, шифрование и дешифрование, а также обмен данными между задачами с помощью канальных потоков (piped streams).

Новые API ввода-вывода (new I/O, или NIO), впервые появившиеся в Java 1.4, обеспечивают дополнительные возможности ввода-вывода, в том числе буферизацию, блокирование файлов, проверку регулярных выражений, поддержку масштабируемых сетевых решений и управление буферами.

Интерфейс NIO 2.0 впервые появился в Java SE 7 (о нем рассказывается в следующей главе). NIO 2.0 является расширением NIO и обеспечивает новый API для работы с файловой системой.

Стандартные потоки `in`, `out` и `err`

В Java используются три стандартных потока: `in`, `out` и `err`.

- `System.in` — стандартный поток ввода, который используется для передачи данных от пользователя к программе.

```
byte teamName[] = new byte[200];
int size = System.in.read(teamName);
System.out.write(teamName,0,size);
```

- `System.out` — стандартный поток вывода, который используется для вывода данных из программы пользователю.

  ```
  System.out.print("Игра окончена!");
  ```

- `System.err` — стандартный поток вывода, который используется для вывода данных об ошибках из программы пользователю.

  ```
  System.err.println("Мало участников");
  ```

Обратите внимание на то, что каждый из рассмотренных выше методов может генерировать исключение `IOException`.

НА ЗАМЕТКУ

Класс `Console`, впервые появившийся в Java SE 6, служит альтернативой стандартным потокам ввода-вывода и предназначен для обмена данными с устройствами, поддерживающими режим командной строки (так называемыми консолями, или текстовыми терминалами).

Иерархия основных классов ввода-вывода

На рис. 12.1 представлена иерархия классов для широко используемых устройств чтения и записи, а также для потоков ввода и вывода. Обратите внимание: классы ввода-вывода могут быть сцеплены друг с другом с целью повышения эффективности операций ввода-вывода.

Классы `Reader` и `Writer` используются для чтения и записи символьных данных (текста). Классы `InputStream` и `OutputStream` обычно используются для чтения и записи двоичных данных.

Чтение и запись файлов

В Java предусмотрены удобные средства для чтения и записи файлов.

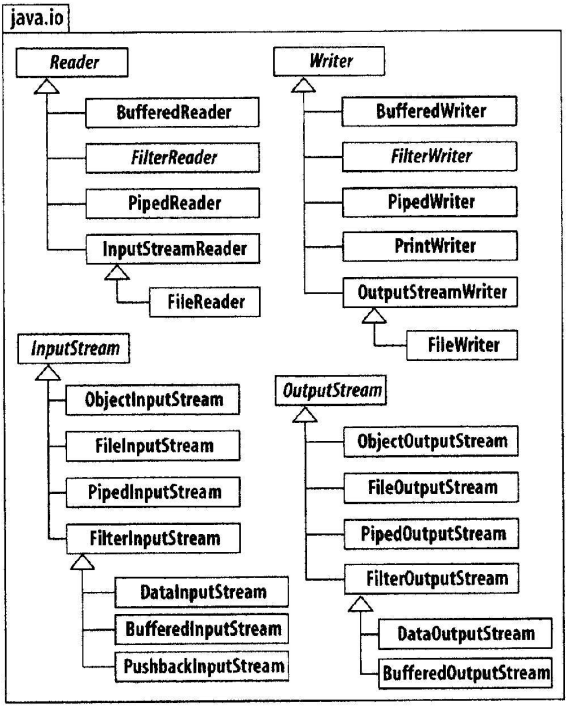


Рис. 12.1. Иерархия типичных классов для чтения и записи, а также потокового ввода-вывода

Чтение символьных данных из файла

Чтобы прочитать символьные данные из файла, нужно воспользоваться классом `BufferedReader`. Можно также воспользоваться классом `FileReader`, но в случае большого объема данных класс `FileReader` будет не так эффективен, как `BufferedReader`. Вызов метода `readLine()` позволяет прочитать строку текста из соответствующего файла. По завершении чтения вызовите метод `close()` для объекта `BufferedReader`.

```
BufferedReader bReader = new BufferedReader
   (new FileReader("Master.txt"));
String lineContents;
while ((lineContents = bReader.readLine())
```

```
   != null) {...}
bReader.close();
```

НА ЗАМЕТКУ

Чтобы избежать проблем с кодировками при обработке текстовых файлов воспользуйтесь средством NIO 2.0 `Files.newBufferedReader(<path>,<charset>)`.

Чтение двоичных данных из файла

Чтобы прочитать двоичные данные из файла, нужно воспользоваться объектом `DataInputStream`. Вызов метода `read()` позволяет прочитать данные из входного потока. Обратите внимание: если должен быть прочитан только некий массив байтов, следует воспользоваться классом `InputStream`.

```
DataInputStream inStream = new DataInputStream
   (new FileInputStream("Team.bin"));
inStream.read();
```

Если необходимо прочитать большой объем данных, то следует воспользоваться классом `BufferedInputStream`, чтобы чтение данных было более эффективным.

```
DataInputStream inStream = new DataInputStream
   (new BufferedInputStream(new FileInputStream(team)));
```

Прочитанные двоичные данные можно возвратить в соответствующий поток с помощью методов класса `PushbackInputStream`.

```
unread(int i); // возвратить один байт
unread(byte[] b); // возвратить массив байтов
```

Запись символьных данных в файл

Чтобы записать символьные данные в файл, воспользуйтесь классом `PrintWriter`. Вызовите метод `close()` класса `PrintWriter`, когда запись в выходной поток будет завершена.

```
String in = "Огромная строка текста";
PrintWriter pWriter = new PrintWriter
   (new FileWriter("CoachList.txt"));
pWriter.println(in);
pWriter.close();
```

Небольшой объем текста можно также записать в файл с помощью класса `FileWriter`. Если файл, передаваемый конструктору класса `FileWriter`, не существует, он будет создан автоматически.

```
FileWriter fWriter = new
   FileWriter("CoachList.txt");
fwriter.write("Это список тренеров");
fwriter.close();
```

Запись двоичных данных в файл

Чтобы записать в файл двоичные данные, воспользуйтесь классом `DataOutputStream`. Вызов метода `writeInt()` позволяет записать массив целых чисел в выходной поток.

```
File positions = new File("Positions.bin");
Int[] pos = {0, 1, 2, 3, 4};
DataOutputStream outStream = new DataOutputStream
   (new FileOutputStream(positions));
for (int i = 0; i < pos.length; i++)
   outStream.writeInt(pos[i]);
```

Если необходимо записать большой объем данных, воспользуйтесь классом `BufferedOutputStream`

```
DataOutputStream outStream = new DataOutputStream
(new BufferedOutputStream(positions));
```

Чтение и запись сокетов

В Java также предусмотрены удобные средства для чтения и записи системных сокетов.

Чтение символьных данных из сокета

Чтобы прочитать символьные данные из сокета, сначала подключитесь к этому сокету, а затем воспользуйтесь классом `BufferedReader`.

```
Socket socket = new Socket("127.0.0.1", 64783);
InputStreamReader reader = new InputStreamReader
   (socket.getInputStream());
BufferedReader bReader = new BufferedReader(reader);
String msg = bReader.readLine();
```

В Java SE 8 в классе `BufferedReader` появился метод `lines()`, относящийся к новому потоковому API (Stream API). Он возвращает объект типа `Stream`, элементы которого являются строками, которые были прочитаны ранее из объекта типа `BufferedReader`.

Чтение двоичных данных из сокета

Чтобы прочитать двоичные данные из сокета, воспользуйтесь классом `DataInputStream`. Вызов метода `read()` позволяет прочитать данные из входного потока. Обратите внимание: класс `Socket` находится в пакете `java.net`.

```
Socket socket = new Socket("127.0.0.1", 64783);
DataInputStream inStream = new DataInputStream
   (socket.getInputStream());
inStream.read();
```

Если требуется прочитать большой объем данных из сокета, то, чтобы повысить эффективность этой операции, можно также воспользоваться классом `BufferedInputStream`.

```
DataInputStream inStream = new DataInputStream
   (new BufferedInputStream(socket.getInputStream()));
```

Запись символьных данных в сокет

Чтобы записать символьные данные в сокет, сначала подключитесь к этому сокету, а затем создайте и используйте объект класса `PrintWriter`.

```
Socket socket = new Socket("127.0.0.1", 64783);
PrintWriter pWriter = new PrintWriter
   (socket.getOutputStream());
pWriter.println("Батя, мы выиграли эту игру!");
```

Запись двоичных данных в сокет

Чтобы записать двоичные данные в сокет, воспользуйтесь классом `DataOutputStream`. Вызов метода `write()` позволяет записать такие данные в выходной поток.

```
byte positions[] = new byte[10];
Socket socket = new Socket("127.0.0.1", 64783);
DataOutputStream outStream = new DataOutputStream
   (socket.getOutputStream());
outStream.write(positions, 0, 10);
```

Если требуется записать большой объем данных в сокет, то можно также воспользоваться классом `BufferedOutputStream`.

```
DataOutputStream outStream = new DataOutputStream
   (new BufferedOutputStream(socket.getOutputStream()));
```

Сериализация

Чтобы сохранить версию какого-либо объекта (и всех связанных с ним данных, которые затем должны быть восстановлены) в массиве байтов, в классе этого объекта должны быть реализованы методы интерфейса `Serializable`. Обратите внимание: временные члены данных, объявленные как `transient`, не будут включены в сериализуемый объект. При использовании сериализации и десериализации (т.е. восстановления сохраненной версии объекта) нужно быть внимательным, поскольку изменения в классе — в том числе перемещения соответствующего класса в иерархии классов, удаление какого-либо поля, изменение атрибута какого-либо поля из временного в постоянное (nontransient) или статическое, а также использование разных JVM — могут повлиять на процесс восстановления объекта.

Для чтения и записи сериализованных данных в файл можно использовать классы `ObjectOutputStream` и `ObjectInputStream`.

Сериализация

Для записи сериализованных данных объекта в файл воспользуемся классом `ObjectOutputStream`.

```
ObjectOutputStream s = new
   ObjectOutputStream(new FileOutputStream("p.ser"));
```

Ниже приведен пример сериализации.

```
ObjectOutputStream oStream = new
   ObjectOutputStream(new
   FileOutputStream("PlayerDat.ser"));
oStream.writeObject(player);
oStream.close();
```

Десериализация

Чтобы восстановить сериализованный объект из файла (т.е. превратить его из плоской версии в объект), воспользуйтесь объектом `ObjectInputStream`, затем прочитайте данные из соответствующего файла и приведите полученный объект к нужному типу.

```
ObjectInputStream d = new
   ObjectInputStream(new FileInputStream("p.ser"));
```

Ниже приведен пример десериализации.

```
ObjectInputStream iStream = new
   ObjectInputStream(new
   FileInputStream("PlayerDat.ser"));
Player p = (Player) iStream.readObject();
```

Сжатие и распаковка файлов

В Java предусмотрены классы, предназначенные для создания сжатых файлов-архивов в формате ZIP или GZIP.

Работа с ZIP-архивами

Для создания ZIP-архивов используется класс `ZipOutputStream`, а для извлечения файлов из этого архива — класс `ZipInputSteam`.

```
ZipOutputStream zipOut = new ZipOutputStream(
        new FileOutputStream("out.zip"));
   String[] fNames = new String[] {"f1", "f2"};
   for (int i = 0; i < fNames.length; i++) {
   ZipEntry entry = new ZipEntry(fNames[i]);
   File InputStream fin =
        new FileInputStream(fNames[i]);
   try {
      zipOut.putNextEntry(entry);
      for (int a = fin.read();
         a != -1; a = fin.read()) {
            zipOut.write(a);
      }
      fin.close();
      zipOut.close();
   } catch (IOException ioe) {...}
}
```

Чтобы разархивировать файлы, создайте объект типа `ZipInputStream`, вызовите его метод `getNextEntry()` и прочитайте соответствующий файл и выведите его в `OutputStream`.

Архивы в формате GZIP

Чтобы создать GZIP-файл, создайте новый объект типа `GZIPOutputStream`, передайте его конструктору имя файла с расширением `.gzip`, а затем передайте соответствующие данные из `GZIPOutputStream` в `FileInputStream`.

Чтобы распаковать GZIP-файл, создайте объект типа `GZipInputStream`, создайте новый объект типа `FileOutputStream` и прочитайте в него соответствующие данные.

Работа с файлами и каталогами

В Java предусмотрен специальный класс `File`, предназначенный для работы с файлами и каталогами. Он позволяет получить доступ к существующим файлам, выполнять позиционирование данных в файле, создавать каталоги и получать их список файлов, а также удалять файлы и каталоги.

Часто используемые методы класса `File`

В табл. 12.1 перечислены часто используемые методы класса `File`.

Таблица 12.1. Часто используемые методы класса `File`

Метод	Описание
`delete()`	Удаляет файл или каталог
`exists()`	Проверяет, существует ли файл
`list()`	Выводит содержимое каталога
`mkdir()`	Создает каталог
`renameTo(File f)`	Переименовывает файл

Доступ к существующим файлам

К существующим файлам можно получить доступ через класс `File`. Данный класс `File` представляет какой-либо файл или каталог; однако он не имеет доступа к содержимому файла.

Чтобы создать объект `File` с помощью лишь имени файла, воспользуйтесь следующим кодом:

```
File roster = new File("Roster.txt");
```

Чтобы создать объект `File`, представляющий определенный файл в некотором каталоге, воспользуйтесь приведенным ниже кодом.

```
File rosterDir = new File("/usr/rosters");
File roster = new File(rosterDir, "Roster.txt");
```

Позиционирование данных в файле

Чтобы прочитать и записать данные в определенной позиции в файле, воспользуйтесь методом `seek()` класса `RandomAccessFile`. Этот класс часто создается с атрибутами только для чтения или для чтения и записи, обозначаемыми символами `'r'` и `'rw'` при вызове конструктора. Большинство файлов с произвольным доступом являются двоичными файлами, содержащими записи фиксированной длины.

```
File team = new File("Team.txt");
RandomAccessFile raf = new
   RandomAccessFile(team, "rw");
raf.seek(10);
byte data = raf.readByte();
```

Новое API ввода-вывода NIO 2.0

Интерфейс NIO 2.0 впервые появился в JDK 7. Целью его разработки было усовершенствование поддержки системы ввода-вывода и доступа к стандартной файловой системе. Интерфейс NIO 2.0 поддерживается пакетами `java.nio.file` и `java.nio.file.attribute`. API NIO 2.0 описан в документе "JSR 203: More New I/O APIs for the Java Platform" ("JSR 203: дополнительные новые API ввода-вывода для платформы Java"). Популярными интерфейсами, которые используются из API, являются `Path`, `PathMatcher`, `FileVisitor` и `WatchService`. Популярными классами, которые используются из API, являются `Paths` и `Files`.

Интерфейс Path

Интерфейс `Path` может использоваться для работы с путями к файлам и каталогам. Этот интерфейс представляет собой усовершенствованную версию класса `java.io.File`. В приведенном ниже коде продемонстрировано использование некоторых методов интерфейса `Path` и класса `Paths` для получения информации.

```
Path p = Paths.get("\\opt\\jpgTools\\README.txt");
System.out.println(p.getParent());     // \opt\jpgTools
System.out.println(p.getRoot());       // \
System.out.println(p.getNameCount()); // 3
System.out.println(p.getName(0));      // opt
System.out.println(p.getName(1));      // jpgTools
System.out.println(p.getFileName());  // README.txt
System.out.println(p.toString());      // Полный путь
```

У интерфейса `Path` есть несколько дополнительных методов; некоторые из них описаны в табл. 13.1.

