

П.Ноутон, Г.Шилдт

## JAVA 2

Подробно излагаются основы нового платформно-независимого, объектно-ориентированного языка программирования Java 2, приведено описание библиотек его классов и методики разработки приложений, апплетов и сервлетов с помощью Java-подсистем Bean и Swing, а также способы миграции приложений из C/C++. В книге рассматриваются интересные и сложные Java-приложения, в том числе многопользовательская сетевая игра в слова (word game). Изложение сопровождается огромным числом примеров и законченных программ, листинги которых Вы можете найти по адресу <http://www.osborne.com>.

## Содержание

Об авторах	1
Благодарности	3
Предисловие	5
ЧАСТЫ. ЯЗЫК JAVA	
<b>Глава 1. Генезис Java</b>	<b>11</b>
Происхождение Java	11
Рождение современного программирования: С	12
Потребность в C++	13
Этап становления Java	15
Создание Java	15
Почему Java важен для Internet	17
Java-апплеты и приложения	18
Защита	18
Мобильность	19
Волшебство Java: байт-код	19
Базовые термины Java	21
Простой	21
Объектно-ориентированный	22
Устойчивый	22
Многопоточный	23
Архитектурно-независимый	23
Интерпретируемость и высокая эффективность	23
Распределенный	24
Динамический	24
Продолжение революции	24
Свойства, добавленные версией 1.1	25
Свойства, исключенные из версии 1.1	25
Свойства, добавленные версией 2	26
Свойства, исключенные из версии 2	27
Java — не расширение HTML	27
<b>Глава 2. Обзор языка Java</b>	<b>29</b>

Объектно-ориентированное программирование	29
Две парадигмы программирования	29
Абстракция	30
Три принципа ООП	31
Первая простая программа	37
Ввод программы	37
Компиляция программы	38
Подробный взгляд на первую программу	39
Вторая короткая программа	41
Два оператора управления	43
Оператор <i>if</i>	43
Цикл <i>for</i>	45
Использование блоков кода	46
Лексические вопросы	48
Пробельные символы	48
Идентификаторы	48
Константы	49
Комментарии	49
Разделители	49
Ключевые слова языка Java	50
Библиотеки классов языка Java	51
<b>Глава 3. Типы данных, переменные и массивы</b>	<b>52</b>
Java — язык со строгой типизацией	52
Простые типы	53
Целые типы	53
Тип <i>byte</i>	54
Тип <i>short</i>	55
Тип <i>int</i>	55
Тип <i>long</i>	56
Типы с плавающей точкой	56
Тип <i>float</i>	57
Тип <i>double</i>	57
Символьный тип ( <i>char</i> )	58
Булевский тип ( <i>boolean</i> )	59
Подробнее о литералах	60
Целочисленные литералы	60
Литералы с плавающей точкой	61
Булевые литералы	62
Символьные литералы	62
Строковые литералы	63
Переменные	63
Обявление переменной	63
Динамическая инициализация	64

Область действия и время жизни переменных	65
Преобразование и приведение типов	68
Автоматическое преобразование типов в Java	68
Приведение несовместимых типов	69
Автоматическое расширение типа в выражениях	70
Правила расширения типов	71
Массивы	72
Одномерные массивы	72
Многомерные массивы	75
Альтернативный синтаксис объявления массива	80
Несколько слов относительно строк	80
Замечание для программистов C/C++ по поводу указателей	81
<b>Глава 4. Операции</b>	<b>82</b>
Арифметические операции	82
Основные арифметические операции	83
Деление по модулю	84
Арифметические операции присваивания	85
Инкремент и декремент	86
Поразрядные операции	88
Поразрядные логические операции	90
Левый сдвиг	92
Правый сдвиг	94
Правый сдвиг без знака	96
Поразрядная операция присваивания	97
Операции отношений	98
Операции булевой логики	100
Короткие логические операции	102
Операция присваивания	102
Условная операция	103
Старшинство операций	104
Использование круглых скобок	105
<b>Глава 5. Управляющие операторы</b>	<b>106</b>
Операторы выбора Java	106
Оператор <i>if</i>	106
Оператор <i>switch</i>	110
Операторы цикла	115
Оператор цикла <i>while</i>	115
Оператор цикла <i>do while</i>	117
Оператор цикла <i>for</i>	120
Вложенные циклы	125
Операторы перехода	125
Использование оператора <i>break</i>	126
Использование оператора <i>continue</i>	130

Оператор <i>return</i>	132
<b>Глава 6. Введение в классы</b>	<b>133</b>
Основы классов	133
Общая форма класса	134
Простой класс	135
Объявление объектов	138
Операция <i>new</i>	139
Назначение ссылочных переменных объекта	140
Представление методов	141
Добавление метода к классу <i>Box</i>	142
Возврат значений	144
Добавление метода с параметрами	146
Конструкторы	148
Параметризованные конструкторы	150
Ключевое слово <i>this</i>	152
Скрытие переменной экземпляра	152
Сборка "мусора"	153
Метод <i>finalize()</i>	153
Класс <i>Stack</i>	154
<b>Глава 7. Методы и классы</b>	<b>158</b>
Перегрузка методов	158
Перегрузка конструкторов	162
Использование объектов в качестве параметров	164
Передача аргументов	167
Возврат объектов	169
Рекурсия	170
Управление доступом	172
Статические элементы	177
Спецификатор <i>final</i>	179
Ревизия массивов	179
Вложенные и внутренние классы	181
Класс <i>String</i>	185
Использование аргументов командной строки	188
<b>Глава 8. Наследование</b>	<b>189</b>
Основы наследования	189
Доступ к элементам и наследование	191
Практический пример	192
Переменная суперкласса может ссылаться на объект подкласса	195
Использование ключевого слова <i>super</i>	196
Вызов конструктора суперкласса с помощью первой формы <i>super</i>	196
Использование второй формы <i>super</i>	200
Создание многоуровневой иерархии	201
Когда вызываются конструкторы	204

Переопределение методов	205
Динамическая диспетчеризация методов	208
Зачем нужны переопределенные методы?	210
Применение переопределения методов	211
Использование абстрактных классов	212
Использование ключевого слова <i>final</i> с наследованием	216
Использование <i>final</i> для отказа от переопределения	216
Использование <i>final</i> для отмены наследования	217
Класс <i>Object</i>	217
<b>Глава 9. Пакеты и интерфейсы</b>	<b>219</b>
Пакеты	220
Определение пакета	220
Использование CLASSPATH	221
Короткий пример пакета	222
Защита доступа	223
Пример управления доступом	224
Импорт пакетов	228
Интерфейсы	230
Определение интерфейса	231
Реализация интерфейсов	232
Применения интерфейсов	235
Переменные в интерфейсах	239
Расширение интерфейсов	241
<b>Глава 10. Обработка исключений</b>	<b>243</b>
Основные принципы обработки исключений	243
Типы исключений	244
Неотловленные исключения	245
Использование операторов <i>try</i> и <i>catch</i>	246
Отображение описания исключения	248
Множественные операторы <i>catch</i>	249
Вложенные операторы <i>try</i>	251
Оператор <i>throw</i>	253
Методы с ключевым словом <i>throws</i>	255
Блок <i>finally</i>	256
Встроенные исключения Java	258
Создание собственных подклассов исключений	260
Использование исключений	262
<b>Глава 11. Многопоточное программирование</b>	<b>263</b>
Поточная модель Java	264
Приоритеты потоков	265
Синхронизация	266
Передача сообщений	266
Класс <i>Thread</i> и интерфейс <i>Runnable</i>	267

Главный поток	268
Создание потока	270
Реализация интерфейса <i>Runnable</i>	270
Расширение <i>Thread</i>	272
Выбор подхода	274
Создание множественных потоков	274
Использование методов <i>isAlive()</i> и <i>join()</i>	276
Приоритеты потоков	278
Синхронизация	281
Использование синхронизированных методов	282
Оператор <i>synchronized</i>	284
Межпоточные связи	286
Блокировка	291
Приостановка, возобновление и остановка потоков	294
Приостановка, возобновление и остановка потоков в Java 1.1 и более ранних версиях	294
Приостановка, возобновление и остановка потока в Java 2	297
Использование многопоточности	299
<b>Глава 12. Ввод/вывод, апплеты и другие темы</b>	<b>301</b>
Основы ввода/вывода	301
Потоки	302
Байтовые и символьные потоки	302
Предопределенные потоки	305
Чтение консольного ввода	306
Чтение символов	307
Чтение строк	308
Запись консольного вывода	309
Класс <i>PrintWriter</i>	310
Чтение и запись файлов	312
Апплеты. Основы программирования	315
Модификаторы <i>transient</i> и <i>volatile</i>	319
Использование <i>instanceof</i>	319
Ключевое слово <i>strictfp</i>	322
Native-методы	323
Проблемы native-методов	327
ЧАСТЬ II. БИБЛИОТЕКА JAVA	329
<b>Глава 13. Обработка строк</b>	<b>331</b>
<i>String</i> -конструкторы	332
Длина строки	334
Специальные строковые операции	334
Строковые литералы	335
Конкатенация строк	335
Конкатенация других типов данных	336

Преобразование строк и метод <i>toString()</i>	337
Извлечение символов	338
Метод <i>charAt()</i>	338
Метод <i>getChars()</i>	339
Метод <i>getBytes()</i>	339
Метод <i>toCharArray()</i>	340
Сравнение строк	340
Методы <i>equals()</i> и <i>equalsIgnoreCase()</i>	340
Метод <i>regionMatches()</i>	341
Методы <i>startsWith()</i> и <i>endsWith()</i>	341
Сравнение <i>equals()</i> и операции <code>= =</code>	343
Метод <i>compareTo()</i>	343
Поиск строк	344
Изменение строки	346
Метод <i>substring()</i>	346
Метод <i>concat()</i>	347
Метод <i>replace()</i>	348
Метод <i>trim()</i>	348
Преобразование данных, использующее метод <i>valueOf()</i>	349
Изменение регистра символов в строке	350
Класс <i>StringBuffer</i>	350
Конструкторы <i>StringBuffer</i>	351
Методы <i>length()</i> и <i>capacity()</i>	351
Метод <i>ensureCapacity()</i>	352
Метод <i>setLength()</i>	352
Методы <i>charAt()</i> и <i>setCharAt()</i>	353
Метод <i>getChars()</i>	353
Метод <i>append()</i>	354
Метод <i>insert()</i>	355
Метод <i>reverse()</i>	355
Методы <i>delete()</i> и <i>deleteCharAt()</i>	356
Метод <i>replace()</i>	357
Метод <i>substring()</i>	357
<b>Глава 14. Пакет java.lang</b>	<b>358</b>
Оболочки простых типов	359
Класс <i>Number</i>	359
Оболочки <i>Double</i> и <i>Float</i>	360
Оболочки <i>Byte</i> , <i>Short</i> , <i>Integer</i> и <i>Long</i>	365
Оболочка <i>Character</i>	374
Оболочка <i>Boolean</i>	378
Класс <i>Void</i>	378
Класс <i>Process</i>	379
Класс <i>Runtime</i>	379

Управление памятью	381
Выполнение других программ	382
Класс <i>System</i>	384
Использование метода <i>currentTimeMillis()</i>	386
Использование метода <i>arraycopy()</i>	387
Свойства среды	387
Класс <i>Object</i>	388
Использование метода <i>clone()</i> и интерфейса <i>Cloneable</i>	389
Класс <i>Class</i>	391
Класс <i>ClassLoader</i>	394
Класс <i>Math</i>	395
Трансцендентные функции	395
Экспоненциальные функции	396
Округление функций	397
Разные методы класса <i>Math</i>	398
Компилятор	399
Классы <i>Thread</i> , <i>ThreadGroup</i> и интерфейс <i>Runnable</i>	399
Интерфейс <i>Runnable</i>	399
Класс <i>Thread</i>	399
Класс <i>ThreadGroup</i>	402
Классы <i>ThreadLocal</i> и <i>InheritableThreadLocal</i>	407
Класс <i>Package</i>	408
Класс <i>RuntimePermission</i>	409
Класс <i>Trowable</i>	409
Класс <i>SecurityManager</i>	410
Интерфейс <i>Comparable</i>	410
Пакеты <i>java.lang.ref</i> и <i>java.lang.reflect</i>	410
Пакет <i>java.lang.ref</i>	410
Пакет <i>java.lang.reflect</i>	411
<b>Глава 15. Пакет <i>java.util</i>: структура коллекций</b>	<b>412</b>
Краткий обзор коллекций	413
Интерфейсы коллекций	415
Интерфейс <i>Collection</i>	416
Интерфейс <i>List</i>	419
Интерфейс <i>Set</i>	421
Интерфейс <i>SortedSet</i>	421
Классы <i>Collection</i>	422
Класс <i>ArrayList</i>	423
Получение массива из <i>ArrayList</i> -объекта	425
Класс <i>LinkedList</i>	427
Класс <i>HashSet</i>	428
Класс <i>TreeSet</i>	430
Доступ к коллекции через итератор	431

Использование итератора	433
Коллекции пользовательских классов	434
Работа с картами отображений	436
Интерфейсы карт	436
Классы карт отображений	440
Компараторы	444
Использование компаратора	445
Алгоритмы коллекций	448
Массивы	452
Наследованные классы и интерфейсы	456
Интерфейс <i>Enumeration</i>	457
Класс <i>Vector</i>	457
Класс <i>Stack</i>	463
Класс <i>Dictionary</i>	465
Класс <i>Hashtable</i>	466
Класс <i>Properties</i>	470
Использование методов <i>store()</i> и <i>load()</i>	474
Резюме	476
<b>Глава 16. Пакет java.util: сервисные классы</b>	477
Класс <i> StringTokenizer</i>	477
Класс <i> BitSet</i>	479
Класс <i> Date</i>	482
Сравнение дат	484
Класс <i> Calendar</i>	484
Класс <i> GregorianCalendar</i>	488
Класс <i> TimeZone</i>	490
Класс <i> SimpleTimeZone</i>	491
Класс <i> Locale</i>	492
Класс <i> Random</i>	493
Класс <i> Observable</i>	496
Интерфейс <i> Observable</i>	497
Пример наблюдателя	498
Пакет <i> java.util.zip</i>	500
Пакет <i> java.util.jar</i>	500
<b>Глава 17. Ввод/вывод: обзор пакета java.io</b>	501
Классы и интерфейсы ввода/вывода Java	501
Класс <i> File</i>	503
Каталоги	506
Использование интерфейса <i> FilenameFilter</i>	507
Альтернативный метод <i> listFiles()</i>	508
Создание каталогов	509
Поточные классы	509
Байтовые потоки	509

Класс <i>InputStream</i>	510
Класс <i>OutputStream</i>	510
Класс <i>FileInputStream</i>	511
Класс <i>FileOutputStream</i>	513
Класс <i>ByteArrayInputStream</i>	515
Класс <i>ByteArrayOutputStream</i>	516
Фильтрованные байтовые потоки	517
Буферизированные байтовые потоки	518
Класс <i>SequenceInputStream</i>	522
Класс <i>PrintStream</i>	523
Класс <i>RandomAccessFile</i>	524
<b>Символьные потоки</b>	525
Класс <i>Reader</i>	525
Класс <i>Writer</i>	526
Класс <i>FileReader</i>	527
Класс <i>FileWriter</i>	528
Класс <i>CharArrayReader</i>	529
Класс <i>CharArrayWriter</i>	530
Класс <i>BufferedReader</i>	531
Класс <i>BufferedWriter</i>	533
Класс <i>PushbackReader</i>	533
Класс <i>PrintWriter</i>	534
<b>Использование поточного ввода/вывода</b>	535
Улучшение метода <i>wc</i> с помощью класса <i>StreamTokenizer</i>	537
<b>Сериализация</b>	519
Интерфейс <i>Serializable</i>	540
Интерфейс <i>Externalizable</i>	540
Интерфейс <i>ObjectOutput</i>	540
Класс <i>ObjectInputStream</i>	541
Интерфейс <i>ObjectInput</i>	54?
Класс <i>ObjectInputStream</i>	541
Пример с сериализацией	545
<b>Преимущества потоков</b>	547
<b>Глава 18. Работа в сети</b>	<b>548</b>
<b>Основы работы в сети</b>	<b>548</b>
Обзор сокетов	544
Клиент-сервер	549
Зарезервированные сокеты	550
Proxy-серверы	551
Адресация Internet	552
<b>Java и сеть</b>	<b>552</b>
Сетевые классы и интерфейсы	553
Класс <i>InetAddress</i>	553

Производственные методы	554
Методы экземпляра	555
Сокеты TCP/IP клиентов	556
Пример работы с сокет-соединением (программа Whois)	558
Использование URL	559
Формат	559
Класс <i>URLConnection</i>	561
Сокеты TCP/IP серверов:	562
Кэширующий proxy HTTP-сервер	563
Исходный код	564
Дейтаграммы	584
Класс <i>DatagramPacket</i>	584
Дейтаграммный сервер и клиент	585
Достоинства сети	587
<b>Глава 19. Класс <i>Applet</i></b>	<b>588</b>
Основы аплетов	588
Класс <i>Applet</i>	589
Архитектура аплета	592
Скелетная схема аплета	593
Инициализация и завершение аплета	594
Переопределение метода <i>update()</i>	595
Простые методы отображения аплетов	596
Требование перерисовки	598
Аплет с бегущим заголовком	599
Использование окна состояния	602
Тег <i>&lt;applet&gt;</i>	603
Пересылка параметров в аплеты	605
Усовершенствованный аплет заголовка	606
Методы <i>getDocumentBase()</i> и <i>getCodeBase()</i>	608
Интерфейс <i>AppletContext</i> и метод <i>showDocument()</i>	609
Интерфейс <i>AudioClip</i>	611
Интерфейс <i>AppletStub</i>	611
Вывод на консоль	611
<b>Глава 20. Обработка событий</b>	<b>612</b>
Два механизма обработки событий	612
Модель делегирования событий	613
События	613
Источники событий	614
Блок прослушивания событий	615
Классы событий	615
Класс <i>ActionEvent</i>	617
Класс <i>AdjustmentEvent</i>	618
Класс <i>ComponentEvent</i>	618

Класс <i>ContainerEvent</i>	619
Класс <i>FocusEvent</i>	620
Класс <i>InputEvent</i>	620
Класс <i>ItemEvent</i>	621
Класс <i>KeyEvent</i>	621
Класс <i>MouseEvent</i>	622
Класс <i>TextEvent</i>	624
Класс <i>WindowEvent</i>	624
Элементы-источники событий	625
Интерфейсы прослушивания событий	625
Интерфейс <i>ActionListener</i>	627
Интерфейс <i>AdjustmentListener</i>	627
Интерфейс <i>ComponentListener</i>	627
Интерфейс <i>ContainerListener</i>	627
Интерфейс <i>FocusListener</i>	627
Интерфейс <i>ItemListener</i>	628
Интерфейс <i>KeyListener</i>	628
Интерфейс <i>MouseListener</i>	628
Интерфейс <i>MouseMotionListener</i>	628
Интерфейс <i>TextListener</i>	629
Интерфейс <i>WindowListener</i>	629
Использование модели делегирования событий	629
Обработка событий мыши	630
Обработка событий клавиатуры	633
Классы-адаптеры	636
Внутренние классы	638
Анонимные внутренние классы	640
<b>Глава 21. Введение в AWT: работа с окнами, графикой и текстом</b>	<b>642</b>
Классы AWT	643
Основы оконной графики	645
Класс <i>Component</i>	646
Класс <i>Container</i>	646
Класс <i>Panel</i>	647
Класс <i>Window</i>	647
Класс <i>Frame</i>	647
Класс <i>Canvas</i>	648
Работа с фреймовыми окнами	648
Установка размеров окна	648
Скрытие и показ окна	649
Установка заголовка окна	649
Закрытие фрейм-окна	649
Создание фрейм-окна в апплете	649
Обработка событий фрейм-окна	651

Создание оконной программы	656
Отображение информации в окне	658
Работа с графикой	658
Рисование линий	659
Рисование прямоугольников	660
Рисование эллипсов и кругов	661
Рисование дуг	662
Рисование многоугольников	663
Установка размеров графики	664
Работа с цветом	665
Цветовые методы	665
Установка текущего цвета графики	666
Апплет с демонстрацией цветов	667
Установка режима рисования	668
Работа со шрифтами	670
Определение доступных шрифтов	671
Создание и выбор шрифта	673
Получение информации о шрифте	675
Управление текстовым выводом с помощью класса <i>FontMetrics</i>	676
Отображение многострочного текста	678
Выравнивание текста по центру	680
Выравнивание многострочного текста	681
Исследование текста и графики	685
<b>Глава 22. Использование элементов управления, менеджеров компоновки и меню AWT</b>	<b>686</b>
Элементы управления. Основные понятия	687
Добавление и удаление элементов управления	687
Реагирование на элементы управления	688
Текстовые метки	688
Использование кнопок	689
Обработка кнопок	690
Применение флажков	693
Обработка флажков	694
Класс <i>CheckboxGroup</i>	695
Элемент управления <i>Choice</i>	697
Обработка списков типа <i>Choice</i>	698
Использование списков	700
Обработка списков	701
Управление полосами прокрутки	703
Обработка полос прокрутки	705
Использование класса <i>TextField</i>	707
Обработка <i>TextField</i>	708
Использование <i>TextArea</i>	709

Понятие менеджера компоновки	711
Менеджер <i>FlowLayout</i>	713
Класс <i>BorderLayout</i>	714
Использование вставок	716
Менеджер <i>GridLayout</i>	718
Класс <i>CardLayout</i>	719
Панели меню и меню	723
Диалоговые окна	729
Класс <i>FileDialog</i>	734
Обработка событий путем расширения AWT-компонентов	736
Расширение класса <i>Button</i>	737
Расширение класса <i>Checkbox</i>	738
Расширение группы флажков	739
Расширение класса <i>Choice</i>	740
Расширение класса <i>List</i>	741
Расширение класса <i>Scrollbar</i>	742
Исследование элементов управления, меню и менеджеров компоновки	743
<b>Глава 23. Работа с изображениями</b>	<b>744</b>
Форматы графических файлов	745
Создание, загрузка и просмотр изображений	745
Создание объекта изображения	745
Загрузка изображения	746
Просмотр изображения	746
Интерфейс <i>ImageObserver</i>	748
Пример с <i>ImageObserver</i>	750
Двойная буферизация	752
Класс <i>MediaTracker</i>	755
Интерфейс <i>ImageProducer</i>	759
Производитель изображений <i>MemoryImageSource</i>	759
Интерфейс <i>ImageConsumer</i>	761
Класс <i>PixelGrabber</i>	761
Класс <i>ImageFilter</i>	764
Фильтр <i>CropImageFilter</i>	765
Фильтр <i>RGBImageFilter</i>	767
Анимация ячеек	779
Дополнительные классы изображений Java 2	782
<b>Глава 24. Дополнительные пакеты</b>	<b>783</b>
Пакеты ядра Java API	783
Отражение	786
Вызов удаленных методов (RMI)	791
Простое RMI-приложение клиент-сервер	791
Текстовое форматирование	796
Класс <i>DateFormat</i>	796

Класс <i>SimpleDateFormat</i>	798
ЧАСТЬ III. РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ	801
<b>Глава 25. Компоненты Java Beans</b>	<b>803</b>
Что такое Java Bean-компонент?	804
Преимущества технологии Java Beans	804
Инструментарий построения приложений	805
Комплект разработчика Bean-компонентов	806
Установка BDK	806
Запуск BDK	806
Использование BDK	807
JAR-файлы	809
Файлы описания	810
Утилита jar	810
Интроспекция	812
Проектные шаблоны для свойств	812
Проектные шаблоны для событий	814
Методы	815
Разработка простого Bean-компонента	815
Создание нового Bean-компонента	816
Использование связанных свойств	819
Алгоритм	820
Использование интерфейса <i>BeanInfo</i>	821
Ограниченные свойства	823
Сохраняемость	823
Конфигураторы	824
Java Beans API	824
Будущее Bean-технологии	827
<b>Глава 26. Система Swing</b>	<b>829</b>
Класс <i>JApplet</i>	830
Значки и метки	831
Текстовые поля	834
Кнопки	834
Класс <i>JButton</i>	834
Флажки	836
Переключатели	838
Поля со списком	840
Панели со вкладками	842
Панели прокрутки	844
Деревья	846
Таблицы	850
Другие возможности и будущее Swing-технологии	852
<b>Глава 27. Сервлеты</b>	<b>854</b>
Предпосылки	854

Жизненный цикл сервлета	855
Java Servlet Development Kit	856
Простой сервlet	857
Создание и компиляция исходного кода сервлета	857
Запуск утилиты <i>servletrunner</i>	858
Запуск Web-браузера и запрос сервлета	858
Servlet API	858
Пакет <i>javax.servlet</i>	859
Интерфейс <i>Servlet</i>	860
Интерфейс <i>ServletConfig</i>	861
Интерфейс <i>ServletContext</i>	861
Интерфейс <i>ServletRequest</i>	862
Интерфейс <i>ServletResponse</i>	863
Интерфейс <i>SingleThreadModel</i>	864
Класс <i>GenericServlet</i>	864
Класс <i>ServletInputStream</i>	864
Класс <i>ServletOutputStream</i>	865
Класс <i>ServletException</i>	865
Класс <i>UnavailableException</i>	865
Чтение параметров сервлета	865
Чтение параметров инициализации	867
Пакет <i>javax.servlet.http</i>	869
Интерфейс <i>HttpServletRequest</i>	870
Интерфейс <i>HttpServletResponse</i>	871
Интерфейс <i>HttpSession</i>	873
Интерфейс <i>HttpSessionBindingListener</i>	874
Интерфейс <i>HttpSessionContext</i>	874
Класс <i>Cookie</i>	874
Класс <i>HttpServlet</i>	876
Класс <i>HttpSessionBindingEvent</i>	877
Класс <i>HttpUtils</i>	878
Обработка запросов и ответов HTTP	878
Обработка GET-запросов HTTP	878
Обработка POST-запросов HTTP	880
Использование cookie-данных	881
Прослеживание сеанса	884
Проблемы защиты	885
Исследование сервлетов	886
<b>Глава 28. Миграция из C++ в Java</b>	<b>887</b>
Различия между C++ и Java	887
Что Java исключил из C++	887
Новые свойства, добавленные в Java	889
Отличающиеся свойства	890

Исключение указателей в C++	891
Преобразование параметров типа указателя	89?
Преобразование указателей, работающих на массивах	894
Ссылочные параметры C++ в сравнении со ссылочными параметрами Java	896
Преобразование абстрактных классов C++ в Java-интерфейсы	899
Преобразование умалчиваемых аргументов	903
Преобразование иерархий множественного наследования C++	905
Деструкторы в сравнении с методом <i>finalize()</i>	907
ЧАСТЬ IV. ПРИМЕНЕНИЕ JAVA	911
<b>Глава 29. Апплет DynamicBillboard</b>	<b>913</b>
Тег <i>&lt;Applet&gt;</i>	913
Обзор исходного кода	915
<i>DynamicBillboard.java</i>	915
<i>BillData.java</i>	923
<i>BillTransition.java</i>	925
<i>ColumnTransition.java</i>	927
<i>FadeTransition.java</i>	930
<i>SmashTransition.java</i>	933
<i>TearTransition.java</i>	937
<i>UnrollTransition.java</i>	941
Динамический код	945
<b>Глава 30. ImageMenu: Web-меню изображений</b>	<b>947</b>
Исходное изображение	949
Тег <i>&lt;applet&gt;</i>	950
Методы	951
Метод <i>init()</i>	951
Метод <i>update()</i>	951
Метод <i>lateInit()</i>	951
Метод <i>paint()</i>	951
Метод <i>mouseExited()</i>	952
Метод <i>mouseDragged()</i>	952
Метод <i>mouseMoved()</i>	952
Метод <i>mouseReleased()</i>	953
Код	953
Резюме	955
<b>Глава 31. Апплет Lavatron: дисплей для спортивной арены</b>	<b>956</b>
Как работает <i>Lavatron</i>	956
Исходный код	958
Тег <i>&lt;applet&gt;</i>	958
<i>Lavatron.java</i>	959
Класс <i>IntHash</i>	963
Апплет <i>HotLava</i>	966

<b>Глава 32. Scrabblet: многопользовательская игра в слова</b>	<b>967</b>
Вопросы сетевой безопасности	967
Игра	968
Подсчет очков	971
Исходный код	975
Ter < <i>applet</i> >	975
<i>Scrabblet.java</i>	976
<i>IntroCanvas.java</i>	987
<i>Board.java</i>	989
<i>Bag.java</i>	1005
<i>Letter.java</i>	1007
<i>ServerConnection.java</i>	1012
Код сервера	1017
<i>Server.java</i>	1017
<i>ClientConnection.java</i>	1020
Совершенствование <i>Scrabblet</i>	1025
<b>Приложение. Использование документационных комментариев Java</b>	<b>1027</b>
Теги <i>javadoc</i>	1027
Тег <i>@author</i>	1028
Тег <i>@deprecated</i>	1029
Тег <i>@exception</i>	1029
Тег <i>{@link}</i>	1029
Тег <i>@param</i>	1029
Тег <i>@return</i>	1029
Тег <i>@see</i>	1030
Тег <i>serial</i>	1030
Тег <i>@serialData</i>	1030
Тег <i>@serialField</i>	1030
Тег <i>@since</i>	1031
Тег <i>@throws</i>	1031
Тег <i>@version</i>	1031
Общая форма документационного комментария	1031
Что выводит <i>javadoc</i>	1032
Пример использования документационных комментариев	1032
Предметный указатель	1034

## Предметный указатель

A	appletviewer (программа просмотра апплетов) 588
Accessibility API 853	
Adapter classes 612, 636	
API (Applicatin Programming Interface) 559, 783	AWT (Abstract Window Toolkit) 301, 316,588
API ядро 783, 301	B
	BDK (Bean Developer Kit) 804, 806
	Bean-компонент 804

big-endian (формат коротких целых чисел) 55

C

CGI (Common Gateway Interface) 855

Class members (члены класса) 134

collections framework (строктура коллекций) 413

collection-view (представление в виде коллекции) 415, 438

Convenience routine (подпрограмма удобств) 570

Cookie-элементы 874

CORBA (Common Object Request Broker Architecture) 27

CPU (Central Processor Unit) 20

D

Daemon thread ("демонический" поток) 401

Delegation event model (модель делегирования событий) 613

Default access level 173

constructor 150

package 220

DLL (Dynamic Link Library) 324

DNS (Domain Naming Service), служба доменных имен 552

Drag-and-Drop API 853

E

Endianness 55

escape-последовательности символов, таблица 62

Event loop with polling 264

F

Factory methods (производственные методы) 554

Finalization, завершение работы с объектом 154

firewall (компьютер межсетевой защиты) 967

G

GMT (Greenwich Mean Time) 490

GUI (Graphical User Interface — графический интерфейс пользователя) 25, 642

H

HSB (Hue-Saturation-Brightness — цветовая модель "тон-насыщенность-яркость") 665

HTML (HyperText Markup Language) теги

<applet> 28, 317, 603, 604

<img> 604

<param> 604

файл 317, 589

HTTP (HyperText Transport Protocol) 854

I

IDL (Interface Definition Language) 27

image-based menu 947

instance variables (переменные экземпляра) 134

IP (Internet Protocol) 549

ISO (International Standardization Organization) 493

J

JAR (Java ARchive) 26, 809

Java

Internet 16, 17

JAR-файл (архивный файл Java) 809

апплет Java 18, 315

байт-код (bytecode) 19

библиотеки

Java 2D 26

доступности (Accessibility library) 26

ввод/вывод

консольный (текстовый) 301

оконный (графический) 301

версия 1.0 (исходная) 24

версия 1.1 24

список добавлений 25

версия 2 24

список добавлений 26

Всемирная Паутина (WWW) 16  
единица компиляции (compilation unit) 38  
исключенные свойства C++ 887  
исполнительная (run-time) система 20  
история создания 15  
коллекции 26  
межсетевая защита 19  
мобильность (переносимость)  
программ 19  
независимость от платформ 17  
новые свойства (по сравнению с C++) 889  
приложение Java 18, 315  
родной (native) код 20  
свойства, различающиеся с C++ 890  
связь с языками С и C++ 11  
список базовых терминов 21  
строгая типизация 52  
файл(ы)  
исходный (.java) 37  
откомпилированный (.class) 38, 39  
цели разработки 11  
язык свободной формы 48  
языки-предшественники 11, 13  
Java Beans 25, 787 API 812  
Bean-компоненты  
ActiveX 828  
булевы свойства 813  
индексированные свойства 813  
конфигуратор (customizer) 824  
ограниченные (constrained)  
свойства 823  
простые свойства 812  
связанные свойства (bound properties) 819  
сохраняемость (persistence) 823  
инструменты  
Bridge for ActiveX (мост для ActiveX) 828

Java Beans Migration Assistant for ActiveX 828  
интерфейс BeanInfo 821  
getEventSetDescriptors() 821  
getMethodDescriptors() 821  
getPropertyDescriptorsQ 821  
интроспекция 812  
класс SimpleBeanInfo 821  
проектные шаблоны (design patterns) 812  
Java I/O system 302, 501  
Java Security Manager 855  
Java 2D API 853  
java.awt.event (пакет) 612, 637  
java.lang (пакет)  
интерфейсы  
Cloneable 389  
Comparable 410  
Runnable 399  
классы  
Class 391-393, 787-789  
ClassLoader 394, 395  
Compiler 399  
Component 589, 632, 633  
Container 589  
InheritableThreadLocal 407  
Math 65, 395  
Modifier 789  
Object 217, 218, 388, 389  
Panel 589  
Package 408, 409  
Process 379  
Random 240, 398  
Runtime 379—381  
Runtime Permission 409  
SecurityManager 410  
System 384-386  
Thread 399-402  
ThreadGroup 402-404  
Throwable 409  
Void 378  
оболочки  
Boolean 378

Byte 366, 367  
Character 374-376  
Double 360, 362-364  
Float 360-362  
Integer 369-371  
Long 371-373  
Short 367-369

java.lang.reflect (пакет ядра API)  
    Conctructor 787  
    Field 787  
    Method 787

java.lang.rmi (пакет ядра API)  
    Naming 792, 793

java.util

- интерфейсы
  - Cloneable 479
  - Collection 416
  - Comparable 482
  - Comparator 445
  - Enumeration 477
  - Iterator 431
  - List 419
  - ListIterator 431
  - Map 437
  - Map.Entry 439, 440
  - Observer 496, 497
  - Set 421
  - SortedMap 439
  - Sorted Set 421
- классы
  - AbstractMap 441
  - Arrays 452
  - BitSet 479
  - Calendar 484
  - Collections 448
  - Date 482
  - Date Format 796—798
  - Event Listener 413
  - EventObject 413
  - GregorianCalendar 488
  - HashMap 441
  - ListResourceBundle 413
  - Locale 492, 493, 796
- методы (для получения Properties-объектов среды)  
    System.getProperties() 470
- методы (для работы с ArrayList)  
    ensureCapacity() 425  
    toArray() 426  
    toString() 425  
    trimToSize() 425
- методы (для работы с Arrays)  
    AsListQ 453  
    binarySearch() 453  
    equals() 453  
    fill() 454  
    sort() 454
- методы (для работы с LinkedList)  
    addFirst() 427  
    addLast() 427  
    removeFirst() 427  
    removeLast() 427
- методы (для работы с коллекциями)  
    add() 418

addAll() 418  
clear() 418  
contains() 418  
containsAll() 418  
equals() 419  
isEmpty() 418  
iterator() 419, 432  
remove() 418  
removeAll() 418  
retainAll() 418  
size() 418  
subList() 421  
to Array () 418

методы (для работы с сортированными наборами)  
first() 421  
headSet() 421  
last() 421  
subSet() 421  
tailSet() 421

методы (для работы со списками)  
get() 419  
indexOf() 419  
lastIndexOf() 419  
set() 419

методы (интерфейса Comparator)  
compare() 445  
equals() 445

методы (интерфейса Map)  
entrySet() 439  
keySet() 438  
values() 438

методы (интерфейса SoitedMap)  
firstKey() 439  
headMap() 439  
lastKey() 439  
subMap() 439

методы (интерфейса Map)  
entrySet() 438

методы (класса Collections)  
synchronized List() 451  
synchronizedSet() 451

методы (наследованного интерфейса Enumeration)  
hasMoreElements() 457  
nextElement() 457

JDBC (Java Database Connectivity) 25, 886

JDBC API 886

JDK (Java Developer's Kit) 27, 37, 588, 806

appletviewer (программа просмотра аплетов) 317

утилиты  
jar (генерация JAR-файлов) 810  
Java (запуск приложений, интерпретатор) 27, 38  
javac (компилятор) 27, 38  
javadoc (создание документации) 27, 1027  
javah.exe (построение .h файлов) 324

JFC (Java Foundation Class) 853

JIT (Just In Time) 20, 27

JNI (Java Native Interface) 25, 323

JRE (Java Runtime Environment) 27

JSDK (Java Servlet Development Kit) 854, 856

JVM (Java Virtual Machine) 19, 27

L

Listener (блок прослушивания событий) 613

M

Manifest file 810

MIME (Multipurpose Internet Mail Extensions) 854

Multicasting 614

Mutex (взаимоисключающая блокировка) 281

N

Native-методы 885

NCSA 744

O

OOP (Object-Oriented Programming) 14, 29

ORB (Object Request Broker) 27  
P  
Pluggable look-and-feel 853  
Preemptive multitasking  
(упреждающая  
многозадачность) 265  
Protection domain (домен защиты) 393  
Proxy (сетевой посредник) 968  
R  
RGB (Red-Green-Blue — цветовая  
модель "красный-зелёный-  
синий") 666  
RMI 24, 25, 783  
    rmiregistry 795  
    nm-time-состояние 391  
    динамическая загрузка классов 795  
    интерфейс Remote 792  
    компилятор RMI 794  
    объекты  
        заглушка (stub) 793  
        скелет (skelton) 794  
    простое приложение клиент-  
    сервер 791  
    сериализация 539, 794

S  
Scope (область видимости  
идентификаторов) 65  
Security manager 384  
Servlet (сервлет) 854  
Servlet API 855, 858  
set-view (представление в виде  
набора) 441, 469  
skelton (скелет), RMI-объект 794  
subclasser responsibility method  
(метод, находящийся на  
ответственности подкласса) 213  
Swing 26  
    API 829  
     Swing-компоненты 846, 850  
    интерфейсы Icon 831  
        MutableTreeNode 847  
    SwingConstants 831  
    TreeExpansionListener 848

TreeNode 847  
ScrollPaneConstants 844  
классы  
    AbstractButton 834  
    Container, метод Add() 831  
    DefaultMutableTreeNode 847  
    Image Icon 831  
    JApplet 830  
     JButton 834  
    JCheckBox 836  
    JComboBox 840  
    JComponent 831  
    JLabel 831  
    JRadioButton 838  
     JScrollPane 844  
    JTabbedPane 842  
    JTable 850  
    JTextComponent 833  
    JTextField 833  
    JTree 846  
    TreeExpansionEvent 848  
    Tree Path 847

компоненты  
    pluf-свойство (pluggable look-and-feel) 852  
    подсказки кнопочных команд (tooltips) 852  
    прогресс-полоски (progress bars) 852

пакеты  
    javax.swing 830  
    javax.swing.event 848  
    javax.swing.tree 830

панели  
    корневая (root pane) 830  
    прозрачная (glass pane) 830  
    прокрутки (scroll pane) 844  
    со вкладками (tabbed pane) 842  
    содержания (content pane) 830  
таблица классов 829  
System.in.read() 119  
T

TCP (Transmission Control Protocol) 549  
Thread-safe (поточно-безопасный) 451  
Type wrappers (оболочки простых типов) 359  
**U**  
UDP (User Datagram Protocol) 549  
unicasting, 614  
Unicode 25, 58, 302  
URI (User Resource Identifier) 870  
URL (Uniform Resource Locator) 559, 856  
UTC (Coordinated Universal Time) 490  
**W**  
Web 28, 559  
whitespace (пробельный символ) 48  
**X**  
XOR-режим рисования 668  
**A**  
Абстрактный класс 899  
Абстрактный метод 900  
Автоматическое преобразование типов 68, 160  
расширяющее (widening conversion) 68  
сужающее (narrowing conversion) 69  
Алгоритмы (коллекций) 414  
Алгоритмы синхронизации (коллекций) 451  
Апплет(ы)  
    HTML-тег <applet> 589  
    базовые методы работы с апплетами 594  
    destroy() 595  
    init() 594  
    paint() 595  
    start() 594  
    stop() 595  
    update() 595  
    интерфейсы  
        AppletContext 609  
AppletStub 611  
AudioClip 611  
класс Applet  
    методы 608—609  
консольный вывод 611  
ненадежные 885  
пересылка параметров в 605  
простые методы отображения 596  
    drawstring() 596  
    getBackground() 597  
    getForeground() 597  
    getGraphics() 599  
    repaint() 598  
    setBackground() 596  
    setForeground() 596  
    цветовые константы 596  
с бегущим заголовком 599, 606  
скелетная схема 593  
Аргументы  
    командной строки 188  
    метода 142, 146  
**Б**  
Библиотеки классов 51  
Блоки кода (кодовые блоки) 46  
Браузер Web 18  
**В**  
Ввод/вывод  
    байтовые потоки 502  
        буферизованные 518  
    Externalizable (интерфейс) 540  
    FileFilter (интерфейс) 509  
    FilenameFilter (интерфейс) 507  
    ObjectInput (интерфейс) 542  
    ObjectOutput (интерфейс) 540  
    Serializable (интерфейс) 540  
    ArrayOutputStream (класс) 516  
    BufferedInputStream (класс) 518  
    BufferedOutputStream (класс) 518, 520  
    ByteArrayInputStream (класс) 515  
    File (класс) 503  
    FileInputStream (класс) 312, 511  
     FileOutputStream (класс) 312, 513

FilterInputStream (класс) 518  
FilterOutputStream (класс) 518  
InputStream (класс) 510  
ObjectInputStream (класс) 543  
ObjectInputStream.GetField (класс)  
    502  
ObjectOutput.Stream.GetField  
    (класс) 502  
ObjectOutputStream (класс) 541  
OnlyExt (класс) 507  
OutputStream (класс) 510  
OutputStream (класс) 303  
PrintStream (класс) 309  
PrintStream (класс) 523  
PushbackInputStream (класс) 520  
PushbackInputStream (класс) 518  
RandomAccessFile (класс) 524  
SequenceInputStream (класс) 522  
System (класс) 303  
исключения  
    FileNotFoundException 312, 527  
    IOException 307, 513, 528  
    SecurityException 513, 528  
поточный  
    байтовый поток 302  
    буферизованный 306  
    поток (stream) 302, 501  
    поток ввода 302  
    поток вывода 302  
    символьный поток 302, 525  
сериализация объектов 502  
символьные потоки 502  
    BufferedReader (класс) 307, 308  
    BufferedReader (класс) 531  
    BufferWriter (класс) 533  
    CharArrayReader (класс) 529  
    CharArrayWriter (класс) 530  
    FileReader (класс) 527  
    FileWriter (класс) 528  
    InputStremReader (класс) 306  
    PrintWriter (класс) 534  
    PushbackReader (класс) 533  
    Reader (класс) 525  
StreamTokenizer (класс) 537  
Writer (класс) 303, 526  
таблица классов символьного  
ввода/вывода 303  
Вызов удаленных методов (RMI) 791  
Вызовы методов  
    встроенные (inline) 217  
Выражения с авторасширением  
    типов 70  
    правила 71  
Г  
Графический контекст 658  
Д  
"Демонический" (demon) процесс  
    1017  
Десериализация 794, 539  
Динамическая диспетчеризация  
    методов 208  
Домен защиты 393  
Е  
Емкость (размер) массива 424  
Емкость загрузки хэш-набора 429  
Естественное упорядочение объектов  
    410  
3  
Завершение 154  
И  
Идентификаторы Java 48  
Интернационализация 492  
Интерфейс (interface) 899  
    AWT, LayoutManager 712  
    java.io.Serializable 824  
    альтернатива множественного  
        наследования 231  
    определение 231  
    реализация 232  
Исключения 243  
    ArrayIndexOutOfBoundsException  
        455  
    ArrayStoreException 417, 418  
    ClassCastException 362, 377, 416—  
        455  
    ClassFormatError 394

ClassNotFoundException 392, 395  
CloneNotSupportedException 389  
EmptyStackException 463  
IllegalAccessException 393  
IllegalArgumentException 454, 455  
IllegalStateException 432  
InstantiationException 393  
InterruptedException 389, 401  
NoSuchElementException 421—439  
NullPointerException 421, 437, 439  
NumberFormatException 362  
SecurityException 392  
SecurityException 379, 384  
UnsupportedOperationException  
    416, 419, 451  
контролируемые 258  
    ClassNotFoundException 259  
    CloneNotSupportedException 259  
    IllegalAccessException 259  
    InstantiationException 259  
    InterruptedException 259  
    NoSuchFieldException 259  
    NoSuchMethodException 259  
    таблица 258  
неконтролируемые 258  
    Arithmetiс Exception 258  
    ArrayIndexOutOfBoundsException  
        258  
        Array Store Exception 259  
        ClassCastException 259  
        IllegalArgumentExcepcion 259  
        IllegalMonitorStateException 259  
        IllegalStateException 259  
        IllegalThreadStateException 259  
        IndexOutOfBoundsException 259  
        NegativeArraySizeException 259  
        NullPointerException 259  
        NumberFormatException 259  
        SecurityException 259  
        StringIndexOutOfBoundsException  
            259  
    UnsupportedOperationException  
        259

таблица 258  
обработчик по умолчанию 245  
объект(ы) 243  
собственные методы 260  
    String getLocalizedMessage() 260  
    String getMessage() 260  
    String toString() 260  
    Throwable fillInStackTrace() 260  
    void printStackTrace() 260  
    void printStackTrace(PrintStream  
        stream) 260  
    void  
    printStackTrace(PrintWriterstrea  
        m) 260

средства работы  
    catch-блок 246  
    finally-блок 256  
    throw-оператор 253  
    throws-методы 255  
    try-блок 246  
    вложенные try-блоки 251  
    множественные catch-блоки 249  
    общий формат 244

типы (классы)  
    Error 245  
    Exception  
        IllegalAccessExcepcion 255  
        Exception 244  
        ArithmetiсException 245  
        RuntimeException 244  
        Runtime Exception  
        ArrayIndexOutOfBoundsException  
            249  
        NullPointerException 254  
        Throwable 244

Итератор коллекции 414, 419, 431  
К  
Карта отображений (map) 415, 436  
Классы 133  
    Applet 316, 588  
        AppletContext (интерфейс) 588  
        AppletStub (интерфейс) 588  
        AudioClip (интерфейс) 588

таблица методов 589  
AppletContext 588  
AppletStub 588  
AudioClip 588  
таблица методов 589  
AWT 643  
    AWTEvent 736  
    Button 690  
    Canvas (окно) 648  
    Checkbox 693  
    CheckboxGroup 696  
    CheckboxMenuItem 724, 725  
    Choice 697, 698  
    Color 665, 666  
    Component 636—673, 736  
    Container 646, 687, 715-720  
    Dialog (модальный,  
    немодальный) 729  
    Dimension 648, 664  
    Dimention 787  
    FileDialog 734, 735  
    Font 670  
    FontMetrics 676—678  
    Frame 647—649  
    Graphics 660—668  
    GraphicsEnvironment 671, 672  
    ItemEvent 725  
    Label 688  
    List 700, 701  
    Menu 723  
    MenuBar 723  
    MenuItem 723  
    Panel (окно) 647  
    PopupMenu 729  
    Scrillbar 704, 705  
    TextArea 710  
    TextField 707, 708  
    Window 647, 730  
    работа с фреймовыми окнами  
    648  
    режим рисования 668  
    таблица 643  
Component 589  
Container 589  
java.util 796  
java.util.DateFormat 796-798  
java.util.SimpleDateFormat 798  
LANG  
    Class 787, 789  
    Modifier 789  
Math 65  
Object 217, 218  
Panel 589  
public 224  
Random 240  
REFLECT  
    Conctructor 787  
    Field 787  
    Method 787  
RMI, Naming 792, 793  
String 80, 185, 186, 331  
StringBuffer 85, 331  
System 51, 305  
абстрактный 213, 235  
вложенные 181  
    нестатические 182  
    статические 182  
внутренние 25, 182, 638  
    анонимные 185, 640  
иерархия 33  
    подклассы 33  
    суперклассы 33  
интерфейсы 230  
как новый тип данных 133  
как шаблон для объектов 133  
наследованные 457  
определение 32  
подкласс  
определение 189  
просмотра (peer classes) 185  
реализация интерфейсов 232  
сетевые 548  
суперкласс  
    определение 189  
члены 32  
    методы 32

переменные (переменные экземпляра) 32

Клон (clone) 389

Ключевые слова 50

- abstract 213, 235
- class 39, 134
- extends 189, 241
- final 179, 216
- interface 219, 230
- native 323
- static 40, 177
- strictfp 322
- super 177, 196
- this 152
- transient 319, 824
- volatile 319

для работы с исключениями

- catch 244
- finally 256
- throw 244
- throws 244, 255
- try 244

спецификаторы доступа

- private 40, 173, 224
- protected 173, 224
- public 40, 173, 224

Кодовые блоки (блоки кода) 46

Коллекция (объектов) 412

- изменяемая (modifiable) 416
- неизменяемая (unmodifiable) 416
- несинхронизированная 451
- синхронизированная (поточно-безопасная) 451

Комментарий 39

- документационный 49
- использование 1027
- многострочный 39
- однострочный 40

Компаратор (comparator) 444

Компоненты Java Beans 787

Константы 49

Конструктор (constructor) 139, 148

- super() 196, 204

по умолчанию 139, 150

Контейнер C++ 415

Коэффициент заполнения хеш-набора (fillRatio) 429

Л

Лексема (token) 477, 537

Лексический анализатор (сканер) 477

Литералы 60

Локализация 493

М

Массив(ы) ,72

- многомерные 75
- инициализация 74, 78
- одномерные 72
- альтернативный формат 80
- общий формат 72

Менеджер безопасности (security manager) 384, 410

Менеджеры компоновки (layout manager) 712

- BorderLayout 714
- CardLayout 719, 720
- FlowLayout 713
- GridLayout 718

использование вставок 716

Меню изображений (image-based menu) 947

Методы 134

- clone() 218
- equals() 218
- finalize() 154, 218
- getClass() 218
- hashCode() 218
- main() 65
- notify() 218
- notifyAll() 218
- toString() 218
- wait() 218

абстрактный 213

динамическая диспетчеризация 208

заглушка 591

как члены класса 134

на ответственности подкласса 213  
перегруженные 158  
 переопределенные (overrided) 206  
 производственные (factory) 554,  
 925  
 `getAllByName()` 554  
 `getAllByName()` 554  
 `getLocalHost()` 554  
 рекурсивные 170  
 удобств (convenience method) 998  
 `makeMimeHeader()` 570  
 `toBytes()` 570  
 `writeTo()` 517

Многозадачность блокировка задач  
 291  
 основанная на потоках 263  
 поток (как единица  
 диспетчеризации) 263  
 основанная на процессах 263  
 программа (как единица  
 диспетчеризации) 263  
 упреждающая 265

Многопоточность 889

Модификатор `abstract` 213

`transient` 319

`volatile` 319

Н

Начальное числр (псевдослучайного  
 генератора) 494

О

Облегченные (Swing) компоненты  
 829

Оболочки простых типов (`type  
 wrappers`) 359

Обработка событий

блок(и) прослушивания 613, 625  
 метод отказа от регистрации,  
 формат 614  
 методы регистрации 614, 632

интерфейсы прослушивания 626  
 `Action Listener` 627, 690  
 `AdjustmentListener` 627  
 `ComponentListener` 627

ContainerListener 627  
 FocusListener 627  
 ItemListener 628, 694, 698  
 KeyListener 628  
 MouseListener 628  
 MouseMotionListener 628  
 TextListener  
 WindowListener 629  
 таблица 626

классы-адаптеры 612, 636  
 внутренние, анонимные 612  
 таблица 637

модель делегирования событий  
 613, 625

модель расширения AWT-  
 компонентов 736

таблица методов 737

мультивещание (multicasting)  
 событий 614

обработчики

`keyPressed()` 633  
 `keyReleased()` 633  
 `keyTyped()` 633

пакет поддержки  
 `java.awt.event` 612, 637

унивещание (unicasting) событий  
 614

Обработка строк 331  
 извлечение символов 338

классы

`Object` 337  
 `String` 331-350  
 `StringBuffer` 331-357

конкатенация строк 335  
 с другими типами данных 336

поиск строк 344

строковый литерал 335

Объект(ы) 133  
 возврат методами 169  
 как экземпляр класса 32, 133  
 передача методу по ссылке 168  
 состав 32

# ООП (Объектно-Ориентированное Программирование)

базовые принципы

наследование 189

## Оператор(ы)

import 228

null (пустой) оператор 116

package 220

управления 43

выбора 106–109

перехода 126—132

повторения (циклов) 115, 117, 120

## Операции

() — круглые скобки 104

. — точка (dot) 104

[ ] — квадратные скобки 104

new — распределение памяти для объекта 138

арифметические

таблица 82

беззнакового сдвига 96

дополнение до двух 89

дополнение до единицы 89

логические

instanceof 82

таблица 100

отношений

таблица 98

поразрядные

таблица 88

поразрядные логические

таблица 90

постфиксная форма 86

префиксная форма 87

присваивания (назначения) 102

расширение знака 95

старшинство (таблица) 104

укороченные (shortCircuit) 102

Отражение (reflection) 411, 783, 786

П

Пакет(ы)

AWT642

менеджер компоновки (layout manager) 686, 712

работа с графикой 658

работа с меню, методы 724

работа с цветом 665

работа со шрифтами 670

строка меню (menu bar) 686

элементы управления (controls) 686

java.applet 301, 316, 588

java.awt 588

java.awt.dnd 853

java.awt.event 612, 636

java.beans 824

таблица интерфейсов 824

таблица классов 824

java.io 301, 303, 501

список интерфейсов 502

список классов 501

java.lang 258, 305, 331, 358

Number (класс) 359

список интерфейсов 358

список классов 358

java.lang.ref 410

java.lang.reflect 411, 783

Member (интерфейс) 787

таблица методов 787

java.net (программирование для Internet) 548

java.rmi 783

java.text 783, 796

java.util 412, 477

таблица интерфейсов 413

таблица классов 412

java.util.jar 500

java.util.zip 500

javax.servlet 857, 858

javax.servlet.http 869

иерархия 221

именованные 228

как группа классов 173

определение 220

по умолчанию (default package) 220

типы меню

- иерархическое 723
- плавающее 723

ядро API (таблица) 783

Параметр(ы) метода 40, 141, 146

Параметры указателя (pointer parameters) 892

Перегрузка (overloading) методов 158

Передача аргумента

- по значению 167
- по ссылке 167

Переменные

- время жизни 65
- выражение инициализации 64
- динамическая инициализация 64
- инициализация 64
- массива (array variable) 72
- область видимости
  - идентификатора 65
  - определенная классом (class scope) 65
  - определенная методом (method scope) 65
- окружения
  - CLASSPATH 221
- определение 41, 63
- ссылочные (объекта) 140
- управления циклом 120
- формат объявления 63
- экземпляра
  - length 180
- определение 134

Переопределение (overriding)

- методов 593

Песочница (sandbox) 885

Полиморфизм 899

- перегруженных функций 158

Потоки 263

- асинхронные 266
- выполнения 267
- главный 268

группа 269

не синхронизированные

состояние состязаний (гонок) 284

приоритеты 265

синхронизация 281

неявный монитор 286

синхронные 266

состояния 265

текущий 268

Поточное программирование

- межпоточные связи
- использование 287
- многопоточная многозадачность в Java 264
- многопоточное
  - Runnable (интерфейс) 267
  - synchronized (ключевое слово) 282
  - Thread (класс), таблица методов 267, 279
  - многопоточность 263
  - монитор (семафор) 266, 281
  - синхронизированные методы 284
  - синхронизированные операторы, блоки 285
  - правила переключения контекста 265
  - работа с приоритетами 278
  - сериализация (преобразование в последовательную форму) 284
- однопоточное
  - цикл событий с опросом 264

Представление в виде коллекции (collection-view) 415, 438

Представление в виде набора (set-view) 441, 469

Преобразование типов усечение (truncation) 69

Приведение (явное преобразование) типов (cast) формат 69

Пробельные символы (whitespace) space, tab, newline 48

## Программа

- как модель, ориентированная на процесс 30
- как управляемый данными доступ к коду 30

## Программирование

- компонентное 803
- объектно-ориентированное (ООП)
  - 14, 29, 30
  - абстракция 30
  - основные механизмы (инкапсуляция, наследование, полиморфизм) 31, 33, 34
  - сущность 31

### парадигмы

- объектно-ориентированная 30
- ориентированная на процессы 30

### языки

- процедурные 30

## Прототип функции 899

## Процесс (выполняющаяся программа) 379

## Псевдослучайные числа 493

## Р

## Работа в сети Internet 548

### DNS (Domain Naming Service)

### MIME

- заголовок 564
- стандарт 564
- строка User-Agent 573
- типы 564

### адрес

- IP 549, 552
- URL 559
- групповой (multicast) 555
- доменный адрес 552
- определение 552

### браузер

- Web 550
- действия
- определенение 584

### интерфейсы 553

## классы 553

- DatagramPacket 584
- DatagramSocket 584
- InetAddress 553—555
- ServerSocket 556, 562
- Socket 556, 557
- URL 559
- URLConnection 561
- клиент 549
- клиент-сервер 549
- масштабирование 554
- межсетевая защита
  - компьютер для (firewall) 967
- пакеты данных 549
- порт 550
- протокол(ы)
  - file 559
  - ftp 559
  - gopher 559
  - HTTP (HyperText Transfer Protocol) 550, 559
  - IP (Internet Protocol) 549
  - TCP (Transmission Control Protocol) 549, 553
  - TCP/IP 549
  - UDP (User Datagram Protocol) 549, 553
  - whois 559
  - WWW (Web-протоколы) 559
- дейтаграммные 584
- номера портов для разных протоколов 550
- сервер(ы) 549
  - DNS 556
  - InterNIC 556
  - proxy 551
  - Web 549, 550
  - вычислительные 549
  - дисковые 549
  - каптированные 555
  - кэширующий proxy HTTP 551, 563
  - однонаправленные (unieast) 792

печати 549  
реплицированные (replicated) 792  
сетевой посредник (proxy) 968  
сокет(ы)  
    Berkeley 549  
    TCP/IP (для клиентов и серверов) 556  
парадигма 548  
сетевой 549  
хост-компьютер 553, 554, 968

Работа с изображениями  
    Web-дизайн 744  
    двойная буферизация 753  
изображение (как графический объект) 744  
интерфейсы  
    ImageObserver 748  
    ImageProducer 746, 759

классы  
    Applet 746  
    Component 765  
    FilteredImageSource 764  
    Graphics 746  
    Image 744, 745  
    ImageFilter 764  
    ImageFilter.AreaAveragingScaleFilter 764  
    ImageFilter.CropImageFilter 764, 765  
    ImageFilter.RePLICATEScaleFilter 764  
    ImageFilter.RGBImageFilter 764, 767  
    MediaTracker 756  
    MemoryImageSource 759  
    PixelGrabber 762  
    анимация ячеек 779  
    загрузка объекта изображения 746  
    наблюдатель изображения (image observer) 747

создание объекта изображения 745  
пакеты  
    java.awt 744  
    Java.awt.image 744  
    javax.servlet 857—859  
    javax.servlet.http 858, 869

производители изображений (image producers) 759  
    FilteredImageSource 759  
    MemoryImageSource 759

форматы графических файлов  
    GIF 745  
    JPEG 745

Разделители таблица 49

Расширение (повышение) типов 55, 71

Рекурсия 170

С  
Сборка "мусора" 153  
Связывание вызовов  
    позднее 217  
    раннее 217  
Сеанс (session) 884  
Сервлет(ы)  
    HTTP-заголовок 854  
    HTTP-запрос 854  
    HTTP-запросы  
        GET 878  
        POST 878, 880  
        строка запроса (query string) 880  
HTTP-ответ 854  
MIME-типы text/html 854  
    text/plain 854  
    Servlet API 859

инструменты JSDK 856

интерфейсы  
    HttpServlet, таблица методов 876  
    HttpServletRequest, таблица методов 870  
    HttpServletResponse, таблица методов 871  
    HttpSession, таблица методов 873

HttpSessionBindingListener 874  
HttpSessionContext 874  
Servlet, таблица методов 860, 864  
ServletConfig, таблица методов  
861, 864  
ServletContext, таблица методов  
861  
Servlet Request, таблица методов  
862  
ServletResponse, таблица методов  
863

классы

- GenericServlet 855—864
- HttpServletResponse 876
- HttpSessionBindingEvent 877
- HttpUtils, таблица методов 878
- ServletException 865
- ServletInputStream 864
- ServletOutputStream 865
- ServletRequest 865
- UnavailableException 865

надежные 885  
ненадежные 885

работа с сеансами

- класс HttpSession 884
- метод getSession() 884
- метод getValue() 884
- метод putValue() 884
- метод removeValue() 884

сеансы

- создание 884

утилита servletrunner 856

сериализация 794

Сериализация (serialization) 539

Сигнатура

- типов (type signature) 205

Синтаксический анализ

- пробельные (whitespace) символы  
477
- разделители 477

Синтаксический анализ (parsing) 477

Сканер (лексический анализатор) 477

События

источники событий 614  
таблица 625

классы событий

- ActionEvent 617, 690, 701
- AdjustmentEvent 618, 705
- ComponentEvent 618, 619
- ContainerEvent 619
- EventObject (суперкласс) 615, 616
- FocusEvent 620
- InputEvent 620, 621
- ItemEvent 621, 701, 702
- KeyEvent 621, 622
- Mouse Event 622, 623
- TextEvent 624
- WindowEvent 624

иерархия 615  
таблица конструкторов и  
методов 616

определение 613

Сокет (socket)-сетевое соединение  
302

Спецификаторы доступа 173, 224

- private 173, 224
- protected 173, 224
- public 40, 173, 224

Ссылки

- интерфейсные 233
- мягкие 411
- слабые 411
- phantomные 411

Статические члены класса

- блоки 177
- методы 177
- переменные 177

Строки

- как объекты 185
- массивы строк 187

Структура коллекций (collections framework) 413

Т

Типы данных, простые 53

У

Управление доступом 173  
Уровень доступа по умолчанию 173  
Ф  
Файл  
CAB (file cabinet) 842  
JAR (архив Java) 809  
описания (manifest file) 810  
Фрейм (окно класса Frame) 648  
Функция удобств setDim 148  
Х

Хост-приложение 647  
Хэширование (рандомизация) 429  
Хэш-код 361, 429  
Ц  
Цветовые модели  
    HSB 665  
    RGB 666  
Цифровые подписи 810  
Ч  
Чистая виртуальная функция 899

# Об авторах

Патрик Ноутон (Patric Naughton) начал консультирование как программный инженер в 1982 г., оплачивая это через школу и приобретая "испытанные огнем" перспективы в РС-индустрии, так что все это началось еще с юношеских лет. После экспериментов с X Window System (MIT) он присоединился в 1988 г. к группе оконных систем Sun Microsystems. В конце 1990 г. Ноутон приступил к разработке секретного проекта под названием "Green" в SunLabs. Этот маленький проект подразумевал создание совершенно новой платформы для разработки программного обеспечения, которая решила бы многие из проблем существующих систем. Java была наиболее значительной технологией, вышедшей из проекта Green. Ноутон способствовал созданию и оценке Java-технологии от ее начала до революционного перехода в язык Internet.

Патрик Ноутон в настоящее время — исполнительный вице-президент программ для Infoseek Corporation и лидер группы, которая создает значительные разработки для GO Network. До работы с Infoseek, он был президентом и главным технологом Starwave Corporation Пола Аллена (Paul Allen), где вел развитие стратегий платформ, системного программного обеспечения, приложений и инструментальных средств, издав набор побеждавших на конкурсах online-служб, включая ESPN.com, ABCNEWS.com, Mr Showbiz, NBA.com, NFL.com и др.

Ноутон — автор *The Java Handbook* и соавтор *Java 1.1: The Complete Reference*, бестселлеров издательства Osume/McGraw-Hill. Он имеет степень бакалавра в области вычислительной техники (B.S.) от Университета Clarkson.

Герберт Шилдт (Herbert Schildt) — всемирно известный автор книг по программированию. Их было продано более двух миллионов копий по всему миру, и они были переведены на все основные иностранные языки. Герберт Шилдт — автор бестселлеров: *C++: The Complete Reference*, *C: The Complete Reference*, *Java Programmer's Reference*, *Teach Yourself C*, *Teach Yourself C++*, *C++ from the Ground Up*, *Windows 98 Programming from the Ground Up*, и *STL Programming from the Ground Up*. Он также соавтор *C/C++ Annotated Archives*.

Герберт Шилдт — президент Universal Computing Laboratories, консалтинговой фирмы по программному обеспечению в Mahomet, штат Иллинойс. Он имеет степень магистра по информатике от Университета штата Иллинойс.

## **Специальные благодарности**

Специальная благодарность — Джо О'Нейлу (Joe O'Neil) за его помощь в подготовке третьего издания этой книги. В дополнение к обработке обновлений, требуемых новыми спецификациями Java 2, Джо также обеспечил начальные черновые проекты для глав 24—27. Как всегда, его усилия очень ценные.

# Благодарности

Написание моей первой книги, *The Java Handbook*, было духовным опытом, завершающим пять лет усилий по проекту Java. Она больше похожа на повествование о Java, чем на "полный справочник". Не все из библиотечных классов Java были затронуты и не каждый метод во всех классах был перечислен. Завершаясь исторической главой "Длинное странное путешествие к Java", книга дает мое персональное мнение о том, как делается язык. Эта книга — другая. Она представляет собой сбалансированное, объективное и всестороннее представление Java.

Героизм людей, которые заставили этот язык появиться, описан недостаточно. Прессы имеет тенденцию мыслить слишком узко, чтобы делать чистую историю. Успех языка обязан не какому-то одному человеку, а успехам и неудачам группы преданных и вдохновенных личностей, включая James Gosling, Arthur van Hoff, Jonathan Payne, Chris Warth, Tim Lindholm, Frank Yellin, Sami Shaio, Patrick Chan, Kim Polese, Richard Tuck, Eugene Kuemer, Bill Joy и многих других.

Мое уважение к тому, что эта группа делает, растет с каждым прошедшим днем и каждым классом Java, который я записываю.

Internet — странное место для работы. Для первого издания этой книги я потратил шесть месяцев, работая близко с человеком, которого я никогда не встречал. Герберт Шилдт написал замечательную книгу, *C++: The Complete Reference*, об очень трудном языке. Когда стало ясно, что программисты нуждаются в комбинации, которую я предложил в *The Java Handbook* и в книге Герберта, решение было простым. Мы объединились, чтобы создать книгу, наилучшую из этих двух. Герберт понимает, как представить трудные концепции способом, который не оскорбляет опытного программиста и не оставляет новичка позади. Его выдержка для записи каждой небольшой детали, объединенная с моим пониманием того, как эти подробности возникли, сделали книгу такой, что, мы думаем, вы будете довольны. Мы модифицировали эту книгу для Java 2 так, что она остается текущим документом и продолжает являться солидным справочником в надежном формате.

Герберт выполнил большую часть трудной работы для этого (третьего) издания. Моей главной целью было включение законченных примеров превосходного программирования на Java. Чтобы избежать узко личных представлений о том, как записывать программы на Java, я также использовал некоторых из первоклассных аплетов моих друзей, и они заслуживают хорошей оценки за работу.

- Роберт Темпл (Robert Tample) — одаренный Java-программист, создающий удивительные аплеты с минимальным временем загрузки. Я увидел его код в сети и послал ему электронную почту, чтобы предложить работу, но, по случайности, он уже нашел в сети план работ и уже спланировал свое резюме. Его аплет `DynamicBillboard`, который мы рассматриваем в главе 29, полон неочевидных рабочих трюков, что возобновило мое доверие к Java в первый же раз, когда я увидел его.
- Дэвид Лавалли (David LaVallee), известный также как Бойскаут — один из четырех, кто создавал коды Java в 1991 г. Его аплеты `ImageMenu` и `Lavatron` в главах 30 и 31 — это классические проекты Бойскаута. Известный по его "безделушкам и пустякам", он всегда привносит творческий дизайн в свои аплеты.
- Дэвид Геллер (David Geller) — давний Windows-программист и автор, который имеет острый взгляд на инструментальные средства разработчика. Его проникновение и вклад в развитие среды разработки были неоценимы.
- Джоанна Кэрр (Johanna Carr) остается самым сильным непрограммистом в мире, когда она берется за программные системы и языки. Она старательно читала и комментировала каждую страницу этой книги, по крайней мере, дважды. Если Джоанна чего-то не понимает, то это, вероятно, плохо написано.

Благодарности также направляются Мэттью Нейтонсу (Matthew Naythons), который разыскивает все, что представляет для меня интерес, так что я избавлен от этого, и Кенна Мозер (Kenna Moser), которая продолжает поддерживать мои стремления с любовью, поддержкой и большой двойной порцией кофе.

ПАТРИК НОУТОН

# Предисловие

Языки программирования, парадигмы, и практика не стоят на месте слишком долго. Часто оказывается, что методы и технические приемы, которые мы применяли вчера, сегодня уже устарели. Конечно, большая скорость изменений — это также одно из обстоятельств, которые делают занятие программированием весьма привлекательным времяпрожождением. Всегда есть что-то новое на горизонте.

Возможно, никакой язык не иллюстрирует предшествующие утверждения лучше, чем Java. В последние несколько лет Java вырос из концепции в один из всемирно доминирующих машинных языков. Ко всему прочему, в течение того же короткого периода времени Java прошел через две крупные ревизии. Первая произошла, когда была выпущена версия 1.1. Изменение в младшем номере версии (от 1.0 до 1.1) не отражает значимости спецификаций этой версии. Например, Java 1.1 существенно изменил способ обработки событий, были добавлены такие свойства, как Java Beans и усовершенствованный API.

Темой этой книги является вторая крупная версия — Java 2. В Java 2 сохраняются все функциональные возможности Java 1.1 и добавляется большое количество новых и усовершенствованных старых. Например, прибавляется структура коллекций (collections framework), более мощный, чем AWT, набор классов Swing, новая поточная (threading) модель и многочисленные API-методы и классы. Фактически, было добавлено так много новых свойств, что в данной книге невозможно их полное обсуждение.

Чтобы сохранять темп вместе с Java, книга также прошла несколько версий. Первоначальная версия описывала Java 1.0. Второе издание описывало версию Java 1.1. Это, третье издание охватывает Java 2. Java развивается очень быстро: время от первого издания до третьего меньше чем два с половиной года!<sup>1</sup>

---

<sup>1</sup> Здесь приводится предыстория американских изданий. — Примеч. ред.

## Книга для всех программистов

Чтобы использовать эту книгу, не требуется никакого предыдущего опыта программирования. Если, однако, вы идете от C/C++, тогда сможете продвигаться немножко быстрее. Как убедится большинство читателей, Java по-добен по форме и духу языкам С и С++. Таким образом, их (языков) знание помогает, но не является необходимым. Даже если вы никогда не программировали прежде, то, используя эту книгу, сможете научиться программированию на Java.

## Что внутри

Книга охватывает все аспекты языка программирования Java. Часть I представляет подробный учебник языка. Он начинается с основ, включая типы данных, операторы управления и классы. Часть I обсуждает также механизм обработки особых ситуаций Java, подсистему многопоточных процессов, пакеты и интерфейсы.

Часть II рассматривает стандартную библиотеку Java. Как вы увидите, многое из мозги Java можно найти в его библиотеке. Разделы данной части обсуждают строки, ввод/вывод, работу в сети, стандартные утилиты, структуру коллекций, апплеты, элементы управления, основанные на графическом интерфейсе пользователя (GUI) и формирование изображений.

Часть III содержит обсуждение некоторых вопросов, касающихся среды разработки Java, включая краткий обзор Java Beans, Swing и сервлетов.

Часть IV представляет ряд высокопроизводительных апплетов Java, которые служат расширенными примерами различных применений Java. Заключительный апплет, называемый Scrabblet, является законченной многопользовательской сетевой игрой. Он демонстрирует, как можно решать некоторые из самых жестких проблем, включенных в Web-программирование.

## Что является новым в третьем издании

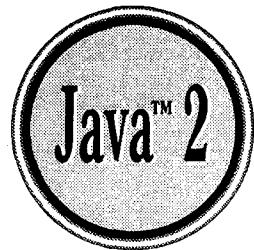
Главные различия между этим и предыдущими изданиями данной книги включают свойства, добавленные языком Java 2. К ним относятся структура коллекций, пакеты Swing и изменения способов работы с многопоточностью. Однако имеется также много более мелких трансформаций, которые вкраплены повсюду в Java API. Другой новый элемент, включенный в книгу — глава о сервлетах, которые являются маленькими программами, расширяющими функциональные возможности Web-серверов. Я думаю, что это будет особенно интересное добавление.

## **Усилие группы**

Я писал о программировании много лет и редко работаю с соавтором. Однако из-за специфического характера этой книги я объединился с Патриком Ноутоном — одним из создателей языка Java. Проницательность, опыт и энергия Патрика очень способствовали успеху проекта. Благодаря детальному знанию Патриком языка Java, его проекта и реализации в этой книге можно найти советы и методики, которые трудно (если не невозможно) обнаружить в другом месте.

**ГЕРБЕРТ ШИЛДТ**





# Язык JAVA

## ЧАСТЬ I

1. Генезис Java
2. Обзор языка Java
3. Типы данных, переменные и массивы
4. Операции
5. Управляющие операторы
6. Введение в классы
7. Методы и классы
8. Наследование
9. Пакеты и интерфейсы
10. Обработка исключений
11. Многопоточное программирование
12. Ввод/вывод, апплеты и другие темы





## ГЛАВА 1

### Генезис Java

Хронику машинных языков можно записать так: язык В привел к языку С, С был включен в C++, а C++ можно рассматривать как этап в развитии языка Java. Чтобы понять язык Java, нужно выяснить причины, которые управляли его созданием, силы, которые его формировали, и наследство, которое он получает. Подобно успешным машинным языкам, появившимся раньше, язык Java — смесь лучших элементов их богатого наследия, объединенного с творческими концепциями, востребованными ее уникальной средой. В то время как остальные главы данной книги описывают практические аспекты Java, включая синтаксис, библиотеки и приложения, в этой главе вы узнаете, как и почему Java возник, и что делает его настолько важным.

Хотя Java стал неразрывно связанным со средой реального времени Internet, необходимо помнить, что Java является прежде всего языком программирования. Все новшества машинного языка и его развитие преследуют две фундаментальные цели:

- адаптироваться к изменяющимся средам и применением;
- осуществить усовершенствования и уточнения в искусстве программирования.