Таблица 13.1. Методы интерфейса `Path`

Метод	Описание
`path.toUri()`	Преобразует путь в объект URI
`path.resolve(Path)`	Объединяет два пути
`path.relativize(Path)`	Формирует путь из одного места в другое
`path.compareTo(Path)`	Сравнивает между собой два пути

Класс `Files`

Класс `Files` может использоваться для создания, проверки, удаления, копирования или перемещения какого-либо файла или каталога. В приведенном ниже коде показаны некоторые широко используемые методы класса `Files` в действии.

```
// Создать каталог
Files.createDirectories("\\opt\\jpg");

// Создать экземпляры объектов пути
Path target1 = Paths.get("\\opt\\jpg\\README1.txt");
Path p1 = Files.createFile(target1);
Path target2 = Paths.get("\\opt\\jpg\\README2.txt");
Path p2 = Files.createFile(target2);

// Проверить атрибуты файла
System.out.println(Files.isReadable(p1));
System.out.println(Files.isReadable(p2));
System.out.println(Files.isExecutable(p1));
System.out.println(Files.isSymbolicLink(p1));
System.out.println(Files.isWritable(p1));
System.out.println(Files.isHidden(p1));
System.out.println(Files.isSameFile(p1, p2));

// Удалить, переместить и копировать экземпляры
Files.delete(p2);
System.out.println(Files.move(p1, p2));
System.out.println(Files.copy(p2, p1));
Files.delete(p1);
Files.delete(p2);
```

Кроме двух обязательных параметров, методу `move()` можно передать дополнительные параметры — константы, входящие в перечисление, такие как `REPLACE_EXISTING` или `ATOMIC_MOVE`, которые влияют на выполнение операции перемещения. Например, если указать в качестве третьего параметра константу `REPLACE_EXISTING`, операция перемещения файла завершится успешно, даже если выходной файл уже существует. Указание константы `ATOMIC_MOVE` гарантирует, что любой процесс, который "видит" соответствующий каталог, сможет получить доступ ко всему файлу.

Методу `copy()` также можно передать дополнительные параметры — константы, входящие в перечисление, такие как `REPLACE_EXISTING`, `COPY_ATTRIBUTES` или `NOFOLLOW_LINKS`. При указании константы `REPLACE_EXISTING` файл будет скопирован, даже если уже существует целевой файл с таким же именем. При указании константы `COPY_ATTRIBUTES` при копировании будут скопированы атрибуты файла, а при указании `NOFOLLOW_LINKS` копируются ссылки, а не сами файлы.

В новом Stream API к классу `Files` были добавлены методы `lines()`, `list()`, `walk()` и `find()`. Метод `lines()` позволяет прочитать набор строк. Метод `list()` предназначен для анализа содержимого каталога, а метод `walk()` позволяет совершить рекурсивный обход элементов файловой системы. С помощью метода `find()` класса `Path` можно выполнить поиск файлов в дереве файловой системы, которое начинается с указанного узла.

Дополнительные возможности

Кроме того, API NIO 2.0 обеспечивает перечисленные ниже возможности, о которых следует знать программисту. Вопросы, касающиеся этих возможностей, включены также в экзамен для программистов, желающих получить звание "Сертифицированный специалист Oracle по Java SE 8" (Oracle Certified Professional Java SE 8 Exam). В настоящей книге эти вопросы не освещаются,

поскольку они в большей степени подходят для руководства или ресурса, имеющего характер учебного пособия.

- Просмотр каталога с помощью интерфейса `WatchService`.
- Рекурсивный обход деревьев каталогов с помощью интерфейса `FileVisitor`.
- Поиск файлов с помощью интерфейса `PathMatcher`.

Поскольку `PathMatcher` относится к функциональным интерфейсам, его можно использовать совместно с лямбда-выражениями.

```
PathMatcher matcher = (Path p) -> {
   // Возвращается булево значение
   return (p.toString().contains("World"));
};
Path path = FileSystems.getDefault().getPath(
   "\\opt\\jpg\\HelloWorld.java");
   System.out.print("Соответствует: " +
    matcher.matches(path));

$ Соответствует: true
```

НА ЗАМЕТКУ

Для циклического перебора элементов каталога воспользуйтесь новым функциональным интерфейсом `java.nio.file.DirectoryStream`.

НА ЗАМЕТКУ

Исчерпывающую информацию по интерфейсу NIO 2.0 можно найти в книге Кея Хорстманна и Гари Корнелла *Java. Библиотека профессионала, том 1. Основы*, 9-е издание, выпущенной ИД "Вильямс" в 2014 году.

Параллелизм

Многопоточное программирование в Java позволяет более эффективно использовать несколько процессоров или несколько ядер одного процессора. Благодаря потокам на одном процессоре можно выполнять параллельные операции, такие как ввод-вывод и одновременную обработку данных.

Многопоточное программирование в Java поддерживается с помощью класса `Thread` и интерфейса `Runnable`.

Создание потоков

Потоки можно создать двумя способами: либо путем расширения класса `java.lang.Thread`, либо реализации в классе методов интерфейса `java.lang.Runnable`.

Расширение класса `Thread`

Путем расширения класса `Thread` и переопределения его метода `run()` можно создать класс, позволяющий запускать параллельно выполняющиеся потоки. Ниже показан удобный способ инициализации потока.

```
class Comet extends Thread {
   public void run() {
      System.out.println("Выход на орбиту");
      orbit();
   }
}
Comet halley = new Comet();
hally.run();
```

Учтите, что в Java не поддерживается множественное наследование. Поэтому расширить можно только один суперкласс, а

значит, класс, расширяющий Thread, не может одновременно расширять какой-либо другой суперкласс.

Реализация интерфейса Runnable

При реализации функционального интерфейса Runnable и определении его метода run() появляется возможность запускать параллельно выполняющиеся потоки в текущем классе. Для создания потока нужно создать новый объект типа Thread и передать его конструктору экземпляр класса, поддерживающий интерфейс Runnable, как показано ниже.

```
class Asteroid implements Runnable {
   public void run() {
      System.out.println("Выход на орбиту");
      orbit();
   }
}

Asteroid majaAsteroid = new Asteroid();
Thread majaThread = new Thread(majaAsteroid);
majaThread.run();
```

Отдельно взятый экземпляр объекта, поддерживающий интерфейс Runnable, можно передать нескольким объектам потока. В результате каждый поток будет выполнять одну и ту же задачу, как показано ниже на примере лямбда-выражения.

```
Runnable asteroid = () -> {
   System.out.println("Выход на орбиту!");
   orbit();
};

Thread asteroidThread1 = new Thread(asteroid);
Thread asteroidThread2 = new Thread(asteroid);
asteroidThread1.run();
asteroidThread2.run();
```

Состояния потока

Как следует из табл. 14.1, в перечислении Thread.state предусмотрено шесть состояний потока.

Таблица 14.1. Состояния потока

Состояние потока	Описание
NEW	Поток, который создан, но не запущен
RUNNABLE	Поток, который готов к выполнению
BLOCKED	Работающий поток, который заблокирован до снятия блокировки
WAITING	Работающий поток, в котором вызван метод `wait()` или `join()`, для ожидания завершения какого-либо другого потока
TIMED_WAITING	Работающий поток, который находится в ожидании какого-либо другого потока в течение заданного периода времени (состояние "сна")
TERMINATED	Поток, выполнение которого завершено

Приоритеты потоков

Допустимый диапазон приоритетов потока — от 1 до 10 включительно; по умолчанию приоритет равен 5. Приоритеты потоков являются одним из наименее переносимых аспектов Java, поскольку их диапазон и значения, предусмотренные по умолчанию, могут различаться у разных виртуальных машин Java. Определить приоритеты можно с помощью констант `MIN_PRIORITY`, `NORM_PRIORITY` и `MAX_PRIORITY`.

```
System.out.print(Thread.MAX_PRIORITY);
```

Потоки с более низкими приоритетами пропускают вперед потоки с более высокими приоритетами.

Типичные методы

В табл. 14.2 перечислены типичные методы класса `Thread`, используемые при работе с потоками.

Таблица 14.2. Методы для работы с потоками

Метод	Описание
`getPriority()`	Возвращает приоритет потока
`getState()`	Возвращает состояние потока

Метод	Описание
`interrupt()`	Прерывает выполнение потока
`isAlive()`	Возвращает действующее состояние потока
`isInterrupted()`	Проверяет наличие прерывания в потоке
`join()`	Заставляет поток, вызвавший этот метод, ждать момента завершения другого потока, который этот объект представляет
`setPriority(int)`	Устанавливает приоритет потока
`start()`	Переводит поток в выполнимое состояние

В табл. 14.3 перечислены типичные методы класса `Object`, используемые для работы с потоками.

Таблица 14.3. Методы класса `Object` для работы с потоками

Метод	Описание
`notify()`	Выводит поток из состояния сна и запускает его на выполнение
`notifyAll()`	Выводит все потоки, которые ожидают окончания работы какого-либо потока или освобождения ресурса, из состояния сна; затем планировщик выберет один из этих потоков для выполнения
`wait()`	Переводит поток в состояние ожидания до тех пор, пока какой-либо другой поток не вызовет метод `notify()` или `notifyAll()`

НА ЗАМЕТКУ

В результате вызовов методов `wait()` и `notify()` генерируется исключение `InterruptedException`, если эти вызовы были сделаны в потоке, находящемся в прерванном состоянии (флаг `Interrupted` установлен в значение `true`).

В табл. 14.4 перечислены типичные статические методы класса `Thread`, используемые для работы с потоками, такие как `Thread.sleep(1000)`.

Таблица 14.4. Статические методы для работы с потоками

Метод	Описание
`activeCount()`	Возвращает количество потоков в группе текущего потока
`currentThread()`	Возвращает ссылку на поток, выполняющийся в данный момент
`interrupted()`	Проверяет, не находится ли поток, выполняющийся в данный момент, в состоянии прерывания
`sleep(`*время*`)`	Блокирует поток, выполняющийся в данный момент, на количество миллисекунд, указанное *параметром*
`yield()`	Переводит поток, выполняющийся в данный момент, в неактивное состояние, чтобы дать возможность выполниться другим потокам

Синхронизация

Ключевое слово `synchronized` позволяет применить блокировки к блокам и методам. Это следует делать в случае, если в блоках и методах есть обращения к совместно используемым критическим ресурсам. В результате применения ключевого слова `synchronized` блокировка будет автоматически ставиться в коде после открытия фигурной скобки и сниматься после ее закрытия. Ниже приведено несколько примеров синхронизированных блоков и методов.

- Экземпляр объекта `t` с синхронизированной блокировкой:

```
synchronized (t) {
   // Тело блока
}
```

- Экземпляр объекта `this` с синхронизированной блокировкой:

```
synchronized (this) {
   // Тело блока
}
```

- Метод `raise()` с синхронизированной блокировкой:

```
// Эквивалентный сегмент кода 1
synchronized void raise() {
   // Тело метода
}
```

```
// Эквивалентный сегмент кода 2
void raise() {
   synchronized (this) {
      // Тело метода
   }
}
```

- Статический метод `calibrate()` с синхронизированной блокировкой:

```
class Telescope {
   synchronized static void calibrate() {
      // Тело метода
   }
}
```

НА ЗАМЕТКУ

Блокировку называют также *мьютексом* (от mutually exclusive lock — взаимоисключающая блокировка).

Для работы и управления параллелизмом используется также ряд дополнительных классов, описанных ниже.

Классы для поддержки параллелизма

В Java 2 SE 5.0 впервые появились служебные классы, используемые при создании параллельно выполняющихся программ. Они находятся в пакете `java.util.concurrent`. К их числу относятся исполнители (executors), классы коллекций для параллельного выполнения, синхронизаторы и средства хронометража.

Исполнители

В классе `ThreadPoolExecutor`, а также его производном классе `ScheduledThreadPoolExecutor` реализованы методы интерфейса `Executor`, обеспечивающего конфигурируемые и гибкие пулы потоков. Пулы потоков позволяют компонентам сервера пользоваться возможностью повторного использования потоков.

В классе Executors предусмотрены методы фабрики (создатели объектов) и служебные методы. Из их числа для создания пулов потоков используются перечисленные ниже методы.

`newCachedThreadPool()`

Создает неограниченный пул потоков, который автоматически повторно использует потоки.

`newFixedThreadPool(int nThreads)`

Создает пул потоков фиксированного размера, который автоматически использует повторно потоки из совместно используемой неограниченной очереди.

`newScheduledThreadPool(int corePoolSize)`

Создает пул потоков, в котором могут быть команды, предназначенные для периодического выполнения или для выполнения с указанной задержкой.

`newSingleThreadExecutor()`

Создает однопотоковый исполнитель, который действует на основе неограниченной очереди.

`newSingleThreadScheduledExecutor()`

Создает однопотоковый исполнитель, в котором могут быть команды, предназначенные для периодического выполнения или выполнения с указанной задержкой.

В приведенном ниже примере продемонстрировано использование метода фабрики `newFixedThreadPool()`.

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class ThreadPoolExample {
   public static void main() {
      // Создать задачи
      // из класса Rtask, реализующего
      // интерфейс Runnable

      RTask t1 = new RTask("thread1");
      RTask t2 = new RTask("thread2");
```

```
        // Создать диспетчер потоков
        ExecutorService threadExecutor =
           Executors.newFixedThreadPool(2);

        // Сделать потоки выполнимыми
        threadExecutor.execute(t1);
        threadExecutor.execute(t2);

        // Завершить потоки
        threadExecutor.shutdown();
    }
}
```

Классы коллекций для параллельного выполнения

Несмотря на то что классы коллекций можно синхронизировать, лучше воспользоваться классами с потоко-безопасным параллельным выполнением, которые обеспечивают выполнение аналогичных функций. Соответствие между классами коллекций и их потоко-безопасными эквивалентами указано в табл. 14.5.

Таблица 14.5. Коллекции и их потоко-безопасные эквиваленты

Класс коллекции	Потоко-безопасный эквивалент
`HashMap`	`ConcurrentHashMap`
`TreeMap`	`ConcurrentSkipListMap`
`TreeSet`	`ConcurrentSkipListSet`
Подтипы `Map`	`ConcurrentMap`
Подтипы `List`	`CopyOnWriteArrayList`
Подтипы `Set`	`CopyOnWriteArraySet`
`PriorityQueue`	`PriorityBlockingQueue`
`Deque`	`BlockingDeque`
`Queue`	`BlockingQueue`

Синхронизаторы

Синхронизаторы представляют собой специализированные средства, предназначенные для синхронизации выполнения потоков. Имеющиеся в наличии синхронизаторы перечислены в табл. 14.6.

Таблица 14.6. Синхронизаторы

Синхронизатор	Описание
`Semaphore`	Обслуживает совокупность разрешений
`CountDownLatch`	Реализует режим ожидания в соответствии с набором выполняемых операций
`CyclicBarrer`	Реализует режим ожидания в соответствии с типичными барьерными точками
`Exchanger`	Реализует точку синхронизации, в которой потоки могут обмениваться элементами

Средства хронометража

Перечисление `TimeUnit` обычно используется для информирования методов, работающих со временем, о том, как следует трактовать переданный интервал времени (см. приведенный ниже пример). Имеющиеся в наличии константы перечисления `TimeUnit` указаны в табл. 14.7.