Создатели Java руководствовались этими задачами почти в равной мере.

### Происхождение Java

Java связан с C++, который является прямым потомком С. Многое в характере Java унаследовано от этих двух языков. От С Java получил его синтаксис. На многие из объектно-ориентированных свойств Java повлиял C++. Некоторые из определяющих характеристик Java происходят от его предшественников. Кроме того, создание Java глубоко внедрилось в процессы усовершенствования и адаптации, которые проявились в языках машинного программирования в течение последних трех десятилетий. По этим причи-

нам в данном разделе предлагается обзор последовательности событий и усилий, которые привели к созданию Java. Каждое новшество в проекте языка управлялось потребностью решить фундаментальную проблему, с которой не справились предшествующие языки. Java не является исключением.

## Рождение современного программирования: С

Язык С потряс компьютерный мир. Его воздействие не должно быть недооценено, потому что он фундаментально изменил подход к программированию и даже способ мышления в нем. Создание С было прямым результатом потребности в структурном, эффективном и высокоуровневом языке, который мог бы заменять ассемблерный код при создании системных программ. Когда машинный язык только разрабатывается, часто делаются компромиссы, подобно следующим:

- между легкостью в использовании и мощностью;
- между безопасностью и эффективностью;
- между жесткостью и расширяемостью.

До С программисты обычно должны были выбирать между языками, которые оптимизировали тот или другой набор характерных черт. Например, хотя ФОРТРАН мог использоваться при написании довольно эффективных программ для научных приложений, он был не очень хорош для системного программирования. И хотя БЕЙСИК был прост для изучения, он оказался не очень мощным, а его недостаточная структурированность делала его пользу весьма сомнительной для больших программ. Ассемблер может использоваться для создания высоко эффективных программ, но он не прост для изучения или результативного использования, а отладка ассемблерного кода может быть весьма трудной.

Другая проблема состояла в том, что ранние машинные языки типа БЕЙСИКА, КОБОЛА и ФОРТРАНА не были построены вокруг структурных принципов. Вместо этого они полагались на оператор GOTO, как на первичное средство программного управления. В результате программы, написанные с использованием этих языков, имели тенденцию производить "спагетти-код" — массу запутанных переходов и условных ветвей, из-за которых программу, фактически, невозможно понять. В то же время языки, подобные Паскалю, структурированы, но они не были разработаны для высокой эффективности и не включали некоторых свойств, делающих их применимыми к широкому диапазону программ. (В частности, учитывая стандартные диалекты Паскаля, доступные в то время, было бы непрактично его использование для кодирования на системном уровне.)

Так, до изобретения С никакой язык не примирял конфликтующих атрибутов, хотя к этому прилагались упорные усилия уже давно. Все же потребность в таком языке возрастала. В начале 1970-х началась компьютерная

революция, и запрос на программное обеспечение быстро стал опережать способность программистов создавать его. Много усилий расходовалось в академических кругах в попытке создать лучший машинный язык. Но, что, возможно, наиболее важно, начинали чувствоватьсь вторичные силы. Компьютерные аппаратные средства становились достаточно обычными, так что критическая масса была достигнута. Больше не было компьютеров, хранимых за блокированными дверьми. Впервые программисты получили фактически неограниченный доступ к машинам. Это обеспечило свободу эксперименту и разрешило программистам создавать их собственные инструментальные средства. Накануне появления С в машинных языках наступил этап существенного скачка вперед.

Изобретенный и впервые реализованный Дэннисом Ритчи (Dennis Ritchie) на машине PDP-11 фирмы DEC для разработки операционной системы UNIX, С был результатом процесса развития, который начинался с более старого языка, называемого BCPL и разработанного Мартином Ричардсом (Martin Richards). На BCPL повлиял язык с именем B, изобретенный Кеном Томпсоном (Ken Thompson), который и привел к разработке С в 1970-х гг. Много лет фактическим стандартом для С был язык, поставляемый с операционной системой UNIX и описанный в книге "Язык программирования С" Брайеном Керниганом (Brian Kernigan) и Дэннисом Ритчи (Prentice-Hall, 1978). С был формально стандартизирован в декабре 1989 г. Американским национальным институтом стандартов (ANSI).

Создание С, по мнению многих, отмечает начало современной эры машинных языков. Он успешно синтезирует противоречивые атрибуты, которые так беспокоили в более ранних языках. Результатом был мощный, эффективный, структурный язык, который оказался относительно простым для изучения. Он также включил еще один, почти неосозаемый аспект: это был язык *программиста*. До изобретения С машинные языки вообще разрабатывались или как академические упражнения, или бюрократическими комитетами. С — другой. Он был спроектирован, реализован и развит реальными, работающими программистами, отражая методику их подхода к работе по программированию. Его свойства были отточены, проверены, обдуманы и обсуждены заново людьми, которые фактически пользовались языком. Результатом стал язык, приемлемый для программирования. Действительно, С быстро привлек много последователей, которые имели к нему почти религиозное отношение. Он также был широко и быстро принят в сообществе программистов. Иными словами, С — это язык, разработанный программистами и для программистов. Как вы увидите, и Java принял это наследство.

## Потребность в C++

В течение поздних 70-х и ранних 80-х С стал доминирующим языком программирования и все еще широко используется сегодня. Так как С — успешный и полезный язык, вы могли бы спросить, почему существовала

потребность в чем-то еще. Ответ может быть таким: потому что существует проблема *сложности* программ. Через всю историю программирования увеличивающаяся сложность программ привела к потребности в лучших способах управления этой сложностью. C++ — ответ на эту потребность. Чтобы лучше понять, почему управление сложностью программы фундаментально для создания C++, приведем следующие рассуждения.

Подходы к программированию драматично изменились, начиная с изобретения компьютера. Например, когда компьютеры были впервые изобретены, программирование выполнялось ручным переключением разрядов двоичных машинных команд в регистрах лицевой вычислительной системы. Пока программы содержали несколько сотен команд, этот подход работал. Когда их объем увеличился, был изобретен ассемблер, так что программист мог иметь дело с большими, все более и более комплексными программами, используя символические представления машинных команд. Поскольку программы продолжали расти, были введены языки высокого уровня, которые предоставили программисту больше инструментов, чтобы реализовывать сложные программные системы.

Первый широко распространенный язык был, конечно, ФОРТРАН. Хотя ФОРТРАН оказался внушительным первым шагом, это едва ли язык, который поощряет чистые и легкие для понимания программы. 1960-е гг. породили *структурное программирование*. Это — метод, поддерживаемый языками типа С. Первое время использование структурных языков давало возможность программистам довольно легко записывать умеренно сложные программы. Однако даже с методами структурного программирования проект когда-то достигает внушительного размера, а его сложность превосходит то, с чем программист может справиться. К началу 1980-х годов во многих проектах отказались от структурного подхода из-за его ограничений. Чтобы решить эту проблему, был изобретен новый подход к программам, названный *объектно-ориентированным программированием* (OOP, Object-Oriented Programming). Объектно-ориентированное программирование подробно обсуждается далее в этой книге, но здесь приведем лишь краткое определение: OOP — это методология программирования, которая помогает организовать сложные программы с помощью наследования, инкапсуляции и полиморфизма.

Наконец, хотя С — один из всемирно используемых языков программирования, существует предел его способности обработать сложность. Как только код программы превосходит размер, находящийся в пределах где-то между 25 000 и 100 000 строк, программа становится столь сложной, что ее трудно воспринимать целиком. C++ позволяет разрушить этот барьер и помогает программисту управлять более крупными программами.

C++ был изобретен Бьярни Страуструпом (Bjarne Stroustrup) в 1979 г., в то время когда он работал в Лабораториях Бэлла (Bell Laboratories) в Мюррей Хилле (Murray Hill), Нью-Джерси. Страуструп первоначально назвал новый язык "С с классами". Однако в 1983 г. это название было заменено на C++. C++ — это расширение С, с добавлением объектно-ориентированных свойств.

Поскольку C++ построен на основе C, он включает все свойства, атрибуты и выгоды C. Это является решающей причиной успеха C++ как языка. Изобретение C++ не было попыткой создать полностью новый язык программирования. Это было улучшением уже довольно успешного языка C. Язык C++ был стандартизирован в ноябре 1997 года, и теперь для него доступен стандарт ANSI/ISO.

## Этап становления Java

К концу 1980-х и началу 1990-х годов объектно-ориентированное программирование, использующее C++, еще сохранялось. Действительно, некоторое время казалось, что программисты наконец нашли совершенный язык. Поскольку C++ объединял высокую эффективность и стилистические элементы C с объектно-ориентированной парадигмой, это был язык, который мог использоваться, чтобы создавать широкий диапазон программ. Однако, так же как в прошлом, назревали силы, которые будут еще не раз продвигать эволюцию машинного языка вперед. В течение нескольких лет Всемирная Паутина (WWW) и Internet достигли критической массы. Это событие ускорило другую революцию в программировании.

## Создание Java

Java был задуман Джеймсом Гослингом (James Gosling), Патриком Ноутоном (Patrick Naughton), Крисом Вартом (Chris Warth), Эдом Франком (Ed Frank) и Майком Шериданом (Mike Sheridan) из компании Sun Microsystems, Inc. в 1991 г. Потребовалось восемнадцать месяцев, чтобы создать первую рабочую версию. Этот язык первоначально назывался "Oak" (Дуб), но был переименован в "Java" (Ява) в 1995 году. Между начальной реализацией Oak в конце 1992 г. и публичным объявлением Java весной 1995 г., намного больше людей внесли вклад в проект и эволюцию языка. Билл Джой (Bill Joy), Артур ван Хофф (Arthur van Hoff), Джонатан Пайн (Jonathan Payn), Фрэнк Еллин (Frank Yellin) и Тим Линдхолм (Tim Lindholm) внесли ключевой вклад в созревание прототипа.

Удивительно, но первоначальным стимулом для Java не был Internet. Вместо этого, первичным побуждением была потребность в независимом от платформы (т. е. архитектурно-нейтральном) языке, который мог использоваться для создания программного обеспечения с целью внедрения в электронные устройства различных потребителей (типа микроволновых печей и дистанционных пультов управления). Как известно, много различных типов CPU (Central processor unit, процессор) используются как контроллеры. Неприятности с C и C++ (и большинством других языков) заключаются в том, что они спроектированы так, чтобы компилироваться для определенного адресата. Хотя возможно компиляция программы C++ для почти любого типа

CPU требует полного компилятора C++, нацеленного на это CPU. Проблема состоит в том, что создание таких компиляторов дорого и отнимает много времени. Требовалось более простое и дешевое решение. В попытке найти такое решение Гослинг и др. начали работу над переносимым, независящим от платформы языком, который мог бы использоваться для производства кода и стал бы выполняться на разных CPU в отличающихся средах. Эти усилия, в конечном счете, и привели к созданию Java.

Когда разрабатывались детали Java, всплыл второй и, в общем, более важный фактор, который позднее стал играть ключевую роль в языке Java. Этим фактором оказалась, конечно, Всемирная Паутина (WWW или просто Web). Если бы она не обрела своей формы примерно в то же самое время, когда был реализован язык Java, то он мог бы остаться полезным, но незаметным для программирования бытовой электроники. Однако с появлением Всемирной Паутины Java тоже начал продвигаться на передний край проектирования машинного языка, потому что в Web также требовалась переносимость программ.

Большинство программистов на ранних этапах своих карьер осознано, что переносимые программы столь же иллюзорны, как и желательны. В то время как поиски способа создания эффективных, переносимых (независимых от платформы) программ оказались почти столь же стары как сама дисциплина программирования, потребовалось возвратиться к другим более неотложным проблемам. Далее, из-за того, что большая часть компьютерного мира разделила себя на три конкурирующих лагеря — Intel, Macintosh и UNIX, большинство программистов осталось в пределах их укрепленных границ, и срочная потребность в переносном коде была уменьшена. Однако с появлением Internet и Web, старая проблема мобильности вернулась. В конце концов, Internet состоит из различных, распределенных областей, заполненных многими типами компьютеров, операционных систем и CPU. Даже при том, что к Internet присоединены различные типы платформ, пользователи хотели бы, чтобы они все были способны выполнить одну и ту же программу. То, что было когда-то раздражающей, но низкоприоритетной проблемой, теперь стало заметной потребностью.

К 1993 г. для членов группы проекта Java стало очевидным, что проблемы мобильности, с которыми часто сталкиваются при создании кода для встроенных контроллеров, также возникают и при попытке создать код для Internet. Действительно, та же проблема, ради устранения которой Java был первоначально разработан, могла бы возникнуть и в Internet, но в крупном масштабе. Такое понимание заставило переключить фокус Java с бытовой электроники на Internet-программирование. В то время как желание архитектурно-нейтрального языка программирования обеспечило начальный успех, Internet, в конечном счете, привел к крупномасштабному успеху Java.

Как было упомянуто ранее, Java получил многие свои свойства из С и C++. Это сделано намеренно. Проектировщики Java знали, что использование

знакомого синтаксиса С и отображение объектно-ориентированных особенностей С++ сделают их язык обращенным к легионам опытных программистов С/С++. В дополнение к поверхностным подобиям, Java совместно использует некоторые из других атрибутов, которые помогли сделать С и С++ успешными. Во-первых, Java был спроектирован, проверен и усовершенствован реально работающими программистами. Это язык, основанный на потребностях и опыта людей, которые его изобрели. Таким образом, Java — тоже язык программистов. Во-вторых, Java связан и логически непротиворечив. В-третьих, за исключением ограничений, наложенных средой Internet, Java дает вам, как программисту, полный контроль. Если вы программируете хорошо, ваши программы это отражают. Если вы программируете плохо, ваши программы отражают и это тоже. Иными словами, Java — не язык для тренировок. Это язык для профессиональных программистов.

Из-за подобий между Java и С++ соблазнительно думать о Java просто как об Internet-версии С++. Однако делать так было бы большой ошибкой. Java имеет существенные практические и философские отличия. Хотя и правда, что Java был под влиянием С++, он — не усовершенствованная версия С++. Например, Java ни вверх, ни вниз не совместим с С++. Конечно, подобия с С++ существенны, и если вы — программист С++, то будете чувствовать себя в Java прямо как дома. Еще одно: Java не был разработан для замены С++. Он был создан, чтобы решить некоторый набор проблем, в то время как С++ был рассчитан для решения другого набора проблем. И оба будут сосуществовать много лет.

Как упомянуто в начале этой главы, машинные языки развиваются по двум причинам: чтобы адаптироваться к изменениям среды и осуществлять усовершенствования в искусстве программирования. Изменение окружающей среды, которое подтолкнуло Java, нуждалось в независимых от платформы программах, спроектированных для распределенной работы в Internet. Java изменяет также саму методику подхода человека к записи программ. Действительно, Java расширяет и совершенствует объектно-ориентированную парадигму, используемую в С++. Таким образом, Java — не язык, который существует в изоляции. Скорее, это текущий образец продолжающегося процесса, начатого много лет назад. Один этот факт достаточен, чтобы гарантировать Java место в истории машинных языков. Java является по отношению к программированию для Internet тем же, чем С был по отношению к системному программированию — революционной силой, которая изменит мир.

## Почему Java важен для Internet

Internet помог катапультировать Java на передний край программирования, а Java, в свою очередь, имел глубокое влияние на Internet. Этому есть простое объяснение: Java разворачивает вселенную объектов, которые могут свободно перемещаться в киберпространстве. В сети две очень широких ка-

тегории объектов передаются между сервером и вашим персональным компьютером — пассивная информация и динамические, активные программы. Например, когда вы читаете вашу электронную почту, то рассматриваете пассивные данные. Даже, когда вы загружаете программу, ее код — это все еще только пассивные данные до тех пор, пока вы их не начнете выполнять. Однако на ваш компьютер может быть передан объект второго типа — динамическая, самовыполняющаяся программа. Такая программа — активный агент на компьютере клиента, все же инициализируется сервером. Например, сервер мог бы предоставить (клиенту) программу, чтобыенным образом отображать данные, посылаемые клиенту.

Столь же желательными, как и динамические, являются *сетевые* программы. Они также порождают серьезные проблемы в области защиты и мобильности. До Java, киберпространство было эффективно закрыто для половины объектов, которые теперь живут там. Кроме того, Java имеет дело с захватывающе новой формой программ — апплетами.

## Java-апплеты и приложения

Java можно использовать, чтобы создать два типа программ — приложения и апплеты. *Приложение* — это программа, которая выполняется на вашем компьютере с помощью его операционной системы. То есть, приложение, созданное с помощью Java, более или менее подобно приложению, созданному с использованием С или С++. При создании приложения Java не намного отличается от любого другого машинного языка. Более важной является способность Java создавать апплеты. *Апплет* — это приложение, разработанное для передачи по Internet и выполняемое совместимым с Java Web-браузером<sup>1</sup>. Апплет — это, фактически, крошечная программа Java, динамически загружаемая через сеть, подобная изображению, звуковому файлу, или видеоклипу. Важное отличие заключается в том, что апплет является интеллектуальной программой, а не просто мультипликацией (анимацией) или media-файлом. Другими словами, апплет — это программа, которая может реагировать на ввод пользователя и динамически изменять, а не просто выполнять ту же самую мультипликацию или звук много раз.

Апплеты были бы не более чем мысленным желанием, если бы Java не был способен адресоваться к двум фундаментальным проблемам, связанным с ними — защите и мобильности. Прежде чем продолжить, давайте определим, что эти два термина означают для Internet.

## Защита

Не секрет, что каждый раз, когда вы загружаете "нормальную" программу, вы рискуете обзавестись вирусной инфекцией. До Java большинство пользо-

<sup>1</sup> *Браузер* (browser) — программа просмотра документов в формате HTML. — Примеч. пер.

вателей не загружало выполняемые программы часто и сканировало их до выполнения для поиска вирусов. Даже в этом случае большинство пользователей все еще волновалось относительно возможности инфицирования их систем. В дополнение к вирусам существует другой тип злонамеренных программ, от которых должна быть защита. Они могут собирать частную информацию, типа номеров кредитных карточек, балансов банковских счетов и паролей, путем их поиска в каталогах локальной файловой системы вашего компьютера. Java отвечает за обе эти ситуации, обеспечивая межсетевую защиту (между сетевым приложением и вашим компьютером).

Когда вы используете совместимый с Java Web-браузер, то можете безопасно загружать Java-апплеты, не опасаясь вирусной инфекции или злых намерений. Язык обеспечивает защиту, ограничивая Java-программу средой ее выполнения, и не позволяет ей получить доступ к другим частям компьютера. (Вы вскоре увидите, как это делается.) Способность загружать аплеты с уверенностью, что никакой вред не будет причинен и никакая защита не будет нарушена, рассматривается многими как наиболее важный аспект Java.

## Мобильность

Как обсуждалось ранее, во всем мире используются различные типы компьютеров и операционных систем, причем многие связаны с Internet. Для программ, динамически загружаемых во все различные типы платформ и связанных с Internet, необходимы некоторые средства производства мобильного (переносимого) выполняемого кода. Как вы вскоре увидите, тот же самый механизм, который гарантирует защиту, помогает также создавать и мобильность. Действительно, решение Java этих двух проблем является как изящным, так и эффективным.

## Волшебство Java: байт-код

Ключом, который позволяет Java решать только что описанные проблемы защиты и мобильности, является байт-код (bytecode). Дело в том, что выход компилятора Java является не выполняемым кодом, а байт-кодом. *Байт-код* — это высоко оптимизированный набор команд, предназначенных для выполнения специальной системой<sup>1</sup> Java, которая называется *виртуальной Java-машиной* (JVM, Java Virtual Machine)<sup>2</sup>. То есть, в его стандартной форме JVM — это интерпретатор байт-кода. Это немного неожиданно. Как известно, исходный текст C++ компилируется в выполняемый код. Фактически,

<sup>1</sup> Run-time system — исполняющая система: — Примеч. пер.

<sup>2</sup> Фактически, JVM — виртуальный компьютер, размещающийся только в оперативной памяти. — Примеч. ред.

большинство современных языков разработано для компиляции, а не интерпретации — главным образом по соображениям эффективности выполнения. Однако факт, что программа Java, выполняемая с помощью JVM, помогает решить большинство проблем, связанных с загрузкой программ через Internet. И вот почему.

Перевод программы Java в байт-код делает более простым ее выполнение в широком разнообразии сред. Единственное условие: JVM должен быть реализован для каждой платформы. Как только пакет времени выполнения<sup>1</sup> появляется для данной платформы, на нем может выполняться любая программа Java. Помните, хотя подробности JVM отличаются от платформы к платформе, все они интерпретируют тот же самый байт-код Java. Если программа Java была откомпилирована с приведением к родному (native) коду, то различные версии той же самой программы должны будут существовать для каждого типа CPU, присоединенного к Internet. Это, конечно, невыполнимое решение. Таким образом, интерпретация байт-кода — самый простой способ создавать истинно переносимые программы.

Факт, что программа Java интерпретируется, также помогает сделать ее и безопасной. Поскольку выполнение каждой программы Java находится под управлением JVM, то JVM может как содержать программу, так и оберегать ее от генерации побочных эффектов вне системы. Безопасность также усилена некоторыми ограничениями, которые существуют в языке Java.

Когда программа интерпретируется, она, в общем случае, выполняется существенно медленнее, чем при компиляции к выполняемому коду. Однако в случае Java разница между двумя способами трансляции не так велика. Использование байт-кода дает возможность исполнительной (run-time) системе Java выполнять программы намного быстрее, чем можно было бы ожидать.

Хотя Java был разработан для интерпретации, не существует технических препятствий для компиляции "на лету" байт-кода в родной код. Поэтому Sun только что разработала свой Just In Time (JIT) компилятор для байт-кода, который включен в выпуск Java 2. Поскольку JIT-компилятор — часть JVM, он компилирует байт-код в выполнимый код в реальном времени "часть-за-частью" по запросу. Необходимо понять, что невозможно компилировать полную Java-программу сразу в выполняемый код, потому что Java организует различные проверки, которые могут быть сделаны только во время выполнения. Вместо этого JIT компилирует код по мере необходимости, во время выполнения. Однако современный подход все еще дает существенное увеличение эффективности. Даже когда к байт-коду применяется динамическая компиляция, мобильность и безопасность все еще обеспечиваются, потому что исполнительная система (которая осуществляет компи-

<sup>1</sup> В языке программирования Java принято делить средства JDK на средства времени проектирования и средства времени выполнения. — Примеч. пер.

ляцию) все еще отвечает за среду выполнения. Интерпретируется ли ваша Java-программа фактически традиционным способом или откомпилирована "на лету", ее функциональные возможности одинаковы.

## Базовые термины Java

Никакое обсуждение происхождения Java нельзя закончить без взгляда на базовые термины (buzzwords) Java. Хотя основными причинами, которые привели к необходимости изобретения Java, являлись мобильность и защита, другие факторы также играли важную роль в формировании окончательной формы языка. Ключевые соображения были суммированы группой Java в следующем списке базовых терминов:

- простой;
- безопасный;
- переносимый;
- объектно-ориентированный;
- устойчивый;
- многопоточный;
- архитектурно-независимый;
- интерпретируемость;
- высокая эффективность;
- распределенный;
- динамический.

Два из этих терминов уже были обсуждены: безопасный и переносимый. Давайте рассмотрим, что означают остальные.

## Простой

Java был спроектирован, чтобы быть легким в изучении и эффективным в использовании для профессионального программиста. Предполагается, что, имея некоторый опыт программирования, вы не найдете Java трудным для мастера. Если вы уже понимаете основные концепции объектно-ориентированного программирования, изучение Java будет даже проще. Лучше всего, если вы — опытный программист C++, переход к Java потребует тогда очень небольшого усилия. Поскольку Java наследует синтаксис C/C ++ и многие из объектно-ориентированных свойств C++, большинство программистов имеет немного неприятностей при изучении Java. Кроме того, некоторые из наиболее запутывающих концепций C++ или изъяты из Java, или реализованы в более ясной, более доступной манере.

Вне своих подобий с C/C ++ Java имеет и другой атрибут, который делает его легким для изучения: разработчики предприняли специальные усилия, чтобы язык не обладал "сюрпризными" свойствами. В Java существует небольшое количество ясно определенных способов, чтобы выполнить данную задачу.

## Объектно-ориентированный

Даже под влиянием своих предшественников Java не разрабатывалася так, чтобы его исходный текст был похож на текст любого другого языка. Это обеспечило группе Java свободу проектировать "с чистого листа". Одним из результатов этого был чистый, пригодный для использования, прагматический подход к объектам. Свободно заимствуя из многих плодотворных объектно-программных сред нескольких последних десятилетий, Java ухитряется найти равновесие между парадигмой пуриста "все — объекты" и моделью прагматика "не стой на моем пути". Объектная модель в Java проста и легко расширяема, в то же время простые типы, такие как целые, сохраняются как высокоеффективные "необъекты".

## Устойчивый

Многоплатформенная среда Web предъявляет экстраординарные требования к программе, потому что та должна выполниться надежно в самых разнообразных системах. Поэтому способности создавать устойчивые программы был дан высокий приоритет в проекте Java. Чтобы обеспечить надежность, Java ограничивает вас в нескольких ключевых областях, вынуждая рано находить ошибки при разработке программы. В то же самое время, Java освобождает от необходимости волноваться относительно многих из наиболее общих причин ошибок программирования. Поскольку Java — язык со строгой типизацией, он проверяет ваш код во время компиляции. Однако он также проверяет ваш код и во время выполнения. В действительности, множество трудно прослеживаемых ошибок, которые часто обнаруживаются в трудно воспроизводимых ситуациях во времена выполнения, просто невозможно создать в Java. Знание того, что программа, которую вы написали, будет вести себя предсказуемым образом при разных условиях, является ключевым свойством Java.

Чтобы лучше понимать, насколько устойчив Java, рассмотрим две из главных причин отказа программы: ошибки управления памятью и неуправляемые исключительные состояния (т. е. ошибки во время выполнения). Управление памятью может быть трудной и утомительной задачей в традиционных средах программирования. Например, на C/C++ программист должен вручную распределять и освобождать всю динамическую память. Это иногда ведет к проблемам, потому что программисты или забывают освобождать память, которая была предварительно распределена, или, хуже, пытаются освободить некоторую память, которую другая часть их кода все еще использует. Java фактически устранил эти проблемы, управляя распределением и освобождением памяти. (Фактически, освобождение полностью автоматическое, потому что Java обеспечивает сборку "мусора" для неиспользованных объектов.) Исключительные состояния в традиционных средах часто возникают в ситуациях типа деления на нуль или "файл, не най-

ден", и они должны управляться неуклюжими и трудно читаемыми конструкциями. Java помогает и в этой области, обеспечивая объектно-ориентированную обработку особых ситуаций. В хорошо написанной Java-программе все ошибки времени выполнения могут — и должны — управляться вашей программой.

## Многопоточный

Java был спроектирован так, чтобы выполнить реальное требование — создавать интерактивные сетевые программы. Чтобы выполнить это, Java поддерживает *многопоточное программирование*, которое позволяет вам писать программы, выполняющие одновременно несколько операций. Исполняющая система Java подходит с изящным и все же искушенным решением к синхронизации мультипроцесса, что дает возможность создавать гладко работающие интерактивные системы. Удобный в работе подход Java к многопоточности позволяет вам поразмыслить над спецификой поведения вашей программы, а не заботиться о многозадачной подсистеме.

## Архитектурно-независимый

Центральной проблемой для проектировщиков Java была долговечность и мобильность кода. Одна из главных проблем, возникающих перед программистами, заключается в том, что нет гарантии, что после написания программы сегодня она будет работать завтра — даже на той же самой машине. Модернизация операционной системы и процессора, изменения в основных системных ресурсах могут объединяться так, что вызовут сбой программы. Проектировщики Java сделали несколько жестких решений в языке и виртуальной Java-машине в попытке изменить эту ситуацию. Их цель можно сформулировать так: "запись — однажды; выполнение — везде, в любое время, всегда". В значительной степени эта цель была достигнута.

## Интерпретируемость и высокая эффективность

Как было упомянуто выше, Java делает возможным создание кросс-платформенных программ, компилируя в промежуточное представление, названное байт-кодом Java. Этот код может интерпретироваться в любой системе, которая обеспечена виртуальной Java-машиной. Большинство предыдущих попыток для кросс-платформенных решений делали это за счет дорогостоящей эффективности. Другие интерпретируемые системы, типа БЭЙСИК, ТСЛ и ПЕРЛ страдают от почти непреодолимого дефицита эффективности. Java, однако, был разработан так, чтобы эффективно работать на очень маломощных CPU. Как объяснялось ранее, хотя и верно, что Java разрабатывалася для интерпретации, байт-код Java был тщательно спроектирован так, чтобы не возникало проблем при трансляции его непосредственно в родной

машинный код (очень высоко эффективный), используя синхронный компилятор. Исполнительная система Java, которая обеспечивает это свойство, не теряет ни одного из преимуществ кода, независимого от платформы. "Высокоэффективная кросс-платформа — больше не оксюморон<sup>1</sup>".

## Распределенный

Java разработан для распределенной среды Internet, потому что он обрабатывает протоколы TCP/IP. Фактически, доступ к ресурсу с использованием URL не намного отличается от доступа к файлу. Первоначальная версия Java (Oak) поддерживала свойство передачи сообщений во внутриадресном пространстве. Это позволяло объектам на двух различных компьютерах выполнять процедуры дистанционно. Java недавно восстановил эти интерфейсы в пакете с именем *Remote Method Invocation* (RMI, удаленный вызов методов). Данное свойство вносит беспрецедентный уровень абстракции в программирование на основе клиент-сервер.

## Динамический

Программы Java несут в себе существенное количество информации времени выполнения, которая используется, чтобы проверять и разрешать доступ к объектам в период работы программы. Это дает возможность динамически связывать код в безопасной и целесообразной манере, и имеет решающее значение для устойчивости среды апплета, в которой маленькие фрагменты байт-кода могут динамически обновляться исполнительной системой.

## Продолжение революции

Начальная версия Java была далека от революционности, но это не отмечало конца эры быстрого обновления Java. В отличие от большинства других программных систем, которые обычно укладываются в шаблон мелких улучшений, Java продолжал развиваться во взрывном темпе. Вскоре после версии Java 1.0 проектировщики уже создали Java 1.1. Свойства, добавленные этой версией, были более существенны, чем просто увеличение следующего номера версии. Java 1.1 приобрел новые библиотечные элементы, для него переопределены способы обработки событий в апплетах и реконфигурированы многие свойства библиотеки версии 1.0. Из версии также исключены (объявлены устаревшими) некоторые свойства, первоначально определенные в Java 1.0. Таким образом, в Java 1.1 как добавлены, так и исключены атрибуты первоначальных технических спецификаций. Продолжая эту эволюцию, Java 2 также добавляет и исключает некоторые свойства.

<sup>1</sup> Оксюморон (окхютогон) — сочетание противоположных по значению слов. — Примеч. пер.

Изменения в Java очень важны, потому что старые версии браузеров не будут способны выполнить код, который использует новое свойство. По этой причине хорошо иметь общее представление о том, когда произошли различные изменения. Следующий раздел представляет краткий обзор эволюции Java, начиная с его исходной спецификации 1.0.

## Свойства, добавленные версией 1.1

Версия 1.1 добавила в Java некоторые важные элементы. Большинство добавлений произошло в библиотеке Java. Однако были также включены несколько новых свойств самого языка. Список наиболее важных, добавленных версией 1.1, таков:

- Java Beans, которые являются программными компонентами, написанными на языке Java.
- СерIALIZАЦИЯ, которая позволяет сохранять и восстанавливать состояние объекта.
- Remote Method Invocation (RMI, вызов удаленных методов), который позволяет объекту Java вызывать методы иного Java-объекта, расположенного на другой машине. Это важное средство для построения распределенных приложений.
- Java Database Connectivity (JDBC, средство связи с базами данных), позволяющее программам обращаться к SQL базам данных от многих различных продавцов.
- Java Native Interface (JNI, интерфейс "родных" (native) программ), который обеспечивает новый способ взаимодействия ваших программ с библиотеками кода, написанными на других языках.
- Отражение, которое является процессом определения полей, конструкторов, и методов объекта Java во время выполнения.
- Различные свойства защиты, такие как цифровые подписи, обзоры сообщений, списков управления доступом, и генерация ключей.
- Встроенная поддержка для 16-разрядных символьных потоков, которые обрабатывают символы Unicode.
- Существенные изменения обработки событий, которые улучшают способ обработки событий, сгенерированных компонентами графического интерфейса пользователя (GUI, Graphic Use Interface).
- Внутренние классы, которые позволяют определять один класс внутри другого.

## Свойства, исключенные из версии 1.1

Как только что было упомянуто, из Java 1.1 исключено много ранних библиотечных элементов. Например, самый большой из первоначальных классов

Date был исключен. Однако исключенные свойства не исчезли. Вместо этого, они были заменены модифицированными альтернативами. Вообще, исключенные свойства версии 1.0 все еще доступны в Java, чтобы поддерживать существующие коды, но они не должны использоваться новыми приложениями. Эта книга описывает некоторые из наиболее важных исключенных элементов из библиотеки 1.0 ради программистов, обновляющих старые коды.

## Свойства, добавленные версией 2

Построенная на версии 1.1, Java 2 приобрела много важных новых свойств. Вот их частичный список:

- Swing — набор компонентов интерфейса пользователя, который реализован полностью на языке Java. Вы можете использовать средства "look-and-feel", которые или специфичны для конкретной операционной системы, или одинаковы для всех операционных систем. Вы можете также проектировать ваши собственные средства "look-and-feel".
- Коллекции — это группы объектов. Java 2 обеспечивает несколько типов коллекций, таких как связанные списки, динамические массивы и хэштаблицы. Коллекции предлагают новый способ решения нескольких общих проблем программирования.
- Для Java-программ теперь доступны более гибкие механизмы защиты. Файлы политики (policy files) могут определять разрешения для кода из различных источников. Они определяют, например, возможен ли доступ к специальному файлу или каталогу, или можно ли установить соединение с определенным главным компьютером и портом.
- Цифровые удостоверения обеспечивают механизм установки тождества пользователя. Вы можете представлять их как электронные паспорта. Программы Java могут анализировать и использовать удостоверения, чтобы предписать политику защиты.
- Доступны различные инструментальные средства защиты, которые дают возможность создавать и хранить криптографические ключи и цифровые удостоверения, подписывать файлы Java-архива (JAR) и проверять сигнатуру JAR-файла.
- Библиотека доступности (Accessibility library) обеспечивает свойства, которые делают ее проще во время работы с компьютером для людей с ухудшениями зрения или другими недостатками. Конечно, эти возможности могут быть полезны для любого пользователя.
- Библиотека Java 2D обеспечивает усовершенствованные свойства для работы с формами, изображениями и текстом.
- Возможность перетаскивания мышью (drag-and-drop) позволяет перемещать данные внутри или между приложениями.

- Текстовые компоненты могут теперь принимать с клавиатуры японские, китайские и корейские символы. Для ввода таких символов используются последовательности нажатий клавиш.
- Теперь можно воспроизводить аудиофайлы WAV, AIFF, AU, MIDI и RMF.
- Архитектура посредника запросов типового объекта (CORBA, Common Object Request Broker Architecture) определяет посредника запросов объекта (ORB, Object Request Broker) и язык определения интерфейса (IDL, Interface Definition Language). Java 2 включает ORB и компилятор **idltojava**. Последний генерирует код по IDL-спецификациям.
- Улучшение эффективности было сделано в нескольких областях. В JDK включен JIT-компилятор (Just-In-Time, только в реальном времени).
- Многие браузеры включают виртуальную Java-машину, которая используется для выполнения апплетов. К сожалению, браузерные JVM обычно не включают самых последних свойств Java. Эту проблему решает подключаемый (Plug-In) блок Java. Он нацеливает браузер на использование JRE (Java Runtime Environment, среда выполнения Java) вместо браузерной JVM. JRE является подмножеством JDK. Он не включает инструментальные средства и классы, которые используются в среде разработки.
- Были усовершенствованы различные инструментальные средства типа **javac**, **java** и **javadoc**. Для JVM доступны интерфейсы отладчика и системы построения профиля программы (**profiler**).

## Свойства, исключенные из версии 2

Из Java 2 исключены некоторые свойства Java 1.1, хотя исключения и не столь обширны, как в версиях 1.0 и 1.1. Например, методы `suspend()`, `resume()` и `stop()` класса `Thread` не должны использоваться в новом коде. В данной книге исключенные особенности всегда указаны, а их Java 2 альтернативы описаны. Это будет полезно для любого программиста, ответственного за обновление кода Java 1.1.

## Java — не расширение HTML

Прежде чем двигаться дальше, необходимо рассеять общее недоразумение. Поскольку Java используется в создании Web-страниц, новички иногда путают Java с языком разметки гипертекста (HTML) или думают, что Java — просто некоторое расширение HTML. К счастью, это неправильные представления. HTML — это, в сущности, средство определения логической организации информации и обеспечения ссылок, называемых гипертекстовыми ссылками, к связанной информации. *Гипертекстовая ссылка* (также называемая *гиперссылкой*) — это ссылка к другому гипертекстовому докумен-

ту, который может существовать или локально, или где-то в Web-сети. Определяющим свойством гипертекстового документа является то, что он может читаться нелинейным способом. Пользователь отыскивает связанный документ, просто выбирая соответствующую гипертекстовую ссылку к нему.

Хотя HTML дает возможность пользователю читать документы динамическим способом, HTML не является и никогда не был языком программирования. Конечно, верно, что HTML помог стимулировать популярность Web, но он был также катализатором для создания языка Java, хотя непосредственно и не влиял на проект языка или стоящие за ним концепции. Единственной связью HTML с Java является тег `<applet>`, выполняющий Java-апплет.