```
// tyrLock (long time, TimeUnit unit)
if (lock.tryLock(15L, TimeUnit.DAYS)) {...} //15 дней
```

Таблица 14.7. Константы перечисления `TimeUnit`

Константа	Определение единицы	Единица (с)	Обозначение
NANOSECONDS	1/1000 мкс	.000000001	ns
MICROSECONDS	1/1000 мс	.000001	μs
MILLISECONDS	1/1000 с	.001	ms
SECONDS	с	1	sec
MINUTES	60 с	60	min
HOURS	60 мин	3600	hr
DAYS	24 ч	86400	d

Коллекции Java

Инфраструктура Java Collections Framework предназначена для поддержки многочисленных коллекций в иерархическом стиле. По сути, она состоит из интерфейсов, реализаций и статических методов (алгоритмов).

Интерфейс `Collection`

Коллекции — это объекты, которые позволяют сгруппировать многочисленные элементы в одном объекте, а также хранить их, получить доступ к объектам и выполнять манипуляции с ними. Интерфейс `Collection` является родоначальником иерархии коллекций. Кроме него существуют также производные интерфейсы, включая `List`, `Queue` и `Set` (табл. 15.1). В этой же таблице указано, поддерживают ли интерфейсы упорядочиваемость и допускают ли они наличие дубликатов элементов. В эту таблицу также включен интерфейс `Map`, поскольку он является частью инфраструктуры коллекций.

Таблица 15.1. Часто используемые коллекции

Интерфейс	Упорядоченность	Дубликаты	Примечание
`List`	Да	Да	Позиционный доступ; управление вставкой элементов
`Map`	Может быть	Нет (ключи)	Уникальные ключи; сопоставление каждому ключу не более одного значения
`Queue`	Да	Да	Содержит элементы, поставленные в очередь в порядке поступления
`Set`	Может быть	Нет	Уникальность имеет значение

Реализации

В табл. 15.2 перечислены часто используемые реализации классов коллекций и их интерфейсы, а также указано, поддерживают ли интерфейсы упорядочиваемость и допускают ли они наличие дубликатов элементов.

Таблица 15.2. Реализации классов коллекций

Реализация	Интерфейс	Упорядоченность	Сортировка	Дубликаты	Примечания
`ArrayList`	`List`	Индекс	Нет	Да	Массив с быстро изменяемым размером
`LinkedList`	`List`	Индекс	Нет	Да	Двухсвязный список
`Vector`	`List`	Индекс	Нет	Да	Устаревший, синхронизированный
`HashMap`	`Map`	Нет	Нет	Нет	Пары "ключ/значение"
`Hashtable`	`Map`	Нет	Нет	Нет	Устаревший, синхронизированный
`LinkedHashMap`	`Map`	Вставка, по последнему	Нет	Нет	Связанный список/хеш-таблица
`TreeMap`	`Map`	Сбалансированный	Да	Нет	Отображение на основе красно-черного дерева
`PriorityQueue`	`Queue`	Приоритет	Да	Да	Реализация на основе кучи
`HashSet`	`Set`	Нет	Нет	Нет	Набор с быстрым доступом
`LinkedHashSet`	`Set`	Вставка	Нет	Нет	Связанный список/хеш-множество
`TreeSet`	`Set`	Отсортированный	Да	Нет	Множество на основе красно-черного дерева

Методы инфраструктуры коллекций

В интерфейсах, унаследованных от интерфейса `Collection`, предусмотрено несколько важных сигнатур методов (табл. 15.3).

Метод `Collection.stream()` возвращает объект типа `Stream` для выполнения последовательных операций по отношению к исходной коллекции. Метод `Collection.parallelStream()` возвращает объект типа `Stream` для выполнения параллельных операций по отношению к исходной коллекции.

Таблица 15.3. Важные методы интерфейса `Collection`

Метод	Параметры `List`	Параметры `Set`	Параметры `Map`	Возвращает
`add()`	Индекс, элемент	Элемент	—	`boolean`
`contains()`	Объект	Объект	—	`boolean`
`containsKey()`	—	—	Ключ	`boolean`
`containsValue()`	—	—	Значение	`boolean`
`get()`	Индекс	—	Ключ	`Object`
`indexOf()`	Объект	—	—	`int`
`iterator()`	Нет	Нет	—	`Iterator`
`keySet()`	—	—	Нет	`Set`
`put()`	—	—	Ключ, значение	`void`
`remove()`	Индекс или объект	Объект	Ключ	`void`
`size()`	Нет	Нет	Нет	`int`

Алгоритмы класса `Collections`

Класс `Collections` (не путать с интерфейсом `Collection`) содержит несколько важных статических методов (т.е. алгоритмов), которые можно вызвать для самых разных типов коллекций. В табл. 15.4 представлены широко используемые методы класса `Collections`, параметры, которые им передаются, и возвращаемые ими значения.

Таблица 15.4. Статические методы (алгоритмы) класса `Collections`

Метод	Параметры	Возвращает
`addAll()`	`Collection<? super T>, T...`	`boolean`
`max()`	`Collection, [Comparator]`	`<T>`
`min()`	`Collection, [Comparator]`	`<T>`
`disjoint()`	`Collection, Collection`	`boolean`
`frequency()`	`Collection, Object`	`int`
`asLifoQueue()`	`Deque`	`Queue<T>`
`reverse()`	`List`	`void`
`shuffle()`	`List`	`void`
`copy()`	`List destination, List source`	`void`

Окончание табл. 15.4

Метод	Параметры	Возвращает
`rotate()`	`List, int distance`	`void`
`swap()`	`List, int position, int position`	`void`
`binarySearch()`	`List, Object`	`int`
`fill()`	`List, Object`	`void`
`sort()`	`List, Object, [Comparator]`	`void`
`replaceAll()`	`List, Object oldValue, Object newValue`	`boolean`
`newSetFromMap()`	`Map`	`Set<E>`

Подробнее о типизированных параметрах (т.е. <T>) речь пойдет в главе 16.

Эффективность алгоритмов

Алгоритмы и структуры данных оптимизируются по разным критериям: некоторые — для произвольного доступа к элементам или вставки/удаления элементов, другие — для их упорядочивания. В зависимости от потребностей вы должны уметь правильно выбирать нужные алгоритмы и структуры данных.

В табл. 15.5 представлены важные статические методы класса `Collections`, их типы и средние показатели временной эффективности.

Табл. 15.5. Оценка эффективности алгоритмов

Методы	Конкретный тип	Время
`get(),set()`	`ArrayList`	O(1)
`add(),remove()`	`ArrayList`	O(*n*)
`contains(),indexOf()`	`ArrayList`	O(*n*)
`get(),put(),remove(), containsKey()`	`HashMap`	O(1)
`add(),remove(),contains()`	`HashSet`	O(1)
`add(),remove(),contains()`	`LinkedHashSet`	O(1)
`get(),set(),add(),remove()` (с любого конца)	`LinkedList`	O(1)

Окончание табл. 15.5

Методы	Конкретный тип	Время
`get(),set(),add(),remove()` (от индекса)	`LinkedList`	O(*n*)
`contains(),indexOf()`	`LinkedList`	O(*n*)
`peek()`	`PriorityQueue`	O(1)
`add(),remove()`	`PriorityQueue`	O(log *n*)
`remove(),get(),put(), containsKey()`	`TreeMap`	O(log *n*)
`add(),remove(),contains()`	`TreeSet`	O(log *n*)

Символ большого "О" используется для обозначения показателей временной эффективности алгоритма, причем *n* обозначает количество элементов (табл. 15.6).

Таблица 15.6. Смысл обозначения большого "О"

Обозначение	Описание
O(1)	Время является константой независимо от количества элементов
O(*n*)	Время является линейной функцией от количества элементов
O(log *n*)	Время является логарифмической функцией от количества элементов
O(*n* log *n*)	Время является линейно-логарифмической функцией от количества элементов

Функциональный интерфейс `Comparator`

В ряде методов из класса `Collections` предполагается, что объекты из данной коллекции можно сравнивать. Если естественный порядок расположения элементов отсутствует, то можно воспользоваться вспомогательным классом, в котором реализованы методы функционального интерфейса `Comparator`, чтобы указать способ, в соответствии с которым должны быть упорядочены объекты, как показано в примере ниже.

```
public class Crayon {
   private String color;
   public Crayon(String color) {
      this.color = color;
```

```
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public String toString() {
        return this.color;
    }
}

import java.util.Comparator;
public class CrayonSort implements Comparator<Crayon> {
    @Override
    public int compare (Crayon c1, Crayon c2) {
        return c1.getColor().compareTo(c2.getColor());
    }
}

import java.util.ArrayList;
import java.util.Collections;
public class CrayonApp {
    public static void main(String[] args) {
        Crayon crayon1 = new Crayon("фиолетовый");
        Crayon crayon2 = new Crayon("синий");
        Crayon crayon3 = new Crayon("красный");
        Crayon crayon4 = new Crayon("зеленый");
        Crayon crayon5 = new Crayon("желтый");

        ArrayList <Crayon> cList = new ArrayList <>();
        cList.add(crayon1);
        cList.add(crayon2);
        cList.add(crayon3);
        cList.add(crayon4);
        cList.add(crayon5);

        System.out.println("Исходные цвета: " + cList );
        CrayonSort cSort = new CrayonSort(); // Here
        Collections.sort(cList, cSort);
        System.out.println("Отсортировано: " + cList );
    }
}
```

После компиляции и запуска программы на экране отобразится следующее:

```
$ Исходные цвета: [фиолетовый, синий, красный, зеленый,
желтый]
$ Отсортировано: [желтый, зеленый, красный, синий,
фиолетовый]
```

В классе `CrayonSort` реализованы методы интерфейса `Comparator`, которые используются для экземпляра объекта `cSort`. Можно и не создавать отдельный класс `CrayonSort`, а воспользоваться анонимным внутренним классом, как показано ниже.

```
Comparator<Crayon> cSort = new Comparator <Crayon>()
{
   public int compare(Crayon c1, Crayon c2) {
      return c1.getColor().compareTo(c2.getColor());
   }
};
```

Поскольку `Comparator` является функциональным интерфейсом, можно воспользоваться лямбда-выражением, чтобы сделать код более понятным.

```
Comparator <Crayon> cSort = (Crayon c1, Crayon c2)
   -> c1.getColor().compareTo(c2.getColor());
```

Имя класса не обязательно должно явно указываться в списке аргументов, так как лямбда-выражению известны целевые типы аргументов. Другими словами, вместо записи `(Crayon c1, Crayon c2)` можно использовать `(c1, c2)`:

```
// Пример 1
Comparator <Crayon> cSort = (c1, c2)
   -> c1.getColor().compareTo(c2.getColor());
Collections.sort(cList, cSort);

// Пример 2
Collections.sort(cList, (c1, c2)
   -> c1.getColor().compareTo(c2.getColor()));
```

Обобщения

Инфраструктура обобщений (Generics Framework) впервые появилась в Java SE 5.0 и предназначалась для поддержки параметризации типов. Она была обновлена в Java SE 7 и Java SE 8.

Ценность обобщений заключается в значительном сокращении объема кода, который приходится писать при разработке той или иной библиотеки. Еще одним преимуществом обобщений является устранение операции приведения типов во многих случаях.

Были обновлены классы коллекций, класс `Class` и другие библиотеки Java с целью включения в них обобщений.

Более подробно обобщения рассмотрены в книге Кея Хорстманна и Гари Корнелла *Java. Библиотека профессионала, том 1. Основы*, 9-е издание, выпущенной ИД "Вильямс" в 2014 году.

Обобщенные классы и интерфейсы

Для создания параметризованных типов на основе обобщенных классов и интерфейсов нужно после имени класса указать параметр типа, заключенный в угловые скобки (например, `<T>`). При создании экземпляра класса вместо параметра в угловых скобках будет подставлен соответствующий тип.

После создания экземпляра класса обобщенный параметр типа будет применен ко всем методам класса, где он указан. В приведенном ниже примере в методах `add()` и `get()` используется параметризованный тип в качестве аргумента для входного параметра и возвращаемого типа данных соответственно.

```
public interface List <E> extends Collection<E>{
   public boolean add(E e);
   E get(int index);
}
```

При объявлении переменной параметризованного типа указывается ее конкретный тип (например, <Integer>), который будет использоваться вместо параметра типа (т.е., <E>).

Впоследствии, при извлечении элементов из таких объектов, как коллекции, будет устранена необходимость приведения типов, как показано в примере ниже.

```
// Коллекция List/ArrayList с обобщениями
List<Integer> iList = new ArrayList<Integer>();
iList.add(1000);
// Явное приведение типов не требуется
Integer i = iList.get(0);

// Коллекция List/ArrayList без обобщений
List iList = new ArrayList();
iList.add(1000);
// Явное приведение типов требуется
Integer i = (Integer)iList.get(0);
```

В Java SE 7 впервые появился ромбический оператор (оператор, обозначаемый парой угловых скобок, <>), призванный упростить создание обобщенных типов за счет устранения необходимости в дополнительном наборе символов с клавиатуры:

```
// Без использования ромбического оператора
List<Integer> iList1 = new ArrayList<Integer>();

// С использованием ромбического оператора
List<Integer> iList2 = new ArrayList<>();
```

Конструкторы с обобщениями

Конструкторы обобщенных классов не требуют параметров обобщенного типа в качестве аргументов.

```
// Обобщенный класс
public class SpecialList <E> {
   // Конструктор без аргументов
   public SpecialList() {...}
   public SpecialList(String s) {...}
}
```

Обобщенный объект этого класса можно создать следующим образом:

```
SecialList<String> b = new
   SpecialList<String>();
```

Если у конструктора какого-либо обобщенного класса имеется параметр типа, такой как `String`, то обобщенный объект можно создать так:

```
SpecialList<String> b = new
   SpecialList<String>("Джон Мартин");
```

Принцип подстановки

Принцип подстановки разрешает использовать подтипы в случаях, когда их супертип параметризован.

- Переменной данного типа может быть присвоено значение любого подтипа этого типа.
- Метод с параметром данного типа может быть вызван с использованием аргумента любого подтипа этого типа.

Например, подтипами класса `Number` являются следующие типы: `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `BigInteger` и `BigDecimal`.

```
// Переменная типа List объявлена с помощью обобщенного
типа Number
List<Number> nList = new ArrayList<Number>();
nList.add((byte)27); // Byte (автоупаковка)
nList.add((short)30000); // Short
nList.add(1234567890); // Integer
nList.add((long)2e62); // Long
nList.add((float)3.4); // Float
nList.add(4000.8); // Double
nList.add(new BigInteger("92233720368547758l0"));
nList.add(new BigDecimal("2.1e309"));

// Вывести значения подтипа Number из этого списка
for( Number n : nList )
System.out.println(n);
```

Параметры типа, символы подстановки и ограничения

Проще всего объявить обобщенный класс с помощью какого-либо параметра неограниченного типа, например T:

```
public class GenericClass <T> {...}
```

Границы (ограничения) и символы подстановки можно применять к параметру (параметрам) типа, как показано в табл. 16.1.

Таблица 16.1. Параметры типа, границы и символы подстановки

Параметры типа	Описание
`<T>`	Неограниченный тип; то же, что и `<T extends Object>`
`<T, P>`	Неограниченные типы; `<T extends Object>` и `<P extends Object>`
`<T extends P>`	Тип, ограниченный сверху; особый тип T, который является подтипом типа P
`<T extends P & S>`	Тип, ограниченный сверху; особый тип T, который является подтипом типа P и реализует тип S
`<T super P>`	Тип, ограниченный снизу; особый тип T, который является супертипом типа P
`<?>`	Неограниченный символ подстановки; любой тип объекта, такой же, как `<? extends Object>`
`<? extends P>`	Ограниченный символ подстановки; некий неизвестный тип, который является подтипом типа P
`<? extends P & S>`	Ограниченный символ подстановки; некий неизвестный тип, который является подтипом типа P и реализует тип S
`<? super P>`	Символ подстановки, ограниченный снизу; некий неизвестный тип, который является супертипом типа P

Принцип "взять и положить"

Этот принцип уточняет оптимальный способ использования шаблонов с ключевыми словами `extends` и `super`.

- Используйте шаблон с ключевым словом `extends`, когда нужно извлечь значения из какой-либо структуры.

- Используйте шаблон с ключевым словом super, когда нужно записать значения в какую-либо структуру.
- Не используйте символ подстановки, когда нужно и записывать, и извлекать значения из какой-либо структуры.

Шаблон с ключевым словом extends используется в объявлении метода addAll() коллекции List, поскольку этот метод *извлекает* значения из коллекции:

```
public interface List <E> extends Collection<E>{
   boolean addALL(Collection <? extends E> c)
}

List<Integer> srcList = new ArrayList<Integer>();
srcList.add(0);
srcList.add(1);
srcList.add(2);

// Использование метода addAll() с шаблоном с ключевым
словом extends
List<Integer> destList = new ArrayList<Integer>();
destList.addAll(srcList);
```

Шаблон с ключевым словом super используется в объявлении метода addAll() класса Collections, поскольку этот метод *записывает* значения в коллекцию:

```
public class Collections {
   public static <T> boolean addAll
      (Collection<? super T> c, T... elements){...}
}
// Использование метода addAll() с шаблоном с ключевым
словом super
List<Number> sList = new ArrayList<Number>();
sList.add(0);
Collections.addAll(sList, (byte)1, (short)2 );
```

Обобщенная специализация

Обобщенный тип можно расширить множеством способов.

Предположим, задан параметризованный абстрактный класс AbstractSet <E>.

`class SpecialSet<E> extends AbstractSet<E> {...}`

Класс `SpecialSet` расширяет класс `AbstractSet` с помощью параметра типа `E`. Это типичный способ объявления обобщенных классов с помощью обобщенных типов.

`class SpecialSet extends AbstractSet<String> {...}`

Класс `SpecialSet` расширяет класс `AbstractSet` с помощью параметризованного типа `String`.

`class SpecialSet<E,P> extends AbstractSet<E> {...}`

Класс `SpecialSet` расширяет класс `AbstractSet` с помощью параметра типа `E`. Тип `P` уникален для класса `SpecialSet`.

`class SpecialSet<E> extends AbstractSet {...}`

Класс `SpecialSet` является обобщенным классом, который параметризует обобщенный тип класса `AbstractSet`. Поскольку первичный (raw) тип класса `AbstractSet` был расширен, а не обобщен, параметризация не может произойти. При вызове соответствующего метода компилятор выдаст предупреждения.

`class SpecialSet extends AbstractSet {...}`

Класс `SpecialSet` расширяет первичный тип класса `AbstractSet`. Поскольку ожидалась обобщенная версия класса `AbstractSet`, в результате попытки вызвать соответствующий метод компилятор выдаст предупреждения.

Обобщенные методы в первичных типах

Статические и нестатические методы, а также конструкторы, которые являются частью необобщенных или первичных (raw) типов классов, можно объявить как обобщенные. Первичный тип класса является необобщенным классом-двойником обобщенного класса.

В случае обобщенных методов необобщенных классов типу, возвращаемому соответствующим методом, должен предшествовать параметр обобщенного типа (например, <E>). Однако между

типом параметра и возвращаемым типом нет функциональной зависимости, если только этот возвращаемый тип не является обобщенным типом:

```
public class SpecialQueue {
   public static <E> boolean add(E e) {...}
   public static <E> E peek() {...}
}
```

При вызове обобщенного метода параметр обобщенного типа помещается перед именем метода. В данном случае `<String>` используется для указания аргумента обобщенного типа:

```
SpecialQueue.<String>add("Белая гвоздика");
```

Языки сценариев Java

API языков сценариев Java (Java Scripting API) появился в Java SE 6 и предназначен для взаимодействия посредством стандартного интерфейса между Java-приложениями и приложениями, написанными на интерпретируемых языках (сценариях).

Этот API подробно описан в документе JSR 223, "Scripting for the Java Platform" ("Написание сценариев для платформы Java") и содержится в пакете `javax.script`.

Языки сценариев

Существует ряд сценарных языков, для которых реализованы движки (интерпретаторы), удовлетворяющие требованиям JSR 223. Их подмножество описано в разделе "Интерпретируемые языки (совместимые с JSR-223)" приложения Б.

Реализации интерпретаторов сценарных языков

Интерфейс `ScriptEngine` обеспечивает фундаментальные методы для поддержки API. Класс `ScriptEngineManager` работает совместно с этим интерфейсом и позволяет задавать желаемые интерпретаторы сценарных языков, которые предстоит использовать.

Встраивание сценариев в Java-программы

В API языков подготовки сценариев предусмотрена возможность встраивания сценариев и/или сценарных компонентов в приложения Java.

В приведенном ниже примере проиллюстрированы два способа встраивания сценарных компонентов в Java-приложение.

1. Непосредственная передача кода сценария методу `eval()` интерпретатора сценарных языков.
2. Считывание кода сценария методом `eval()` из указанного файла.

```
import java.io.FileReader;
import java.nio.file.Path;
import java.nio.file.Paths;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class HelloWorld {
   public static void main(String[] args) throws
         Exception {
      ScriptEngineManager m
         = new ScriptEngineManager();

      // Создает машину Nashhorn JavaScript.
      ScriptEngine e = m.getEngineByExtension("js");

      // Синтаксис Nashhorn JavaScript.
      e.eval("print ('Hello, ')");

      // Содержимое world.js: print('World!\n');
      Path p1 = Paths.get("/opt/jpg2/world.js");
      e.eval(new FileReader(p1.toString()));
   }
}

$ Hello, World!
```

Вызов методов сценарных языков

Интерпретаторы сценарных языков, в которых реализованы методы дополнительного интерфейса `Invocable`, позволяют вызывать методы сценарных языков (т.е. запускать их на выполнение), которые уже обработаны интерпретатором.

В представленном ниже фрагменте кода используется Java-метод `invokeFunction()`, вызывающий созданную нами

функцию сценарного языка `greet()`, после интерпретации ее движком Nashhorn.

```
ScriptEngineManager m = new ScriptEngineManager();
ScriptEngine e = m.getEngineByExtension("js");

e.eval("function greet(message)
   + "{" + "println(message)" + "}");

Invocable i = (Invocable) e;
i.invokeFunction("greet", "Привет с Марса!");

$ Привет с Марса!
```

Доступ к ресурсам Java из сценариев

Java Scripting API обеспечивает возможность доступа к ресурсам (объектам) Java и управления этими ресурсами из сценарного кода, прошедшего интерпретацию. Одним из способов обеспечить такой доступ и управление является использование привязок на основе ключей и значений в интерпретаторах сценарных языков.

В приведенном ниже примере JavaScript-кода, прошедшего интерпретацию движком Nashhorn, ключу с именем `"nameKey"` ставится в соответствие значение переменной `world`. Это позволяет получить доступ из программы на JavaScript к переменной `world`, объявленной в Java-программе, и вывести ее значение на экран.

```
ScriptEngineManager m = new ScriptEngineManager();
ScriptEngine e = m.getEngineByExtension("js");

String world = "Gliese 581 c";
e.put("nameKey", world);

e.eval("var w = nameKey" );
e.eval("println(w)");

$ Gliese 581 c
```

Используя подобные привязки ключей и значений, вы можете изменить значения переменных Java из интерпретируемого кода.

```
ScriptEngineManager m = new ScriptEngineManager();
ScriptEngine e = m.getEngineByExtension("js");
```

```
List<String> worldList = new ArrayList<>();
worldList.add ("Earth");
worldList.add ("Mars");

e.put("nameKey", worldList);
e.eval("var w = nameKey.toArray();");
e.eval(" nameKey.add (\"Gliese 581 c\")");
System.out.println(worldList);
$ [Earth, Gliese 581 c]
```

Установка сценарных языков и интерпретаторов

Прежде чем использовать API языков сценариев Java, вы должны загрузить и развернуть желаемые реализации интерпретаторов. В дистрибутивную поставку многих сценарных языков входит интерпретатор, совместимый со стандартом JSR-223. Он может находиться либо в отдельном JAR-файле, либо в основном JAR-архиве, как в случае с JRuby.

Установка сценарного языка

Ниже описаны этапы установки сценарного языка в Java.

1. Установить поддержку сценарного языка в своей операционной системе. В разделе "Интерпретируемые языки (совместимые с JSR-223)" приложения Б приведен список сайтов, с которых можно загрузить интерпретаторы некоторых из совместимых сценарных языков. Выполните прилагаемые к ним инструкции по установке.
2. Вызовите один из сценарных интерпретаторов, чтобы убедиться, что все функционируют правильно. Обычно имеется интерпретатор с интерфейсом командной строки, а также интерпретатор с графическим интерфейсом пользователя.

В случае JRuby (воспользуемся им в качестве примера), чтобы убедиться в правильности установки, нужно проверить наличие следующих команд:

```
jruby [file.rb] //Файл командной строки
jruby.bat //пакетный файл Windows
```

Установка интерпретатора сценарного языка

Ниже описаны этапы установки интерпретатора сценарного языка.

1. Определите, включен ли в дистрибутив вашего сценарного языка интерпретатор, поддерживающий API JSR-223. Если это так, то этапы 2 и 3 выполнять не нужно.
2. Найдите на каком-либо из внешних ресурсов (например, на веб-сайте) файл интерпретатора и загрузите его.
3. Поместите загруженный вами файл в какой-либо каталог и распакуйте его, чтобы получить необходимый JAR-файл. Обратите внимание, что в качестве дистрибутивного обычно используется каталог для дополнительного программного обеспечения `/opt`.

НА ЗАМЕТКУ

Чтобы установить и настроить некоторые сценарные языки на Windows-машине, может понадобиться минимальная POSIX-совместимая оболочка, например MSYS или Cygwin.

Проверка правильности установки интерпретатора

Проверить правильность установки интерпретатора можно, выполнив компиляцию и/или интерпретацию библиотек сценарного языка и библиотек самого интерпретатора. Ниже представлен пример того, как это можно было сделать в старых версиях JRuby, в которых использовался внешний интерпретатор.

```
javac -cp c:\opt\jruby-1.0\lib\jruby.jar;c:\opt\
jruby-engine.jar;. Engines
```

Дополнительное тестирование можно также выполнить с помощью коротких программ. Приведенная ниже программа выводит список имен имеющихся в наличии интерпретаторов сценарных языков, номера версий языка и расширения. Обратите внимание: в данном случае речь идет об обновленной версии JRuby, в которую включена поддержка JSR-223 в основном JAR-файле. Поэтому JAR-файл интерпретатора языка не требуется отдельно включать в путь поиска классов.

```
$ java -cp c:\opt\jruby-1.6.7.2\lib\jruby.jar;.
   EngineReport

import java.util.List;
import javax.script.ScriptEngineManager;
import javax.script.ScriptEngineFactory;

public class EngineReport {
   public static void main(String[] args) {
      ScriptEngineManager m =
         new ScriptEngineManager();
      List<ScriptEngineFactory> s =
         m.getEngineFactories();

      // Просмотр перечня фабрик
      for (ScriptEngineFactory f: s) {
         // Имя и версия релиза
         String en = f.getEngineName();
         String ev = f.getEngineVersion();
         System.out.println("Машина: "
            + en + " " + ev);
         // Имя и версия языка
         String ln = f.getLanguageName();
         String lv = f.getLanguageVersion();
         System.out.println("Язык: "
            + ln + " " + lv);
         // Расширения
         List<String> l = f.getExtensions();
         for (String x: l) {
            System.out.println("Расширения: " + x);
```

```
            }
        }
    }
}
$ Машина: Oracle Nashorn 1.8.0
$ Язык: ECMAScript ECMA - 262 Edition 5.1
$ Расширения: js

$ Машина: JSR 223 JRuby Engine 1.6.7.2
$ Язык: ruby jruby 1.6.7.2
$ Расширения: rb
```

НА ЗАМЕТКУ

Nashorn JavaScript — это API сценариев, содержащийся в одном пакете с Java SE 8; он доступен по умолчанию. Он заменил собой API сценариев Rhino JavaScript, который использовался в предыдущей версии JDK.

Дата и время

Прикладной программный интерфейс для работы с датой и временем (Date and Time API, JSR 310) обеспечивает набор функций, с помощью которых можно выполнять разнообразные операции с датой и временем, а также осуществлять вычисления на основе календаря. Эталонная реализация для этого JSR была выполнена в проекте "три-десять" (*ThreeTen Project*) и была предоставлена для включения в JDK 1.8. Программный интерфейс для работы с датой и временем размещается в пакете `java.time` и вложенных пакетах `java.time.chrono`, `java.time.format`, `java.time.temporal` и `java.time.zone`.

При разработке JSR 310 были поставлены следующие цели.

- Текучий интерфейс API (Fluent API), нацеленный на повышение читабельности исходного кода программы путем использования цепочки методов.
- Концепция потоковой безопасности (Thread-safe design), ориентированная на неизменяемость объектов классов.
- Расширяемый API, позволяющий работать с новыми системами календарей, корректировщиками и запросами.
- Предсказуемое поведение.

В API для работы с датой и временем используется модель обмена данными ISO 8601. Стандарт ISO 8601 формально называется "Data elements and interchange formats — Information interchange — Representation of dates and times" ("Элементы даты и форматы обмена — Информационный обмен — Представление даты и времени"). В его основу положен григорианский календарь. Также поддерживаются и региональные календари.

Подробнее текучий интерфейс API описан в приложении А.

Устаревшая функциональная совместимость

Проект JSR 310 заменил собой, но не отменил использование следующих классов: `java.util.Date`, `java.util.Calendar`, `java.util.DateFormat`, `java.util.GregorianCalendar`, `java.util.TimeZone` и `java.sql.Date`. С целью совместимости и поддержки устаревших приложений в JDK 8 предусмотрены методы для этих классов, позволяющие выполнить преобразование типов в и из формата JSR 310.

```
// Устаревший -> Новый -> Устаревший
Calendar c = Calendar.getInstance();
Instant i = c.toInstant();
Date d = Date.from(i);

// Новый -> Устаревший -> Новый =
ZonedDateTime zdt
    ZonedDateTime.parse("2014-02-24T11:17:00+01:00" +
                        "[Europe/Gibraltar]")
GregorianCalendar gc = GregorianCalendar.from(zdt);
LocalDateTime ldt =
    gc.toZonedDateTime().toLocalDateTime();
```

Региональные календари

В проекте JSR 310 была предусмотрена возможность подключения новых календарей. При создании нового календаря в классах должны быть реализованы интерфейсы `Era`, `Chronology` и `ChronoLocalDate`.

В пакеты поддержки API даты и времени включены четыре региональных календаря:

- хиджра;
- японский императорский;
- миньго;
- тайский буддийский.

Вместе с региональными календарями вам не нужно использовать основные классы календаря ISO.

Календарь ISO

В основном пакете API `java.time` обеспечивается поддержка календарной системы ISO 8601, которая соответствует правилам григорианского календаря. В этом и связанном с ним пакетах API поддерживается простой интерфейс с пользователем, который мы продемонстрируем ниже на примере вычисления разницы в возрасте двух президентов.

```
public final static String DISNEY_BIRTH_YEAR = "1901";
public final static String TEMPLE_BIRTH_YEAR = "1928";
...
Year birthYear1 = Year.parse(DISNEY_BIRTH_YEAR);
Year birthYear2 = Year.parse(TEMPLE_BIRTH_YEAR);
long diff = ChronoUnit.YEARS.
between(birthYear1,birthYear2);

System.out.println("Разница в возрасте - "
+ Math.abs(diff) + " лет." );
$ Разница в возрасте - 27 лет.
```

Основные классы API

В этом разделе перечислены основные классы API для работы с датой и временем, а также их краткое описание, взятое из электронной документации. В последующих разделах мы опишем их основные свойства, а также приведем примеры использования некоторых из этих классов.

`Instant`

: Представляет текущую точку на временной шкале. Отсчет выполняется относительно начала эпохи Java — `1970-01-01T00:00:00Z`.

`LocalDate`

: Неизменяемый объект даты и времени, представляющий дату в виде года, месяца и дня.

`LocalTime`

: Неизменяемый объект даты и времени, представляющий время в виде часов, минут и секунд.

`LocalDateTime`

Неизменяемый объект даты и времени, представляющий дату и время в виде года, месяца, дня, а также часов, минут и секунд.

`OffsetTime`

Неизменяемый объект даты и времени, представляющий время в виде часов, минут, секунд и временного сдвига.

`OffsetDateTime`

Неизменяемый объект даты и времени, учитывающий сдвиг. Позволяет хранить все компоненты даты и времени с точностью до нескольких наносекунд, а также временной сдвиг относительно Гринвичского меридиана (зоны UTC/Greenwich).

`ZonedDateTime`

Неизменяемый объект даты и времени, учитывающий временную зону. Позволяет хранить все компоненты даты и времени с точностью до нескольких наносекунд, а также информацию о временной зоне и соответствующий ей временной сдвиг, использующийся для устранения неоднозначности представления локальной даты и времени.

`ZoneOffset`

Объект, хранящий сдвиг временной зоны — разницу во времени между текущей зоной и Гринвичским меридианом (зоной UTC/Greenwich).

`ZonedId`

Объект, хранящий идентификатор зоны. Используется для определения правил преобразования значений, полученных из объектов `Instant` и `LocalDateTime`.

`Year`

Неизменяемый объект даты и времени, представляющий год.

`YearMonth`

Неизменяемый объект даты и времени, представляющий комбинацию года и месяца.

`MonthDay`

Неизменяемый объект даты и времени, представляющий комбинацию месяца и дня.

`DayOfWeek`

Объект, хранящий список названий дней недели — понедельник, вторник, среда и т.д.

`Month`

Объект, хранящий список названий месяцев — январь, февраль, март и т.д.

`Duration`

Объект, хранящий длительность интервала времени в секундах.

`Period`

Объект, хранящий длительность периода времени между двумя датами.

`Clock`

Объект, позволяющий определить текущий момент времени, дату и время с учетом временной зоны. Без него вполне можно обойтись.

Машинный интерфейс

В JSR 310 для представления стандартного календаря ISO 8301 используется точка отсчета времени, принятая в системе Unix — `1970-01-01T00:00:00Z`. По определению время является непрерывной величиной. Отрицательные значения представляют дату и время, наступившие до начала эпохи Unix.

Чтобы определить текущий момент времени, нужно вызвать метод `now()` объекта `Instant`, как показано ниже.

```
Instant i = Instant.now();

System.out.println("Машинное представление: " +
i.toEpochMilli());
$ Машинное представление: 1392859358793
```

```
System.out.println("Человеческое представление: " + i);
$ Человеческое представление: 2014-02-20T01:20:41.402Z
```

Объект `Clock` позволяет определить текущий момент времени, дату и время с учетом временной зоны.

```
Clock clock1 = Clock.systemUTC();
Instant i1 = Instant.now(clock1);

ZoneId zid = ZoneId.of("Europe/Vienna");
Clock clock2 = Clock.system(zid);
Instant i2 = Instant.now(clock2);
```

В API даты и времени используется база данных временных зон TZDB.

Длительности и периоды

Объект `Duration` позволяет задать длительность интервала времени между двумя датами в днях, часах, минутах, секундах и наносекундах.

Длительность задается в виде синтаксически анализируемой строки `PnDTnHnMnS`, где `P` задает период, `T` — время. Буквы `D`, `H`, `M` и `S` соответствуют дням, часам, минутам и секундам и указываются после цифр *n*.

```
Duration d1 = Duration.parse("P2DT3H4M1.1S");
```

Длительность можно также задать с помощью метода `of`*Тип*. После этого к длительности можно прибавлять или вычитать часы, минуты, секунды и наносекунды, как показано ниже.

```
Duration d2 = Duration.of(41, ChronoUnit.YEARS);

Duration d3 = Duration.ofDays(8);
d3 = d3.plusHours(3);
d3 = d3.plusMinutes(30);
d3 = d3.plusSeconds(55).minusNanos(300);
```

С помощью метода `between()` класса `Duration` можно создать новый объект типа `Duration` на основе начального и конечного значений даты и времени:

```
Instant birth   = Instant.parse("1967-09-15T10:30:00Z");
Instant current = Instant.now();
```

```
Duration d4 = Duration.between(birth, current);
System.out.print("Прожито дней: " + d4.toDays());
```

Объект `Period` позволяет задать период между двумя датами, выраженный в годах, месяцах и днях.

Длительность периода задается в виде синтаксически анализируемой строки `PnYnMnD`, где `P` задает период. Буквы `Y`, `M` и `D` соответствуют годам, месяцам и дням и указываются после цифр *n*.

```
Period p1 = Period.parse("P10Y5M2D");
```

Периоды можно также задать с помощью метода `of`*Тип*. После этого к периоду можно прибавлять или вычитать годы, месяцы и дни, как показано ниже.

```
Period p2 = Period.of(5, 10, 40);
p2 = p2.plusYears(100);
p2 = p2.plusMonths(5).minusDays(30);
```

Соответствие типов JDBC и XSD

В новой версии Java достигнуто соответствие между типами `java.time` и `java.sql`. В табл. 18.1 приведено соответствие между типами JSR 310 и SQL, а также схемой XML (XSD).

Таблица 18.1. Соответствие типов JSR 310, JDBC и XSD

JSR 310	SQL	XSD
`LocalDate`	`DATE`	`xs:time`
`LocalTime`	`TIME`	`xs:time`
`LocalDateTime`	`TIMESTAMP WITHOUT TIMEZONE`	`xs:dateTime`
`OffsetTime`	`TIME WITH TIMEZONE`	`xs:time`
`OffsetDateTime`	`TIMESTAMP WITH TIMEZONE`	`xs:dateTime`
`Period`	`INTERVAL`	

Форматирование

Класс `DateTimeFormatter` обеспечивает средства форматирования временных данных при выводе их на печать, а также синтаксического анализа при создании объектов даты и времени. В приведенном ниже примере показано использование строки

образца в методе ofPattern() этого класса. Подробное описание букв в строке образца приведено в документации Javadoc для класса DateTimeFormatter.

```
LocalDateTime input = LocalDateTime.now();
DateTimeFormatter format =
            DateTimeFormatter.
ofPattern("yyyyMMddhhmmss");
String date = input.format(format);
String logFile = "simple-log-" + date + ".txt";
```

В табл. 18.2 приведены примеры предопределенных форматов, которые можно использовать в приведенном ниже фрагменте кода.

```
System.out.print(LocalDateTime.now()
   .format(DateTimeFormatter.BASIC_ISO_DATE));
```

Таблица 18.2. Предопределенные форматы

Класс	Формат	Пример
LocalDateTime	BASIC_ISO_DATE	20140215
LocalDateTime	ISO_LOCAL_DATE	2014-02-15
OffsetDateTime	ISO_OFFSET_DATE	2014-02-15-05:00
LocalDateTime	ISO_DATE	2014-02-15
OffsetDateTime	ISO_DATE	2014-02-15-05:00
LocalDateTime	ISO_LOCAL_TIME	23:39:07.884
OffsetTime	ISO_OFFSET_TIME	23:39:07.888-05:00
LocalDateTime	ISO_TIME	23:39:07.888
OffsetDateTime	ISO_TIME	23:39:07.888-05:00
LocalDateTime	ISO_LOCAL_DATE_TIME	2014-02-15T23:39:07.888
OffsetDateTime	ISO_OFFSET_DATE_TIME	2014-02-15T23:39:07.888-05:00
ZonedDateTime	ISO_ZONED_DATE_TIME	2014-02-15T23:39:07.89-05:00 [America/New_York]
LocalDateTime	ISO_DATE_TIME	2014-02-15T23:39:07.891
ZonedDateTime	ISO_DATE_TIME	2014-02-15T23:39:07.891-05:00 [America/New_York]
LocalDateTime	ISO_ORDINAL_DATE	2014-046
LocalDate	ISO_WEEK_DATE	2014-W07-6
ZonedDateTime	RFC_1123_DATE_TIME	Sat, 15 Feb 2014 23:39:07 -0500

Лямбда-выражения

Лямбда-выражения, которые еще называют замыканиями (closures), используются для представления анонимных методов. Лямбда-выражения поддерживаются специальным проектом *Project Lambda* и позволяют создавать и использовать классы с одним методом. Базовый синтаксис этих методов позволяет не указывать модификаторы, тип возвращаемого параметра и необязательные аргументы. Спецификация лямбда-выражений описана в документе JSR 335, который состоит из семи частей: функциональные интерфейсы, лямбда-выражения, ссылки на методы и конструкторы, поливыражения, типизация и интерпретация, выведение типов и стандартные методы. В этой главе будут описаны первые две части.

Основы лямбда-выражений

Лямбда-выражения должны иметь функциональный интерфейс. Функциональным называется такой интерфейс, который имеет только один абстрактный метод и несколько стандартных методов (default methods) либо вовсе не имеет стандартных методов. Благодаря функциональному интерфейсу в лямбда-выражении можно определить целевой тип и ссылки на методы. В идеале для функционального интерфейса следует использовать аннотацию `@FunctionalInterface`, чтобы помочь разработчикам и компилятору прояснить цели проекта.

```
@FunctionalInterface
public interface Comparator<T> {
   // Разрешен только один абстрактный метод
   int compare(T o1, T o2);
   // Переопределение разрешено
   boolean equals(Object obj);
   // Разрешены опциональные стандартные методы
}
```

Синтаксис и примеры

Как правило, лямбда-выражение состоит из списка параметров, возвращаемого типа и тела.

```
(список параметров) -> { операторы; }
```

Ниже приведено несколько примеров лямбда-выражений.

```
() -> 66
(x,y) -> x + y
(Integer x, Integer y) -> x*y
(String s) -> { System.out.println(s); }
```

Приведенное ниже простое графическое приложение, в котором используются классы JavaFX, помещает текст в строке заголовка после щелчка на кнопке. В коде используется функциональный интерфейс `EventHandler`, содержащий один абстрактный метод `handle()`.

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class JavaFxApp extends Application {

   @Override
   public void start(Stage stage) {
      Button b = new Button();
      b.setText("Нажмите кнопку");

      // Используется анонимный внутренний класс
      b.setOnAction(new EventHandler<ActionEvent>() {

         @Override
         public void handle(ActionEvent event) {
            stage.setTitle("Работает лямбда-выражение!");
         }
      });
```

```
        StackPane root = new StackPane();
        root.getChildren().add(b);
        Scene scene = new Scene(root, 200, 50);
        stage.setScene(scene);
        stage.show();
    }

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

Изменим приведенный выше код так, чтобы вместо анонимного внутреннего класса в нем использовалось лямбда-выражение. В качестве типа параметра нужно указать либо `(ActionEvent event)`, либо просто `(event)`, а всю нужную функциональность поместим в тело лямбда-выражения, как показано ниже.

```
// Используется лямбда-выражение
b.setOnAction((ActionEvent event) -> {
stage.setTitle("Работает лямбда-выражение!");
});
```

НА ЗАМЕТКУ

В большинстве современных интегрированных сред разработки (IDE) есть средства для автоматического преобразования анонимных внутренних классов в лямбда-выражения.

В разделе "Функциональный интерфейс Comparator" главы 15 мы уже рассматривали пример использования лямбда-выражений совместно с функциональным интерфейсом `Comparator`.

Ссылки на методы и конструкторы

Понятие *ссылки на метод* означает наличие указателя на существующий метод, но без его вызова. Существуют различные типы ссылок на методы, включая ссылку на статический метод, метод экземпляра конкретного объекта, метод родительского объекта

и метод экземпляра произвольного объекта определенного типа. К ссылкам на методы относятся также ссылки на конструкторы класса и ссылки на конструкторы массивов.

```
"текст"::length // Определить длину строки
String::length // Определить длину строки
CheckAcct::compareByBalance // Ссылка на статический метод
myComparator::compareByName // Метод экземпляра
конкретного объекта
super::toString // Метод родительского объекта
String::compareToIgnoreCase // Метод экземпляра
произвольного объекта определенного типа
ArrayList<String>::new // Конструктор New ArrayList
Arrays::sort // Сортировка массива элементов
```

Функциональные интерфейсы специального назначения

В табл. 19.1 перечислены аннотированные функциональные интерфейсы, которые были введены для решения специфических задач в тех пакетах/API, к которым они относятся. Обратите внимание, что не все функциональные интерфейсы в Java SE API являются аннотированными.

Таблица 19.1. Специфические функциональные интерфейсы

API	Класс	Метод
AWT	KeyEventDispacter	dispatchKeyEvent (KeyEvent e)
AWT	KeyEventPostProcessor	postProcessKeyEvent (KeyEvent e)
IO	FileFilter	accept(File pathname)
IO	FilenameFilter	accept(File dir, String name)
LANG	Runnable	run()
NIO	DirectorStream	iterator()
NIO	PathMatcher	matches(Path path)
TIME	TemporalAdjuster	adjustInto(Temporal temporal)

Окончание табл. 19.1

API	Класс	Метод
TIME	TemporalQuery	queryFrom(TemporalAccessor temporal)
UTIL	Comparator	compare(T o1, T o2)
CONC	Callable	call()
LOG	Filter	isLoggable(LogRecord record)
PREF	PreferenceChangeListener	preferenceChange(PreferenceChangeEvent evt)

Функциональные интерфейсы общего назначения

В пакет `java.util.function` включены функциональные интерфейсы общего назначения, которые предназначены в основном для использования средств JDK (табл. 19.2).

Таблица 19.2. Универсальные функциональные интерфейсы

Клиент	accept(T t)
BiConsumer	accept(T t, U u)
ObjDoubleConsumer	accept(T t, double value)
ObjIntConsumer	accept(T t, int value)
ObjLongConsumer	accept(T t, long value)
DoubleConsumer	accept(double value)
IntConsumer	accept(int value)
LongConsumer	accept(long value)
Function	apply(T t)
BiFunction	apply(T t, U u)
DoubleFunction	apply(double value)
IntFunction	apply(int value)
LongFunction	apply(long value)
BinaryOperator	apply(Object, Object)
ToDoubleBiFunction	applyAsDouble(T t, U u)
ToDoubleFunction	applyAsDouble(T value)
IntToDoubleFunction	applyAsDouble(int value)
LongToDoubleFunction	applyAsDouble(long value)

Окончание табл. 19.2

Клиент	accept(T t)
DoubleBinaryOperator	applyAsDouble(double left, double right)
ToIntBiFunction	applyAsInt(T t, U u)
ToIntFunction	applyAsInt(T value)
LongToIntFunction	applyAsInt(long value)
DoubleToIntFunction	applyAsInt(double value)
IntBinaryOperator	applyAsInt(int left, int right)
ToLongBiFunction	applyAsLong(T t, U u)
ToLongFunction	applyAsLong(T value)
DoubleToLongFunction	applyAsLong(double value)
IntToLongFunction	applyAsLong(int value)
LongBinaryOperator	applyAsLong(long left, long right)
BiPredicate	test(T t, U u)
Predicate	test(T t)
DoublePredicate	test(double value)
IntPredicate	test(int value)
LongPredicate	test(long value)
Supplier	get()
BooleanSupplier	getAsBoolean()
DoubleSupplier	getAsDouble()
IntSupplier	getAsInt()
LongSupplier	getAsLong()
UnaryOperator	identity()
DoubleUnaryOperator	identity()
IntUnaryOperator	applyAsInt(int operand)
LongUnaryOperator	applyAsInt(long value)

Дополнительная информация по лямбда-выражениям

В этом разделе приводятся ссылки на руководства и общедоступные ресурсы в Интернете, посвященные лямбда-выражениям.

Руководства

Полноценные учебные пособия по лямбда-выражениям разработаны Oracle и Maurice Naftalin.

- The Java Tutorials: Lambda Expressions
- Maurice Naftalin's Lambda FAQ: "Your questions answered: all about Lambdas and friends"

Общедоступные ресурсы

Для изучения лямбда-выражений можно воспользоваться многочисленными форумами, списками рассылки и видеоуроками, представленными в Интернете. Некоторые из них перечислены ниже.

- Форум, посвященный лямбда-выражениям, на CodeRanch
- Список рассылки: *Technical discussions related to Project Lambda*
- Учебный канал Oracle на YouTube.

ЧАСТЬ III

Приложения

Текучие интерфейсы API

Текучие API, или просто текучие интерфейсы, являются объектно-ориентированными программными интерфейсами приложений, разработанными с целью повышения читабельности исходного года и, как следствие, облегчения их использования. Для этой цели используется "сшивание" объектов посредством сцепления методов. При таком подходе цепочки методов, как правило, возвращают значения одного того же типа.

```
// StringBuilder API
StringBuilder sb = new StringBuilder("Палиндром!");

// Цепочка методов
sb.delete(10, 10).append("ы").reverse();
System.out.println("Значение: " + sb);
$ Значение: ыморднилаП
```

Перечислим несколько популярных текучих API, написанных на Java: API объектно-ориентированных запросов на Java (Java Object Oriented Querying, jOOQ), API тестирования jMock, API тестирования Calculon Android, API шаблонов интеграции Apache Camel, API даты и времени Java 8 (JSR 310) и API работы с денежными единицами (JSR 354), которое планируется включить в Java 9. Можно считать, что каждое из этих API является предметно-ориентированным языком Java (domain specific language, DSL).

Внешние предметно-ориентированные языки могут быть легко отображены на новый внутренний предметно-ориентированный язык Java путем использования текучего API.

В текучем API для работы с объектами используются следующие широко распространенные префиксы: `at`, `format`, `from`, `get`, `to` и `with`.

Ниже представлены два примера использования класса `LocalDateTime` из API даты и времени без цепочки методов (первый пример) и с цепочкой методов (второй пример).