## ГЛАВА 2



# Обзор языка Java

Подобно другим машинным языкам, элементы языка Java не существуют изолированно друг от друга. Скорее они работают совместно, чтобы сформировать язык в целом. Однако эта взаимосвязь может затруднить описание одного аспекта Java без учета некоторых других. Часто обсуждение одного свойства означает априорное знание другого. Поэтому в данной главе представлен краткий обзор нескольких ключевых свойств языка Java. Приведенный здесь материал даст точку опоры, которая позволит вам писать и понимать простые программы. Большинство обсужденных здесь тем будет более детально рассмотрено в остальных главах Части I.

## Объектно-ориентированное программирование

Сущность языка Java составляет *объектно-ориентированное программирование* (OOP, Object-Oriented Programming). Фактически все программы Java объектно-ориентированы (тогда как, например, в языке C++ это не так). Объектно-ориентированное программирование так интегрировано с языком Java, что вы должны понять его основные принципы, прежде чем сможете написать даже самую простую Java-программу. Потому эта глава начинается с обсуждения теоретических аспектов ООР.

## Две парадигмы программирования

Как вы знаете, все компьютерные программы состоят из двух элементов: *кода и данных*<sup>1</sup>. Любая программа может быть концептуально организована

<sup>1</sup> В записи программы на исходном языке под *кодом* понимают набор исполняемых утверждений, определяющих алгоритм обработки данных, а под *данными* — описатели переменных, используемых в этом алгоритме. — *Примеч. пер.*

либо вокруг ее кода, либо вокруг ее данных. Иначе говоря, некоторые программы концентрируют свою запись вокруг того, "что делается с данными"<sup>1</sup>, а другие — вокруг того, "на что этот процесс влияет"<sup>2</sup>. Существуют две парадигмы (основополагающих подходов), которые управляют конструированием программ. Первый подход называет программу *моделью, которая ориентирована на процесс* (*process-oriented model*). При этом подходе программу определяют последовательности операторов ее кода. Модель, ориентированную на процесс, можно представлять как *кодовое воздействие на данные* (*code acting on data*). Процедурные языки, такие как C, успешно эксплуатируют такую модель. Однако, как указывалось в главе 1, при этом подходе возникают проблемы, когда возрастает размер и сложность программ.

Второй подход, названный *объектно-ориентированным программированием*, был задуман для управления возрастающей сложностью программ. Объектно-ориентированное программирование организует программу вокруг своих данных (т. е. вокруг *объектов*) и набора хорошо определенных *интерфейсов* (взаимодействий) с этими данными. Объектно-ориентированную программу можно характеризовать как *управляемый данными доступ к коду* (*data controlling access to code*). Как вы увидите далее, переключая управление на данные, можно получить некоторые организационные преимущества.

## Абстракция

Существенным элементом объектно-ориентированного программирования является *абстракция*. Человечество управляет сложностью через абстракцию. Например, люди не представляют себе автомобиль как набор десятков тысяч индивидуальных частей (деталей). В их воображении автомобиль — хорошо определенный *объект* со своим собственным уникальным *поведением*. Эта абстракция позволяет людям использовать автомобиль для поездки в бакалейный магазин, не задумываясь над сложностью частей, из которых он состоит. Игнорируя подробности работы двигателя, трансмиссионной и тормозной системы, они могут свободно пользоваться объектом в целом.

Мощным способом управления абстракцией является применение *иерархических классификаций*. Они позволяют расслоить семантику сложных систем, разбивая их на более управляемые части. Извне автомобиль представляется как единый объект. Изнутри же он содержит несколько систем — рулевого управления, тормозной и звуковой систем, пристяжных ремней, нагревателей, сотового телефона и т. п. В свою очередь каждая из этих субсистем состоит из более специализированных узлов. Например, звуковая система имеет радиоприемник, CD-плейер и кассетный магнитофон. Суть здесь в том, что вы управляете сложностью автомобиля (или любой другой сложной системой) через использование иерархических абстракций.

<sup>1</sup> То есть вокруг процесса обработки, определяемого кодом. — Примеч. пер.

<sup>2</sup> То есть вокруг самих данных. — Примеч. пер.

Иерархические абстракции сложных систем можно также применить к компьютерным программам. Данные из традиционной программы, ориентированной на процесс, могут быть трансформированы (путем абстракции) в компоненты ее объектов. Последовательность шагов алгоритмического процесса может стать набором *сообщений* между этими объектами. Таким образом, каждый из этих объектов описывает свое собственное уникальное *поведение*. Вы можете трактовать эти объекты как конкретные сущности, которые откликаются на сообщения, говорящие им, *что нужно делать*. В этом и состоит сущность объектно-ориентированного программирования.

Объектно-ориентированные концепции формируют основу языка Java, точно так же, как они формируют основу человеческого понимания. Важно, чтобы вы понимали, как эти концепции транслируются в программы. Итак, объектно-ориентированное программирование — это мощная и естественная парадигма для создания программ, которые переживают неизбежные изменения, сопровождающие жизненный цикл любого большого программного проекта (на всех этапах этого цикла, включая разработку концепций, рост и старение). Например, как только вы получаете хорошо определенные объекты и ясные и надежные интерфейсы к этим объектам, вы можете изящно и безболезненно удалить или заменить части старой системы.

## Три принципа ООП

Все языки объектно-ориентированного программирования обеспечивают механизмы, которые помогают вам реализовать объектно-ориентированную модель. К ним относятся *инкапсуляция*, *наследование* и *полиморфизм*.

### Инкапсуляция

Инкапсуляция — это механизм, который связывает код вместе с обрабатываемыми им данными и сохраняет их в безопасности как от внешнего влияния, так от ошибочного использования. Можно представить инкапсуляцию как *защитную оболочку*, которая предохраняет код и данные от произвольного доступа из других кодов, определенных *вне* этой оболочки. Доступ к коду и данным *внутри* оболочки строго контролируется через хорошо определенный интерфейс. Чтобы соотнести это с реальным миром, рассмотрим автоматическую трансмиссию автомобиля. Она инкапсулирует сотни бит информации о вашей машине, например, величину ускорения, наклон поверхности, на которой находится машина, и позицию рукоятки передач. Как пользователь вы имеете только один метод воздействия на эту сложную инкапсуляцию — перемещение рукоятки передач. Вы не можете, например, повлиять на трансмиссию, используя сигналы поворота или стеклоочистители. Таким образом, рукоятка передач есть удачно определенный (на самом деле — уникальный) *интерфейс* к трансмиссии. Далее, все, что происходит внутри трансмиссии, никак не влияет на ее внешние объекты. Например, смещение рычага передач никак не сказывается на сигналах поворота.

Именно потому, что автоматическая трансмиссия инкапсулирована, масса производителей автомобилей может реализовать ее любым способом, каким они хотят. Однако с точки зрения водителей все трансмиссии работают одинаково. Ту же самую идею можно применить и к программированию. Мощь инкапсулированного кода состоит в том, что каждый знает, как получить к нему доступ, и может пользоваться им независимо от деталей его реализации и без боязни неожиданных побочных эффектов.

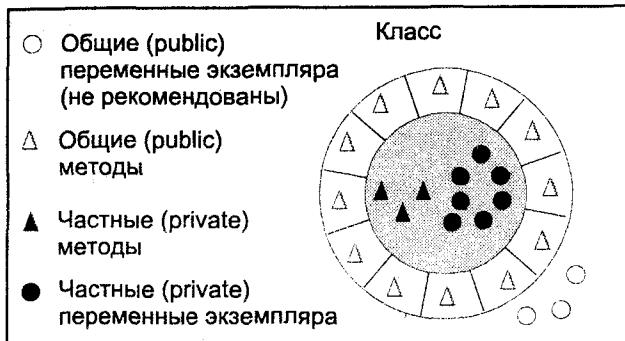
Основой инкапсуляции в языке Java является класс. Хотя классы будут подробно рассмотрены дальше, здесь будет полезно их краткое обсуждение. Класс определяет структуру и поведение (данные и код) некоторого набора объектов. Каждый объект заданного класса содержит как структуру (данные), так и поведение, определяемые классом (как если бы они были проштампованы некоторым шаблоном в форме класса). По этой причине об объекте иногда говорят как об *экземпляре класса*. Таким образом, *класс* — это логическая конструкция, а *объект* — это физическая реальность<sup>1</sup>.

Когда вы создаете класс, нужно специфицировать код и данные, которые составляют этот класс. Все вместе эти элементы называют *членами* (*members*) *класса*. Данные, определяемые в классе, называют *членами-переменными* (*member variables*) или *переменными экземпляра* (*instance variables*). Код, оперирующий с этими данными, называют *членами-методами* (*member methods*) или просто *методами*<sup>2</sup> (*methods*). В правильно записанных Java-программах методы определяют, как можно использовать члены-переменные. Это означает, что поведение и интерфейс класса определяются методами, оперирующими на данных его экземпляров.

Так как цель класса — инкапсуляция сложности, то существуют механизмы скрытия сложности реализации внутри класса. Каждый метод или переменная в классе может быть помечена как *private* (частный или локальный) или *public* (общий). Модификатор *public* указывает на все, что нужно или можно знать внешним пользователям класса. Методы и данные с модификатором *private* могут быть доступны только в коде, являющемся членом данного класса. Любой код, не являющийся членом класса, не имеет доступа к *private*-методам или переменным. Так как *private*-члены класса оказываются доступными другим частям вашей программы только через *public*-методы класса, вы можете быть уверены, что никакие неподходящие действия не выполняются. Конечно, это означает, что *public*-интерфейс должен быть тщательно спроектирован, чтобы не раскрывать слишком много внутренних членов класса (рис. 2.1).

<sup>1</sup> Поскольку речь идет о программировании, точнее трактовать *объект* как *экземпляр программы* некоторого реального (не обязательно физического) объекта. Класс в этом случае естественно трактуется как *тип* объекта. — Примеч. пер.

<sup>2</sup> Если вы знакомы с языками С/С++, полезно знать следующее: то, что Java-программист называет *методом*, С/С++-программист называет *функцией* (*function*). — Примеч. пер.



**Рис. 2.1.** Инкапсуляция:  
public-методы можно  
использовать для защиты  
private-данных

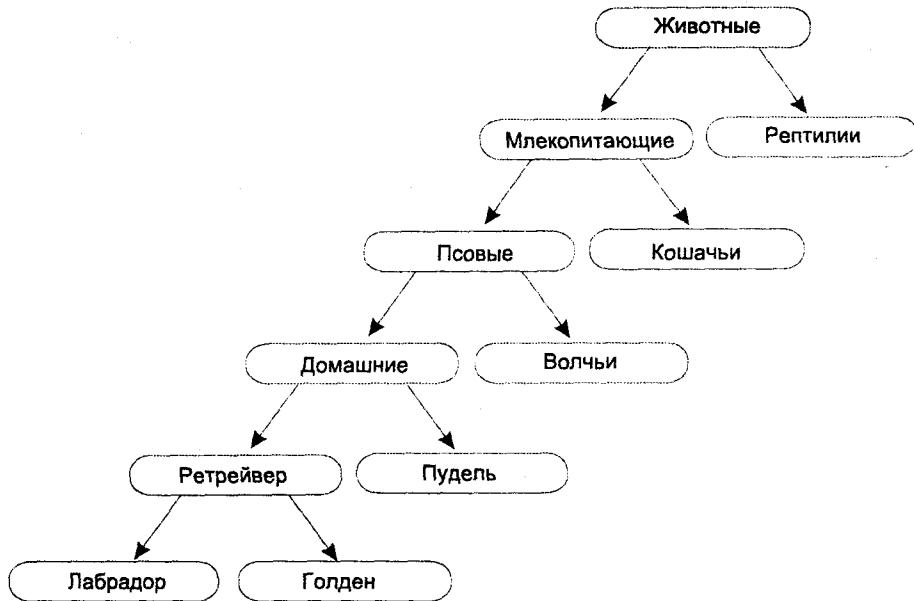
## Наследование

Наследование есть процесс, с помощью которого один объект приобретает свойства другого объекта. Оно важно потому, что поддерживает концепцию иерархической классификации. Как уже говорилось выше, наибольшая часть знаний становится управляемой только с помощью иерархических (т. е. организованных "сверху вниз") классификаций. Например, порода поисковых охотничьих собак *Golden Retriever* является частью классификации *dog* (собаки), которая, в свою очередь, есть часть класса *Mammal* ( млекопитающие ) — подкласса большого класса *Animal* (животные). Без применения классификаций каждый объект нуждался бы в явном определении всех своих характеристик. При использовании наследования объект нуждается в определении только тех качеств, которые делают его уникальным в собственном классе. Он может наследовать общие свойства от своего родителя. Поэтому именно механизм наследования дает возможность одному объекту быть специфическим экземпляром более общего случая. Рассмотрим этот процесс поподробнее.

Большинство людей, естественно, видит мир, состоящим из объектов, которые связаны друг с другом иерархическим способом, например, из животных, млекопитающих и собак. Если бы вы хотели описать животных абстрактным образом, вам следовало бы определить их некоторые атрибуты, например размер, интеллект и тип скелетной системы. Животные также обладают некоторыми поведенческими аспектами — они едят, дышат и спят. Такое описание атрибутов и поведения и определяет *класс* для животных.

Если бы вы хотели описать более специфичный класс животных, такой как *млекопитающие*, они бы должны были иметь более специфичные атрибуты, такие как тип зубов и молочные железы. Подобный класс известен как *подкласс* животных, тогда как о классе животных говорят как о *суперклассе*.

Так как *млекопитающие* есть более точно специфицированные *животные*, то говорят, что они *наследуют* все атрибуты животных. Подкласс, находящийся на более глубоком уровне иерархии, наследует все атрибуты каждого своего предка в *иерархии классов* (рис. 2.2).



**Рис. 2.2.** Схема иерархии классов

Наследование взаимодействует также и с инкапсуляцией. Если данный класс инкапсулирует некоторые атрибуты, то любой подкласс будет иметь те же атрибуты *плюс* атрибут, который он добавляет как часть своей специализации (рис. 2.3). Это ключевая концепция, которая позволяет объектно-ориентированным программам расти по сложности *линейно*, а не *геометрически*. Новый подкласс наследует все атрибуты всех его предков. Он не имеет непредсказуемых взаимодействий с большинством остальных кодов в системе.

## Полиморфизм

**Полиморфизм** (от греч. — "имеющий много форм") — свойство, которое позволяет использовать один интерфейс для общего класса действий. Специфическое действие определяется точной природой ситуации. Рассмотрим стек (список типа LIFO — Last-In, First-Out). Вы можете иметь программу, которая требует три типа стеков. Один стек используется для целых значений, другой — для значений с плавающей точкой, и третий — для символов. Алгоритм, который реализует каждый стек — один и тот же, хотя хранимые данные различны. В не объектно-ориентированном языке от вас бы потребовалось создать три различных набора стековых подпрограмм, каждый из которых имел бы собственное имя. Вследствие полиморфизма в языке Java можно специфицировать общий для всех типов данных набор стековых подпрограмм, использующих одно и то же имя.

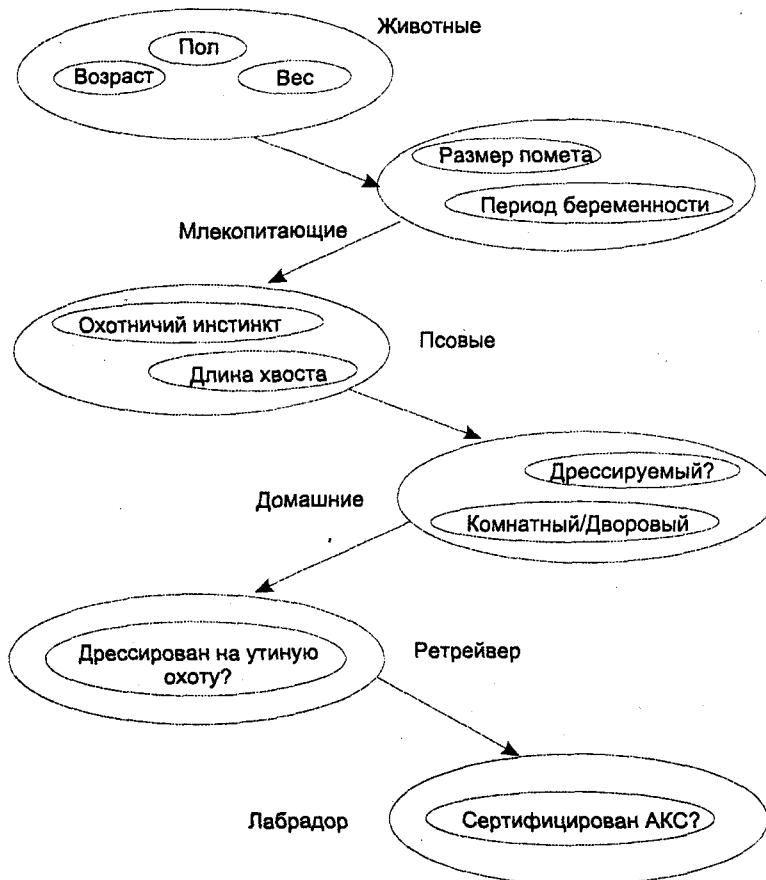


Рис. 2.3. Лабрадор наследует свойства всех своих суперклассов

В общем смысле, концепцию полиморфизма часто выражают фразой "один интерфейс, много методов". Это означает, что возможно спроектировать родовой интерфейс для группы связанных объектов. Это позволяет уменьшить сложность, допуская использование одного и того же интерфейса для *общего класса действий*. Задача компилятора — выбрать специфическое действие (т. е. метод) для его использования в каждой конкретной ситуации. Вы — программист — не должны делать этот выбор "вручную". Вам нужно только помнить и использовать общий интерфейс.

Расширяя аналогию с собакой, можно сказать, что обоняние у собаки полиморфно. Если она чует кошку, то лает и бежит за ней. Если чует пищу, выделяет слюну и бежит к миске. В обеих ситуациях одно и то же чувство — обоняние. Разница в том, что именно она нюхает, т. е. в типе данных, с которым оперирует нос собаки! Та же общая концепция реализована и в языке Java относительно методов в Java-программах.

## **Полиморфизм, инкапсуляция и наследование работают вместе**

При правильном применении полиморфизм, инкапсуляция и наследование комбинируются так, что создают некую среду программирования, которая обеспечивает намного более устойчивые и масштабируемые программы, чем модель, ориентированная на процесс. Удачно спроектированная иерархия классов является базисом для повторно используемого кода, в который вы вложили время и усилия при разработке и тестировании. Инкапсуляция позволяет реализациям мигрировать во времени без разрушения кода, который зависит от public-интерфейса классов. Полиморфизм позволяет создавать ясный, чувствительный и читабельный код.

Из двух примеров реальных объектов, приведенных выше, автомобиль полнее иллюстрирует мощь объектно-ориентированного проектирования. Пример с собаками удобно использовать при обдумывании наследования, а пример с автомобилями больше подходит для программирования. При возжении разных типов (подклассов) транспортных средств все водители полагаются на наследование. Независимо от того, является ли транспортное средство школьным автобусом, Mercedes-седаном, Porsche или представителем семейства minivan (крытый фургон), водитель везде может работать с рулем, колесами, тормозами и акселератором. Наслушавшись скрежетаний в коробке передач, большинство людей может даже справиться с различием между рычажным и автоматическим переключением скоростей, потому что они ясно представляют их общий суперкласс — трансмиссию.

Люди постоянно взаимодействуют с инкапсулированными свойствами автомобиля. Педали тормоза и газа скрывают невероятный набор сложностей с настолько простым интерфейсом, что вы можете работать с ним с помощью ног. Работа двигателя, тип тормозов, размер колес не влияют на то, как вы взаимодействуете с определением класса педалей.

Последний атрибут — полиморфизм — ясно отражен в способности производителей автомобилей предлагать широкий набор возможностей для примерно одинаковых транспортных средств. Например, вы можете получить противоблокировочную тормозную систему или традиционные тормоза, рулевое управление с усилителем или реечное управление, 4-, 6- или 8-цилиндровый двигатель. В любом случае, когда вы не хотите двигаться, то будете нажимать педаль тормоза для остановки, поворачивать руль для изменения направления и нажимать акселератор. Один и тот же интерфейс можно использовать для управления различными реализациями.

Как можно было видеть, именно через применение инкапсуляции, наследования и полиморфизма отдельные части трансформируются в объект, известный как *автомобиль*. То же самое справедливо и для компьютерных программ. Применяя объектно-ориентированные принципы, разные части сложной программы можно объединить вместе в форму взаимосвязанного, устойчивого и легко обслуживаемого целого.

Как упоминалось в начале этого раздела, каждая Java-программа объектно-ориентирована. Или точнее, каждая Java-программа включает инкапсуляцию, наследование и полиморфизм. Хотя примеры коротких программ, показанных в конце этой и в нескольких следующих главах, возможно, не проявляют этих свойств, они, тем не менее, присутствуют в них. Как вы увидите далее, многие из свойств, обеспечиваемых языком Java, являются частью встроенных в него библиотек классов, которые широко используют инкапсуляцию, наследование и полиморфизм.

## Первая простая программа

Теперь, когда объектно-ориентированные основы языка Java были обсуждены, посмотрим на некоторые реальные программы. Начнем с компиляции и выполнения короткой программы, представленной ниже. Как вы увидите, она включает немного больше работы, чем можно было бы представить.

```
/*
Это простая Java-программа.
Назовите этот файл "Example.java".
*/
class Example {
    // Программа начинается с вызова main().
    public static void main(String args[]) {
        System.out.println("Это простая Java-программа.");
    }
}
```

### Замечание

Последующие описания используют стандартный набор инструментальных средств разработчика JDK (Java Developer's Kit) фирмы Sun Microsystems. Если вы используете другую среду разработки, то, возможно, придется следовать иным процедурам для компиляции и выполнения Java-программ.

## Ввод программы

Для большинства компьютерных языков имя файла, который содержит исходный код программы, является произвольным. Для языка Java это не так. Первое, что вы должны изучить в языке Java, это то, что имя, которое вы

присваиваете исходному файлу, очень важно. Например, имя исходного файла должно быть Example.java. Посмотрим почему.

В Java исходный файл официально называют *единицей компиляции* (compilation unit). Это текстовый файл, который содержит одно или несколько определений классов. Компилятор Java требует, чтобы исходный файл использовал расширение .java. Заметим, что расширение имени файла имеет длину в четыре символа. Как вы можете заметить, ваша операционная система должна поддерживать длинные имена файлов. Это означает, что DOS и Windows 3.1 не способны поддерживать язык Java (по крайней мере, в настоящее время). Однако Windows 95/98 и Windows NT имеют такую возможность.

Глядя на программу можно заметить, что имя класса, определенное в программе, тоже Example. Это не совпадение. В Java весь код должен находиться внутри класса. По соглашению имя этого класса должно быть согласовано с именем файла, который содержит программу. Вы должны также убедиться в согласовании прописных букв в именах файла и класса. Причина в том, что язык Java чувствителен к регистру клавиатуры. Здесь соглашение о том, чтобы имена файлов соответствовали именам классов, может показаться произволом. Однако это соглашение делает более простым поддержку и организацию ваших программ.

## Компиляция программы

Чтобы откомпилировать программу Example, запустите компилятор javac, указав в параметре командной строки имя исходного файла:

```
C:\>javac Example.java
```

Компилятор javac создает файл с именем Example.class, который содержит программу в виде байт-кода. Как обсуждалось ранее, байт-код Java — это промежуточное представление программы, состоящее из инструкций, которые будет выполнять интерпретатор Java. Таким образом, результат работы компилятора javac не является непосредственно выполняемым кодом.

Для действительного выполнения программы следует использовать Java-интерпретатор с именем java. Командная строка запуска интерпретатора выглядит так:

```
C:\>java Example
```

После выполнения программы на экран выводится строка:

Это простая Java-программа.

После компиляции исходного кода каждый индивидуальный класс помещается в собственный выходной файл, имя которого совпадает с именем этого класса, и расширением .class. Присвоение исходному файлу того же имени, что и содержащемуся в нем классу — неплохая идея, так как имя исходного

файла будет согласовано с именем class-файла. Когда вы запускаете интерпретатор Java, вы в действительности специфицируете имя класса, который должен исполнить интерпретатор. Он автоматически отыскивает файл с тем же именем и расширением .class. Если интерпретатор находит такой файл, то он выполняет код, содержащийся в указанном классе.

## Подробный взгляд на первую программу

Хотя программа Example.java очень короткая, она содержит некоторые ключевые свойства, общие для всех Java-программ. Рассмотрим поближе каждую часть программы.

```
/*
Это простая Java-программа.
Назовите этот файл "Example.java".
*/
```

Это *комментарий*. Как и большинство других языков программирования, Java позволяет вводить замечания в исходный программный файл. Содержимое комментария игнорируется компилятором. Комментарий описывает и объясняет работу программы всем, кто читает исходный код. В нашем случае комментарий описывает программу и напоминает, что исходный файл должен быть назван Example.java. Конечно, в реальных приложениях комментарий, в общем случае, объясняет, как работает некоторая часть программы или какими специфическими свойствами она обладает.

Java поддерживает три стиля комментариев. Первый, показанный в начале нашей программы, называют *многострочным комментарием*. Этот тип комментария должен начинаться символами /\* и заканчиваться \*/. Все символы, находящиеся между этими парами, игнорируются компилятором. Как понятно из названия, многострочный комментарий может состоять из нескольких строк.

Следующая строка кода программы:

```
class Example {
```

Эта строка использует ключевое слово `class` для объявления, что определяет новый класс. Example — это идентификатор, являющийся именем класса. Полное определение класса, включающее все его члены, размещается между открывающей ({) и закрывающей (}) фигурными скобками. Использование фигурных скобок в Java аналогично способу их использования в C и C++.

Сейчас не очень беспокойтесь о деталях класса, за исключением того, что в Java вся программная активность происходит внутри класса. Это одна из причин, почему Java-программы (хотя бы немного) объектно-ориентированы.

Следующая строка в программе — *однострочный комментарий*:

```
// Программа начинается с вызова main().
```

Это второй тип комментариев, поддерживаемых в Java. Однострочный комментарий начинается символами // и завершается в конце строки. Вообще, программисты используют многострочные комментарии для длинных заметок, а однострочные — для кратких построчных описаний.

Следующая строка кода:

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

Эта строка заголовка метода main(). Как подсказывает предшествующий комментарий, отсюда начнется выполнение программы. Все приложения Java начинают выполнение с вызова main(). (Точно так же, как в C/C++.) Определенного значения каждой части этой строки сейчас дать нельзя потому, что это требует детального понимания подхода Java к инкапсуляции. Однако, поскольку большинство примеров Части I этой книги будет использовать подобную строку кода, давайте бросим беглый взгляд на каждую часть сейчас.

Ключевое слово public — это *спецификатор доступа*, который позволяет программисту управлять видимостью членов класса. Когда члену класса предшествует public, то к этому члену возможен доступ из кода, внешнего по отношению к классу, в котором данный метод описан. (Противоположностью public является ключевое слово private, которое предохраняет член от использования кодом, определенным вне данного класса.) В рассматриваемом случае main() необходимо объявить как public, т. к. он должен вызываться кодом извне своего класса при старте программы. Ключевое слово static позволяет методу main() быть вызванным без наличия экземпляров специфического класса. Оно необходимо, т. к. main() вызывается интерпретатором Java перед тем, как создается какой-либо объект. Ключевое слово void просто говорит компилятору, что main() не возвращает значения. Как вы увидите далее, методы могут также и возвращать значения. Если все это кажется немного сбивающим с толку, не волнуйтесь. Все эти концепции будут обсуждены детально в следующих главах.

Итак, main() — это метод,ываемый, когда Java-приложение начинает выполняться. Имейте в виду, что Java чувствителен к регистру. Таким образом, Main отличается от main. Важно также понимать, что компилятор Java будет компилировать классы, которые не содержат метода main(). Но у него нет способа выполнять эти классы. Так, если бы вы ввели Main вместо main, компилятор еще бы откомпилировал вашу программу. Однако интерпретатор Java сообщил бы об ошибке потому, что у него не было бы возможности найти метод main().

Любая информация, которую нужно передать методу, принимается переменными, специфицированными в скобках, следующих за именем метода. Эти переменные называют *параметрами*. Если нет параметров, необходимых для данного метода, все равно нужно указывать пустые скобки. Для main() существует только один параметр, хотя и сложный. String args[] объявляет параметр с именем args, являющийся массивом экземпляров класса String.

(Массив — это набор однотипных объектов.) Объекты типа `String` хранят символьные строки. В данном случае массив `args` принимает любые аргументы из командной строки, запускающей программу на выполнение. Данная программа не использует эту информацию, но примеры, представленные далее в этой книге, будут использовать.

Последний символ в строке это `{`. Он отмечает начало тела `main()`. Весь код, который содержит метод, будет располагаться между открывающей и закрывающей фигурными скобками.

С другой точки зрения, `main()` — это просто место старта для интерпретатора. Сложная программа будет иметь много классов, но только в одном из них нужно указать метод `main()`, чтобы запустить систему. Когда вы начинаете создание апплетов (Java-программ, которые внедрены в Web-браузеры), вы вообще не используете `main()`, т. к. Web-браузер пользуется иными средствами начала выполнения апплетов.

Следующая строка кода (заметим, она находится внутри `main()`):

```
System.out.println("Это простая Java-программа.");
```

Она выводит на экран строку "Это простая Java-программа.", за которой следует строка новой команды. Вывод в действительности выполняется встроенным методом `println()`. В данном случае `println()` выводит на экран строку, которая ей переслана. Далее вы увидите, что `println()` можно использовать также для показа других типов информации. Стока начинается с `System.out`. Пока слишком сложно объяснять это в деталях, но, если кратко, то `System` — это предопределенный класс, который обеспечивает доступ к системе, а `out` — выходной поток, который соединен с консолью.

Как вы уже, вероятно, догадались, консольный вывод (и ввод) не используется часто в реальных Java-программах и апплетах. Так как наиболее современные компьютерные среды используют окна и являются графическими по природе, консоль ввода/вывода (В/В) применяется, в основном, для простых, вспомогательных и демонстрационных программ. Позднее в этой книге вы изучите другие способы генерации вывода, используя язык Java. Но сейчас мы продолжим использование консольных методов ввода/вывода.

Заметим, что утверждение `println()` заканчивается точкой с запятой `(;)`. Все утверждения в Java заканчиваются этим символом. Строки, которые не являются утверждениями, данным символом не заканчиваются.

Первая скобка `}` в программе заканчивает `main()`, а вторая скобка `}` завершает определение класса `Example`.

## Вторая короткая программа

Вероятно, нет другой более фундаментальной для языка программирования концепции, чем переменная. Как вы, наверное, знаете, *переменная* — это именованная область памяти, в которой ваша программа может установить

некоторое значение. Значение переменной может изменяться во время выполнения программы. Следующая программа показывает, как переменная определяется и как ей приписывается некоторое значение. Дополнительно программа также иллюстрирует новые аспекты консольного вывода.

```
/*
Другой короткий пример.
Назовите этот файл "Example2.java".
*/
class Example2 {
    public static void main(String args[]) {
        int num; // объявляет переменную с именем num
        num = 100; // присваивает num значение 100
        System.out.println("Значение num: " + num);
        num = num * 2;
        System.out.print("Удвоенное значение num: ");
        System.out.println(num);
    }
}
```

Когда вы выполните эту программу, то получите следующий вывод:

```
Значение num: 100
Удвоенное значение num: 200
```

Посмотрим внимательно, как получен этот вывод. Первая новая строка в этой программе:

```
int num; // объявляет переменную с именем num
```

В ней объявляется целочисленная переменная с именем num. Java (подобно большинству других языков) требует, чтобы переменные объявлялись *перед* тем, как они используются.

Общая форма объявления переменной:

```
type var-name;
```

где **type** специфицирует тип объявляемой переменной, а **var-name** — имя этой переменной. Если вам нужно объявить более одной переменной указанного типа, то можно использовать разделенный запятыми список имен. В Java определены несколько типов данных, включая *целый* (integer), *символьный* (character) и *с плавающей точкой* (floating-point). Ключевое слово **int** специфицирует целый тип.

В нашей программе строка

```
num = 100; // присваивает num значение 100
```

присваивает переменной num значение 100. В языке Java оператор присваивания выражается одиночным знаком равенства (=).

Следующая строка кода выводит значение `num`, предваряя его строкой "Значение `num`".

```
System.out.println("Значение num: " + num);
```

В этом утверждении знак плюс (+) добавляет к предшествующей строке значение `num`, а затем результирующая строка выводится. (В действительности, `num` сначала преобразуется из целого типа в строчный эквивалент и затем склеивается с предшествующей строкой. Далее в этой книге подобный процесс описывается детально.) Данной подход можно обобщить. Используя операцию +, вы можете склеивать вместе в одном операторе `println()` несколько operandов.

Следующая строка кода назначает переменной `num` значение `num`, умноженное на 2. Как и большинство других языков, чтобы указать умножение, Java использует операцию \*. После выполнения этой строки кода `num` будет содержать значение 200.

Две следующие строки в программе:

```
System.out.print("Удвоенное значение num: ");
System.out.println(num);
```

Здесь два новых момента. Во-первых, для вывода на экран строки "Удвоенное значение `num`:" используется встроенный метод `print()`. За этой строкой не следует символ newline (перевод строки). Это означает, что следующий вывод будет начинаться на той же строке. Метод `print()` очень похож на `println()` за исключением того, что он не выводит символа newline после каждого вызова. Теперь взгляните на обращение к `println()`. Обратите внимание, что аргументом здесь является имя переменной `num`. Как `print()`, так и `println()` можно использовать для вывода любого встроенного типа языка Java.

## Два оператора управления

Хотя в главе 5 операторы управления будут рассмотрены подробно, два из них приводятся здесь, чтобы их можно было использовать в примерах программ в главах 3 и 4. Они помогут также иллюстрировать важный аспект Java: блоки кодов.

### Оператор `if`

Оператор `if` языка Java во многом работает подобно оператору `if` в любом другом языке. Он синтаксически идентичен оператору `if` в С и C++. Простейшая форма этого оператора выглядит так:

```
if(condition) statement;
```

Здесь **condition** представляет собой логическое выражение. Если оно истинно (true), то **statement** исполняется. Если **condition** ложно (false), то **statement** пропускается. Например:

```
if(num < 100) println("num is less than 100");
```

В данном случае, если num содержит значение, которое меньше, чем 100, то условное выражение истинно (true), и println() будет выполняться. Если num содержит значение, большее или равное 100, то метод println() пропускается.

Как вы увидите в главе 4, Java определяет полный набор операций отношения, которые можно использовать в условных выражениях. Некоторые из них представлены в табл. 2.1.

**Таблица 2.1. Некоторые операции отношений в Java**

Операция отношения	Значение
<	Меньше чем
>	Больше чем
==	Равно

Заметьте, что проверка равенства обозначается *двойным* знаком равенства.

Приведем пример программы, иллюстрирующей оператор if:

```
/* демонстрирует оператор if.
```

Назовите этот файл "IfSample.java".\*/

```
class IfSample {
    public static void main(String args[]) {
        int x, y;
        x = 10;
        y = 20;
        if(x < y) System.out.println("x меньше чем y");
        x = x * 2;
        if(x == y) System.out.println("x равно y");
        x = x * 2;
        if(x > y) System.out.println("x больше чем y");
        // следующий оператор не выводит свое сообщение
        if(x == y) System.out.println("вы ничего не увидите");
    }
}
```

Вывод, генерируемый этой программой:

х меньше чем у  
х равно у  
х больше чем у

Обратите внимание еще на одну строку:

`int x, y;`

В ней объявлены две переменные, `x` и `y`, разделенные запятой.

## Цикл `for`

Как вы, возможно, знаете из своего прошлого программистского опыта, оператор цикла (`loop`) является важной частью почти любого языка программирования. Java — не исключение. В действительности, как вы увидите в главе 5, Java обеспечивает мощный ассортимент конструкций циклов. Вероятно, наиболее гибким является цикл `for`. Если вы знакомы с C или C++, то вам приятно будет узнать, что цикл `for` в Java работает так же, как в этих языках. Если вы не знаете C/C++, то использовать цикл `for` еще проще.

Формат цикла `for`:

`for(initialization; condition; iteration) statement;`

В этой наиболее общей форме инициализирующая часть цикла устанавливает начальное значение переменной цикла. Логическое выражение `condition` проверяет эту переменную. Если результат проверки — истина (`true`), цикл `for` продолжает итерации, если — ложь (`false`), цикл завершается. Выражение `iteration` определяет, как переменная цикла изменяется на каждом шаге выполнения цикла. Приведем пример программы, иллюстрирующей цикл `for`:

```
/* Демонстрирует цикл for.
   Назовите этот файл "ForTest.java".*/
class ForTest {
    public static void main(String args[]) {
        int x;

        for(x = 0; x<10; x = x+1)
            System.out.println("Значение x: " + x);
    }
}
```

Программа генерирует следующий вывод:

Значение x: 0

Значение x: 1

```
Значение x: 2
Значение x: 3
Значение x: 4
Значение x: 5
Значение x: 6
Значение x: 7
Значение x: 8
Значение x: 9
```

В этом примере `x` — это переменная цикла. Ей присваивается ноль в инициализирующей части `for`. В начале каждой итерации, включая первую, осуществляется проверка условия `x < 10`. Если результат проверки `true`, выполняется оператор `println()`, и затем значение переменной `x` увеличивается. Этот процесс продолжается до тех пор, пока условие не станет ложным (`false`).