```
// Автономный статический method
LocalDateTime ldt1 = LocalDateTime.now();
System.out.println(ldt1);

$ 2014-02-26T09:33:25.676

// Статический метод с цепочкой
LocalDateTime ldt2 = LocalDateTime.now()
   .withDayOfMonth(1).withYear(1878)
   .plusWeeks(2).minus(3, ChronoUnit.HOURS);
System.out.println(ldt2);

$ 1878-02-15T06:33:25.724
```

НА ЗАМЕТКУ

Подробнее предметно-ориентированные языки описаны в книге Мартина Фаулера *Предметно-ориентированные языки программирования* (пер. с англ., ИД "Вильямс", 2011, ISBN 978-5-8459-1738-6).

Средства сторонних разработчиков

В настоящее время в наличии имеется множество средств с открытым исходным кодом, а также коммерческих утилит и технологий сторонних фирм, призванных помочь в разработке приложений на Java.

Ресурсы, перечисленные в настоящем приложении, являются и эффективными, и популярными. Если вы собираетесь пользоваться какими-либо средствами с открытым исходным кодом, обязательно ознакомьтесь с соответствующими лицензионными соглашениями, в которых, как правило, указываются ограничения на использование соответствующих средств в коммерческих организациях.

Средства разработки, конфигурирования и тестирования

Ant

Утилита от Apache, поддерживающая формат XML, предназначенная для разработки и развертывания приложений на языке Java. Она напоминает хорошо известную утилиту Unix `make`.

Bloodhound

Система с открытым исходным кодом от Apache, предназначенная для управления веб-проектами и отслеживания ошибок.

Continuum

Сервер непрерывной интеграции от Apache, который выполняет построение проекта и тестирование кода на частой, регулярной основе.

CruiseControl

Инфраструктура для процесса непрерывного построения приложения. Она включает веб-интерфейс для просмотра деталей процесса построения, а также дополнительные модули для Ant, систему управления версиями и рассылку уведомлений по электронной почте.

Enterprise Architect

Коммерческий продукт системы автоматизированного проектирования и создания программ (Computer-Aided Software Engineering — CASE), который обеспечивает прямую и обратную разработку кода Java с помощью UML.

FindBugs

Программа, которая отыскивает ошибки в коде Java.

Git

Распределенная система управления версиями с открытым исходным кодом.

Gradle

Система построения приложений, позволяющая выполнить их тестирование, публикацию и развертывание.

Heatlamp

Строит понятные, информационно-насыщенные и интерактивные диаграммы прямо из кода Java.

Hudson

Масштабируемый сервер непрерывной интеграции.

Ivy

Диспетчер, отслеживающий транзитивные связи и зависимости в исходном коде. Он интегрирован в Apache Ant.

Jalopy

Форматер исходного кода на Java, в котором имеются дополнительные модули для Eclipse, jEdit, NetBeans и других интегрированных средств разработки.

jClarity

Средство анализа производительности и мониторинга облачных систем.

JDocs

Хранилище документов, которое обеспечивает веб-доступ к документации API Java библиотек с открытым исходным кодом.

jEdit

Текстовый редактор, предназначенный для программистов. В нем имеется несколько дополнительных модулей, к которым можно обращаться посредством специального диспетчера.

JavaFX SceneBuilder

Средство визуальной компоновки, предназначенное для разработки приложений с графическим интерфейсом JavaFX.

Jenkins

Сервер непрерывной интеграции с открытым исходным кодом, формально называемый "Hudson Labs".

JIRA

Коммерческое приложение, предназначенное для отслеживания программных ошибок, генерирования соответствующих запросов на исправление и управления проектами.

JMeter

Утилита от Apache, оценивающая скорость работы приложения и его функциональные возможности, а также общую производительность.

JUnit

Инфраструктура для блочного тестирования, которая служит для написания и выполнения регулярно повторяющихся тестов.

Maven

Средство от Apache управления программными проектами на Java уровня предприятия. Maven может управлять сборками, отчетами и документацией.

Nemo

Онлайн-версия утилиты Sonar, предназначенная для проектов с открытым исходным кодом.

PMD

Сканирует исходный код Java с целью выявления ошибок, недостаточно эффективного кода и слишком сложных выражений.

SonarQube

Платформа управления качеством с открытым исходным кодом.

Subversion

Централизованная система управления версиями от Apache, которая отслеживает работу и изменения в определенном наборе файлов.

Библиотеки

ActiveMQ

Брокер сообщений от Apache, который поддерживает много разноязыковых клиентов и протоколов.

BIRT

Система генерирования отчетов с открытым исходным кодом на основе Eclipse, которая используется в приложениях Java EE.

Camel

Движок от Apache для маршрутизации на основе правил и посредничества.

Hibernate

Служба объектно-реляционной сохраняемости и создания запросов. Позволяет разрабатывать классы, поддерживающие сохраняемость значений своих членов.

iText

Библиотека Java, которая позволяет создавать PDF-документы и работать с ними.

Jackrabbit

Система управления информационным хранилищем от Apache, которая обеспечивает хранение и управление контентом с иерархической структурой.

Jakarta Commons

Хранилище повторно используемых компонентов Java.

JasperReports

Движок для генерирования отчетов Java с открытым исходным кодом.

Jasypt

Библиотека Java, которая позволяет разработчику добавить в свое приложение базовые возможности шифрования.

JFreeChart

Библиотека классов Java для генерирования схем и диаграмм.

JFXtras2

Набор элементов управления и дополнений для JavaFX 2.0.

JGoodies

Предоставляет компоненты и решения, которые используются для построения стандартного пользовательского интерфейса.

JIDE

Библиотека, содержащая разнообразные компоненты Java и Swing.

JMonkeyEngine

Коллекция библиотек, на основе которых функционирует игровой движок Java 3D (OpenGL).

JOGL

API Java, поддерживающий спецификации ES и OpenGL.

jOOQ

Текучий API для безопасного в плане типов построения SQL-запросов и их выполнения.

opencsv

Библиотека синтаксического анализатора текстовых файлов, разделенных запятыми (CSV) для языка Java.

POI

Библиотека от Apache Poor Obfuscation Implementation (POI), предназначена для чтения и записи документов в формате Microsoft Office.

RXTX

Обеспечивает машинно-зависимую последовательную и параллельную передачу данных для Java.

Spring Framework

Многоуровневая инфраструктура приложений Java/Java EE.

Интегрированные среды разработки

BlueJ

IDE, предназначенная для начального обучения.

Eclipse IDE

IDE с открытым исходным кодом, предназначенная для создания Java-аплетов и приложений для рабочего стола, мобильных устройств и веб.

Greenfoot

Простая IDE, созданная для изучения ООП на Java.

IntelliJ IDEA

Коммерческая IDE, предназначенная для создания Java-аплетов и приложений для рабочего стола, мобильных устройств и веб.

JBuilder

Коммерческая IDE, предназначенная для создания Java-аплетов и приложений для рабочего стола, мобильных устройств и веб.

JCreator

Коммерческая IDE, предназначенная для создания Java-аплетов и приложений для рабочего стола, мобильных устройств и веб.

JDeveloper

IDE компании Oracle, предназначенная для создания Java-аплетов и приложений для рабочего стола, мобильных устройств и веб.

NetBeans

IDE компании Oracle с открытым исходным кодом, предназначенная для создания Java-аплетов и приложений для рабочего стола, мобильных устройств и веб.

Платформы веб-приложений

Geronimo

Сервер Java EE от Apache, используемый для разработки, интеграции и развертывания приложений, порталов и веб-служб.

Glassfish

Сервер Java EE с открытым исходным кодом, используемый для разработки, интеграции и развертывания приложений, порталов и веб-служб.

IBM WebSphere

Коммерческий сервер Java EE, используемый для разработки, интеграции и развертывания приложений, порталов и веб-служб.

JavaServer Faces

Технология JavaServer Faces упрощает построение пользовательских интерфейсов для приложений JavaServer. JSF-реализации и компоненты включают: Apache MyFaces, ICEFaces, RichFaces и Primefaces.

Jetty

Веб-контейнер для сервлетов Java и JavaServer Pages.

Oracle WebLogic Application Server

Коммерческий сервер Java EE, используемый для разработки, интеграции и развертывания приложений, порталов и веб-служб.

Resin

Высокопроизводительный сервер приложений Java, оптимизированный для облачных вычислений.

Seam Framework

Платформа для веб-разработки с открытым исходным кодом.

ServiceMix

Сервисная шина предприятия от Apache, которая сочетает в себе функциональность сервисно-ориентированной архитектуры и архитектуры, управляемой событиями, по спецификации Java Business Integration.

Sling

Инфраструктура для создания веб-приложений, в которой используется стиль архитектуры программного обеспечения Representational State Transfer (REST).

Struts

Инфраструктура для создания корпоративных веб-приложений на Java на основе архитектуры MVC ("модель–представление–контроллер").

Tapestry

Инфраструктура для создания веб-приложений на основе API сервлетов Java.

Tomcat

Веб-контейнер от Apache для запуска сервлетов Java и приложений JavaServer Pages.

TomEE

Стек сертификации веб-профилей Java EE 6, полностью принадлежащий Apache.

WildFly

WildFly, формально называемый JBoss Application Server, — это сервер Java EE с открытым исходным кодом, используемый для разработки, интеграции и развертывания приложений, порталов и веб-служб.

Интерпретируемые языки (совместимые с JSR-223)

BeanShell

Встраиваемый интерпретатор исходного кода Java, поддерживающий сценарный язык на основе объектов.

Clojure

Язык динамического программирования, ориентированный на машины Java Virtual Machine, Common Language Runtime и JavaScript.

FreeMarker

Движок шаблонов общего назначения, основанный на Java.

Groovy

Сценарный язык со многими возможностями Python, Ruby и Smalltalk, поддерживающий Java-подобный синтаксис.

Jacl

Реализация сценарного языка Tcl на Java.

JEP

Java Math Expression Parser (JEP) — библиотека Java для синтаксического анализа и вычисления математических выражений.

Jawk

Реализация сценарного языка AWK на Java.

Jelly

Средство подготовки сценариев, предназначенное для преобразования XML-кода в исполняемый код.

JRuby

Реализация языка программирования Ruby на Java.

Jython

Реализация языка программирования Python на Java.

Nashorn

Реализация JavaScript на Java. Это *единственный* сценарный язык, интерпретатор которого включен в Java Scripting API по умолчанию.

Scala

Язык программирования общего назначения, предназначенный для представления типичных программных шаблонов в лаконичном, элегантном и типизированном виде.

Sleep

Встраиваемый сценарный язык для приложений на языке Java, написанный на Perl.

Visage

Предметно-ориентированный язык (Domain Specific Language — DSL), предназначенный для написания пользовательских интерфейсов.

Velocity

Движок шаблонов общего назначения от Apache, написанный на Java.

Основы UML

Унифицированный язык моделирования (Unified Modeling Language — UML) — это язык для создания и моделирования объектов, в котором используется совокупность графических обозначений для создания абстрактной модели какой-либо определенной системы. UML поддерживается Консорциумом по разработке и продвижению объектно-ориентированных технологий и стандартов (Object Management Group — OMG). Этот язык моделирования можно применить к программам, написанным на языке Java, чтобы представить в графическом виде такие характеристики, как отношения между классами и диаграммы последовательности. Самые последние спецификации для UML можно найти на веб-сайте OMG `http://www.omg.org/spec/UML/`. Подробные сведения об UML можно найти в третьем издании книги Крэга Лармана *Применение UML 2.0 и шаблонов проектирования* (пер. с англ., ИД "Вилямс", 2013 г, ISBN 978-5-8459-1185-8).

Диаграммы классов

Диаграмма классов представляет статическую структуру системы, на которой отображена информация о классах и взаимосвязях между ними. Отдельно взятая диаграмма классов состоит из трех разделов: наименование, атрибуты (необязательно) и операции (необязательно); см. рис. В.1 и последующий пример.

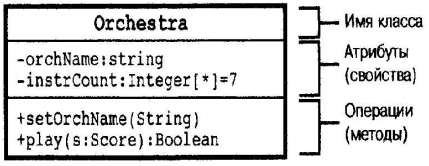


Рис.В.1. Диаграмма классов

```
// Соответствующий сегмент кода
class Orchestra { // Имя класса
   // Атрибуты
   private String orch Name;
   private Integer instrCount = 7;
   // Операции
   public void setOrchName(String name) {...}
   public Boolean play(Score s) {...}
}
```

Наименование

Раздел наименования является обязательным. В нем указывается имя класса или интерфейса, набранное полужирным шрифтом.

Атрибуты

Раздел атрибутов также является необязательным. В нем перечислены переменные-члены (свойства) класса, которые представляют состояние соответствующего объекта. Формат описания атрибутов на языке UML выглядит так:

```
Видимое имя : тип [кратность] = стандартноеЗначение
{строка-свойства}
```

Как правило, на диаграмме отображаются только имена и типы атрибутов.

Операции

Раздел операций является необязательным. В нем перечислены функции-члены класса (методы), которые представляют поведение соответствующей системы. Формат описания операций на языке UML выглядит так:

```
Видимое имя (список-параметров) :
тип-возвращаемого значения
{строка-свойства}
```

Как правило, на диаграмме отображаются только имена операций и списки параметров.

НА ЗАМЕТКУ

Вместо конструкции `{строка-свойства}` указывается любое из доступных свойств, таких как `{ordered}` или `{read-only}`.

Видимость

Для модификаторов доступа можно определить индикаторы видимости (префикс-символы), хотя делать это необязательно. Такие индикаторы можно применить к переменным-членам и функциям-членам диаграммы классов (табл. В.1).

Таблица В.1. Индикаторы видимости

Индикатор	Модификаторы доступа
~	*Пакетный закрытый* (package-private)
#	`protected` (защищенный)
-	`private` (закрытый)
+	`public` (открытый)

Диаграммы объектов

Диаграммы объектов отличаются от диаграмм классов тем, что в них подчеркивается имя соответствующего объекта в разделе наименования. Этот текст может быть представлен тремя способами (табл. В.2).

Таблица В.2. Представление имен объектов

`: ClassName`	Только имя класса
`objectName`	Только имя объекта
`objectName : ClassName`	Имя объекта и класса

Диаграммы объектов используются нечасто, но они могут быть полезны при детализации информации, как показано на рис. В.2.

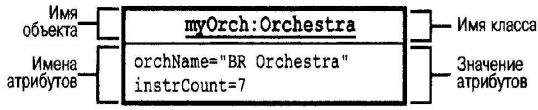


Рис. B.2. Диаграмма объектов

Представление в виде графических символов

Графические символы являются основными компонентами диаграмм UML (рис. B.3).

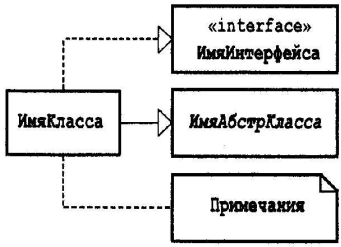


Рис. B.3. Графические символы UML

Классы, абстрактные классы и интерфейсы

Классы, абстрактные классы и интерфейсы представляются их именами, набранными полужирным шрифтом в прямоугольной рамке. В дополнение к этому имена абстрактных классов набираются курсивом. Интерфейсам предшествует слово `interface`, заключенное в угловые кавычки (« »). В кавычки заключаются стереотипы, а в случае интерфейса — классификатор.

Примечания

Это комментарии, помещенные в прямоугольную рамку с загнутым уголком. Они могут быть представлены сами по себе или могут быть соединены пунктирной линией с каким-либо другим символом.

Пакеты

Пакет отображается в виде символа, похожего на папку. Имя пакета указывается в более крупном прямоугольнике, если только в нем не присутствуют какие-либо другие графические элементы (например, символы классов). В последнем случае имя пакета указывается в меньшем прямоугольнике. Зависимости между пакетами отображаются пунктирной линией с широкой стрелкой на конце.

Эта стрелка всегда указывает в направлении пакета, который требуется, чтобы удовлетворить соответствующую зависимость. Диаграммы пакетов представлены на рис. B.4.

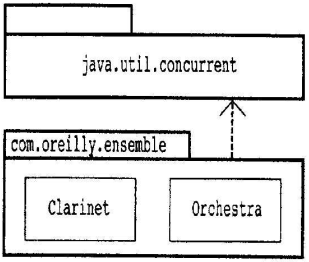


Рис. B.4. Диаграммы пакетов

Соединения

Это графические изображения, которые показывают связи между классами. Соединения подробно описаны в ниже, в разделе "Отношения между классами".

Индикаторы кратности

Индикаторы кратности указывают, сколько объектов участвует в той или иной ассоциации (табл. B.3). Эти индикаторы обычно помещаются после соединения и могут также использоваться как часть той или иной переменной-члена в разделе атрибутов.

Таблица В.3. Индикаторы кратности

Индикатор	Описание
*	Нуль или большее число объектов
0..*	Нуль или большее число объектов
0..1	По выбору (нуль или один объект)
0..n	От нуля до *n* объектов, где *n* > 1
1	В точности один объект
1..*	Один или большее число объектов
1..n	От одного до *n* объектов, где *n* > 1
m..n	Указанный диапазон объектов
n	Только *n* объектов, где *n* > 1

Имена ролей

Имена ролей используются, когда необходимо дополнительно прояснить отношения между классами. Имена ролей часто указываются вместе с индикаторами кратности. На рис. В.5 показан оркестр, где *исполняется* одна или несколько партитур.

Рис. В.5. Имена ролей

Отношения между классами

Отношения между классами представляются с помощью соединений и диаграмм классов (рис. В.6). Кроме того, для отображения отношений между классами могут также использоваться индикаторы кратности и имена ролей.

Ассоциация

Ассоциация обозначает связь между классами и может быть двунаправленной. На целевом конце (концах) могут быть указаны индикаторы кратности и атрибуты классов.

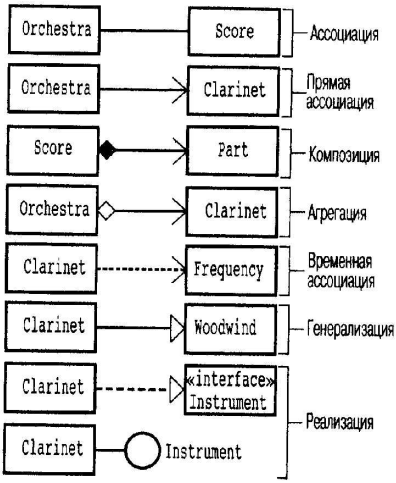


Рис. B.6. Отношения между классами

Прямая ассоциация

Прямая ассоциация, называемая также возможностью навигации, представляет собой связь, направляющую исходный класс к классу-цели. Эта связь может читаться так: "в" оркестре "есть" кларнет. На целевом конце (концах) могут быть помещены индикаторы кратности и атрибуты классов. Возможность навигации между классами может быть двунаправленной.

Композиция

Композиция, называемая также как *включением* (containment), моделирует отношение "целое–часть", где целое управляет продолжительностью существования частей. Части могут существовать лишь как компоненты целого. Это более прочная форма ассоциации, чем агрегация. Можно сказать, что партитура "состоит из" одной или нескольких частей.

Агрегация

Агрегация моделирует отношение "целое–часть", где части могут существовать независимо от целого. Целое не управляет существованием частей. Можно сказать, что оркестр — это целое, а кларнет — "часть" оркестра.

Временная ассоциация

Временная ассоциация, чаще всего называемая *зависимостью* (dependency), используется в случаях, когда один класс требует существования какого-то другого класса. Она также наблюдается в случаях, когда некий объект используется как локальная переменная, возвращаемое значение или аргумент функции-члена. Передача значения частоты методу настройки класса `Clarinet` (Кларнет) может интерпретироваться так: класс `Clarinet` зависит от класса `Frequency` (Частота), или кларнет "использует" частоту.

Генерализация

В случае генерализации (обобщения) некий специализированный класс наследует элементы более общего класса. В Java это называется наследованием, например, класс `Clarinet` расширяет класс `Woodwind` (Деревянные духовые инструменты), или кларнет "является" деревянным духовым инструментом.

Реализация

Реализация моделирует реализацию тем или иным классом некоторого интерфейса, например, в классе `Clarinet` реализованы методы интерфейса `Instrument` (Инструмент).

Диаграммы последовательности

Диаграммы последовательности UML используются для отображения динамического взаимодействия объектов (рис. B.7). Это

взаимодействие начинается сверху диаграммы и направляется в ее нижнюю часть.

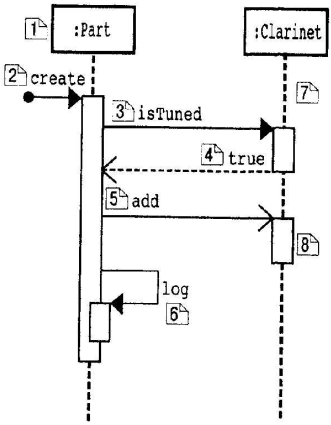


Рис. B.7. Диаграмма последовательности

Участник (1)

Участники считаются объектами.

Найденное сообщение (2)

Найденное сообщение — это сообщение, в котором вызывающая сторона не представлена на диаграмме. Это означает, что отправитель неизвестен или не обязательно должен отображаться на данной диаграмме.

Синхронное сообщение (3)

Синхронное сообщение используется, когда источник ждет, пока получатель завершит обработку соответствующего сообщения.

Ответный вызов (4)

Ответный вызов может отобразить возвращаемое значение и обычно исключается из диаграмм последовательности.

Асинхронное сообщение (5)

Асинхронное сообщение используется, когда источник не ждет, пока получатель завершит обработку соответствующего сообщения.

Сообщение, адресованное самому себе (6)

Сообщение, адресованное самому себе, или *самовызов*, является сообщением, которое остается в соответствующем объекте.

Линия жизни (7)

Линии жизни ассоциируются с каждым объектом и направлены по вертикали. Они связаны со временем и читаются сверху вниз, причем самое раннее событие указано наверху страницы.

Полоса активности (8)

Полоса активности отображается на линии жизни или какой-либо другой полосе активности в виде прямоугольника. Полоса активности показывает, когда соответствующий участник (объект) оказывается активным в данном взаимодействии.

Предметный указатель

A

ActiveMQ *226*
Apache MyFaces *230*
ASCII *25*
ASCII-символы
  неотображаемые *26*
  отображаемые *26*
Assertions *89*

B

BlueJ *228*

C

CASE *224*
CMS Collector *145*
Collections Framework *23*
CORBA *124*
CRL *125*
Cygwin *199*

D

DGC *124*
DSA *125*

E

EDT *97*

G

Garbage collection *143*
GC *143*
Generics Framework *187*
GZIP *160*

H

HotSpot *143*
HPROF *146*

I

ICEFaces *230*
IDE *130*

J

Jackrabbit *227*
Java
  Compatibility Kit *124*
  Development Kit *111*; *129*
  Language Specification *16*
  letter *29*
  Management Extensions *146*
  Math Expression Parser *232*
  Runtime Environment *111*; *129*
  Scripting API *195*
  Virtual Machines *111*; *129*
JavaFX *112*
JavaServer Faces *230*
JBoss Application Server *231*
JCK *124*
JDeveloper *229*
JDK *111*; *129*
JEP *232*
JFreeChart *227*
JLS *16*
JMock *221*
JMX *146*
JNDI *116*
JOOQ *221*
JRE *111*; *129*
JRuby *198*; *199*
JSR-223 *231*
JSR-334 *15*
JVM *111*; *129*

L

LDAP *116*

M

Make *223*
Metaspace *150*
MSYS *199*

N

newSingleThreadExecutor() *175*
newSingleThreadScheduledExecutor() *175*
NIO *153*
  NIO 2.0 *165*

O

Object Management Group *233*
OMG *233*

P

PermGen *150*
PLAF *122*
Pluggable Look-and-Feel *122*
Primefaces *230*
Project Coin *15*

R

Representational State Transfer *230*
REST *230*
RichFaces *230*
RSA *125*

S

SAM *79*
SAX *127*
Service loader *23*
SQL *116*
Swing API *112*

T

Tokens *25*
TZDB *208*

U

UML *15*; *16*; *233*
Unicode *25*
Unified Modeling Language *15*; *16*

V

Varargs *70*

W

WildFly *231*

Z

ZIP *160*

А

Абстрактные
  классы *72*; *236*
  методы *72*
Автоупаковка *47*; *56*
Агрегация *240*
Алгоритмы *181*
Аннотации *76*
  @Deprecated *77*
  @FunctionalInterface *79*; *211*
  @Override *77*
  Retention *78*
  @SuppressWarnings *77*
  встроенные *77*
  разработчика *77*
Ассоциация *238*
  временная *240*
  прямая *239*
Атрибуты *234*

Б

Библиотеки *226*
  java.applet *114*
  java.awt *120*
  java.awt.color *120*
  java.awt.datatransfer *120*
  java.awt.dnd *121*
  java.awt.event *121*
  java.awt.font *121*
  java.awt.geom *121*
  java.awt.im *121*
  java.awt.image *121*
  java.awt.image.renderable *121*
  java.awt.print *121*
  java.beans *114*
  java.beans.beancontext *114*
  javafx.animation *118*
  javafx.application *118*
  javafx.beans *118*
  javafx.beans.binding *118*

javafx.beans.property *118*
javafx.beans.property.adapter *118*
javafx.beans.value *118*
javafx.collections *118*
javafx.concurrent *118*
javafx.embed.swing *118*
javafx.embed.swt *118*
javafx.event *118*
javafx.fxml *118*
javafx.geometry *119*
javafx.scene *119*
javafx.scene.canvas *119*
javafx.scene.chart *119*
javafx.scene.control *119*
javafx.scene.control.cell *119*
javafx.scene.effect *119*
javafx.scene.image *119*
javafx.scene.input *119*
javafx.scene.layout *119*
javafx.scene.media *119*
javafx.scene.paint *119*
javafx.scene.shape *120*
javafx.scene.text *120*
javafx.scene.transform *120*
javafx.scene.web *120*
javafx.stage *120*
javafx.util *120*
javafx.util.converter *120*
java.io *114*
java.lang *112*
java.lang.annotation *112*
java.lang.instrument *112*
java.lang.invoke *112*
java.lang.management *112*
java.lang.ref *112*
java.lang.reflect *112*
java.math *114*
java.net *114*
java.nio *114*
java.nio.channels *114*
java.nio.charset *114*
java.nio.file *114*
java.nio.file.attribute *115*
java.rmi *123*
java.rmi.activation *123*
java.rmi.dgc *124*
java.rmi.registry *124*
java.rmi.server *124*
java.security *125*
java.security.cert *125*
java.security.interfaces *125*
java.security.spec *125*
java.sql *116*
java.text *115*
java.time *115*
java.time.chrono *115*
java.time.format *115*
java.time.temporal *115*
java.time.zone *115*
java.util *113*
java.util.concurrent *113*
java.util.concurrent.atomic *113*
java.util.concurrent.locks *113*
java.util.function *113*
java.util.jar *113*
java.util.logging *113*
java.util.prefs *113*
java.util.regex *113*
java.util.stream *113*
java.util.zip *113*
javax.accessibility *117*
javax.annotation *115*
javax.crypto *125*
javax.crypto.interfaces *125*
javax.crypto.spec *125*
javax.imageio *117*
javax.jws *116*
javax.jws.soap *116*
javax.management *115*
javax.naming *116*
javax.naming.directory *116*
javax.naming.event *116*
javax.naming.ldap *116*
javax.net *115*
javax.net.ssl *115*
javax.print *117*
javax.print.attribute *117*
javax.print.attribute.standard *117*
javax.print.event *117*
javax.rmi *124*

javax.rmi.CORBA *124*
javax.rmi.ssl *124*
javax.script *116*
javax.security.auth *125*
javax.security.auth.callback *125*
javax.security.auth.kerberos *126*
javax.security.auth.login *126*
javax.security.auth.x500 *126*
javax.security.sasl *126*
javax.sound.midi *117*
javax.sound.sampled *117*
javax.sql *116*
javax.sql.rowset *117*
javax.sql.rowset.serial *116*
javax.swing.border *122*
javax.swing.colorchooser *122*
javax.swing.event *122*
javax.swing.filechooser *122*
javax.swing.plaf *122*
javax.swing.plaf.basic *122*
javax.swing.plaf.metal *122*
javax.swing.plaf.multi *122*
javax.swing.plaf.nimbus *122*
javax.swing.plaf.synth *122*
javax.swing.table *123*
javax.swing.text *123*
javax.swing.text.html *123*
javax.swing.text.html.parser *123*
javax.swing.text.rtf *123*
javax.swing.tree *123*
javax.swing.undo *123*
javax.tools *115*
javax.transactions.xa *117*
javax.xml *126*
javax.xml.bind *126*
javax.xml.crypto *126*
javax.xml.crypto.dom *126*
javax.xml.crypto.dsig *127*
javax.xml.datatype *127*
javax.xml.namespace *127*
javax.xml.parsers *127*
javax.xml.soap *127*
javax.xml.transform *127*
javax.xml.transform.dom *127*
javax.xml.transform.sax *127*
javax.xml.transform.stax *127*
javax.xml.validation *127*
javax.xml.ws *127*
javax.xml.ws.handler *128*
javax.xml.ws.handler.soap *128*
javax.xml.ws.http *128*
javax.xml.ws.soap *128*
javax.xml.xpath *128*
org.ietf.jgss *126*
org.omg.CORBA *124*
org.omg.CORBA_2_3 *124*
org.w3c.dom *128*
org.xml.sax *128*
Блок *82*
Буква Java *29*

В

Видимость
индикаторы *235*
Включение *239*

Г

Генерализация *240*

Д

Десериализация *160*
Диаграммы
классов *233*
объектов *235*
последовательности *240*

З

Зависимость *240*

И

Идентификатор *29*
Имена *21*
Инициализаторы
статические *74*
Инкапсуляция *63*
Интегрированные среды
разработки *228*
Интерпретатор
Java *134*

параметры *134*
Интерфейс
функциональный *211*
Интерфейсные классы *46*
Интерфейсы *75*; *236*
Autoclosable *101*
Cloneable *61*
Collection *179*
Comparator *183*; *213*
Executor *174*
FileVisitor *168*
java.lang.Runnable *169*
java.nio.file.DirectoryStream *168*
List *179*
Map *179*
Path *165*
PathMatcher *168*
Queue *179*
Runnable *169*; *170*
ScriptEngine *195*