Интересно, что в написанных профессионально Java-программах вы почти никогда не увидите итерационную часть цикла записанной, как в предшествующей программе. То есть, вы редко встретите оператор, подобный:

```
x = x + 1;
```

Причина в том, что Java предоставляет специальную инкрементную операцию, которая выполняет данное увеличение более эффективно. Инкрементная операция обозначается как `++` (два символа "плюс" рядом) и увеличивает свой операнд на единицу. Используя указанную операцию, предыдущий оператор можно записать как:

```
x++
```

Таким образом, цикл `for` в предыдущей программе будет записан так:

```
for(x = 0; x<10; x++)
```

Если проверить работу этого цикла, то можно убедиться, что он выполняется точно так же, как и прежде.

В Java, кроме того, имеется декрементная операция, которая обозначается как `--`. Она уменьшает значения операнда на единицу.

## Использование блоков кода

Java допускает группировку двух или более операторов в *блоки кода* (*blocks of code*), называемые также *кодовыми блоками* (*code blocks*). Группировка выполняется обрамлением операторов фигурными скобками. Как только блок кода создан, он становится логическим целым, которое можно использовать в любом месте, где может быть указан лишь единственный оператор. Например, блок может быть на месте операнда `statement` в конструкциях операторов `if` и `for`. Посмотрите на следующий пример:

```
if(x < y) { // начало блока
    x = y;
    y = 0;
} // конец блока
```

Если  $x$  меньше, чем  $y$ , то оба оператора внутри блока будут выполнены. Таким образом, два оператора внутри блока образуют логическое целое, и один оператор нельзя выполнять без другого. Здесь важно, что всякий раз, когда необходимо связать логически два и более операторов, создается блок.

Рассмотрим другой пример. Следующая программа использует блок кода в теле цикла `for`.

```
/* Демонстрирует блок кода.
   Назовите этот файл "BlockTest.java" */
class BlockTest {
    public static void main(String args[]) {
        int x, y;
        y = 20;

        // телом этого цикла является блок
        for(x = 0; x<10; x++) {
            System.out.println("Значение x: " + x);
            System.out.println("Значение y: " + y);
            y = y - 2;
        }
    }
}
```

Вывод программы:

```
Значение x: 0
Значение y: 20
Значение x: 1
Значение y: 18
Значение x: 2
Значение y: 16
Значение x: 3
Значение y: 14
Значение x: 4
Значение y: 12
Значение x: 5
Значение y: 10
Значение x: 6
Значение y: 8
Значение x: 7
Значение y: 6
Значение x: 8
```

Значение у: 4

Значение х: 9

Значение у: 2

В этом случае тело цикла `for` содержит блок кода, а не одиночный оператор. Таким образом, во время каждой итерации цикла будут выполняться три оператора внутри блока. Этот факт, конечно, и подтверждает вывод, сгенерированный программой.

Как вы увидите позже в этой книге, блоки кода обладают дополнительными свойствами и применениями.

## Лексические вопросы

Теперь, когда рассмотрены некоторые короткие Java-программы, самое время формально описать мельчайшие элементы языка Java. Java-программа — это набор пробельных символов (whitespace), идентификаторов, комментариев, констант, операций, разделителей и ключевых слов. Операции описаны в следующей главе. Остальное — здесь.

## Пробельные символы

Java является языком *свободной формы* (free-form language<sup>1</sup>). Это означает, что вам не нужно следовать каким бы то ни было правилам структурированного расположения текста. Например, как бы программа `Example` не была введена — вся единой строкой или размещена на экране любым другим способом, который вам понравился при вводе — в ней всегда существует хотя бы один пробельный символ между элементами записи исходного кода. В языке Java к пробельным символам относятся пробел (space), символ табуляции (tab) или новой строки (newline).

## Идентификаторы

Идентификаторы используются в качестве имен классов, методов и переменных. Идентификатор может быть любой последовательностью букв верхнего и нижнего регистров, чисел или символов подчеркивания и знака коммерческого S (\$). Он не должен начинаться с цифры, чтобы не вступать в конфликт с числовой константой. Напомним также, что язык Java чувствителен к регистру, так что `value` есть идентификатор, отличающийся от `Value`. Несколько примеров правильных идентификаторов:

`AvgTemp`      `count`      `a4`      `$test`      `this_is_ok`

<sup>1</sup> Free-form language — язык свободной формы (синтаксис которого не связан позиционными или форматными ограничениями). — Примеч. пер.

## Неправильные имена:

2count            high-temp            Not/ok

## Константы

Постоянные значения в Java создаются с использованием их **литерального представления**. Вот несколько констант:

100            98.6            'X'            "This is a test"

Первая константа специфицирует целое число, следующая — числовое значение с плавающей точкой, третья — символ и последняя — строку. Константу можно использовать **везде**, где допустимо значение ее типа.

## Комментарии

Как говорилось ранее, в Java определены несколько типов комментариев. Два вы уже видели: односторонний и многострочный. Третий тип называется **документационный комментарий** (*documentation comment*). Этот тип комментария используется для производства HTML-файла, который документирует вашу программу. Документационный комментарий начинается с последовательности символов `/**` и заканчивается последовательностью `*/`.

## Разделители

В Java существуют несколько символов, которые используются как разделители. Чаще всего встречается точка с запятой `(;)`. Как вы уже видели, он используется для завершения оператора. Разделители представлены в табл. 2.2.

**Таблица 2.2. Разделители, применяемые в Java**

Символ	Имя	Цель
<code>()</code>	Круглые скобки	Используются, чтобы ограничивать списки параметров в определениях методов и вызовах процедур. Используются также для определения предшествования в выражениях, содержат выражения в управляемых операторах и в операторе приведения типов
<code>{ } [ ]</code>	Фигурные скобки Квадратные скобки	Используются для ограничения значений автоматически инициализированных массивов, а также для определения блоков кода в классах, методах и локальных областях действия
	Квадратные скобки	Используются в объявлении массивов, а также для указания элементов массивов

Таблица 2.2 (окончание)

Символ	Имя	Цель
;	Точка с запятой	Завершает оператор
,	Запятая	Разделяет последовательные идентификаторы в объявлении переменных, а также цепочку выражений внутри оператора for
.	Точка	Используется для разделения имен пакетов и классов, а также для отделения переменной или метода от ссылочной переменной

## Ключевые слова языка Java

Существуют сорок восемь зарезервированных ключевых слов, определенных в настоящее время в языке Java (табл. 2.3). Эти ключевые слова, объединенные с синтаксисом операторов и разделителей, формируют определение языка Java. Ключевые слова нельзя использовать в качестве имен переменных, классов или методов.

Таблица 2.3. Зарезервированные ключевые слова Java

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

Ключевые слова const и goto зарезервированы, но не используются. В ранних версиях языка Java некоторые другие ключевые слова были зарезервированы для будущего использования. Однако текущие спецификации языка Java определяют только ключевые слова, показанные в табл. 2.3.

В дополнение к ключевым в Java зарезервированы следующие слова: true, false и null. Это значения, определенные в Java. Не следует использовать данные слова в качестве имен переменных, классов и т. п.

## Библиотеки классов языка Java

Примеры программ, показанные в этой главе, используют два встроенных Java-метода: `println()` и `print()`. Как уже упоминалось, эти методы являются членами класса `System`, который предопределен в Java и автоматически включается в ваши программы. Если смотреть шире, среда Java зависит от нескольких встроенных библиотек классов, содержащих много встроенных методов, которые обеспечивают поддержку ввода/вывода, обработку строк, работу в сетях и графику. Стандартные классы также обеспечивают поддержку для оконного (графического) вывода. Таким образом, Java в целом — это комбинация собственно языка Java с его стандартными классами. Как будет видно далее, библиотеки классов обеспечивают многое из функциональных возможностей, которые приходят вместе с языком Java. Действительно, обучение использованию стандартных классов Java весьма привлекает Java-программиста. На протяжении Части I этой книги упоминаются, по мере необходимости, различные элементы стандартных библиотек классов и методов. В Части II библиотеки классов описаны подробно.



## ГЛАВА 3

# Типы данных, переменные и массивы

В этой главе рассматриваются три фундаментальных элемента Java: типы данных, переменные и массивы. Как и все языки современного программирования, Java поддерживает несколько типов данных. Вы можете использовать их, чтобы объявлять переменные и создавать массивы. Подход Java к этим элементам ясен, эффективен и связан.

## Java — язык со строгой типизацией

Важно заявить в начале, что Java — это язык со строгой типизацией. Действительно, часть безопасности и устойчивости к ошибкам Java исходит из этого факта. Давайте разберемся, что это означает. Во-первых, каждая переменная и каждое выражение имеют тип, и каждый тип строго определен. Во-вторых, все назначения — явные или через передачу параметров в вызовах методов, проверяются на совместимость типов. Не имеется никаких автоматических приведений или преобразований конфликтующих типов, как в некоторых языках. Компилятор Java проверяет все выражения и параметры, чтобы гарантировать, что типы совместимы. Любые несоответствия типов — ошибки, которые должны быть исправлены прежде, чем компилятор закончит компилировать класс.

### Замечание

Если вы переходите к Java от C или C++, имейте в виду, что Java более строго типизирован, чем любой язык. Например, в C/C++ можно присвоить значение с плавающей точкой целому числу. В Java — нельзя. В С также нет обязательного строгого контроля соответствия типов между параметром и аргументом функции. В Java — есть. Сначала вы могли бы найти строгую проверку соответствия типов в Java ненужного утомительной. Но помните, в конечном счете это поможет уменьшить возможность ошибок в вашем коде.

## Простые типы

Java определяет восемь простых (или элементных) типов данных: `byte`, `short`, `int`, `long`, `char`, `float`, `double` и `boolean`. Их можно объединить в четыре группы.

- **Целые (Integers).** Эта группа включает типы `byte`, `short`, `int` и `long`, которые являются полнозначными целыми числами со знаком.
- **Числа с плавающей точкой (Floating-point number).** Эта группа включает типы `float` и `double`, которые представляют числа с дробной точностью.
- **Символы (Characters).** Эта группа включает тип `char`, который представляет символы в наборе символов, подобные буквам и цифрам.
- **Логические или булевские (Boolean).** Эта группа включает тип `boolean`, который является специальным типом для представления значений `true/false` (истина/ложь).

Вы можете использовать указанные типы сами по себе, создавать массивы или ваши собственные классы типов. Таким образом, они формируют основу для всех других типов данных, которые вы можете создавать.

Простые типы представляют одиночные (не множественные) значения. Хотя Java во всем остальном полностью объектно-ориентирован, простые типы не являются таковыми. Они аналогичны простым типам, которые можно найти в большинстве других не объектно-ориентированных языков. Причина этого кроется в эффективности. Выполнение простых типов в виде объектов слишком ухудшило бы эффективность.

Простые типы определены так, чтобы знать их явный диапазон и математическое поведение. Языки типа C и C++ позволяют значению целого числа изменяться, основываясь на диктате среды выполнения. Однако Java — другой язык. Из-за требования мобильности, все типы данных имеют строго определенный диапазон. Например, `int` — всегда 32-битный, независимо от специфической платформы. Это позволяет программам быть записанными так, чтобы гарантировать ее выполнение *без перевода* на любой машинной архитектуре. Тогда как строгая спецификация размера целого числа может вызывать небольшую потерю производительности, в некоторых средах это необходимо, чтобы достичь мобильности.

Рассмотрим каждый тип данных отдельно.

### Целые типы

Java определяет четыре целых типа: `byte`, `short`, `int` и `long`. Все они со знаками — положительные и отрицательные значения. Java не поддерживает беззнаковых и только положительных целых. (Однако существуют машин-

ные языки, включая C/C++, поддерживающие целые числа как со знаком, так и без знака.) Не смотря на это, проектировщики Java чувствовали, что целые числа без знака все-таки ненужны. В частности, что концепция *целых без знака* (*unsigned*) использовалась главным образом, чтобы специфицировать поведение старшего бита, который определял *знак* целого типа (*int*), когда он выражен как число. Как вы увидите в главе 4, Java управляет значением старшего бита иначе, предоставив специальную операцию "беззнакового правого сдвига". Таким образом, потребность в типе целого числа без знака была устранена.

О *ширине* (или количестве бит, отводимых для хранения значения) целочисленного типа нельзя думать как о количестве памяти, которую он занимает, а скорее, как о *поведении*, которое она определяет для переменных и выражений этого типа. Исполнительная среда Java вольна использовать любой размер, какой она хочет, тогда как типы ведут себя согласно их объявлению. Существует, по крайней мере, одна реализация исполнительной среды, которая хранит числа типа *byte* и *short* как 32-разрядные (а не 8- и 16-разрядные) значения, чтобы улучшить эффективность, потому что этим значением выражается размер слова большинства используемых в настоящее время компьютеров.

Ширина и диапазоны этих целых типов широко изменяются, как показано в табл. 3.1.

**Таблица 3.1. Ширина и диапазоны целых типов данных**

Имя	Ширина	Диапазон
<i>long</i>	64	от -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807
<i>int</i>	32	от -2,147,483,648 до 2,147,483,647
<i>short</i>	16	от -32,768 до 32,767
<i>byte</i>	8	от -128 до 127

Рассмотрим каждый тип целого числа.

## Тип *byte*

Тип *byte* — самый маленький целый тип. Это 8-разрядный тип со знаком, значения которого изменяются в диапазоне от -128 до 127. Переменные типа *byte* особенно полезны, когда вы работаете с сетевым или файловым потоком данных. Они также могут использоваться при работе с рядами двоичных данных, которые не могут быть непосредственно совместимы с другими встроенными типами Java.

Байтовые переменные объявляются при помощи ключевого слова `byte`. Например, следующее предложение объявляет две байтовые переменные с именами `b` и `c`:

```
byte b, c;
```

## Тип `short`

Тип `short` — 16-разрядный тип данных со знаком. Диапазон изменения его значений от  $-32,768$  до  $32,767$ . Пожалуй, это наименее используемый тип Java, так как он определен со старшим байтом в начале (в так называемом формате `big-endian`). Этот тип главным образом применим к 16-разрядным компьютерам, которые становятся все более и более редкими.

Некоторые примеры объявлений `short` переменных:

```
short s;  
short t;
```

### Замечание

Термин "Endianness" описывает, как многобайтовые типы данных, такие как `short`, `int` и `long`, хранятся в памяти. Предположим, имеется два байта для представления типа `short`. Который из них идет первым — наиболее или наименее значащий? Термин "big-endian" означает, что наиболее значащий байт хранится первым, за ним следует наименее значащий. Машины типа SPARC и PowerPC можно назвать машинами `big-endian`, в то время как машины серии Intel x86 — `little-endian` (т. к. первым хранится менее значащий байт).

## Тип `int`

Чаще всего используется тип `int`. Это 32-разрядный тип со знаком, который имеет диапазон от  $-2,147,483,648$  до  $2,147,483,647$ . В дополнение к другим применению, переменные типа `int` обычно используются для управления циклами и индексирования элементов массива. Всякий раз, когда появляется целочисленное выражение, включающее операнды типа `byte`, `short`, `int` и целочисленные константы, тип полного выражения *расширяется* (или *повышается*) до типа `int` прежде, чем выполняется вычисление.

Тип `int` наиболее универсален и эффективен и должен быть использован для расчетов, индексации элементов массива или выполнения целочисленных операций. Может показаться, что использование `short` или `byte` экономит память, но нет никакой гарантии, что Java не будет внутренне так или иначе расширять эти типы до `int`. Помните, что тип определяет поведение, а не размер. (Единственное исключение — массивы, где тип `byte` гарантирует использование только одного байта на элемент массива, в то время как `short` будет забирать два байта, а `int` — четыре байта.)

## Тип *long*

Тип *long* — 64-разрядный тип со знаком. Он полезен в тех случаях, когда тип *int* недостаточен (по размеру памяти), чтобы хранить желаемое значение. Диапазон значений *long* весьма велик. Это делает его полезным при работе с большими целыми числами. Например, вот программа, которая вычисляет мили, пройденные светом за указанное число дней.

```
// Вычисляет расстояние, пройденное световым сигналом,
// используя long-переменные.
class Light {
    public static void main(String args[]) {
        int lightspeed;
        long days;
        long seconds;
        long distance;

        // аппроксимация скорости света в милях в секунду
        lightspeed = 186000;
        days = 1000;                      // задать количество дней
        seconds = days * 24 * 60 * 60;     // преобразовать в секунды
        distance = lightspeed * seconds;  // вычислить расстояние

        System.out.print("За " + days);
        System.out.print(" дней световой сигнал пройдет около ");
        System.out.println(distance + " миль.");
    }
}
```

Эта программа генерирует следующий вывод:

За 1000 дней световой сигнал пройдет около 16070400000000 миль.

Ясно, что результат не мог быть получен в *int*-переменной.

## Типы с плавающей точкой

Числа с плавающей точкой, также известные как *вещественные* числа, используются при оценке выражений, которые требуют дробной точности. Например, вычисления квадратного корня или трансцендентных функций типа синуса и косинуса приводят к вещественному значению с определенной точностью, что и требует типа с плавающей точкой. Java реализует стандартный (IEEE-754) набор типов с плавающей точкой и соответствующие операции с ними. Существует два вида типов с плавающей точкой *float* и *double*, которые представляют числа с одинарной и двойной точностью, соответственно. Их ширина и диапазоны представлены в табл. 3.2.

**Таблица 3.2. Ширина и диапазоны типов с плавающей точкой**

Имя	Ширина в битах	Диапазон
double	64	от 1.7e-308 до 1.7e+308
float	32	от 3.4e-038 до 3.4e+038

Рассмотрим каждый из этих типов.

## Тип *float*

Тип *float* определяет значение с одинарной точностью, которое использует 32 бита памяти. Вычисления с одинарной точностью быстрее на некоторых процессорах, а значения занимают половину пространства значений двойной точности. Но тип оказывается неточным, когда его значения становятся или очень большими, или очень маленькими. Переменные типа *float* полезны, когда нужен дробный компонент, но не требуется большой степени точности. Например, тип *float* может быть полезен при представлении долларовых цен с учетом центов.

Пример объявления *float*-переменных:

```
float hightemp, lowtemp;
```

## Тип *double*

Тип двойной точности, обозначаемый ключевым словом *double*, для хранения значений использует 64 бита памяти. На некоторых современных процессорах, которые были оптимизированы для высокоскоростных математических вычислений, операции с двойной точностью на самом деле выполняются быстрее, чем с одинарной. Все трансцендентные математические функции, типа *sin()*, *cos()* и *sqrt()*, возвращают *double*-значения. Когда нужно поддерживать точность во многих итерационных вычислениях или манипулировать с многозначными числами, *double* — лучший выбор.

Ниже представлена короткая программа, которая использует переменные двойной точности, чтобы вычислить площадь круга.

```
// Вычислить площадь круга.
class Area {
    public static void main(String args[]) {
        double pi, r, a;
        r = 10.8; // радиус круга
        pi = 3.1416; // ПИ приближенно
        a = pi * r * r; // вычисляет площадь
    }
}
```

```

        System.out.println("Площадь круга равна " + a);
    }
}

```

## Символьный тип (*char*)

Для хранения символов Java использует тип *char*. Однако, программисты C/C++, осторегайтесь: тип *char* в Java — не тот же, что в С или С++. В C/C++ *char* — целочисленный тип шириной в 8 бит. В Java это не так. Вместо этого, для представления символов Java использует Unicode. *Unicode* определяет полный набор интернациональных символов, который может представлять все символы, находящиеся во всех человеческих языках. Он объединяет множество наборов символов, таких как латинский, греческий, арабский, кириллица, иврит, Katakana, Hangul и многие другие. Для этой цели он требует 16 бит. Таким образом, *char* в Java — 16-разрядный тип данных. Диапазон его значений варьируется от 0 до 65 536. Нет никаких отрицательных символов. Стандартный набор символов, известный как ASCII, все еще располагается в интервале значений от 0 до 127, как обычно, а расширенный 8-разрядный набор символов, ISO-Latin-1, в диапазоне от 0 до 255. Так как язык Java разработан для записи апплетов "всемирного использования", то имеет смысл, чтобы для представления символов он использовал Unicode. Конечно, использование Unicode несколько неэффективно для языков, подобных английскому, немецкому, испанскому или французскому, чьи символы могут легко содержаться в 8 битах. Но такова цена, которая должна быть заплачена за глобальную мобильность.

### Замечание

Более полную информацию о Unicode можно найти по адресу <http://www.unicode.org>.

Вот программа, которая демонстрирует *char*-переменные:

```

// Демонстрирует тип данных char.
class CharDemo {
    public static void main(String args[]) {
        char ch1, ch2;

        ch1 = 88;                                // код для X
        ch2 = 'Y';

        System.out.print("ch1 и ch2: ");
        System.out.println(ch1 + " " + ch2);
    }
}

```

Эта программа отображает на экране следующий результат:

```
ch1 и ch2: X Y
```

Обратите внимание, что переменной ch1 назначено значение 88, которое является значением ASCII (и Unicode) и соответствует букве X. Как говорилось выше, набор символов ASCII занимает первые 127 значений в наборе символов Unicode. По этой причине, все "старые уловки", которые вы использовали с символами в прошлом, будут работать и в Java тоже.

Хотя символы — не целые числа, во многих случаях можно оперировать с ними, как если бы они были таковыми. Это позволяет складывать два символа вместе, или инкрементировать (увеличивать на 1) значение символьной переменной. Например, рассмотрим следующую программу:

```
// Переменные символа ведут себя подобно целым числам.
class CharDemo2 {
    public static void main(String args[]) {
        char ch1;

        ch1 = 'X';
        System.out.println("ch1 содержит " + ch1);

        ch1++;           // увеличили на 1 значение переменной ch1
        System.out.println("ch1 сейчас содержит " + ch1);
    }
}
```

Вывод, генерируемый программой:

```
ch1 содержит X
ch1 сейчас содержит Y
```

В программе переменной ch1 сначала присваивается значение x. Затем ch1 инкрементируется. Это приводит к тому, что ch1 будет содержать Y — следующий символ в последовательности ASCII (и Unicode).

## Булевский тип (*boolean*)

Java имеет простой тип, называемый boolean (булев или булевский), для логических значений. Он может иметь только одно из двух возможных значений true (истина) или false (ложь). Это тип, возвращаемый всеми операциями отношений, такими как a < b. boolean — также тип, требуемый условными выражениями, которые руководят операторами управления, такими как if и for.

Вот программа, которая демонстрирует тип boolean:

```
// Демонстрирует булевые значения.
class BoolTest {
```

```

public static void main(String args[]) {
    boolean b;

    b = false;
    System.out.println("b равно " + b);
    b = true;
    System.out.println("b равно " + b);

    // булево значение может управлять if-оператором
    if(b) System.out.println("Сравнение выполнено.");

    b = false;
    if(b) System.out.println("Сравнение не выполнено.");

    // вывод операции отношения есть булево значение
    System.out.println("10 > 9 равно " + (10 > 9));
}
}

```

Вывод, сгенерированный этой программой:

```

b равно false
b равно true
Сравнение выполнено.
10 > 9 равно true

```

В этой программе есть три интересных момента. Во-первых, при выводе булева значения методом `println()` на экране отображается "true" или "false". Во-вторых, значения булевой переменной достаточно (самого по себе), чтобы управлять `if`-оператором. Не нужно записывать `if`-оператор как:

```
if(b == true) ...
```

В-третьих, результат операции отношения, такой как `<`, является булевым значением. Именно поэтому выражение `10 > 9` выводит на экран значение `true`. Далее, необходим внешний набор круглых скобок вокруг выражения `10 > 9`, потому что операция `+` (сцепления) имеет более высокий приоритет, чем операция `>` (сравнения).

## Подробнее о литералах

Литералы были упомянуты кратко в главе 2. Теперь, когда встроенные типы формально описаны, рассмотрим их подробнее.

### Целочисленные литералы

Целые числа, вероятно наиболее часто используемый тип в обычных программах. Любое полное числовое значение — целый литерал. Например, 1, 2,

3 и 42. Это все десятичные значения, т. е. числа с основанием 10. Имеются два других основания, которые можно использовать в целых литералах — восьмеричное (octal), с основанием 8, и шестнадцатеричное (hexadecimal), с основанием 16. Восьмеричные значения обозначены в Java ведущим нулем. Обычные десятичные числа не могут иметь ведущий нуль. Таким образом, казалось бы, правильное значение 09 даст ошибку компилятора, так как 9 — вне восьмеричного диапазона от 0 до 7. Более обычное основание для чисел, используемых программистами — шестнадцатеричное, которое четко согласуется с размерами слов по модулю 8, такими как 8, 16, 32 и 64 бита. Шестнадцатеричную константу обозначают с ведущими нулями (0x или 0X). Для представления шестнадцатеричного значения используются цифры от 0 до 9 и буквы латинского алфавита от A до F или от a до f (для значений от 10 до 15).

Целые литералы создают значение типа int, которое в Java является 32-разрядным целым числом. Так как Java строго типизирован, возникает вопрос, как можно назначить целый литерал одному из целых типов, такому как byte или long, без генерации ошибки несоответствия типов? К счастью, такие ситуации легко управляемы. Когда литеральное значение назначается byte- или short-переменной, никакая ошибка не генерируется, если это значение находится в пределах диапазона целевого типа. Аналогично, целый литерал может всегда назначаться переменной типа long. Однако, чтобы определить длинный литерал, нужно будет явно сообщить компилятору, что его значение имеет тип long. Это осуществляется добавлением в конец литерального значения символа L (в верхнем или нижнем регистре). Например, 0xffffffffffffffffL или 9223372036854775807L — самый большой long-литерал.

## Литералы с плавающей точкой

Числа с плавающей точкой представляют десятичные значения с дробным компонентом. Они могут быть выражены или в стандартной или научной (экспоненциальной) форме. Стандартная форма состоит из числового компонента, за которым следует десятичная точка, а далее — дробная компонента. Например, 2.0, 3.14159 и 0.6667 представляют правильные числа с плавающей точкой в стандартной системе обозначений. Научная форма представления использует стандартные обозначения — число с плавающей запятой плюс суффикс, который определяет степень 10, на которую должно быть умножено число. Экспонента обозначается буквой E или e, за которой следует положительное или отрицательное десятичное число. Например: 6.022E23, 314159E-05 и 2e+100.

По умолчанию литералы с плавающей точкой Java имеют double-точность. Чтобы определить float-литерал, следует добавить в конец его записи символ F или f. Можно также явно определить double-литерал, добавляя D или d, но подобное определение будет избыточным. По умолчанию тип double

занимает 64 бита памяти, в то время как менее точный тип с плавающей точкой требует только 32 бита.

## Булевые литералы

Булевые литералы просты. Имеются только два логических значения — `true` и `false`. Значения `true` и `false` не конвертируются в какое-либо числовое представление. Литерал `true` в Java не равняется 1, а литерал `false` не равняется 0. В Java они могут быть назначены только переменным, объявленным типом `boolean`, или использоваться в выражениях с булевскими операциями.

## Символьные литералы

Символы в Java — это индексы в наборе символов Unicode. Они являются 16-разрядными значениями, которые можно преобразовывать в целые числа и с которыми можно манипулировать целочисленными операциями, такими как сложение и вычитание. Литеральный символ представляется внутри пары одиночных кавычек. Все видимые символы ASCII могут быть непосредственно введены внутри кавычек, например: '`'a'`', '`'z'`' и '`'@'`'. Для символов, которые невозможно ввести непосредственно, существуют несколько escape-последовательностей, позволяющих ввести нужный символ, например, '`\\"`' — для самого символа одиночной кавычки, и '`\n`' — для символа `newline`. Имеется также механизм для прямого ввода значения символа в восьмеричном или шестнадцатеричном представлении. Для восьмеричной формы используют обратный слэш (`\`), за которым следует число из трех цифр. Например, '`\141`' — символ '`'a'`'. Для шестнадцатеричного представления нужно ввести обратный слэш с символом и (`\u`), затем четыре шестнадцатеричных цифры. Например, '`\u0061`' вводит символ '`'a'`' набора ISO-LATIN-1, потому что старший байт нулевой. '`\ua432`' — это японский символ Katakana. В табл. 3.3 представлены символьные escape-последовательности.

**Таблица 3.3. Escape-последовательности символов**

Escape-последовательность	Описание
<code>\ddd</code>	Восьмеричный символ (ddd)
<code>\uxxxx</code>	Шестнадцатеричный символ UNICODE (xxxx)
<code>\'</code>	Одиночная кавычка
<code>\"</code>	Двойная кавычка
<code>\\\</code>	Обратный слэш
<code>\r</code>	Возврат каретки
<code>\n</code>	Новая строка (известный так же, как перевод строки)

Таблица 3.3 (окончание)

Escape-последовательность	Описание
\f	Перевод страницы
\t	Символ табуляции (Tab)
\b	Возврат на один символ (Backspace)

## Строковые литералы

Строковые литералы в Java определяются так же, как в большинстве других языков — включением последовательности символов между парой двойных кавычек. Примеры строковых литералов:

```
"Hello World"
"two\nLines"
"\\"This is in quotes\\""
```

Escape-последовательности и восьмеричные/шестнадцатеричные системы обозначений, которые были определены для символьных литералов, работают аналогичным образом и внутри строковых литералов. Обратите внимание на одну важную деталь: строковые литералы должны начинаться и заканчиваться на той же строке. Никакой escape-последовательности продолжения строки не существует, как в других языках.

### Замечание

В большинстве других языков, включая C/C++, строки реализованы как массивы символов. Однако в Java дело обстоит не так. Строки — фактически объектные типы. Как вы увидите позднее, вследствие того, что Java реализует строки как объекты, он включает обширные возможности обработки строк, которые являются мощными и удобными в использовании.

## Переменные

Переменная — базовая единица хранения в Java-программе. Переменная определена комбинацией идентификатора, типа, и необязательного инициализатора. Кроме того, все переменные имеют *область*, которая определяет их *видимость*, и *время жизни*.

## Объявление переменной

В Java все переменные должны быть объявлены прежде, чем они могут быть использованы. Основная форма (формат) объявления переменной выглядит так:

```
type identifier [= value] [, identifier [= value] ...];
```

Здесь *type* — один из типов Java, имя класса или интерфейса. (Понятия "класс" и "интерфейс" обсуждаются позже в Части I этой книги.) *identifier* — это имя переменной. Можно инициализировать переменную, определяя знак равенства (=) и *value* (в виде литерала подходящего типа). Имейте в виду, что выражение инициализации должно привести к значению того же самого (или совместимого) типа, как и значение, которое определено для переменной. Чтобы объявлять несколько переменных указанного типа, используйте список, разделенный запятыми.

Несколько примеров объявлений переменных различных типов (обратите внимание, что некоторые включают выражения инициализации):

```
int a, b, c;           // три int-переменных a, b и с
int d = 3, e, f = 5;   // три int-переменных, d и f инициализируются
byte z = 22;           // инициализирует z
double pi = 3.14159;   // аппроксимация ПИ
char x = 'x';          // переменная x имеет значение 'x'
```

Идентификаторы, которые вы выбираете, не имеют ничего встроенного в своих именах, что бы указывало на их тип. Многие читатели помнят, когда ФОРТРАН предопределял все идентификаторы буквами от I до N, чтобы те имели тип INTEGER, в то время как все другие идентификаторы были REAL. Java позволяет любому нужным образом оформленному идентификатору иметь любой объявленный тип.

## Динамическая инициализация

Хотя предшествующие примеры использовали только константы в качестве инициализаторов, Java позволяет переменным быть инициализированными *динамически*, используя любое выражение, допустимое во время объявления переменной.

Вот, например, короткая программа, которая вычисляет длину гипотенузы прямоугольного треугольника при заданной длине двух его противоположных сторон:

```
// Демонстрирует динамическую инициализацию.
class DynInit {
    public static void main(String args[]) {
        double a = 3.0, b = 4.0;

        // переменная с динамически инициализирована
        double c = Math.sqrt(a * a + b * b);

        System.out.println("Гипотенуза равна " + c);
    }
}
```

Здесь объявлены три локальные переменные — `a`, `b` и `c`. Две первые, `a` и `b`, инициализированы константами. Однако с инициализирована динамически длиной гипотенузы (используя теорему Пифагора). Программа использует один из встроенных методов Java `sqrt()`, который является членом класса `Math`, чтобы вычислить квадратный корень его аргумента. Ключевой момент заключается в том, что выражение инициализации может использовать любой элемент, допустимый во время инициализации, включая вызовы методов, другие переменные или литералы.

## Область действия и время жизни переменных

До сих пор все используемые переменные объявлялись в начале метода `main()`. Однако Java позволяет объявлять переменные в пределах любого блока. Как было объяснено в главе 2, блок начинается с открывающей фигурной скобки и оканчивается закрывающей фигурной скобкой. Блок определяет область видимости (действия) имен переменных (scope). Таким образом, каждый раз, когда запускается новый блок, создается новая область видимости. Область видимости определяет ту часть программы, где имена декларированных (в данном блоке) объектов являются "видимыми" из других частей программы. Эта область определяет также "время (продолжительность) жизни" этих объектов.

Большинство машинных языков определяет две общих категории областей действия — *глобальную* и *локальную*. Однако эти традиционные области действия не совсем согласуются со строгой, объектно-ориентированной моделью Java. Хотя можно показать, что относится к глобальной области действия, но это скорее исключение, а не правило. В Java-программах можно выделить две основных области действия: одна определяется классом, а другая — методом. Однако даже такое различие несколько искусственно. Так как область действия класса (class scope) имеет несколько уникальных свойств и атрибутов, которые не применяются к области действия, определяемой методом, эта дискуссия имеет некоторый смысл. Из-за этих различий обсуждение области действия класса (и переменных, объявленных в ее пределах) отложено до главы 6, где описаны классы. Пока мы будем рассматривать только область действия, определенную в пределах метода.

Область действия, определенная методом, начинается с его открывающей фигурной скобки. Однако, если метод имеет параметры, они также включаются в его область действия. Хотя параметры будут рассмотрены более подробно в главе 5, в данном обсуждении они работают так же, как любая другая переменная метода.

Общее правило: переменные, объявленные внутри области действия, невидимы (т. е. недоступны) коду, который определен вне этой области. Таким образом, когда вы объявляете переменную в пределах области действия, вы локализуете эту переменную и защищаете ее от неправомочного доступа

и/или модификации. Действительно, правила области видимости обеспечивают основу для инкапсуляции.

Области действия могут быть вложены. Например, каждый раз, когда вы создаете блок кода, образуется новая, вложенная область действия. Когда это происходит, внешняя область действия включает внутреннюю. Это означает, что объекты, объявленные во внешней области, будут видимы и во внутренней. Однако, обратное — не верно. Объекты, объявленные во внутренней области действия, не будут видимы во внешней.

Чтобы понять эффект вложенных областей действия, рассмотрим следующую программу:

```
// Демонстрирует область действия блока.
class Scope {
    public static void main(String args[]) {
        int x;                                // известна всему коду внутри main
        x = 10;
        if(x == 10) {                         // начало новой области действия
            int y = 20;                      // известна только в этом блоке
            // здесь известны x и y
            System.out.println("x и y: " + x + " " + y);
            x = y * 2;
        }
        // y = 100;                          //Ошибка! y здесь не известна
        // x здесь еще известна
        System.out.println("x равно " + x);
    }
}
```

Как указывают комментарии, переменная x объявлена в начале области действия main() и доступна для всего последующего кода в main(). В пределах if-блока объявлена переменная y. Так как блок определяет область действия, переменная y видима только кодам в пределах этого блока. Вот почему закомментирована строка y = 100;, находящаяся вне этого блока. Если вы удалите ведущие символы комментария, произойдет ошибка во время компиляции, потому что y — невидима вне этого блока. Внутри блока if можно использовать переменную x, потому что код в блоке (т. е. во вложенной области действия) имеет доступ к переменным, объявленным во включающей области действия.

Внутри блока переменные можно объявлять в любой точке, но область их действия начинается только после того, как они объявлены. Таким образом, если вы определяете переменную в начале метода, она становится доступной всему коду в пределах этого метода. Наоборот, если вы объявляете перемен-

ную в конце блока, это совершенно бесполезно, потому что никакой код не будет иметь к ней доступа. Например, этот фрагмент недопустим, потому что переменная `count` не может использоваться до ее объявления:

```
// Этот фрагмент неправильный!
count = 100;      // count нельзя использовать перед ее объявлением!
int count;
```

Напомним другой важный момент: при выполнении программы переменные создаются во время входа в их область действия и разрушаются при выходе из этой области. Это означает, что переменная не будет сохранять свое значение при выходе из области действия. Поэтому переменные, объявленные внутри метода, не будут сохранять своих значений между вызовами этого метода. Аналогично, переменная, объявленная в блоке, будет терять свое значение при выходе из него. Таким образом, время жизни переменной ограничено ее областью действия.

Если объявление переменной включает инициализатор, то эта переменная будет повторно инициализироваться каждый раз при входе в блок, в котором она объявлена. Например, рассмотрим программу:

```
// Демонстрирует время жизни переменной.
class LifeTime {
    public static void main(String args[]) {
        int x;

        for(x = 0; x < 3; x++) {
            int y = -1;          // y инициализируется каждый раз при входе в блок
            System.out.println("y равен: " + y);    // всегда выводит -1
            y = 100;
            System.out.println("y теперь: " + y);
        }
    }
}
```

Выход, генерируемый этой программой:

```
у равен: -1
у теперь: 100
у равен: -1
у теперь: 100
у равен: -1
у теперь: 100
```

Видно, что у повторно инициализируется (значением `-1`) каждый раз при входе во внутренний цикл `for`. Даже хотя ей затем присваивается значение `100`, это значение теряется.