Serializable *159*
Set *179*
WatchService *168*
имена *21*
функциональные *79*
Инфраструктуры
обобщений *187*
Исключения *91*; *101*
ArithmeticException *94*
ArrayIndexOutOfBoundsException *94*
ClassCastException *55*; *94*
ClassNotFoundException *93*
CloneNotSupportedException *61*; *94*
DateTimeException *95*
EOFException *94*
FileNotFoundException *94*
IllegalArgumentException *95*
IllegalStateException *95*
IndexOutOfBoundsException *95*
InterruptedException *94*
IOException *93*; *153*; *154*
NotSuchMethodException *94*
NullPointerException *95*
NumberFormatException *95*
RuntimeException *93*
SQLException *94*
UncheckedIOException *95*
непроверяемые *92*
определяемые программистом *102*
проверяемые *66*; *92*
Исполнители *174*

К

Календари
ISO *205*
Григорианский *205*
Классы *63*; *236*
Annotation *51*
Array *51*
BlockingDeque *176*
BlockingQueue *176*
BufferedOutputStream *157*; *159*
BufferedReader *155*; *158*
Character *25*
Class *51*; *187*
Clock *207*
Collections *181*; *183*; *191*
ConcurrentHashMap *176*
ConcurrentMap *176*
ConcurrentSkipListMap *176*
ConcurrentSkipListSet *176*
Console *154*
CopyOnWriteArrayList *176*
CopyOnWriteArraySet *176*
DataInputStream *158*
DataOutputStream *157*; *159*
DateTimeFormatter *209*
DayOfWeek *207*
Deque *176*
Double *43*
Duration *207*; *208*
enum *60*; *76*
Enumeration *51*
Error *102*
Exception *102*
Executors *175*
File *162*; *165*

FileReader *155*
Files *166*
FileWriter *157*
Float *43*
HashMap *176*
HashMap() *58*
HashSet *58*
InputStream *154*; *156*
Instant *205*
Interface *51*
java.lang.Object *51*
java.util.Formatter *72*
List *176*
LocalDate *205*
LocalDateTime *206*; *221*
LocalTime *205*
Map *176*
Month *207*
MonthDay *207*
Object *58*; *91*; *151*
ObjectInputStream *160*
ObjectOutputStream *160*
OffsetDateTime *206*
OffsetTime *206*
OutputStream *154*
Paths *165*
Period *207*; *209*
PrintWriter *156*; *158*
PriorityBlockingQueue *176*
PriorityQueue *176*
PushbackInputStream *156*
Queue *176*
RandomAccessFile *163*
Reader *154*
RuntimeException *102*
ScheduledThreadPoolExecutor *174*
ScriptEngineManager *195*
Set *176*
Socket *158*
String *34*; *59*; *60*
StringBuffer *60*
StringBuilder *60*
Thread *169*
ThreadPoolExecutor *174*
Throwable *91*; *103*
TreeMap *176*
TreeSet *176*
Writer *154*
Year *206*
YearMonth *206*
ZipInputSteam *161*
ZipOutputStream *161*
ZonedDateTime *206*
ZonedId *206*
ZoneOffset *206*
абстрактные *72*
имена *21*
отношения *238*
синтаксис *64*
Клонирование *60*; *61*
Ключевые слова *28*
abstract *72*
catch *90*
const *28*
enum *28*
extends *68*
final *74*
finally *90*
goto *28*
interface *75*
new *64*
static *74*
super *68*
synchronized *89*; *173*
this *69*
throw *90*; *97*
transient *159*
try *90*
Коллекции *48*; *179*; *188*
List *191*
Комментарии *27*
Javadoc *27*
Компиляторы
Java *133*
параметры *133*
Композиция *239*
Константы
имена *23*
статические *74*
Конструкторы *67*

копирования *62*
Куча *53*; *150*

Л

Лексемы *25*
Литералы *32*
  false *29*
  null *29*
  true *29*
  булевы *32*
  нулевые *35*
  простого типа *40*
  символьные *32*
  с плавающей точкой *34*
    double *34*
    float *34*
  строковые *34*
  целочисленные *32*
    int *32*
    long *33*
    восьмеричные *33*
    десятичные *33*
    шестнадцатеричные *33*
Локальные
  классы *82*
  переменные *82*

М

Массивы *53*
  анонимные *54*
  многомерные *54*
Метаданные *51*; *76*
Метапространство *150*
Метка *87*
методы
  Remove() *181*
Методы
  activeCount() *173*
  add() *181*
  addAll() *181*
  asLifoQueue() *181*
  between() *208*
  binarySearch() *182*
  clone() *61*; *94*
  close() *155*; *156*
  contains() *181*
  containsKey() *181*
  containsValue() *181*
  copy() *167*; *181*
  currentThread() *173*
  delete() *162*
  disjoint() *181*
  equals() *57*; *58*; *59*
  eval() *196*
  exists() *162*
  fill() *182*
  finalize() *151*
  frequency() *181*
  get() *181*
  getMessage() *103*
  getPriority() *171*
  getState() *171*
  hashCode() *58*
  indexOf() *181*
  intern() *35*
  interrupt() *172*
  interrupted() *173*
  intValue() *49*
  invokeFunction() *196*
  isAlive() *172*
  isInterrupted() *172*
  iterator() *181*
  join() *172*
  keySet() *181*
  list() *162*
  main() *132*
  max() *181*
  min() *181*
  mkdir() *162*
  move() *167*
  newCachedThreadPool() *175*
  newFixedThreadPool() *175*
  newScheduledThreadPool() *175*
  newSetFromMap() *182*
  notify() *172*
  notifyAll() *172*
  now() *207*
  printf() *71*
  println() *36*
  printStackTrace() *103*; *104*

put() *181*
read() *156*; *158*
readLine() *155*
renameTo() *162*
replaceAll() *182*
reverse() *181*
rotate() *182*
run() *170*
seek() *163*
setPriority() *172*
shuffle() *181*
size() *181*
sleep() *173*
sort() *182*
start() *172*
swap() *182*
System.gc() *151*
toString() *103*
values() *76*
wait() *172*
write() *159*
writeInt() *157*
yield() *173*
абстрактные *72*
доступа *63*
защитники *107*
переопределение *66*
правила *67*
по умолчанию *107*
с переменным числом параметров *70*
статические *74*
Модификаторы *105*
abstract *107*
default *107*
final *107*
native *107*
private *106*
protected *106*
public *106*
static *107*
strictfp *107*
synchronized *107*
transient *108*
volatile *108*
пакетный закрытый *106*
Мьютекс *174*

Н

Наименование *234*
Наследование *63*; *240*
Непроверяемые исключения *92*

О

Область действия *82*
Обобщения *187*; *240*
Объекты *63*
Clock *208*
DataInputStream *156*
GZipInputStream *161*
GZIPOutputStream *161*
Instant *207*
ObjectInputStream *160*
ООП *63*
Операторы *30*; *81*
<> *188*
== *58*
break *84*; *87*
continue *88*
do-while *86*
for *85*
расширенный *85*
if *83*
if else *83*
if else if *83*
import *131*
new *54*
return *67*; *88*
switch *84*; *87*
throws *92*
try-catch *97*
try-catch-finally *100*
try-finally *99*
try с ресурсами *100*
while *86*
аддитивные *46*
выражения *81*
мультипликативные *46*
обработки исключений *90*
побитовые *46*

пустой *82*
равенства *46*; *57*
ромбический *188*
сравнения *46*
точка (.) *65*; *73*
условные *46*; *82*
цикла *85*
Операции *234*
Ошибки *93*
AssertionError *95*
ExceptionInInitializeError *95*
inconvertible types *56*
NoClassDefFoundError *96*
OutOfMemoryError *96*
StackOverflowError *96*
VirtualMachineError *95*

П

Пакеты *112*; *131*; *237*
java.lang *46*; *132*
java.lang.annotation *78*
java.time *131*; *203*; *205*
java.util.concurrent *174*
java.util.function *215*
javax.script *195*
имена *24*
Перегрузка *65*
Переменные
локальные *52*
экземпляра *52*
Перечисления *76*
Thread.state *170*
TimeUnit *177*
имена *23*
Подкласс *68*
Полоса активности *242*
Потоки
System.err *154*
System.in *153*
System.out *154*
приоритет *171*
создание *169*
состояние *170*
Преобразование
расширяющие *55*
ссылочных типов *55*
сужающие *55*
Принцип подстановки *189*
Проверяемые исключения *66*; *92*
Простые типы *39*
Пулы потоков *174*

Р

Разделители *30*
Распаковка *47*; *48*; *56*
Расширение *55*
Реализация *240*
Роли *238*
Ромбический оператор *188*

С

Сборка мусора *143*
Сборщик мусора
Garbage-First (G1) *145*
взаимодействие *151*
параллельный *144*
с уплотнением *144*
последовательный *144*
с одновременной маркировкой и очисткой (CMS) *145*
Сервисная шина предприятия *230*
Сервисный загрузчик *23*
Сериализация *62*; *160*
Сигнатура *66*
Символ "О" *183*
Символы валют *36*
Синглтон *76*
Синхронизаторы *176*
CountDownLatch *177*
CyclicBarrer *177*
Exchanger *177*
Semaphore *177*
Скобки *30*
Сокеты *157*
запись
бинарная *159*
символьная *158*
чтение
бинарное *158*
символьное *158*

Сравнение *57*
перечислений *60*
ссылок *57*
строк *59*
Средства разработки *223*
Ссылки
на метод *213*
Ссылочные типы *51*
Статические
данные-члены *73*
константы *74*
Статические методы *74*
Статические инициализаторы *74*
Сужение *55*
Суперкласс *63*; *68*
Сущности
-0.0 *42*
-Infinity *42*
Infinity *42*

Т

Текучий
API *221*
интерфейс *221*
Типы
int *32*
параметризация *187*
простые
boolean *39*
byte *39*
char *39*
double *40*
float *40*
int *39*
long *39*
short *39*

У

Унифицированный язык моделирования *233*
Управляющие
последовательности *35*
символы *26*
Условные операторы *82*
Утверждения *89*; *135*
Утилиты
HPROF Profiler *146*
jar *137*
javadoc *27*; *139*
jdb *146*
jhat *146*
jinfo *146*
jmap *146*
jstack *146*
jstat *146*

Ф

Файлы *154*
MANIFEST.MF *138*
Manifest.txt *138*
запись
бинарная *157*
символьная *156*
Файлы чтение
бинарное *156*
символьное *155*
Функциональный интерфейс *79*; *211*

Ц

Циклы
for
расширенный *72*

Ч

Числовое продвижение *44*
бинарное *45*
унарное *45*

Java SE 8
Вводный курс

Кей С. Хорстманн

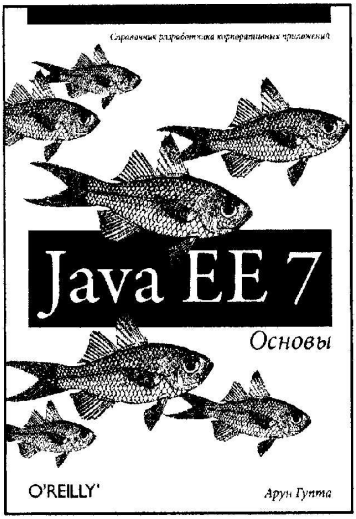
www.williamspublishing.com

В этом кратком руководстве рассматриваются нововведения в версии Java SE 8, главными из которых являются лямбда-выражения и новые потоки ввода-вывода. В книге поясняются на практических примерах исходного кода и снабжаются полезными советами эти и другие не менее важные новшества и усовершенствования в версии Java SE 8, в том числе библиотека для даты, времени и календаря, технология JavaFX для разработки пользовательских интерфейсов, средства параллельного программирования, интерпретатор Nashorn языка JavaScript и прочие мелкие изменения.
Книга рассчитана на опытных программистов, стремящихся писать в недалекой перспективе надежный, эффективный и безопасный код на Java.

ISBN 978-5-8459-1900-7 в продаже

Java EE 7
Основы

Арун Гупта



Книга написана одним из ведущих разработчиков проекта Java EE, и каждая глава в ней посвящена рассмотрению одной из ключевых спецификаций платформы, включая WebSockets, Batch Processing, RESTful Web Services и Java Message Service.

- Ознакомьтесь с ключевыми компонентами платформы Java EE, руководствуясь многочисленными примерами в виде фрагментов кода, сопровождаемых подробными пояснениями автора.
- Изучите все новые технологии, которые были добавлены в версии Java EE 7, включая веб-сокеты, JSON, пакетную обработку и утилиты параллельного выполнения.
- Узнайте о применении веб-служб RESTful, служб на основе SOAP и службы сообщений Java (JMS).
- Изучите технологии Enterprise JavaBeans, CDI (Contexts and Dependency Injection) и Java Persistence.
- Узнайте о том, каким изменениям подверглись различные компоненты при переходе от Java EE 6 к Java EE 7.

www.williamspublishing.com

ISBN 978-5-8459-1896-3 **в продаже**

JAVA
РУКОВОДСТВО ДЛЯ ПРОГРАММИСТА
75 РЕКОМЕНДАЦИЙ ПО НАПИСАНИЮ НАДЕЖНЫХ И ЗАЩИЩЕННЫХ ПРОГРАММ

Фрэд Лонг,
Дхрув Мохиндра,
Роберт С. Сикорд,
Дин Ф. Сазерленд,
Дэвид Свобода

Это справочное руководство составлено из 75 рекомендаций по надежному, безопасному и корректному написанию кода на Java. Каждая рекомендация составлена авторами по одному и тому же образцу: постановка задачи, анализ примера кода, не соответствующего принятым нормам программирования на Java, рассмотрение предлагаемого решения, соответствующего принятым нормам, краткое изложение применимости рекомендации и ссылки на дополнительную литературу. Представленные рекомендации отражают опыт, накопленный в области безопасного и надежного программирования на Java, и поэтому они будут полезны всем, кто занимается разработкой программ на этом языке программирования.

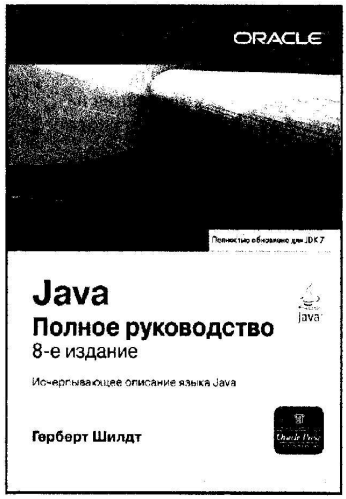
Книга рассчитана на тех, кто имеет определенный опыт написания кода на Java.

www.williamspublishing.com

ISBN 978-5-8459-1897-0 в продаже

JAVA
ПОЛНОЕ РУКОВОДСТВО
8-е издание

Герберт Шилдт



www.williamspublishing.com

Эта книга предназначена для всех программистов – как для новичков, так и для профессионалов. Начинающий программист найдет в ней подробные пошаговые описания и множество чрезвычайно полезных примеров. А углубленное рассмотрение более сложных функций и библиотек Java должно удовлетворить ожидания профессиональных программистов. Для обеих категорий читателей в книге указаны действующие ресурсы и полезные ссылки.

В этой книге автор бестселлеров по программированию Герберт Шилдт знакомит вас со всем необходимым для разработки, компиляции, отладки и запуска Java-программ. Полностью обновленное для платформы Java Platform, Standard Edition 7 (Java SE 7), это исчерпывающее издание рассматривает язык Java в целом, включая его синтаксис, ключевые слова и фундаментальные принципы программирования. Здесь вы найдете информацию о ключевых элементах библиотеки Java API, рассмотрите JavaBeans, сервлеты, аплеты и Swing и ознакомитесь с работой Java в реальных ситуациях. Кроме того, в этой книге подробно обсуждаются такие новые средства Java SE 7, как оператор `try`-с-ресурсами, строки в операторе `switch`, выведение типов с оператором <>, NIO.2 и Fork/Join Framework.

ISBN 978-5-8459-1759-1

в продаже

O'REILLY®

Java 8. Карманный справочник

Если вам нужно получить оперативные ответы по разработке или отладке программ на Java, то эта книга послужит удобным справочником по стандартным возможностям языка программирования Java и его платформы. Вы найдете здесь полезные примеры кодов программ, таблицы, рисунки и списки, а также вспомогательную тематическую информацию, в том числе по Java Scripting API, средствам разработки сторонних фирм и основам унифицированного языка моделирования (Unified Modeling Language — UML). Вы узнаете также о новых возможностях Java 8 — лямбда-выражениях и API для работы с датой и временем.

Эта небольшая книга, включающая описание новых возможностей Java, до Java SE 8 включительно, будет вашим идеальным спутником, где бы вы ни находились — в офисе, в учебном классе или в пути.

- Быстро находите подробные сведения о языке Java, такие как соглашения о присвоении имен, описание простых типов и элементов объектно-ориентированного программирования
- Получите подробные сведения о платформе Java SE, включая основы разработки, управление памятью, параллелизм и обобщения
- Просматривайте базовую информацию, чтобы узнать о возможностях NIO 2.0, инфраструктуре коллекций Java и API языков сценариев Java
- Ознакомьтесь с краткой информацией по текучим API, средствам разработки и тестирования, библиотекам и IDE, а также изучите основы UML

"В книге прекрасно описаны лямбда-выражения и функциональные операции, а также другие новинки Java SE 8 наряду с остальной платформой".

Герт-Ян Виленха,
главный консультант по маркетингу в группе разработки NetBeans IDE компании Oracle

---

Категория: программирование
Предмет рассмотрения: язык Java
Уровень: промежуточный/опытный

ISBN 978-5-8459-2050-8

ВИЛЬЯМС

www.williamspublishing.com

www.oreilly.com