Последний штрих: хотя блоки могут быть вложены, во внутреннем блоке нельзя объявлять переменную с темим же именем, как во внешней области действия. В этом отношении Java отличается от С и С++. Ниже приведен пример, который пробует объявлять две отдельные переменные с одним и тем же именем. В Java это незаконно. В С/C++ это было бы приемлемо, и две переменные `bar` были бы различными.

```
// Эта программа не будет компилироваться.
class ScopeErr {
    public static void main(String args[]) {
        int bar = 1;
        {
            // создается новая область действия
            int bar = 2; // Ошибка времени выполнения — bar уже определена!
        }
    }
}
```

## Преобразование и приведение типов

Опытным программистам уже известно, что довольно обычным делом является присваивание значения одного типа переменной другого типа. Если два типа совместимы, то Java выполнит преобразование автоматически. Например, всегда возможно назначить `int`-значение `long`-переменной. Однако не все типы совместимы и, таким образом, не все преобразования типов неявно позволены. Например, не определено никакого преобразования из `double` в `byte`. К счастью, все еще возможно осуществить преобразование между несовместимыми типами. Чтобы это сделать, следует использовать специальную операцию *приведения типов* (cast), которая выполняет явное преобразование между несовместимыми типами. Рассмотрим как автоматические преобразования типов, так и приведение типов.

### Автоматическое преобразование типов в Java

Когда один тип данных назначается переменной другого типа, будет иметь место *автоматическое преобразование типов*, если выполняются два следующих условия:

- два типа совместимы;
- целевой тип больше чем исходный.

Когда эти два условия выполняются, имеет место *расширяющее преобразование* (widening conversion). Например, тип `int` всегда достаточно большой, чтобы содержать все допустимые `byte`-значения, так что никакой явный оператор приведения не требуется.

Для расширяющих преобразований числовые типы, включая целый и с плавающей точкой, являются совместимыми друг с другом. Однако числовые типы не совместимы с `char` или `boolean`. Типы `char` и `boolean` не совместимы также и друг с другом.

Как говорилось ранее, Java выполняет автоматическое преобразование типов также и при сохранении лiteralной целочисленной константы в переменных типа `byte`, `short` или `long`.

## Приведение несовместимых типов

Хотя автоматическое преобразование типов полезно, оно не удовлетворяет всем потребностям. Например, как быть, если вы захотите назначить `int`-значение `byte`-переменной? Это преобразование не будет выполнено автоматически, потому что тип `byte` меньше, чем `int`. Данный вид преобразования иногда называется *сужающим преобразованием* (narrowing conversion), т. к. вы явно делаете значение более узким, чтобы оно вписалось в целевой тип.

Чтобы создавать преобразование между двумя несовместимыми типами, вы должны использовать *приведение* типов. *Приведение* (cast) — это и есть явное преобразование типов. Оно имеет общий формат:

**(target-type) value**

Здесь `target-type` определяет желаемый тип, к которому следует преобразовать указанное `value`. Например, следующий фрагмент приводит `int` к `byte`. Если целое значение больше, чем диапазон `byte`-типа (256), то оно будет редуцировано по модулю этого диапазона (до остатка от целочисленного деления этого значения на 256).

```
int a;  
byte b;  
// ...  
b = (byte) a;
```

Другой тип преобразований — *усечение* (truncation), произойдет, когда значение с плавающей точкой назначается целому типу. Как вы знаете, целые числа не имеют дробных частей. Таким образом, когда значение с плавающей точкой назначается целому типу, дробная часть теряется. Например, если значение 1.23 назначается целой переменной, результирующее значение будет просто 1, а 0.23 будет усечено. Конечно, если числовое значение слишком велико, чтобы вписаться в целевой целый тип, то оно будет редуцировано по модулю диапазона целевого типа.

Следующая программа демонстрирует некоторые преобразования типов, которые требуют приведений:

```
// Демонстрирует приведения типов.
class Conversion {
    public static void main(String args[]) {
        byte b;
        int i = 257;
        double d = 323.142;

        System.out.println("\nПриведение int к byte.");
        b = (byte) i;
        System.out.println("i и b " + i + " " + b);
        System.out.println("\nПриведение double к int.");
        i = (int) d;
        System.out.println("d и i " + d + " " + i);
        System.out.println("\nПриведение double к byte.");
        b = (byte) d;
        System.out.println("d and b " + d + " " + b);
    }
}
```

Эта программа генерирует следующий вывод:

Приведение int к byte.

i и b 257 1

Приведение double к int.

d и i 323.142 323

Приведение double к byte.

d и b 323.142 67

Посмотрим на каждое преобразование. Когда значение 257 приводится к byte-переменной, результат — остаток от деления 257 на 256 (диапазон типа byte) в этом случае равен 1. Когда d преобразуется к int, его дробный компонент теряется. Когда d преобразуется к byte, его дробный компонент теряется, и значение редуцируется по модулю 256, что в этом случае дает 67.

## Автоматическое расширение типа в выражениях

В дополнение к операции назначения (присваивания) в выражениях есть и другое место, где могут происходить некоторые преобразования типов — в выражениях. Чтобы увидеть, как и почему это происходит, приведем следующие соображения. В выражении точность, требуемая от промежуточного значения, будет иногда превышать диапазон любого операнда. Например, рассмотрим следующие выражения:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

Результат промежуточного выражения  $a*b$  превышает диапазон любого из его `byte`-операндов. Чтобы управлять проблемой этого рода, Java при оценке выражения автоматически расширяет (повышает) тип каждого `byte` или `short` операнда до `int`. Это означает, что подвыражение  $a*b$  вычисляется с использованием целых чисел — не байтов. Таким образом, результат 2000 промежуточного выражения  $50*40$  является законным даже при том, что оба операнда (`a` и `b`) определены с типом `byte`.

Не смотря на свою полезность, автоматические расширения типов могут вызывать плохо распознаваемые ошибки во время компиляции. Например, этот, казалось бы, корректный код порождает проблему:

```
byte b = 50;
b = b * 2; // Ошибка! Невозможно назначить int-значение byte-переменной!
```

Во второй строке кода происходит попытка сохранить совершенно правильное `byte`-значение ( $50*2$ ) в `byte`-переменной `b`. Однако, из-за того что операнды были автоматически расширены до типа `int` (во время оценки выражения), результат также был расширен до `int`. Таким образом, результат выражения имеет теперь тип `int`, который не может быть назначен `byte`-переменной без использования приведения.

Там, где вы осознаете последствия переполнения, следует использовать явное приведение типа:

```
byte b = 50;
b = (byte)(b * 2);
```

что выдает правильное значение 100.

## Правила расширения типов

В дополнение к расширению типов `byte` и `short` до `int` в Java определены несколько правил расширения типов, которые применяются к выражениям. Во-первых, все `byte`- и `short`-значения расширяются до `int`, как только что описано. Затем, если один операнд имеет тип `long`, тип целого выражения расширяется до `long`. Если один операнд — типа `float`, то тип полного выражения расширяется до `float`. Если тип любого из операндов — `double`, то тип результата — также `double`.

Следующая программа демонстрирует, как расширяется каждое значение в выражении, чтобы согласовать второй аргумент `c` каждой бинарной операцией:

```

class Promote {
    public static void main(String args[]) {
        byte b = 42;
        char c = 'a';
        short s = 1024;
        int i = 50000;
        float f = 5.67f;
        double d = .1234;
        double result = (f * b) + (i / c) - (d * s);
        System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
        System.out.println("результат = " + result);
    }
}

```

Рассмотрим подробнее расширение типов в следующей строке программы:

```
double result = (f * b) + (i / c) - (d * s);
```

В первом подвыражении ( $f * b$ ) тип  $b$  расширен до  $float$ , а результат подвыражения имеет тип  $float$ . Далее, в подвыражении ( $i / c$ ) тип  $c$  расширен до  $int$ , а результат имеет тип  $int$ . Затем, в ( $d * s$ ), значение  $s$  расширено до  $double$ , и тип подвыражения тоже  $double$ . Наконец, просматриваются промежуточные значения типов  $float$ ,  $int$  и  $double$ . Результат сложения  $float$ - и  $int$ -значений имеет тип  $float$ . Затем в вычитании  $float$ - и  $double$ -операндов результат расширяется до  $double$ , что и является типом для окончательного результата выражения.

## Массивы

Массив — это набор однотипных переменных, на которые ссылаются по общему имени. Массивы можно создавать из элементов любого типа, и они могут иметь одно или несколько измерений. К определенному элементу в массиве обращаются по его индексу (номеру). Массивы предлагают удобные средства группировки связанной информации.

### Замечание

Если вы знакомы с С/С++, будьте внимательны. Массивы в Java работают иначе, чем в этих языках.

## Одномерные массивы

Одномерный массив — это, по существу, список однотипных переменных. Чтобы создать массив, сначала следует создать переменную массива (array variable) желательного типа. Общий формат объявления одномерного массива:

```
type var-name[ ];
```

Здесь `type` объявляет базовый тип массива; `var-name` — имя переменной массива. Базовый тип определяет тип данных каждого элемента массива. Например, объявление одномерного массива `int`-компонентов с именем `month_days` имеет вид:

```
int month_days[];
```

Хотя это объявление и устанавливает факт, что `month_days` является переменной массива, никакой массив в действительности не существует. Фактически, значение `month_days`<sup>1</sup> установлено в `null` (пустой указатель), который представляет массив без значения. Чтобы связать `month_days` с фактическим, физическим массивом целых чисел, нужно выделить память для него, используя операцию `new`, и назначать ее массиву `month_days`. `new` — это специальная операция, которая распределяет память.

Подробнее операция `new` будет рассмотрена дальше, сейчас же она нужна для выделения памяти под массив. Общий формат `new` в применении к одномерным массивам имеет вид:

```
array-var = new type [size];
```

где `type` — тип распределяемых данных, `size` — число элементов в массиве, `array-var` — переменная, которая связана с массивом. Чтобы использовать `new` для распределения памяти под массив, нужно специфицировать тип и число элементов массива. Элементы в массиве, выделенные операцией `new`, будут автоматически инициализированы нулями. Следующий пример распределяет память для 12-элементного массива целых чисел и связывает его с переменной `month_days`.

```
month_days = new int[12];
```

После того как эта инструкция выполнится, `month_days` будет ссылаться на массив из двенадцати целых чисел. Затем все элементы в массиве будут инициализированы нулями.

Процесс получения массива включает два шага. Во-первых, следует объявить переменную массива желательного типа. Во-вторых, необходимо выделить память, которая будет содержать массив, используя операцию `new`, и назначать ее переменной массива. Таким образом, в Java все массивы являются динамически распределяемыми. Если концепция динамического распределения вам неизвестна, не беспокойтесь. Она будет подробно описана в этой книге позже.

Как только вы выделили память для массива, можно обращаться к определенному элементу в нем, указывая в квадратных скобках индекс. Нумерация элементов массива начинается с нуля. Например, следующий оператор присваивает значение 28 второму элементу массива `month_days`.

```
month_days[1] = 28;
```

<sup>1</sup> То есть в Java переменная массива — это, по-существу, *ссылочная переменная*. — Примеч. пер.

Следующая строка отображает на экран значение, хранимое в элементе с индексом 3.

```
System.out.println(month_days[3]);
```

Собрав все части вместе, получаем программу, которая создает массив для хранения количества дней каждого месяца.

```
// Демонстрирует одномерный массив.
```

```
class Array {  
    public static void main(String args[]) {  
        int month_days[];  
        month_days = new int[12];  
        month_days[0] = 31;  
        month_days[1] = 28;  
        month_days[2] = 31;  
        month_days[3] = 30;  
        month_days[4] = 31;  
        month_days[5] = 30;  
        month_days[6] = 31;  
        month_days[7] = 31;  
        month_days[8] = 30;  
        month_days[9] = 31;  
        month_days[10] = 30;  
        month_days[11] = 31;  
        System.out.println("Апрель содержит " + month_days[3] + " дней.");  
    }  
}
```

После выполнения эта программа выводит число дней в апреле. Как говорилось выше, индексация элементов массива в Java начинается с нуля, так что число дней апреля хранится в элементе `month_days[3]` и равно 30.

Возможна комбинация объявления переменной типа массив с выделением массиву памяти непосредственно в объявлении:

```
int month_days[] = new int[12];
```

Данный способ обычно встречается в профессионально написанных программах Java.

Массивы можно инициализировать во время их объявления. Процесс во многом аналогичен тому, что используется при инициализации простых типов. *Инициализатор массива* — это список разделенных запятыми выражений, окруженный фигурными скобками. Массив будет автоматически создаваться достаточно большим, чтобы содержать столько элементов, сколько вы определяете в инициализаторе массива. Нет необходимости использовать операцию `new`. Например, чтобы хранить число дней в каждом месяце, следующий код создает инициализированный массив целых чисел:

```
// Улучшенная версия предыдущей программы.
class AutoArray {
    public static void main(String args[]) {
        int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
        System.out.println("Апрель содержит " + month_days[3] + " дней.");
    }
}
```

Когда вы выполните эту программу, то увидите тот же самый вывод, как у предыдущей версии программы.

Java делает строгие проверки, чтобы удостовериться, что вы случайно не пробуете сохранять или читать значения вне области хранения массива. Исполнительная система Java тоже делает тщательные проверки, чтобы убедиться, что все индексы массивов находятся в правильном диапазоне. (В этом отношении Java существенно отличается от языков C/C++, которые не обеспечивают проверки границ во время выполнения.) Например, исполнительная система будет проверять значение каждого индекса в `month_days`, чтобы удостовериться, что он находится между 0 и 11 включительно. Если вы попробуете обратиться к элементам вне диапазона массива (индекс меньше нуля или больше, чем длина массива), то вызовете ошибку времени выполнения.

Имеется еще один пример, который использует одномерный массив. Он находит среднее значение набора чисел.

```
// Среднее значение элементов массива.
class Average {
    public static void main(String args[]) {
        double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
        double result = 0;
        int i;

        for(i=0; i<5; i++)
            result = result + nums[i];

        System.out.println("Среднее арифметическое равно " + result / 5);
    }
}
```

## Многомерные массивы

В Java многомерные массивы — это, фактически, массивы массивов. Они выглядят и действуют подобно регулярным многомерным массивам. Однако имеется пара тонких различий. Чтобы объявить многомерную переменную массива, определите каждый дополнительный индекс, используя другой на-

бор квадратных скобок. Например, следующее утверждение объявляет переменную двумерного массива с именем `twoD`:

```
int twoD[][] = new int[4][5];
```

Оно распределяет память для массива  $4 \times 5$  и назначает ее переменной `twoD`. Внутренне эта матрица реализована как массив массивов целых чисел тип `int`. Концептуально этот массив будет выглядеть, как показано на рис. 3.1.

Следующая программа нумерует каждый элемент в массиве слева направо, сверху вниз и затем отображает эти значения:

```
// Демонстрирует двумерный массив.
class TwoDArray {
    public static void main(String args[]) {
        int twoD[][] = new int[4][5];
        int i, j, k = 0;

        for(i=0; i<4; i++)
            for(j=0; j<5; j++) {
                twoD[i][j] = k;
                k++;
            }

        for(i=0; i<4; i++) {
            for(j=0; j<5; j++)
                System.out.print(twoD[i][j] + " ");
            System.out.println();
        }
    }
}
```

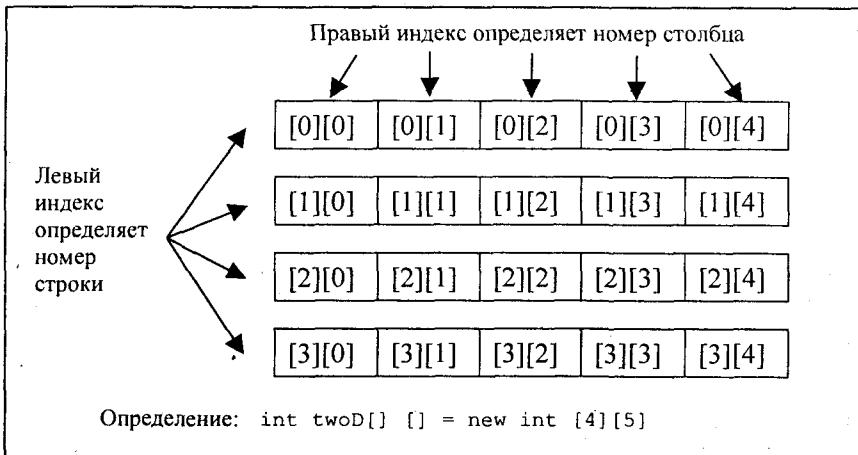


Рис. 3.1. Концептуальный вид двумерного массива  $4 \times 5$

Эта программа генерирует следующий вывод:

```
0 1 2 3 4  
5 6 7 8 9  
10 11 12 13 14  
15 16 17 18 19
```

Когда вы распределяете память для многомерного массива, нужно специфицировать только первое (крайнее левое) измерение. Остающиеся измерения можно распределять отдельно. Например, в следующем тексте память распределяется только для первого измерения переменной `twoD` (когда она объявляется). Распределение для второго измерения выполняется вручную:

```
int twoD[][] = new int[4][];  
twoD[0] = new int[5];  
twoD[1] = new int[5];  
twoD[2] = new int[5];  
twoD[3] = new int[5];
```

Хотя в этой ситуации нет никакого преимущества для индивидуального распределения массивов второго измерения, но оно может появиться в других случаях. Например, когда вы распределяете измерения вручную, не нужно распределить то же самое число элементов для каждого измерения. Как заявлено ранее, многомерные массивы — фактически массивы массивов, длина каждого массива находится под вашим управлением. Например, следующая программа создает двумерный массив, который для разных строк имеет разное количество столбцов.

```
// Ручное распределение различных размеров второго измерения.  
class TwoDAgain {  
    public static void main(String args[]) {  
        int twoD[][] = new int[4][];  
        twoD[0] = new int[1];  
        twoD[1] = new int[2];  
        twoD[2] = new int[3];  
        twoD[3] = new int[4];  
  
        int i, j, k = 0;  
  
        for(i=0; i<4; i++)  
            for(j=0; j<i+1; j++) {  
                twoD[i][j] = k;  
                k++;  
            }  
  
        for(i=0; i<4; i++) {  
            for(j=0; j<i+1; j++)  
                System.out.print(twoD[i][j] + " ");  
        }  
    }  
}
```

```
    System.out.println();  
}  
}  
}
```

Эта программа генерирует следующий вывод:

0  
1 2  
3 4 5  
6 7 8 9

Массив, созданный этой программой, выглядит так, как показано на рис. 3.2.

[0][0]			
[1][0]	[1][1]		
[2][0]	[2][1]	[2][2]	
[3][0]	[3][1]	[3][2]	[3][3]

Рис. 3.2. "Неровный" массив

Использование неровных (или нерегулярных) многомерных массивов не рекомендуется для большинства приложений, потому что они ведут себя противоположно тому, что люди ожидают, когда сталкиваются с многомерным массивом. Однако в некоторых ситуациях их можно использовать эффективно. Например, если вы нуждаетесь в очень большом двухмерном массиве, который редко заполнен (т. е. в котором не все элементы будут содержать ненулевые значения), то нерегулярный массив мог бы быть подходящим решением.

Многомерные массивы возможно инициализировать. Для этого просто включают инициализатор каждого измерения в его собственный набор фигурных скобок. Следующая программа создает матрицу, где каждый элемент содержит произведение индексов строки и столбца. Также обратите внимание, что внутри инициализаторов массива вы можете использовать как выражения, так и лiteralные значения.

// Инициализирует двумерный массив.

```
class Matrix {  
    public static void main(String args[]) {  
        double m[][] = {  
            { 0*0, 1*0, 2*0, 3*0 },  
            { 0*1, 1*1, 2*1, 3*1 },  
            { 0*2, 1*2, 2*2, 3*2 },  
            { 0*3, 1*3, 2*3, 3*3 }  
        };  
        int i, j;
```

```

for(i=0; i<4; i++) {
    for(j=0; j<4; j++)
        System.out.print(m[i][j] + " ");
    System.out.println();
}
}
}

```

Когда вы выполните эту программу, то получите следующий вывод:

```

0.0 0.0 0.0 0.0
0.0 1.0 2.0 3.0
0.0 2.0 4.0 6.0
0.0 3.0 6.0 9.0

```

Как видите, каждая строка в массиве инициализирована, как указано в списках инициализации.

Рассмотрим еще один пример, который использует многомерный массив. Следующая программа создает трехмерный массив  $3 \times 4 \times 5$ , затем заполняет каждый элемент произведением его индексов и, наконец, отображает эти значения на экран.

```

// Демонстрирует трехмерный массив.
class threeDMatrix {
    public static void main(String args[]) {
        int threeD[][][] = new int[3][4][5];
        int i, j, k;

        for(i=0; i<3; i++)
            for(j=0; j<4; j++)
                for(k=0; k<5; k++)
                    threeD[i][j][k] = i * j * k;

        for(i=0; i<3; i++) {
            for(j=0; j<4; j++) {
                for(k=0; k<5; k++)
                    System.out.print(threeD[i][j][k] + " ");
                System.out.println();
            }
            System.out.println();
        }
    }
}

```

Эта программа генерирует следующий вывод:

```

0 0 0 0
0 0 0 0

```

```

0 0 0 0 0
0 0 0 0 0

0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12

0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24

```

## Альтернативный синтаксис объявления массива

Существует иная форма, которая может использоваться для объявления массива:

```
type[ ] var-name;
```

Здесь квадратные скобки следуют за спецификатором типа, а не именем переменной массива. Например, следующие два объявления эквивалентны:

```
int a1[] = new int[3];
int[] a2 = new int[3];
```

Представленные здесь объявления также эквивалентны:

```
char twodi[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

Эта альтернативная форма объявления включена, главным образом, для удобства.

## Несколько слов относительно строк

Как вы, возможно, заметили, в предшествующем обсуждении типов данных и массивов не было никакого упоминания о строках или строковом типе данных. Это не потому, что язык Java не поддерживает такой тип, — он это делает. Дело в том, что строчный тип Java, называемый `String`, — не просто массив символов (как строки в C/C++). Скорее, `String` определяет объект, и полное описание этого требует понимания нескольких объектно-ориентированных свойств. Поэтому о нем будет сказано позже, после описания объектов. Однако, для того чтобы вы могли использовать простые строки в примерах программ, сделаем следующее краткое введение.

Тип `String` используется, чтобы объявлять строковые переменные. Вы можете также объявлять массивы строк. `String`-переменной может быть назначена строковая константа (строка символов в двойных кавычках).

Переменная типа `String` может быть назначена другой переменной типа `String`. Разрешено использовать объект типа `String` как аргумент в методе `println()`. Например, рассмотрим следующий фрагмент:

```
String str = "Это тест";
System.out.println(str);
```

Здесь `str` — объект типа `String`. Ему назначена строка "Это тест". Она выводится на экран методом `println()`.

Как вы увидете позже, объекты типа `String` имеют много специальных свойств и атрибутов, которые делают их весьма мощными и удобными в использовании. Однако в следующих нескольких главах вы будете применять их только в самой простой форме.

## Замечание для программистов C/C++ по поводу указателей

Если вы опытный программист C/C++, то знаете, что эти языки обеспечивают поддержку для указателей. Однако никакого упоминания об указателях не было сделано в текущей главе. Причина проста — Java не поддерживает и не допускает указателей. (Или, более точно, Java не поддерживает указатели, которые доступны и/или изменяются программистом.) Язык Java не разрешает указатели, потому что, сделав это, он позволил бы Java-апплетам нарушать межсетевую защиту между средой выполнения Java и главной ЭВМ. (Вспомните, указателю можно назначать любой адрес в памяти — даже адрес, который мог бы быть вне Java-системы во время выполнения.) Так как C/C++ интенсивно используют указатели, можно подумать, что их потеря — существенный недостаток Java. Однако это не так. Java разработан таким образом, что, пока вы остаетесь в пределах границ среды выполнения, вам никогда не будет нужно ни использовать указатели, ни получать какую-то выгоду от их применения. Советы по преобразованию кода C/C++, включающего указатели, в код Java можно найти в главе 28.



## ГЛАВА 4

# Операции

В языке Java обеспечен богатый набор операций, большинство которых можно разделить на четыре группы: арифметические, поразрядные, отношений и логические. В Java определены также некоторые дополнительные операции, которые обрабатывают специальные ситуации. Эта глава описывает все операции Java за исключением операции сравнения типов `instanceof`, которая рассмотрена в главе 12.

### Замечание

Если вы знакомы с C/C++, то вам будет полезно узнать, что большинство операций в Java работает точно так же, как в C/C++. Однако имеются некоторые тонкие различия, так что советуем читать внимательно.

## Арифметические операции

Арифметические операции используются в математических выражениях таким же образом, как они используются в алгебре. Арифметические операции перечислены в табл. 4.1.

**Таблица 4.1. Арифметические операции Java**

Операция	Результат
+	Сложение (Addition)
-	Вычитание (Subtraction) (унарный минус (Unary minus))
*	Умножение (Multiplication)
/	Деление (Division)
%	Остаток от деления по модулю (Modulus)

Таблица 4.1 (окончание)

Операция	Результат
<code>++</code>	Инкремент (Increment)
<code>+=</code>	Присваивание со сложением (Addition assignment)
<code>-=</code>	Присваивание с вычитанием (Subtraction assignment)
<code>*=</code>	Присваивание с умножением (Multiplication assignment)
<code>/=</code>	Присваивание с делением (Division assignment)
<code>%=</code>	Присваивание с модулем (Modulus assignment)
<code>--</code>	Декремент (Decrement)

Операнды арифметических операций должны иметь числовой тип. Их нельзя применять к переменным типа `boolean`, но можно использовать на типах `char`, т. к. тип `char` в Java, по существу, подмножество `int`.

## Основные арифметические операции

Основные арифметические операции (сложение, вычитание, умножение и деление) ведут себя ожидаемым образом для всех числовых типов. Операция минус имеет также унарную форму, которая отрицает ее одиночный operand. Когда операция деления применяется к целочисленному типу, дробная часть теряется.

Следующий простой пример демонстрирует арифметические операции, а также иллюстрирует различие между делением с плавающей точкой и целочисленным делением.

```
// Демонстрирует основные арифметические операции.
class BasicMath {
    public static void main(String args[]) {
        // арифметика с использованием целых
        System.out.println("Целочисленная арифметика");
        int a = 1 + 1;
        int b = a * 3;
        int c = b / 4;
        int d = c - a;
        int e = -d;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
        System.out.println("e = " + e);
    }
}
```

```

// арифметика с использованием double-типов
System.out.println("\nАрифметика для вещественных значений");
double da = 1 + 1;
double db = da * 3;
double dc = db / 4;
double dd = dc - a;
double de = -dd;
System.out.println("da = " + da);
System.out.println("db = " + db);
System.out.println("dc = " + dc);
System.out.println("dd = " + dd);
System.out.println("de = " + de);
}
}

```

Вывод этой программы:

Целочисленная арифметика

```

a = 2
b = 6
c = 1
d = -1
e = 1

```

Арифметика для вещественных значений

```

da = 2
db = 6
dc = 1.5
dd = -0.5
de = 0.5

```

## Деление по модулю

Операция деления по модулю (%) вычисляет остаток от операции деления. Ее можно применять к типам с плавающей точкой и к целым. (Это отличается от C/C++, в которых данную операцию можно применять только к целым типам.) Следующий пример демонстрирует операцию %:

```

// Демонстрирует операцию %.
class Modulus {
    public static void main(String args[]) {
        int x = 42;
        double y = 42.3;

        System.out.println("x mod 10 = " + x % 10);
        System.out.println("y mod 10 = " + y % 10);
    }
}

```

После выполнения программы вы получите следующий вывод:

```
x mod 10 = 2  
y mod 10 = 2.3
```

## Арифметические операции присваивания

В Java обеспечены специальные операции, которые можно использовать для комбинирования арифметических операций с присваиванием. Как вы, вероятно, знаете, операторы, подобные следующему, весьма обычны в программировании:

```
a = a + 4;
```

В Java вы можете переписать этот оператор так:

```
a += 4;
```

Данная версия использует операцию присваивания со сложением `+=`. Оба оператора выполняют одно и то же — увеличивают значение на 4.

Вот другой пример:

```
a = a % 2;
```

который можно выразить так:

```
a %= 2;
```

В этом случае операция `%=` получает остаток от деления `a` на 2 и помещает этот результат обратно в `a`.

Имеются операции присваивания для всех арифметических и бинарных операций. Таким образом, любой оператор формы

```
var = var op expression;
```

где `var` — идентификатор переменной; `op` — выполняемая операция; `expression` — выражение;

можно переписать как

```
var op = expression
```

Операции присваивания обеспечивают два преимущества. Во-первых, они немного сокращают объем клавиатурного ввода, потому что короче эквивалентных им длинных форм. Во-вторых, исполнительная система Java реализует их гораздо эффективнее, чем их длинные эквиваленты. По этим причинам в профессионально составленных Java-программах вы часто увидите использование именно этих форм операций присваивания.

Пример программы, которая использует несколько операций присваивания с арифметическими (в форме `op =`):

```
// Демонстрирует некоторые операции присваивания.
class OpEquals {
    public static void main(String args[]) {
        int a = 1;
        int b = 2;
        int c = 3;

        a += 5;
        b *= 4;
        c += a * b;
        c %= 6;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
    }
}
```

Вывод этой программы:

```
a = 6
b = 8
c = 3
```

## Инкремент и декремент

В Java операции инкремента и декремента обозначаются как `++` и `--` (по составу операндов это унарные операции). Они были представлены в главе 2, а здесь будут обсуждены подробнее. Указанные операции обладают некоторыми специальными свойствами, которые делают их весьма интересными. Начнем с точного рассмотрения того, что операции инкремента и декремента делают.

Операция инкремента увеличивает свой operand на единицу, а операция декремента — уменьшает (тоже на единицу). Например, следующий оператор:

```
x = x + 1;
```

можно переписать с помощью операции инкремента так:

```
x++;
```

Точно так же оператор

```
x = x - 1;
```

эквивалентен

```
x--;
```

Эти операции уникальны тем, что могут появляться как в *постфиксной* форме, где они *следуют за* operandом, как только что показано, так и в *пре-*

фиксной форме, где они *предшествуют* операнду. В предыдущих примерах нет никакого различия между префиксными и постфиксными формами. Однако, когда инкрементная и/или декрементная операции — часть большего выражения, тогда появляется тонкое, но все же существенное, различие между этими двумя формами. В префиксной форме operand инкрементируется или декрементируется *прежде*, чем значение используется в выражении. В постфиксной форме *предыдущее* значение сначала используется в выражении и только потом operand изменяется. Например:

```
x = 42;  
y = ++x;
```

В этом случае в y устанавливается значение 43, потому что инкремент выполняется прежде, чем x назначается переменной y. Таким образом, строка y = ++x; является эквивалентом следующих двух операторов:

```
x = x + 1;  
y = x;
```

Однако, если записать так:

```
x = 42;  
y = x++;
```

значение x присваивается переменной y *перед* тем, как выполняется операция инкремента, так что значение y равно 42. Конечно, в обоих случаях в x устанавливается значение 43. Здесь строка y = x++; является эквивалентом двух следующих операторов:

```
y = x;  
x = x + 1;
```

Рассмотрим программу, демонстрирующую операцию инкремента:

```
// Демонстрирует ++.  
class IncDec {  
    public static void main(String args[]) {  
        int a = 1;  
        int b = 2;  
        int c;  
        int d;  
  
        c = ++b;  
        d = a++;  
        c++;  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
        System.out.println("c = " + c);  
    }  
}
```

```

        System.out.println("d = " + d);
    }
}

```

Вывод этой программы:

```

a = 2
b = 3
c = 4
d = 1

```

## Поразрядные операции

В Java определено несколько *поразрядных (bitwise) операций*, которые могут применяться к целым типам long, int, short, char и byte. Эти операции действуют на индивидуальные биты<sup>1</sup> их operandов. Они подытожены в табл. 4.2.

**Таблица 4.2. Поразрядные операции Java**

Операция	Результат
<code>~</code>	Поразрядное унарное отрицание (bitwise unary NOT)
<code>&amp;</code>	Поразрядное И (bitwise AND)
<code> </code>	Поразрядное ИЛИ (bitwise OR)
<code>^</code>	Поразрядное исключающее ИЛИ (bitwise exclusive OR (XOR))
<code>&gt;&gt;</code>	Сдвиг вправо (Shift right)
<code>&gt;&gt;&gt;</code>	Сдвиг вправо с заполнением старшего бита нулем (Shift right zero fill)
<code>&lt;&lt;</code>	Сдвиг влево (Shift left)
<code>&amp;=</code>	Присваивание с поразрядным И (bitwise AND assignment)
<code> =</code>	Присваивание с поразрядным ИЛИ (bitwise OR assignment)
<code>^=</code>	Присваивание с поразрядным исключающим ИЛИ (bitwise exclusive OR (XOR) assignment)
<code>&gt;&gt;=</code>	Присваивание со сдвигом вправо (Shift right assignment)
<code>&gt;&gt;&gt;=</code>	Присваивание со сдвигом вправо и заполнением старшего бита нулем (Shift right zero fill assignment)
<code>&lt;&lt;=</code>	Присваивание со сдвигом влево (Shift left assignment)

<sup>1</sup> Поэтому их часто называют *битовыми*. — Примеч. пер.

Так как поразрядные операции манипулируют битами в пределах целого числа, важно понять, каково воздействие таких манипуляций на его значение. В частности, полезно знать, как Java хранит целочисленные значения и как он представляет отрицательные числа. Поэтому, прежде чем продолжить, сделаем краткий обзор этих двух тем.

Все целые типы представляются двоичными числами с разным количеством разрядов. Например, целочисленное значение 42 типа `byte` в двоичном представлении есть 00101010, где каждая позиция представляется степенью двойки, начиная с  $2^0$  в самом правом разряде. Следующая битовая позиция (слева направо) будет представляться как  $2^1$  и т. д. (т. е. разряды нумеруются справа от 0). Двоичное представление числа 42 имеет единицы в позициях 5, 3 и 1, так что 42 есть сумма  $2^5 + 2^3 + 2^1$  или  $32 + 8 + 2$ .

Все целые типы (кроме `char`) — это целые числа со знаком. Это означает, что они могут представлять как отрицательные значения, так и положительные. Java использует кодирование, известное как *дополнение до двух*, означающее, что отрицательные числа представляются инвертированием (заменой 1 на 0 и наоборот) всех битов в (положительном) значении с последующим прибавлением 1 к результату. Например,  $-42$  представляется инвертированием всех битов в двоичном представлении числа 42 (т. е. в 00101010), что дает 11010101, затем прибавляется (двоичная) 1, что приводит к 11010110 (это и есть двоичное представление  $-42$  в дополнительном коде). Чтобы декодировать отрицательное двоичное представление обратно в десятичную форму, сначала инвертируют все биты, затем преобразуют их в десятичную форму и к ней добавляют 1 (не забыв изменить знак результата). Например, инверсия 11010110 дает 00101001 (или десятичное 41), так что, если прибавить 1 и изменить знак, то получим  $-42$ .

Причину использования в Java (и большинстве других машинных языков) дополнения до двух достаточно просто проследить, если рассмотреть проблему *перехода нуля*. Для типа `byte` нулевое значение представляется как 00000000. Простое инвертирование всех битов (которое называют *дополнением до единицы*) дает 11111111, что формально должно создать отрицательный нуль. Неприятность в том, что такой "отрицательный нуль" недопустим в целочисленной математике (мы видим, что с учетом знакового разряда это на самом деле представление  $-1$ ). Данная проблема решается следующим образом: для представления отрицательных значений используется *дополнение до двух*. Дополнение до двух для 00000000 вычисляется так: после поразрядной инверсии этого значения к нему добавляется (двоичная) 1, что дает 100000000. Это приводит к тому, что единичный бит выдвигается слишком далеко влево и выходит за пределы размеров `byte`-значения (8 бит). Поведение становится правильным, т. к.  $-0$  имеет то же двоичное значение, что  $+0$ , а 11111111 кодирует  $-1$ . Хотя в предшествующем примере мы использовали `byte`-значение, тот же основной принцип применяется ко всем целым типам Java.

Из-за того, что Java использует дополнение до двух для хранения отрицательных чисел, а также потому, что в Java все целые значения — со знаками, применение поразрядных операций может легко породить неожиданные результаты. Например, включение старшего бита приведет к тому, что результирующее значение будет интерпретироваться как отрицательное число, независимо от того, входило ли это в ваши намерения или нет. Чтобы избежать неприятных сюрпризов, просто помните, что старший бит определяет знак целого числа независимо от того, как этот старший бит был установлен.

## Поразрядные логические операции

Поразрядные логические операции это `&`, `|`, `^` и `~`. Табл. 4.3 показывает результат каждой операции<sup>1</sup>. При последующем обсуждении имейте в виду, что поразрядные операции применяются к каждому индивидуальному биту в каждом операнде.

**Таблица 4.3. Поразрядные логические операции Java**

A	B	A   B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

## Поразрядное отрицание

Унарная операция поразрядного отрицания (`~`), называемая также *поразрядным дополнением*, инвертирует все биты своего операнда. Например, число 42, которое представляется следующей комбинацией двоичных разрядов:

00101010

после того как применяется операция унарного отрицания, становится

11010101

## Поразрядное И

Операция поразрядного И (`&`) производит единичный бит, если оба операнда также единичные. Во всех других случаях получается нулевой бит. Например:

<sup>1</sup> На самом деле это не одна, а несколько таблиц, называемые *таблицами истинности*. — Примеч. ред.

00101010	(42)
& 00001111	(15)
<hr/>	
00001010	(10)

## Поразрядное ИЛИ

Операция поразрядного ИЛИ ( $\parallel$ ) комбинирует биты так, что, если хотя бы один (любой) из битов в операндах равен 1, то результирующий бит тоже 1, например:

00101010	(42)
00001111	(15)
<hr/>	
00101111	(47)

## Поразрядное исключающее ИЛИ

Операция поразрядного исключающего ИЛИ ( $\wedge$ ) комбинирует биты так, что, если только один operand есть 1, то и результат есть 1. Иначе, результат нулевой. Следующий пример показывает эффект операции  $\wedge$ . Этот пример демонстрирует также одно полезное свойство операции исключающего ИЛИ. Обратите внимание, что комбинация двоичных разрядов числа 42 инвертирована везде, где второй operand 1, а где второй operand 0, биты первого operand'a остаются без изменения. При выполнении некоторых типов разрядных манипуляций вы найдете это свойство весьма полезным.

00101010	(42)
$\wedge$ 00001111	(15)
<hr/>	
00100101	(37)

## Использование поразрядных логических операций

Следующая программа демонстрирует поразрядные логические операции:

```
// Демонстрирует поразрядные логические операции.
class BitLogic {
    public static void main(String args[]) {
        String binary[] = {
            "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
            "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
        };
        int a = 3; // 0 + 2 + 1 или 0011 в двоичной форме
        int b = 6; // 4 + 2 + 0 или 0110 в двоичной форме
        int c = a | b;
        int d = a & b;
```

```

int e = a ^ b;
int f = (~a & b) | (a & ~b);
int g = ~a & 0x0f;

System.out.println("      a = " + binary[a]);
System.out.println("      b = " + binary[b]);
System.out.println("      a|b = " + binary[c]);
System.out.println("      a&b = " + binary[d]);
System.out.println("      a^b = " + binary[e]);
System.out.println("      ~a&b|a&~b = " + binary[f]);
System.out.println("      ~a = " + binary[g]);
}
}

```

В этом примере целые переменные *a* и *b* содержат числа (3 и 6), битовые образы которых хранят (в младших разрядах) четыре возможные комбинации двух бинарных цифр 0-0, 0-1, 1-1 и 1-0. Можно видеть, как операции *|*, *&* и *^*, работая на каждом бите *a* и *b*, размещают результаты в переменных *c*, *d* и *e*. Значение переменной *f* формирует выражение  $(\sim a \& b) | (a \& \sim b)$ , использующее все допустимые поразрядные операции. Битовые значения, присвоенные переменным *e* и *f*, одинаковы. Массив с именем *binary* содержит строки с двоичными представлениями чисел от 0 до 15. Массив построен так, чтобы элемент *binary[n]* хранил правильное строковое представление бинарного значения индекса *n*. Выражение  $\sim a \& 0x0f$  (где шестнадцатеричная константа *0x0f* в бинарной форме представляется как 0000 1111) вычисляется уменьшением своего значения до величины меньшей, чем 16, так чтобы это значение можно было напечатать при помощи массива *binary*. Вывод этой программы:

```

a = 0011
b = 0110
a|b = 0111
a&b = 0010
a^b = 0101
~a&b|a&~b = 0101
~a = 1100

```

## Левый сдвиг

Операция левого сдвига (*<<*) смещает все биты значения влево на указанное количество разрядов. Она имеет следующую общую форму:

*value << num*

где *num* определяет число позиций левого сдвига значения в *value*. То есть *<<* сдвигает все биты в указанном значении влево на число разрядных позиций,

указанное в `листинге`. Для каждого сдвига влево, старший бит выходит за пределы поля значения (и теряется), а справа вводится нуль. Это означает, что, когда левый сдвиг применяется к `int`-операнду, биты теряются сразу же, как только они сдвигаются за 31-ю разрядную позицию. Если операнд — `long`, то биты выше 63-й разрядной позиции теряются.

Когда вы сдвигаете `byte`- и `short`-значения, автоматическое расширение типа в Java дает неожиданные результаты. Как вы знаете, значения `byte` и `short` расширяются до `int` во время вычисления (оценки) выражения. Кроме того, результат вычисления такого выражения также имеет тип `int`. Это означает, что результат левого сдвига на `byte`- или `short`-значении будет `int` и биты, сдвинутые влево, не будут потеряны, пока они не сдвинутся за последнюю 31-ю разрядную позицию. Кроме того, отрицательное `byte`- или `short`-значение при расширении до `int` будет расширено своей знаковой позицией, т. е. его старшие биты будут заполнены единицами. Выполнение левого сдвига в значениях до `byte` или `short` подразумевает, что следует отбросить старшие байты `int`-результата. Например, если сдвигается влево `byte`-значение, то оно будет сначала расширено до `int` и затем сдвинуто. Далее, если нужно сохранить `byte`-тип результата, следует отбросить три верхних байта результата. Проще всего это выполняется с помощью явного приведения типа результата обратно в `byte`. Эту концепцию демонстрирует следующая программа:

```
// Левый сдвиг byte-значения.
class ByteShift {
    public static void main(String args[]) {
        byte a = 64, b;
        int i;

        i = a << 2;
        b = (byte) (a << 2);           // явное приведение типа в byte

        System.out.println("Исходное значение a: " + a);
        System.out.println("i и b: " + i + " " + b);
    }
}
```

Вывод, сгенерированный этой программой:

```
Исходное значение a: 64
i и b: 256 0
```

Так как переменная `a` расширяется до `int` с целью вычислений, двойной левый сдвиг значения 64 (0100 0000) приводит к тому, что переменная `i` содержит значение 256 (1 0000 0000). Однако значение в переменной `b` содержит 0, потому что после сдвига младший байт становится нулевым, а его единственный единичный бит был выдвинут из поля хранения `byte`-значения.

Так как каждый левый сдвиг имеет эффект удвоения первоначального значения, программисты часто используют этот факт как эффективную альтернативу умножению на 2. Но следует соблюдать осторожность: если вы сдвинете бит со значением 1 в самую старшую позицию (бит 31 или 63), то значение станет отрицательным. Следующая программа иллюстрирует это положение:

```
// Левый сдвиг, как быстрый способ умножения на 2.
class MultByTwo {
    public static void main(String args[]) {
        int i;
        int num = 0xFFFFFFF;
        for(i=0; i<4; i++) {
            num = num << 1;
            System.out.println(num);
        }
    }
}
```

Программа генерирует следующий вывод:

```
536870908
1073741816
2147483632
-32
```

Начальное значение было выбрано так, чтобы после смещения влево на 4 позиции был получен результат  $-32$ . Видно, когда единичный бит сдвинулся в позицию 31, число стало интерпретироваться как отрицательное.

## Правый сдвиг

Операция правого сдвига (`>>`) сдвигает все биты значения вправо на указанное количество разрядов. Ее общая форма:

`value >> num`

Здесь `num` определяет число позиций правого сдвига в значении `value`. То есть `>>` передвигает все биты в указанном значении вправо на число разрядных позиций, указанных в `num`.

Следующий кодовый фрагмент сдвигает значение 32 вправо на две позиции, приводя это значение к числу 8:

```
int a = 32;
a = a >> 2; // теперь содержит 8
```

Когда значение имеет биты, которые "выдвигаются" (в данном случае — вправо), такие биты теряются. Например, следующий кодовый фрагмент сдвигает значение 35 вправо на две позиции, что приводит к потере двух младших битов, заканчиваясь снова установкой значения 8.

```
int a = 35;  
a = a >> 2; // снова содержит 8
```

Взгляд на ту же операцию в двоичном представлении яснее показывает, как это происходит:

```
00100011      (35)  
>> 2  
00001000      (8)
```

Каждый раз, когда вы сдвигаете значение вправо, оно делится на два и отбрасывает любой остаток. Этим можно воспользоваться для высокоэффективного целочисленного деления на 2. Конечно, вы должны убедиться, что не выдвигаете никакие биты с правого конца.

При сдвиге вправо старшие (крайние левые) биты освобождаются и заполняются предыдущим содержимым старшего (знакового) бита. Это так называемое *расширение знака*. Оно служит для того, чтобы сохранить знак отрицательных чисел, когда вы сдвигаете их вправо. Например,  $-8 \gg 1$  дает  $-4$ , что в двоичной форме выглядит так:

```
11111000      (-8)  
>>1  
11111100      (-4)
```

Интересно обратить внимание, что, если вы сдвигаете  $-1$  вправо, результат всегда остается  $-1$ , так как расширение знака продолжает вводить все больше единиц в старшие биты.

Иногда расширение знака при сдвиге значений вправо не желательно. Например, следующая программа преобразует byte-значение к его шестнадцатеричному строчному представлению. Заметим, что сдвинутое значение маскируется выполнением операции  $(b \gg 4) \& 0x0f$ , чтобы отказаться от любых расширенных знаком битов (это делается для того, чтобы значение могло использоваться как индекс в массиве шестнадцатеричных символов).

```
// Маскировка расширения знака.  
class HexByte {  
    static public void main(String args[]) {  
        char hex[] = {  
            '0', '1', '2', '3', '4', '5', '6', '7',  
            '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'  
        };  
        byte b = (byte) 0xf1;
```

```

        System.out.println("b = 0x" + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
    }
}

```

Вывод этой программы:

```
b = 0xf1
```

## Правый сдвиг без знака

Как вы только что видели, операция `>>` автоматически наполняет старший бит его предыдущим содержанием каждый раз, когда происходит сдвиг. Это сохраняет знак числа. Правда, иногда это не желательно. Например, если вы сдвигаете что-то, что не представляет числового значения, вы можете не хотеть расширения знака. Эта ситуация обычно имеет место, когда вы работаете с пиксельными значениями и графикой. В этих случаях вы вообще захотите поместить нуль в старший бит, независимо от того, каково было его начальное значение. Такая процедура известна как *беззнаковый сдвиг*. Чтобы выполнить ее, нужно использовать Java-операцию *беззнакового сдвига вправо* `>>>`, которая всегда вставляет нуль в старший бит.

Следующий кодовый фрагмент демонстрирует операцию `>>>`. Здесь переменная `a` установлена в `-1`, что устанавливает все 32 бита ее двоичного представления в 1. Затем это значение сдвигается вправо на 24 бита, заполняя 24 верхних бита нулями и игнорируя нормальное расширение знака. Это устанавливает `a` значение 255.

```

int a = -1;
a = a >>> 24;

```

Чтобы проиллюстрировать, что происходит, ниже показана та же операция в двоичной форме:

11111111 11111111 11111111 11111111	(-1 в двоичной форме типа int)
<code>&gt;&gt;&gt;24</code>	
00000000 00000000 00000000 11111111	(255 в двоичной форме типа int)

Операция `>>>` часто не столь полезна, как вам бы хотелось, т. к. она значима только для 32- и 64-разрядных значений. Напомним, что меньшие значения в выражениях автоматически расширяются до `int`. Это означает, что происходит расширение знака, и что сдвиг будет иметь место скорее на 32-разрядном, чем на 8- или 16-разрядном значении. То есть, можно ожидать, что беззнаковый правый сдвиг на `byte`-значении заполнит нулем все разряды, начиная с седьмого. Но дело обстоит не так, поскольку фактически сдвигается 32-разрядное значение. Следующая программа демонстрирует этот эффект:

```
// Беззнаковый сдвиг byte-значения.
class ByteUShift {
    static public void main(String args[]) {
        char hex[] = {
            '0', '1', '2', '3', '4', '5', '6', '7',
            '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
        };
        byte b = (byte) 0xf1;
        byte c = (byte) (b >> 4);
        byte d = (byte) (b >>> 4);
        byte e = (byte) ((b & 0xff) >> 4);

        System.out.println("          b = 0x" +
            + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
        System.out.println("      b >> 4 = 0x" +
            + hex[(c >> 4) & 0x0f] + hex[c & 0x0f]);
        System.out.println("      b >>> 4 = 0x" +
            + hex[(d >> 4) & 0x0f] + hex[d & 0x0f]);
        System.out.println("(b & 0xff) >> 4 = 0x" +
            + hex[(e >> 4) & 0x0f] + hex[e & 0x0f]);
    }
}
```

По выводу кажется, будто операция `>>>` ничего не делает, работая с байтовыми operandами. Для этой демонстрации в переменную `b` устанавливается произвольное отрицательное byte-значение. Затем `c` присваивается byte-значение `b`, сдвинутое вправо на четыре разряда, что дает `0xff` из-за ожидавшегося расширения знака. Затем `d` присваивается byte-значение `b`, сдвинутое без знака вправо на четыре, что, по вашим ожиданиям, должно быть `0x0f`, но фактически есть `0xff` из-за расширения знака, которое произошло, когда `b` была расширена перед сдвигом до типа `int`. Последнее выражение устанавливает в `e` byte-значение `b` (типа `byte`), маскированное до 8 бит (с помощью операции И) и затем сдвинутое вправо на четыре разряда, что и дает ожидаемое значение `0x0f`. Обратите внимание, что операция беззнакового сдвига вправо для `d` не использовалась, т. к. состояние знакового разряда после операции И было известно.

```
b = 0xf1
b >> 4 = 0xff
b >>> 4 = 0xff(b & 0xff) >> 4 = 0x0f
```

## Поразрядная операция присваивания

Все бинарные поразрядные операции имеют форму, которая объединяет присваивание с поразрядной операцией. Например, два следующих опера-

тора, которые сдвигают значение вправо на четыре разряда, являются эквивалентными:

```
a = a >> 4;
a >>= 4;
```

Аналогично, следующие два оператора, которые присваивают переменной a значение поразрядного выражения a OR b, являются эквивалентными:

```
a = a | b;
a |= b;
```

Представленная ниже программа создает несколько целых переменных и затем для манипулирования ими использует сокращенную форму присваивания с поразрядными операциями:

```
class OpBitEquals {
    public static void main(String args[]) {
        int a = 1;
        int b = 2;
        int c = 3;

        a |= 4;
        b >>= 1;
        c <<= 1;
        a ^= c;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
    }
}
```

Вывод этой программы:

```
a = 3
b = 1
c = 6
```

## Операции отношений

*Операции отношений*<sup>1</sup> (relational operators) определяют отношения, которые один operand имеет к другому. В частности, они определяют равенство и упорядочивания. В Java используются операции отношений, представленные в табл. 4.4.

---

<sup>1</sup> В отечественной практике часто используется термин *операции сравнения*. — Примеч. пер.

**Таблица 4.4. Операции отношений**

Операция	Результат
<code>==</code>	Равно
<code>!=</code>	Не равно
<code>&gt;</code>	Больше чем
<code>&lt;</code>	Меньше чем
<code>&gt;=</code>	Больше чем или равно
<code>&lt;=</code>	Меньше чем или равно

Результат этих операций — значение типа `boolean`. Операции отношений часто используются в выражениях, которые управляют условными операторами и различными операторами цикла.

Данные любых Java-типов, включая целые числа, числа с плавающей точкой, символы и булевские переменные, можно сравнивать, используя операции проверки равенства (`==`) и проверки неравенства (`!=`). Обратите внимание, что в Java (как в C и C++) равенство обозначается двумя знаками равенства, а не одним. (Напомним, что одиночный знак равенства (`=`) используется для операции присваивания.) С помощью операций упорядочивания (`<`, `>`, `<=`, `>=`) можно сравнивать только числовые типы (т. е., чтобы увидеть, какой operand больше или меньше, чем другой, можно сравнивать только целочисленные, с плавающей точкой и символьные operandы).

Как было сказано ранее, результат, полученный операциями отношений, представляет собой значение типа `boolean`. Например, следующий кодовый фрагмент содержит совершенно правильные утверждения:

```
int a = 4;
int b = 1;
boolean c = a < b;
```

В этом случае результат выражения `a < b` (который есть `false`) сохраняется в `c`.

Если вы переходите к Java из среды C/C++, обратите внимание на следующее. В программах C/C++ часто встречаются следующие типы операторов:

```
int done;
// ...
if(!done) ...           // Верно в C/C++
if(done) ...            // но не в Java.
```

На языке Java эти операторы должны быть записаны так:

```
if(done == 0) ... // Это Java-стиль.  
if(done != 0) ...
```

Причина в том, что Java не определяет `true` и `false` таким же образом, как C/C++. В C/C++ `true` — любое значение, отличное от нуля, а `false` — нуль. В Java `true` и `false` — нечисловые значения, которые не имеют отношения к нулю или не нулю. Поэтому, чтобы выполнить проверку на равенство нулю, вы должны явно использовать одну или несколько операций отношений.

## Операции булевой логики

Представленные в табл. 4.5 операции булевой логики работают только с операндами типа `boolean`. Чтобы сформировать результат типа `boolean`, все бинарные логические операции комбинируют два `boolean`-значения.

**Таблица 4.5. Операции булевой логики**

Операция	Результат
&	Логическое И (Logical AND)
	Логическое ИЛИ (Logical OR)
^	Логическое исключающее ИЛИ (Logical XOR (exclusive OR)) Укороченное ИЛИ (Short-circuit OR)
&&	Укороченное И (Short-circuit AND)
!	Логическое унарное отрицание (Logical unary NOT)
&=	Присваивание с И (AND assignment)
=	Присваивание с ИЛИ (OR assignment)
^=	Присваивание с исключающим ИЛИ (XOR assignment)
==	Равно (Equal to)
!=	Не равно (Not equal to)
? :	Троичная условная операция (Ternary if-then-else)

Действия операций булевой логики `&`, `|` и `^` на булевых значениях аналогичны их действиям на битах целых чисел. Например, логическая операция отрицания (`!`) инвертирует булевское значение на противоположное (`!true` дает значение `false`, а `!false` — значение `true`).

Табл. 4.6 показывает эффект каждой логической операции.

Таблица 4.6. Применение логических операций

A	B	A   B	A & B	A ^ B	!A
false	false	false	false	false	true
true	false	true	false	true	false
false	true	true	false	true	true
true	true	true	true	false	false

Программа почти такая же, как в примере с классом BitLogic, показанном ранее, но оперирующая с логическими значениями типа boolean вместо двоичных битов:

```
// Демонстрирует операции булевой логики.
class BoolLogic {
    public static void main(String args[]) {
        boolean a = true;
        boolean b = false;
        boolean c = a | b;
        boolean d = a & b;
        boolean e = a ^ b;
        boolean f = (!a & b) | (a & !b);
        boolean g = !a;

        System.out.println("      a = " + a);
        System.out.println("      b = " + b);
        System.out.println("a|b = " + c);
        System.out.println("a&b = " + d);
        System.out.println("a^b = " + e);
        System.out.println("!a&b|a&!b = " + f);
        System.out.println("      !a = " + g);
    }
}
```

После выполнения этой программы вы заметите, что те же логические правила применяются как к boolean-значениям, так и к битам. Как можно видеть в следующем выводе, строковым представлением boolean-значений в Java является одно из постоянных значений — true или false:

```
a = true
b = false
a | b = true
a&b = false
a^b = true
a&b|a&!b = true
!a = false
```

## Короткие логические операции

Java обеспечивает две интересные булевские операции, которые нельзя найти в большинстве других машинных языков. Это вторые версии булевыхских операций И и ИЛИ, известные как *укороченные* (short-circuit) логические операции. Как можно видеть в предшествующей таблице, односимвольная операция ИЛИ приводит к `true`-результату, когда операнд A — `true`, независимо от того, каков операнд B. Точно так же операция И приводит к `false`-результату, когда A — `false`, независимо от того, каков B. Если вы используете двухзначные (`||` и `&&`), а не однозначные (`|` и `&`) формы логических операций ИЛИ и И, Java вообще не будет выполнять оценку правого операнда перед вычислением всего выражения, так как результат выражения определяется одним левым операндом. Это полезно, когда правильное функционирование правого операнда зависит от того, является ли левый операнд `true` или `false`. Например, следующий кодовый фрагмент показывает, как вы можете воспользоваться преимуществом сокращения логической оценки, чтобы заранее обеспечить правильное выполнение операции деления (обходя любые оценки ее второго операнда во время выполнения):

```
if (denom != 0 && num / denom > 10)
```

Так как используется короткая форма операции И (`&&`), нет никакого риска порождения исключительной ситуации при работе программы, когда `denom` окажется нулевым. Если бы эта строка программы была написана с использованием версии И, состоящей из одного знака `&`, то во время выполнения программы нужно было бы оценивать обе стороны соответствующего выражения, что при нулевом `denom` вызвало бы исключительную ситуацию.

Использование короткой формы И и ИЛИ стало стандартной практикой в случаях использования булевой логики, оставляя версии с одиночным символом исключительно для поразрядных операций. Однако имеются исключения из этого правила. Например, рассмотрим следующий оператор:

```
if(c==1 & e++ < 100) d = 100;
```

Использование однозначной операции (`&`) гарантирует здесь, что операция инкремента будет применяться к e вне зависимости от того, равна переменная с единице или нет.

## Операция присваивания

Вы уже использовали операцию присваивания, начиная с главы 2. Теперь пришло время взглянуть на нее более формально. Операция *присваивания* (*назначения*) (assignment operator) — это одиночный знак равенства (`=`). Операция присваивания работает в Java во многом так же, как в любом другом машинном языке. Она имеет следующую общую форму:

```
var = expression;
```

Здесь тип переменной `var` должен быть совместим с типом выражения `expression`.

Операция присваивания имеет один интересный атрибут, с которым вы, может быть, не знакомы. Он позволяет создавать цепочку присваиваний. Например, рассмотрим следующий фрагмент:

```
int x, y, z;  
x = y = z = 100;      // установить в x, y, и z значение 100
```

Этот фрагмент устанавливает в переменных `x`, `y`, и `z` значение 100 с помощью одиночной операции присваивания. Это работает, потому что `=` представляет собой операцию, которая выдает значение правого выражения. Таким образом, значением выражения `z=100` является 100, которое присваивается переменной `y`, а затем, в свою очередь, присваивается переменной `x`. Использование "цепочки присваиваний" — это просто способ установки общего значения в группе переменных.

## Условная операция

Java включает специальную троичную (с тремя operandами) **условную операцию**, которая может заменять некоторые типы условных операторов (например, `if-then-else`). Данная операция использует два символа (`? :`) и три операнда. Эта конструкция работает в Java во многом подобно аналогичной операции в C и C++. Она может показаться сначала довольно туманной, но однажды усвоенная может использоваться очень эффективно. Условная операция (`? :`) имеет следующую общую форму:

`expression1 ? expression2 : expression3`

Здесь `expression1` может быть любым выражением, которое производит boolean-значение. Если оно `true`, то вычисляется `expression2`, иначе (если `expression1` — `false`) вычисляется `expression3`. Результатом условной операции является значение вычисляемого выражения. Требуется, чтобы как `expression2`, так и `expression3` возвращали один и тот же тип, который не может быть `void`. Пример использования условной операции:

```
ratio = denom == 0 ? 0 : num / denom;
```

Когда Java оценивает это выражение присваивания, сначала просматривается выражение слева от вопросительного знака. Если `denom` равняется нулю, то оценивается и используется в качестве значения всего условного выражения выражение между вопросительным знаком и двоеточием. Если `denom` не равняется нулю, то оценивается и используется в качестве значения всего условного выражения выражение *после* двоеточия. Затем результат условной операции присваивается переменной `ratio`.

Ниже показана программа, которая демонстрирует условную операцию. Она используется для получения абсолютного значения переменной.

```
// Демонстрирует (? :)
class Ternary {
    public static void main(String args[]) {
        int i, k;

        i = 10;
        k = i < 0 ? -i : i;           // получить абсолютное значение i
        System.out.print("Абсолютное значение ");
        System.out.println(i + " равно " + k);

        i = -10;
        k = i < 0 ? -i : i;           // получить абсолютное значение i
        System.out.print("Абсолютное значение ");
        System.out.println(i + " равно " + k);
    }
}
```

Вывод, генерируемый программой:

Абсолютное значение 10 равно 10  
Абсолютное значение -10 равно 10

## Старшинство операций

Ниже представлен порядок выполнения (старшинство, приоритет) операций Java в выражениях. (Самый высокий приоритет имеют скобки, самый низкий — оператор присваивания.) Обратите внимание, что в первой строке показаны элементы, о которых вы, возможно, обычно не думаете, как об операциях — круглые скобки, квадратные скобки и точка. Круглые скобки используются, чтобы изменить старшинство операции. Как вы знаете из предыдущей главы, квадратные скобки обеспечивают индексацию элементов массива. Точечная операция используется для разыменования объектов и будет обсуждена позже.

- |              |           |
|--------------|-----------|
| 1. () [ ] .  | 8. &      |
| 2. ++ -- ~ ! | 9. ^      |
| 3. * / %     | 10.       |
| 4. + -       | 11. &&    |
| 5. >> >>> << | 12.       |
| 6. > >= < <= | 13. ?:    |
| 7. == !=     | 14. = op= |

## Использование круглых скобок

Круглые скобки повышают старшинство операций, которые находятся внутри них. Это часто необходимо для получения желаемого результата. Например, рассмотрим следующее выражение:

```
a >> b + 3
```

Это выражение сначала добавляет 3 к *b* и затем сдвигает вправо *a* на полученнное в результате этого сложения число позиций. Данное выражение можно переписать, используя избыточные круглые скобки, например, так:

```
a >> (b + 3)
```

Однако, если вы хотите сначала сдвинуть *a* право на *b* позиций и затем добавить 3 к этому результату, вам нужно будет расставить скобки в выражение следующим образом:

```
(a >> b) + 3
```

В дополнение к изменению нормального старшинства операций, круглые скобки иногда можно использовать просто для прояснения выражения. Для постороннего читателя вашего кода сложное выражение, возможно, будет трудно понять. Добавление избыточных, но что-то проясняющих круглых скобок в сложных выражениях, поможет избежать путаницы. Например, какое из следующих выражений легче для чтения?

```
a | 4 + c >> b & 7 || b > a % 3  
(a | (((4 + c) >> b) & 7)) || (b > (a % 3))
```

С другой стороны, круглые скобки (избыточные или нет) не ухудшают эффективности вашей программы. Поэтому их добавление для уменьшения неоднозначности не воздействует негативно на вашу программу.



## ГЛАВА 5

# Управляющие операторы

*Управляющие операторы* используются в языках программирования для того, чтобы продвигать и разветвлять поток выполнения, основываясь на изменениях состояния программы. Операторы программного управления Java можно разделить на следующие категории: ветвление, повторение (итерация) и переход. Операторы ветвления выбирают различные пути работы программ, основываясь на результатах вычисления выражений или на состояниях переменных. Операторы повторения дают возможность выполнять несколько раз операторы программы (эти операторы формируют циклы). Операторы перехода позволяют выполнять программу непоследовательно. В этой главе рассматриваются все операторы управления Java.

### Замечание

Если вы знаете C/C++, то операторы управления вам знакомы. Операторы управления Java почти идентичны их C/C++ аналогам. Однако существует несколько различий, особенно в операторах `continue` и `break`.

## Операторы выбора Java

Java поддерживает два оператора выбора `if` и `switch`. Эти операторы позволяют управлять потоком выполнения программы, основываясь на условиях, проверяемых только во время выполнения. Если ваш прошлый опыт в программировании не включает C/C++, вы будете приятно удивлены мощью и гибкостью этих двух операторов.

### Оператор `if`

Оператор `if` был представлен в главе 2. Здесь он рассматривается подробнее. `if` — это оператор условного перехода. Он направляет выполнение программы по двум разным маршрутам. Общая форма оператора `if`:

```
if (condition) statement1;
else statement2;
```

где *statementN* — одиночный или составной (включенный в фигурные скобки, т. е. в блок) оператор маршрута N (N — номер маршрута, для if он равен 1 или 2); *condition* — любое выражение, которое возвращает значение типа boolean. Предложение else — необязательное.

Оператор if работает так: если *condition* имеет значение true, то выполняется *statement1*, иначе — выполняется *statement2* (если он присутствует). Оба оператора вместе не будут выполняться ни в коем случае. Например, рассмотрим следующий фрагмент:

```
int a, b;
// ...
if (a < b) a = 0;
else b = 0;
```

Здесь, если a меньше, чем b, то a устанавливается в нуль, иначе — b устанавливается в нуль. Обе переменные сразу в нуль никогда не устанавливаются.

Чаще всего для управления оператором if используется выражение, включающее операции отношений. Однако в этом нет технической необходимости. Можно управлять оператором if, используя одиночную boolean переменную, как показано в следующем кодовом фрагменте:

```
boolean dataAvailable;
// ...
if (dataAvailable)
    ProcessData();
else
    waitForMoreData();
```

Помните, что только один оператор может появляться сразу после if или else. Если нужно включить в ветви большее количество операторов, то следует создать блок, как сделано, например, в следующем фрагменте:

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
    ProcessData();
    bytesAvailable -= n;
}
else
    waitForMoreData();
```

Здесь будут выполняться оба оператора в блоке if, если bytesAvailable больше нуля.

Некоторые программисты находят удобным включать фигурные скобки в каждую ветвь оператора `if`, даже когда там имеется только один оператор. Это облегчает последующее добавление других операторов в ветви `if`, и не нужно волноваться относительно пропуска фигурных скобок. Действительно, их пропуск при определении блока, когда они необходимы, — обычная причина ошибок. Например, рассмотрим следующий кодовый фрагмент:

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
    ProcessData();
    bytesAvailable -= n;
}
else
    waitForMoreData();
bytesAvailable = n;
```

Кажется ясно, что, судя по уровню отступа, оператор `bytesAvailable = n` предназначался для выполнения внутри предложения `else`. Однако, как вы помните, пробелы незначимы для Java, и у компилятора нет никакого способа распознать ваши намерения. Этот код будет скомпилирован без предупреждений, но во время выполнения будет вести себя неправильно. Предшествующий пример исправляется так:

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
    ProcessData();
    bytesAvailable -= n;
}
else {
    waitForMoreData();
    bytesAvailable = n;
}
```

Здесь всего-навсего вставлены пропущенные блочные скобки.

## **Вложенные if**

**Вложенный if** — это оператор `if`, который размещен внутри другого `if`- или `else`-оператора. Вложение `if` — обычный прием в программировании. Когда вы вкладываете `if`, нужно помнить, что оператор `else` всегда относится к самому близкому оператору `if`, который находится в том же блоке, что и `else`. Например:

```
if(i == 10) {
    if(j < 20) a = b;
```

```
if(k > 100) c = d; // этот if
else a = c;        // связан с этим else
}
else a = d;        // этот else относится к if(i == 10)
```

Как указывают комментарии, заключительный `else` не связан с `if(j < 20)`, потому что он не в одном с ним блоке (хотя это и самый близкий `if` без `else`). Последний `else` связан с `if(i == 10)`. Внутренний `else` относится к `if(k > 100)`, потому что это самый близкий `if` в пределах данного блока.

## Многозвездный `if-else-if`

Общую программную конструкцию, которая основана на последовательности вложенных `if`, называют *многозвездным (ladder) if-else-if*. Эта конструкция выглядит примерно так:

```
if(condition1)
    statement1;
else if(condition2)
    statement2;
else if(condition3)
    statement3;

.
.

else
    statementN;
```

Операторы `if` выполняются сверху вниз. Как только одно из условий, управляющих оператором `if`, становится `true`, оператор, связанный с этим `if`, выполняется, а остальная часть многозвездной схемы пропускается. Если ни одно из условий не `true`, то будет выполнен последний оператор `else`. Заключительный `else` действует как условие по умолчанию: если все другие условные проверки не успешны, то выполняется последний оператор `else`. Если последний `else` отсутствует и все другие условия — `false`, то никакого действия выполняться не будет.

Программа, использующая многозвездный `if-else-if` для определения времени года, в котором находится указанный (в переменной `month`) месяц:

```
// Демонстрирует операторы if-else-if.
class IfElse {
    public static void main(String args[]) {
        int month = 4; // Апрель
        String season;

        if(month == 12 || month == 1 || month == 2)
            season = "Зима";
        else if(month == 3 || month == 4 || month == 5)
            season = "Весна";
        else if(month == 6 || month == 7 || month == 8)
            season = "Лето";
        else
            season = "Осень";
    }
}
```

```

else if(month == 3 || month == 4 || month == 5)
    season = "Весна";
else if(month == 6 || month == 7 || month == 8)
    season = "Лето";
else if(month == 9 || month == 10 || month == 11)
    season = "Осень";
else
    season = "Непонятный месяц";

System.out.println("Апрель это " + season + ".");
}
}

```

Пример вывода этой программой:

Апрель это Весна.

Независимо от того, какое значение вы задаете для переменной `month`, в многозначной схеме будет выполняться один и только один оператор присваивания.

## Оператор `switch`

Оператор `switch` — это Java-оператор множественного ветвления. Он переключает выполнение на различные части кода программы, основываясь на значении выражения, и часто обеспечивает лучшую альтернативу, чем длинный ряд операторов `if-else-if`. Общая форма оператора `switch`:

```

switch (expression) {
    case value1:
        // последовательность операторов1
        break;
    case value2:
        // последовательность операторов2
        break;
    .
    .
    .
    case valueN:
        // последовательность операторовN
        break;
    default:
        // последовательность операторов по умолчанию
}

```

Здесь `expression` должно иметь тип `byte`, `short`, `int` или `char`; каждое `value`, указанное в операторах `case`, должно иметь тип, совместимый с типом

выражения. Каждое значение case должно быть уникальной константой (а не переменной). Дублирование значений case недопустимо.

Оператор switch работает следующим образом. Значение выражения сравнивается с каждым из указанных значений в case-операторах. Если соответствие найдено, то выполняется кодовая последовательность, следующая после этого оператора case. Если ни одна из case-констант не соответствует значению выражения, то выполняется оператор default. Однако оператор default необязателен. Если согласующихся case нет, и default не присутствует, то никаких дальнейших действий не выполняется.

Оператор break используется внутри switch, чтобы закончить последовательность операторов. Когда встречается оператор break, выполнение переходит к первой строке кода, которая следует за полным оператором switch. Он создает эффект досрочного выхода из switch.

Простой пример, который использует оператор switch:

```
// Простой пример с оператором switch.
class SampleSwitch {
    public static void main(String args[]) {
        for(int i=0; i<6; i++)
            switch(i) {
                case 0:
                    System.out.println("i равно нулю.");
                    break;
                case 1:
                    System.out.println("i равно единице.");
                    break;
                case 2:
                    System.out.println("i равно двум.");
                    break;
                case 3:
                    System.out.println("i равно трем.");
                    break;
                default:
                    System.out.println("i больше трех.");
            }
    }
}
```

Вывод этой программы:

```
i равно нулю.
i равно единице.
i равно двум.
i равно трем.
```

i больше трех.  
i больше трех.

Не трудно заметить, что на каждом проходе цикла выполняются операторы, связанные с case-меткой, которая соответствует переменной цикла i. Все другие case-ветви обходятся. После того как i становится больше трех, никакого соответствия case-меткам не возникает, поэтому выполняется оператор default.

Оператор break — необязательный. Если он пропускается, выполнение будет продолжено со следующей case-метки. Иногда желательно иметь множественные case-ветви без операторов break между ними. Например, рассмотрим следующую программу:

```
// switch с пропущенными операторами break.
class MissingBreak {
    public static void main(String args[]) {
        for(int i=0; i<12; i++)
            switch(i) {
                case 0:
                case 1:
                case 2:
                case 3:
                case 4:
                    System.out.println("i меньше 5");
                    break;
                case 5:
                case 6:
                case 7:
                case 8:
                case 9:
                    System.out.println("i меньше 10");
                    break;
                default:
                    System.out.println("i равно 10 или больше");
            }
    }
}
```

Эта программа генерирует следующий вывод:

```
i меньше 5
i меньше 10
i меньше 10
```

```
i  меньше 10
i  меньше 10
i  меньше 10
i  равно 10 или больше
i  равно 10 или больше
```

Можно видеть, что выполнение проходит через каждый `case`, пока не встречается оператор `break` (или не будет достигнут конец оператора `switch`).

Хотя предшествующий пример придуман, конечно, ради иллюстрации, пропуск оператора `break` имеет много практических приложений в реальных программах. Чтобы показать более жизненное использование `switch`, рассмотрим следующий вариант примера с временами года, приведенного ранее. Эта версия использует `switch`, чтобы обеспечить более эффективную реализацию.

```
// Улучшенная версия программы с временами года.
class Switch {
    public static void main(String args[]) {
        int month = 4;
        String season;
        switch (month) {
            case 12:
            case 1:
            case 2:
                season = "Зима";
                break;
            case 3:
            case 4:
            case 5:
                season = "Весна";
                break;
            case 6:
            case 7:
            case 8:
                season = "Лето";
                break;
            case 9:
            case 10:
            case 11:
                season = "Осень";
                break;
            default:
                season = "Непонятный месяц";
        }
        System.out.println("Апрель это " + season + ".");
    }
}
```

## Вложенные switch-операторы

Вы можете использовать `switch` как часть последовательности операторов некоторого внешнего `switch`. Это называется *вложенным* `switch`. Так как `switch`-оператор определяет свой собственный блок, никакие конфликты между `case`-метками во внутреннем и внешнем `switch` не возникают. Например, следующий фрагмент совершенно законен:

```
switch(count) {
    case 1:
        switch(target) { // вложенный switch
            case 0:
                System.out.println("target равен нулю");
                break;
            case 1: // нет конфликтов с внешним switch
                System.out.printIn("target равен единице");
                break;
        }
        break;
    case 2: //...
}
```

Здесь утверждение с меткой `case 1` во внутреннем `switch` не находится в противоречии с утверждением `case 1` во внешнем `switch`. Переменная `count` сравнивается только с `case`-меткой на внешнем уровне. Если `count` равна 1, то переменная `target` сравнивается с внутренней `case`-меткой. Обратите внимание на три важные особенности оператора `switch`.

- `Switch` отличается от `if` тем, что может проверять только равенство (своей переменной с `case`-метками), тогда как `if` может оценивать любой тип булевского выражения. То есть, `switch` отыскивает только соответствие между значением выражения и одной из его `case`-меток.
- Никакие две `case`-метки внутри `switch` не могут иметь идентичных значений. Однако операторы `switch`, включенные во внешний `switch`, могут, конечно, иметь общие `case`-метки с внутренним `switch`.
- Оператор `switch` обычно более эффективен, чем набор вложенных `if`.

Последний пункт особенно интересен, потому что он дает возможность понять, как работает компилятор Java. Когда компилируется оператор `switch`, компилятор Java просматривает каждую из `case`-констант и создает "таблицу переходов", которая будет использоваться для выбора пути выполнения в зависимости от значения выражения. Поэтому, если нужно выбирать среди большой группы значений, оператор `switch`, выполнится намного быстрее, чем эквивалентная кодированная логика, использующая последовательности `if-else`. Компилятор может делать это, т. к. он знает, что все `case`-константы имеют один и тот же тип и просто должны быть проверены на

равенство с выражением оператора `switch`. Компилятор не имеет аналогичных знаний длинного списка выражений `if`.

## Операторы цикла

Операторы цикла (итераций) Java — это `for`, `while` и `do while`. Как вы, вероятно, знаете, цикл повторно выполняет один и тот же набор команд, пока не выполнится условие завершения. Java имеет набор циклических операторов для удовлетворения любой потребности программирования.

### Оператор цикла `while`

Цикл `while` — наиболее фундаментальный оператор цикла Java. Он повторяет оператор или блок операторов, пока его управляющее выражение имеет значение `true`. Вот его общая форма:

```
while(condition) {  
    // тело цикла  
}
```

Здесь `condition` может быть любым булевским выражением. Тело цикла будет выполняться до тех пор, пока условное выражение имеет значение `true`. Когда `condition` становится `false`, управление передается строке программы, следующей непосредственно за циклом. Фигурные скобки не нужны, если повторяется одиночный оператор.

Следующий `while`-цикл считает в обратном порядке от 10, печатая точно десять "тик"-строк:

```
// Демонстрирует while-цикл.  
class While {  
    public static void main(String args[]) {  
        int n = 10;  
  
        while(n > 0) {  
            System.out.println("тик " + n);  
            n--;  
        }  
    }  
}
```

Когда вы выполните эту программу, она "тикнет" ровно десять раз:

```
тик 10  
тик 9  
тик 8  
тик 7
```

```
тик 6
тик 5
тик 4
тик 3
тик 2
тик 1
```

Так как цикл `while` оценивает условное выражение в самом начале своей работы, тело цикла не будет выполняться ни одного раза, если условие сразу же имеет значение `false`. Например, в следующем фрагменте обращение к `println()` никогда не выполняется:

```
int a = 10, b = 20;

while(a > b)
    System.out.println("Эта строка никогда не выведется");
```

Тело цикла `while` (или любого другого цикла Java) может быть пустым. Это потому, что в Java синтаксически допустим *пустой (null) оператор* (который состоит только из точки с запятой). Например, рассмотрим следующую программу:

```
// Тело цикла может быть пустым.
class NoBody {
    public static void main(String args[]) {
        int i, j;

        i = 100;
        j = 200;

        // найти среднюю точку между i и j
        while(++i < --j);    // в этом цикле нет тела

        System.out.println("Средняя точка " + i);
    }
}
```

Эта программа находит среднюю точку между `i` и `j`. Она генерирует следующий вывод:

Средняя точка 150

Посмотрим, как цикл `while` работает. Значения переменной `i` формирует операция инкремента, а значение `j` — декремента. Затем эти значения сравниваются друг с другом. Если новое значение `i` все еще меньше, чем новое значение `j`, цикл повторяется. Если `i` равно или больше `j`, цикл останавливается. На выходе из цикла `i` будет содержать значение, которое является средним между первоначальными значениями `i` и `j`. (Заметим, что эта процедура работает только тогда, когда в самом начале `i` меньше, чем `j`.) Не

трудно видеть, что нет никакой потребности в теле цикла, все действия происходят непосредственно в самом условном выражении. В профессионально составленном коде Java короткие циклы часто кодируются без тел, когда управляющее выражение может само обрабатывать все детали.

## Оператор цикла *do while*

Как вы только что видели, если условное выражение, управляющее циклом *while*, первоначально имеет значение *false*, то тело цикла не будет выполняться вообще. Однако иногда желательно выполнить тело *while*-цикла по крайней мере один раз, даже если условное выражение сначала имеет значение *false*. Другими словами, существуют ситуации, когда вы хотели бы проверить выражение завершения в конце цикла, а не в начале. К счастью, в Java имеется цикл *do while*, который делает именно это. Цикл *do while* всегда выполняет свое тело по крайней мере один раз, потому что его условное выражение размещается в конце цикла. Общая форма цикла:

```
do{  
    // тело цикла  
} while (condition);
```

Каждая итерация цикла *do while* сначала выполняет тело цикла, а затем оценивает условное выражение. Если это выражение — *true*, цикл повторится. Иначе, цикл заканчивается. Как и у всех Java-циклов, *condition* должно быть булевским выражением.

Ниже показана переделанная версия программы с "тиками", которая демонстрирует цикл *do while*. Она генерирует тот же вывод, что и прежде.

```
// Демонстрирует do-while цикл.  
class DoWhile {  
    public static void main(String args[]) {  
        int n = 10;  
  
        do {  
            System.out.println("тик " + n);  
            n--;  
        } while(n > 0);  
    }  
}
```

Цикл в этой программе, хотя и технически корректный, можно записать более эффективно следующим образом:

```
do {  
    System.out.println("тик " + n);  
} while(--n > 0);
```

В этом примере, выражение `(--n > 0)` объединяет декремент `n` и проверку на нуль в одно выражение. Вот как это работает. Сначала выполняется оператор `--n`, декрементируя `n` и возвращая его новое значение. Затем это значение сравнивается с нулем. Если оно больше нуля, цикл продолжается, в противном случае цикл заканчивается.

Цикл `do while` особенно полезен, когда вы обрабатываете выбор из меню, потому что обычно требуется, чтобы тело цикла меню выполнялось по крайней мере один раз. Рассмотрим следующую программу, которая реализует очень простую справочную систему для Java-операторов выбора и повторения:

```
// Использование do-while для обработки выборки из меню
// -- простая справочная (help) система.
class Menu {
    public static void main(String args[])
        throws java.io.IOException {
        char choice;

        do {
            System.out.println("Справка:");
            System.out.println(" 1. if");
            System.out.println(" 2. switch");
            System.out.println(" 3. while");
            System.out.println(" 4. do while");
            System.out.println(" 5. for\n");
            System.out.println("Выберите раздел:");
            choice = (char) System.in.read();
        } while( choice < '1' || choice > '5');

        System.out.println("\n");

        switch(choice) {
            case '1':
                System.out.println("Условный оператор if:\n");
                System.out.println("if(condition) statement1;");
                System.out.println("else statement2;");
                break;
            case '2':
                System.out.println("Оператор выбора switch:\n");
                System.out.println("switch(expression) {");
                System.out.println("  case value:");
                System.out.println("      последовательность операторов");
                System.out.println("      break;");
                System.out.println("  // ...");
                System.out.println("}");
                break;
        }
    }
}
```

```
case '3':
    System.out.println("Оператор цикла while:\n");
    System.out.println("while(condition) тело цикла;");
    break;
case '4':
    System.out.println("Оператор цикла do while:\n");
    System.out.println("do (");
    System.out.println(" тело цикла;");
    System.out.println("} while (condition);");
    break;
case '5':
    System.out.println("Оператор цикла for:\n");
    System.out.print("for(init; condition; iteration));");
    System.out.println(" тело цикла;");
    break;
}
}
}
```

Пример вывода, выполненного этой программой:

Справка:

1. if
2. switch
3. while
4. do while
5. for

Выберите раздел:

4

Оператор цикла do while:

```
do {
    тело цикла;
} while (condition);
```

Чтобы проверить, верно ли пользователь сделал правильный выбор, здесь используется цикл do while. Если правильного выбора нет, то номер раздела справки запрашивается повторно. Так как меню должно быть отображено по крайней мере один раз, то do while — идеальный цикл для выполнения этой задачи.

Несколько других замечаний относительно этого примера: обратите внимание, что символы читаются с клавиатуры с помощью вызова `System.in.read()`. Это одна из функций консольного ввода языка Java. Хотя консольные методы ввода/вывода Java не будут обсуждаться подробно до главы 12, `System.in.read()` используется здесь для того, чтобы узнать о выборе пользователя. Он читает символы при стандартном вводе (возвращая целые числа, что объясняет, почему возвращаемое значение приводится к типу `char`).

По умолчанию стандартный ввод является буферизированной строкой, так что вы должны нажать клавишу <Enter>, прежде чем любые символы, которые вы печатаете, будут переданы вашей программе.

Консольный ввод Java весьма ограничен и неудобен для работы. Кроме того, большинство реальных Java-программ и апплетов являются графическими и работают с окнами. По этим причинам, в данной книге консольный ввод используется не слишком часто. Однако в этом контексте он полезен. Еще один момент: поскольку используется метод `System.in.read()`, программа должна определить предложение

```
throws java.io.IOException
```

Эта строка необходима для обработки ошибок ввода и является частью системы обработки особых ситуаций языка Java, которая обсуждается в главе 10.

## Оператор цикла `for`

Простая форма цикла `for` была представлена в главе 2. Далее вы увидите, что это мощная и гибкая конструкция. Общая форма оператора `for`:

```
for(initialization; condition; iteration) {
    // тело цикла
}
```

Если повторяется только один оператор, фигурные скобки не нужны.

Цикл `for` работает следующим образом. В начале работы цикла выполняется выражение `initialization`. В общем случае это выражение устанавливает значение *переменной управления циклом*, которая действует как счетчик. Важно понять, что выражение инициализации выполняется только один раз. Затем оценивается `condition`. Оно должно быть булевским выражением и обычно сравнивает переменную управления циклом с некоторым граничным значением. Если это выражение — `true`, то отрабатывают операторы из тела цикла, если — `false`, цикл заканчивается. Далее выполняется часть цикла `iteration`. Обычно это выражение, которое осуществляет инкрементные или декрементные операции с переменной управления циклом. Затем цикл реализовывает итерации. В каждом проходе цикла — сначала оценивается условное выражение, потом выполняется тело цикла и затем — выражение `iteration`. Этот процесс повторяется до тех пор, пока управляющее выражение не станет `false`.

Ниже приводится версия программы "тиков", которая использует цикл `for`:

```
// Демонстрирует for-цикл.
class ForTick {
    public static void main(String args[]) {
        int n;
```

```

for(n=10; n>0; n--)
    System.out.println("тик " + n);
}
}

```

## Объявление переменных управления внутри цикла for

Часто переменная, которая управляет циклом `for`, необходима только для целей цикла и не используется в другом месте. Когда дело обстоит так, можно объявить переменную внутри инициализационной части `for`. Например, предшествующая программа перекодирована так, чтобы переменная управления циклом `n` объявлялась типом `int` внутри заголовка `for`:

```

// Определение переменной управления циклом внутри for.
class ForTick {
    public static void main(String args[]) {
        // здесь n объявлена внутри заголовка for-цикла
        for(int n=10; n>0; n--)
            System.out.println("тик " + n);
    }
}

```

Когда переменная объявляется внутри заголовка `for`-цикла, важно помнить, что область ее действия заканчивается там же, где и у `for`-оператора (т. е. область действия переменной ограничена циклом `for`). Вне цикла `for` переменная прекратит существование. Если нужно использовать управляющую переменную цикла в другом месте программы, ее не следует объявлять внутри цикла `for`.

Когда управляющая переменная цикла не нужна где-то в другом месте, большинство Java-программистов объявляет ее внутри `for`-заголовка. Ниже показана небольшая программа, которая проверяет, является ли число простым<sup>1</sup>. Обратите внимание, что управляющая переменная цикла `i` объявлена внутри `for`-заголовка, т. к. она нигде больше не нужна.

```

// Проверка на принадлежность к категории простых чисел.
class FindPrime {
    public static void main(String args[]) {
        int num;
        boolean isPrime = true;

```

---

<sup>1</sup> Простым называется число, которое делится только на собственное значение и на 1 (например, простыми являются числа 1, 2, 3, 5, 7, 11, 13 и т. д.). — Примеч. ред.

```

num = 14;
for(int i=2; i < num/2; i++) {
    if((num % i) == 0) {
        isPrime = false;
        break;
    }
}
if(isPrime) System.out.println("Простое число");
else System.out.println("Число не является простым");
}
}

```

## Использование запятой

Может возникнуть желание включить более одного оператора в инициализационную и итерационную части цикла for. Например, рассмотрим цикл следующей программы:

```

class Sample {
    public static void main(String args[]) {
        int a, b;

        b = 4;
        for(a=1; a<b; a++) {
            System.out.println("a = " + a);
            System.out.println("b = " + b);
            b--;
        }
    }
}

```

Не трудно заметить, что цикл зависит от взаимодействия двух переменных. Так как цикл управляет двумя переменными, было бы полезно включить их непосредственно в заголовок оператора for вместо того, чтобы обрабатывать переменную b вручную. К счастью, в Java обеспечена такая возможность. Чтобы позволить двум или большему числу переменных управлять циклом for, Java разрешает вам включать множественные операторы как в инициализационную, так и в итерационную части for-заголовка. Каждый оператор отделяется от следующего запятой.

Используя запятую, предыдущий цикл for можно закодировать более эффективно:

```

// Использование запятой.
class Comma {
    public static void main(String args[]) {
        int a, b;

```

```
for(a=1, b=4; a<b; a++, b--) {
    System.out.println("a = " + a);
    System.out.println("b = " + b);
}
```

В этом примере инициализационная часть устанавливает значения как для *a*, так и для *b*. Два разделенных запятыми оператора в итерационной части выполняются каждый раз, когда цикл повторяется. Программа генерирует следующий вывод:

```
a = 1
b = 4
a = 2
b = 3
```

### Замечание

Если вы знакомы с C/C++, то знаете, что на этих языках запятая — это оператор, который можно использовать в любом правильном выражении. В Java, однако, дело обстоит не так. В Java запятая — это разделитель, который применяется только в цикле *for*.

## Некоторые разновидности цикла *for*

Цикл *for* поддерживает ряд разновидностей, которые усиливают его гибкость и расширяют применимость. Причина такой гибкости заключается в том, что три части его заголовка — инициализация, проверка условия и итерация — могут использоваться не только для внутренних целей. На самом деле эти три части цикла *for* можно применять для решения любой задачи. Рассмотрим некоторые примеры.

Одна из наиболее общих разновидностей включает условное выражение. Это выражение не нуждается в сравнении переменной управления циклом с некоторым значением. На самом деле, условие, управляющее циклом *for*, может быть любым булевским выражением. Например, рассмотрим следующий фрагмент:

```
boolean done = false;

for(int i=1; !done; i++) {
    // ...
    if(interrupted()) done = true;
}
```

В этом примере цикл *for* продолжает выполняться до тех пор, пока boolean-переменная *done* имеет значение *true*. Он не проверяет значения *i*.

Другая интересная разновидность цикла `for` — когда инициализационное или итерационное выражение (или оба) могут отсутствовать, как показано в следующей программе:

```
// Части цикла for могут быть пустыми.
class ForVar {
    public static void main(String args[]) {
        int i;
        boolean done = false;

        i = 0;
        for(; !done;) {
            System.out.println("i равно " + i);
            if(i == 10) done = true;
            i++;
        }
    }
}
```

Здесь инициализационное и итерационное выражения были удалены из `for`, так что эти части `for` пусты. В данном примере это не имеет никакого значения и может рассматриваться просто как ограниченный стиль программирования. Однако могут возникнуть ситуации, когда такой подход оправдан. Например, если начальное условие устанавливается через сложное выражение в другом месте программы или управляющая переменная цикла изменяется непоследовательным способом, определяемым действиями, происходящими в теле цикла, то удобнее оставить эти части заголовка цикла `for` пустыми.

Еще одна разновидность цикла `for` — можно преднамеренно создать бесконечный цикл (цикл, который никогда не заканчивается), если оставить все три части заголовка `for`-цикла пустыми. Например:

```
for(;;) {
    // ...
}
```

Этот цикл будет выполняться всегда, потому что нет условия, при котором он завершится. Хотя существуют некоторые программы, такие как командные процессоры операционных систем, требующие бесконечного цикла, большинство "бесконечных циклов" в действительности является просто циклами со специальными требованиями завершения. Как вы вскоре увидите, существует способ завершить цикл — даже бесконечный (подобный показанному выше), который не использует нормальное условное выражение цикла.

## **Вложенные циклы**

Подобно всем другим языкам программирования Java допускает вложение циклов. То есть один цикл может быть внутри другого. Например, следующая программа, вкладывает `for`-циклы:

```
// Циклы могут быть вложенными.
class Nested {
    public static void main(String args[]) {
        int i, j;

        for(i=0; i<10; i++) {
            for(j=i; j<10; j++)
                System.out.print(".");
            System.out.println();
        }
    }
}
```

Вывод этой программы:

A 10x10 grid of black dots, representing a 10x10 matrix or a 10x10 grid of data points.

## Операторы перехода

Java поддерживает три оператора перехода — `break`, `continue` и `return`. Они передают управление в другую часть вашей программы. Рассмотрим каждый из операторов подробно.

## **Замечание**

В дополнение к операторам перехода, обсуждаемым здесь, Java поддерживает другой способ, с помощью которого вы можете изменять поток выполнения программы — обработку исключений (особых ситуаций). Обработка исключений обеспечивает структурированный метод, с помощью которого ваша программа может отлавливать и обрабатывать ошибки во время выполнения. Она поддерживается ключевыми словами `try`, `catch`, `throw`, `throws` и `finally`. Помимо существующего механизма обработки исключений позволяет вашей программе выполнять нелокальный переход. Так как обработка особых ситуаций — большая тема, она обсуждается в своей собственной главе (см. гл. 10).

## Использование оператора *break*

Оператор *break* в Java используется в трех случаях. Во-первых, как вы уже видели, он заканчивает последовательность операторов в ветвях оператора *switch*. Во-вторых, его можно использовать для выхода из цикла. В-третьих, он может применяться как "цивилизованная" форма оператора безусловного перехода *goto*. В этом разделе объясняются два последних случая.

### Использование *break* для выхода из цикла

Используя *break*, вы можете форсировать немедленное завершение цикла, обходя условное выражение и любой остающийся код в теле цикла. Когда оператор *break* встречается внутри цикла, второй заканчивается и программное управление передается оператору, следующему за ним. Простой пример:

```
// Использование break для выхода из цикла.
class BreakLoop {
    public static void main(String args[]) {
        for(int i=0; i<100; i++) {
            if(i == 10) break; // завершить цикл, если i = 10
            System.out.println("i: " + i);
        }
        System.out.println("Цикл завершен.");
    }
}
```

Эта программа генерирует следующий вывод:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Цикл завершен.
```

Хотя цикл *for* разработан здесь для выполнения своих операторов от 0 до 99 раз, оператор *break* заставляет его завершиться раньше, когда *i* равно 10.

Оператор *break* может использоваться с любым из циклов Java, включая преднамеренно бесконечные циклы. Например, ниже показана предыдущая программа, закодированная при помощи цикла *while*. Вывод этой программы такой же, как у ее предшественницы.

```
// Использование break для выхода из while-цикла.
class BreakLoop2 {
    public static void main(String args[]) {
        int i = 0;

        while(i < 100) {
            if(i == 10) break; // закончить цикл, если i = 10
            System.out.println("i: " + i);
            i++;
        }
        System.out.println("Цикл завершен.");
    }
}
```

При использовании внутри набора вложенных циклов оператор `break` будет выходить только из самого внутреннего цикла. Например:

```
// Использование break во вложенных циклах.
class BreakLoop3 {
    public static void main(String args[]) {
        for(int i=0; i<3; i++) {
            System.out.print("Итерация " + i + ": ");
            for(int j=0; j<100; j++) {
                if(j == 10) break; // закончить цикл, если j = 10
                System.out.print(j + " ");
            }
            System.out.println();
        }
        System.out.println("Цикл завершен.");
    }
}
```

Эта программа генерирует следующий вывод:

Итерация 0: 0 1 2 3 4 5 6 7 8 9

Итерация 1: 0 1 2 3 4 5 6 7 8 9

Итерация 2: 0 1 2 3 4 5 6 7 8 9

Цикл завершен.

Как вы видите, оператор `break` во внутреннем цикле вызывает завершение только этого цикла. Внешний цикл не затрагивается.

Сделаем еще два замечания относительно `break`. Во-первых, в цикле может появиться несколько операторов `break`. Однако будьте осторожны. Слишком большое их количество создает тенденцию к деструктуризации вашего кода. Во-вторых, `break`, который завершает оператор `switch`, воздействует только на данный `switch`-оператор (но не на включающие его циклы).

## Замечание

Break не проектировался в качестве нормального средства завершения цикла. Эту цель обслуживает условное выражение заголовка цикла. Оператор `break` следует использовать для прерывания цикла только тогда, когда возникают некоторые специальные ситуации.

## Использование `break` как формы `goto`

В дополнение к применению в операторах `switch` и циклах `break` можно также использовать отдельно, в качестве "цивилизованной" формы оператора `goto`. Java не содержит оператора `goto`, потому что он выполняет переход произвольным и неструктурированным способом. Код, интенсивно использующий `goto`, обычно трудно понять и поддерживать. Он также отменяет некоторые оптимизации компилятора. Существует, однако, несколько мест в программе, где `goto` — ценная и законная конструкция управления потоком выполнения. Например, `goto` может быть полезен, когда вы выходите из глубоко вложенного набора циклов. Чтобы обрабатывать такие ситуации, Java определяет расширенную форму оператора `break`. Используя ее, вы можете выйти из одного или большего количества блоков кода. Этим блокам не нужно быть частью цикла или оператора `switch`. Это может быть любой блок. Далее, вы можете определить точно, где выполнение будет продолжено, потому что данная форма `break` работает с меткой и обеспечивает преимущества `goto`, минуя его проблемы. Оператор `break` с меткой имеет следующую общую форму:

```
break label;
```

Здесь `label` — имя метки, которая идентифицирует некоторый блок кода. Когда эта форма `break` выполняется, управление передается из именованного блока кода (чья метка указана в операторе `break`) на следующий за этим блоком оператор. Помеченный блок кода обязательно должен включать данный оператор `break`, но не требуется, чтобы это включение было непосредственным (т. е. `break` может включаться не прямо в блок со своей меткой, а во вложенный в него блок, возможно, тоже помеченный). Это означает, что вы можете использовать помеченный оператор `break`, чтобы выйти из набора вложенных блоков. Но вы не имеете возможности использовать `break` для передачи управления блоку кода, который не включает оператор `break`.

Для именования блока поместите метку в его начале (перед открывающей блок фигурной скобкой). *Метка* — это любой допустимый идентификатор Java, за которым следует двоеточие. После маркировки блока, его метку можно использовать как аргумент оператора `break`. Это приведет к тому, что выполнение будет продолжено с конца помеченного блока. Например, следующая программа содержит три вложенных блока, причем каждый имеет свою собственную метку. Оператор `break` осуществляет переход вперед, за

конец блока, маркированного меткой second, пропуская два оператора `println()`.

```
// Использование break как цивилизованной формы goto.
class Break {
    public static void main(String args[]) {
        boolean t = true;

        first: {
            second: {
                third: {
                    System.out.println("Перед оператором break.");
                    if(t) break second; // выход из блока second
                    System.out.println("Данный оператор никогда не выполнится");
                }
                System.out.println("Данный оператор никогда не выполнится ");
            }
            System.out.println("Данный оператор размещен после блока second.");
        }
    }
}
```

Выполнение этой программы генерирует следующий вывод:

Перед оператором break.

Данный оператор размещен после блока second.

Одним из наиболее частых применений помеченного оператора `break` является выход из вложенных циклов. Например, в следующей программе внешний цикл выполняется только один раз:

```
// Использование break для выхода из вложенных циклов.
class BreakLoop4 {
    public static void main(String args[]) {
        outer: for(int i=0; i<3; i++) {
            System.out.print("Итерация " + i + ": ");
            for(int j=0; j<100; j++) {
                if(j == 10) break outer; // выйти из обоих циклов
                System.out.print(j + " ");
            }
            System.out.println("Эта строка никогда не будет выведена");
        }
        System.out.println("Цикл завершен.");
    }
}
```

Программа генерирует следующий вывод:

Итерация 0: 0 1 2 3 4 5 6 7 8 9 Цикл завершен.

Нетрудно заметить, что при прерывании внутреннего цикла до конца внешнего заканчиваются оба цикла.

Имейте в виду, что вы не можете сделать break-переход к любой метке, которая не определена для включающего блока. Например, следующая программа недопустима и не будет компилироваться:

```
// Эта программа содержит ошибку.
class BreakErr {
    public static void main(String args[]) {
        one: for(int i=0; i<3; i++) {
            System.out.print("Итерация " + i + ": ");
        }

        for(int j=0; j<100; j++) {
            if(j == 10) break one; // Не верно!
            System.out.print(j + " ");
        }
    }
}
```

Так как цикл, помеченный как `one`, не включает оператор `break`, передать управление этому блоку невозможно.

## Использование оператора *continue*

Иногда полезно начать очередную итерацию цикла пораньше. То есть нужно продолжить выполнение цикла, но прекратить обработку остатка кода в его теле для данной частной итерации. На самом деле это — goto-переход мимо следующих операций тела в конец блока цикла. Такое действие выполняет оператор `continue`. В циклах `while` и `do while` оператор `continue` вызывает передачу управления непосредственно условному выражению, которое управляет циклом. В цикле `for` управление переходит сначала к итерационной части оператора `for` и затем к условному выражению. Для всех трех циклов любой промежуточный код обходится.

Пример программы, которая использует `continue` для печати двух чисел на каждой строке, приведен ниже:

```
// Демонстрирует continue.
class Continue {
    public static void main(String args[]) {
        for(int i=0; i<10; i++) {
            System.out.print(i + " ");
            if (i%2 == 0) continue;
            System.out.println("");
        }
    }
}
```

Этот код использует операцию % (остаток по модулю), чтобы проверять, является ли *i* четным. Если это так, цикл продолжается без печати символа новой строки (newline). Вывод программы:

```
0 1  
2 3  
4 5  
6 7  
8 9
```

Как и в операторе `break`, в `continue` можно определить метку, указывающую, какой включающий цикл следует продолжить. Пример программы, которая использует `continue` для печати треугольной таблицы умножения от 0 до 9.

```
// Использование continue с меткой.  
class ContinueLabel {  
    public static void main(String args[]) {  
outer: for (int i=0; i<10; i++) {  
    for(int j=0; j<10; j++) {  
        if(j > i) {  
            System.out.println();  
            continue outer;  
        }  
        System.out.print(" " + (i * j));  
    }  
    System.out.println();  
}  
}
```

Оператор `continue` в этом примере завершает цикл, вычисляющий *j*, и продолжает со следующей итерации цикла, управляемого *i*. Вывод этой программы:

```
0  
0 1  
0 2 4  
0 3 6 9  
0 4 8 12 16  
0 5 10 15 20 25  
0 6 12 18 24 30 36  
0 7 14 21 28 35 42 49  
0 8 16 24 32 40 48 56 64  
0 9 18 27 36 45 54 63 72 81
```

В использовании `continue` полезен крайне редко. Одна из причин этого заключается в том, что Java обеспечивает богатый набор операторов цикла,

которые устраивают большинство приложений. Однако, для тех специальных ситуаций, в которых необходимо досрочное прекращение итерации, оператор `continue` обеспечивает структурный способ выполнения этой задачи.

## Оператор `return`

Последний оператор управления — `return`. Он используется для явного возврата из метода, т. е. передает программное управление обратно в вызывающую программу. Оператор `return` относят к категории операторов перехода. Хотя его полное обсуждение должно подождать до обсуждения методов (см. гл. 7), здесь представлен краткий обзор `return`.

Оператор `return` можно использовать в любом месте метода для выполнения перехода обратно в вызывающую этот метод программу. Оператор `return` немедленно заканчивает выполнение метода, в котором он находится. Это иллюстрирует следующий пример:

```
// Демонстрирует return.
class Return {
    public static void main(String args[]) {
        boolean t = true;

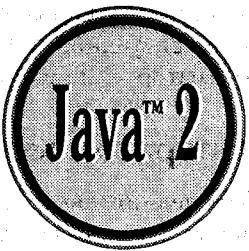
        System.out.println("Перед оператором return.");
        if(t) return; // возврат в вызывающую программу
        System.out.println("Этот оператор никогда не выполнится.");
    }
}
```

Здесь `return` выполняет возврат к Java-системе во время выполнения, т. к. именно эта система вызывает метод `main()`. Вывод этой программы:

Перед оператором `return`.

Можно заметить, что заключительный оператор `println()` не выполняется. В момент выполнения `return` управление передается обратно в вызывающую программу.

И последнее замечание. В предыдущей программе оператор `if(t)` необходим. Без него Java-компилятор выдал бы ошибку "недостижимый код", потому что он знал бы, что последний оператор `println()` никогда не будет выполняться. Чтобы предотвратить указанную ошибку, как раз и используется оператор `if` (он обманывает компилятор ради данной демонстрации).



## ГЛАВА 6

# Введение в классы

В основе языка Java лежит класс. *Класс* — это логическая конструкция, на которой построен весь язык Java, потому что такая конструкция определяет форму и природу объекта. Класс формирует также основу для объектно-ориентированного программирования в Java. Любая концепция, которую вы желаете реализовать в Java-программе, должна быть инкапсулирована в класс.

Из-за того, что класс достаточно фундаментален для Java, данная и следующие несколько глав будут посвящены этому важнейшему понятию языка. Здесь вам будут представлены основные элементы класса, и вы узнаете, как класс может использоваться для создания объектов. Вы также познакомитесь с методами, конструкторами, и ключевым словом `this`.

## Основы классов

Классы в примерах программ использовались с самого начала этой книги. Однако до сих пор применялась только самая элементарная их (классов) форма. Классы, созданные в предшествующих главах, первоначально существуют просто для того, чтобы включать метод `main()`, который вводился для демонстрации основ синтаксиса Java. Как вы увидите далее, классы являются более мощными компонентами языка, чем представленные до сих пор.

Пожалуй, наиболее важным в понятии класса является то, что он определяет *новый тип данных*. После определения новый тип можно использовать для создания *объектов* этого типа. Таким образом, класс — это *шаблон для объекта*, а объект — это *экземпляр* класса. Поскольку объект — экземпляр класса, два слова *объект* и *экземпляр* часто будут использоваться как взаимозаменяемые.

## Общая форма класса

Когда вы определяете класс, то объявляете его точную форму и природу. Делаете вы это, формируя данные, которые он содержит, и код, который оперирует с этими данными. В то время как очень простые классы могут содержать только код или только данные, наиболее реальные классы содержат и то и другое. Код класса определяет интерфейс к его данным.

Класс объявляется при помощи ключевого слова `class`. Классы, которые использовались до сих пор, фактически являются очень ограниченными примерами их полной формы. Классы могут быть (и обычно являются) более сложными конструкциями. Ниже показана общая форма определения класса:

```
class classname {
    type instance-variable1;
    type instance-variable2;
    //...
    type instance-variableN;
    type methodname1(parameter-list) {
        // тело метода
    }
    type methodname2(parameter-list) {
        // тело метода
    }
    //...
    type methodnameN(parameter-list) {
        // тело метода
    }
}
```

Данные или переменные, определенные в классе, называются *переменными экземпляра* или *экземплярными переменными* (instance variables). Код содержится внутри *методов* (methods). Все вместе, методы и переменные, определенные внутри класса, называются *членами класса* (class members). В большинстве классов на переменные экземпляра действуют методы, определенные в этих классах. Таким образом, именно методы определяют, как могут использоваться данные класса.

Переменные, определенные в классе, называются *переменными экземпляра* потому, что каждый экземпляр класса (т. е. каждый объект класса) содержит свою собственную копию этих переменных. Таким образом, данные одного объекта отделены от данных другого.

Все методы имеют ту же общую форму, что метод `main()`, который мы использовали до сих пор. Однако большинство методов не будут определяться как `static` или `public`. Обратите внимание, что общая форма класса не оп-

ределяет метод `main()`. Классы Java вообще могут включать метод `main()`. Вы определяете этот метод, только если класс является стартовой точкой вашей программы. Кроме того, апплеты вообще не используют метод `main()`.

### Замечание

Программисты C++ обратят внимание, что объявление класса и реализация методов хранятся в одном месте и не определяются отдельно. Это иногда приводит к очень большим исходным (.java) файлам, т. к. любой класс должен быть полностью определен в одном исходном файле. Данное проектное свойство было встроено в Java потому, что, в конечном счете, хранение спецификации, объявления и реализации в одном месте упрощает поддержку кода.

## Простой класс

Начнем наше изучение класса с простого примера. Предположим, имеется класс с именем `Box`, который определяет три переменных экземпляра: `width`, `height` и `depth`. Пока что класс `Box` не содержит никаких методов (но вскоре они будут добавлены).

```
class Box {  
    double width;  
    double height;  
    double depth;  
}
```

Как было сказано выше, класс определяет новый тип данных. В этом случае новый тип данных называется `Box`. Вы будете использовать это имя для объявления объектов типа `Box`. Важно помнить, что объявление класса создает только шаблон, а не фактический объект. Таким образом, предшествующий код не приводит к появлению каких-либо объектов типа `Box`. Чтобы фактически создать `Box`-объект, можно воспользоваться следующим утверждением:

```
Box mybox = new Box(); // создать Box-объект с именем mybox
```

После выполнения этого утверждения переменная `mybox` станет экземпляром класса `Box`, становясь той самой "физической" реальностью. Пока не беспокойтесь относительно деталей данного утверждения.

Всякий раз, когда вы создаете экземпляра класса, образуется объект, который содержит свою собственную копию каждой экземплярной переменной, определенной в классе. Таким образом, каждый `Box`-объект будет содержать свою собственную копию переменных `width`, `height` и `depth`. Для доступа к этим переменным необходимо использовать операцию "точка" `(.)`. Она связывает имя объекта с именем переменной экземпляра. Например, чтобы назначить переменной `width` объекта `mybox` значение 100, нужно использовать следующий оператор:

```
mybox.width = 100;
```

Этот оператор просит компилятор назначать копии переменной width, которая содержится в объекте mybox, значение 100. В общем случае, чтобы обращаться как к переменным экземпляра, так и к методам объекта, следует указывать операцию "точка". Далее предлагается законченная программа, которая использует Box-класс.

```
/* Программа, которая использует Box-класс.
Назовите этот файл BoxDemo.java
*/
class Box {
    double width;
    double height;
    double depth;
}

// Этот класс объявляет объект типа Box.
class BoxDemo {
    public static void main(String args[]) {
        Box mybox = new Box();
        double vol;

        // присвоить значения экземплярным переменным объекта mybox
        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;

        // вычислить объем блока
        vol = mybox.width * mybox.height * mybox.depth;

        System.out.println("Объем равен " + vol);
    }
}
```

Вы должны назвать файл, который содержит эту программу, BoxDemo.java, потому что метод main() находится в классе с именем BoxDemo, а не в классе с именем Box. После компиляции программы вы обнаружите, что были созданы два файла с расширением .class — один для Box-класса и один для класса BoxDemo. Java-компилятор автоматически помещает каждый класс в его собственный class-файл. Нет необходимости в том, чтобы классы Box и BoxDemo находились в одном исходном файле. Можно было поместить каждый класс в свой собственный файл с именами Box.java и BoxDemo.java, соответственно.

Чтобы выполнить данную программу, нужно выполнить BoxDemo.class. Когда вы сделаете это, то увидите следующий вывод:

Объем равен 3000.0

Как указано ранее, каждый объект имеет свои собственные копии переменных экземпляра. Это означает, что, если вы имеете два Box-объекта, каждый из них обладает своей собственной копией depth, width и height. Важно понять, что изменения экземплярных переменных одного объекта не имеют никакого влияния на экземплярные переменные другого. Например, следующая программа объявляет два Box-объекта:

```
// Эта программа объявляет два Box-объекта.

class Box {
    double width;
    double height;
    double depth;
}

class BoxDemo2 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // присвоить значения экземплярным переменным объекта mybox1
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* присвоить другие значения экземплярным переменным
         * объекта mybox2 */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // вычислить объем первого блока
        vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Объем равен " + vol);

        // вычислить объем второго блока
        vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Объем равен " + vol);
    }
}
```

Вывод, выполненный этой программой:

Объем равен 3000.0

Объем равен 162.0

Можно видеть, что данные `mybox1` объекта полностью отделены от данных, содержащихся в `mybox2`.

## Объявление объектов

Как только что было объяснено, когда вы создаете класс, вы создаете новый тип данных. Этот тип можно использовать для объявления соответствующих объектов. Однако получение объектов класса это двухшаговый процесс. Во-первых, нужно объявить переменную типа "класс". Она не определяет объект. Это просто переменная, которая может ссылаться на объект. Во-вторых, вы должны получить актуальную, физическую копию объекта и назначать ее этой переменной. Вы можете сделать это с помощью операции `new`. Операция `new` распределяет динамически (т. е. во время выполнения) память для объекта и возвращает ссылку на нее. Данная ссылка является адресом ячейки памяти, выделенной объекту вышеуказанной операцией. Затем эта ссылка сохраняется в переменной. Таким образом, в Java все объекты класса должны быть распределены динамически. Рассмотрим подробности данной процедуры.

В предшествующих примерах программ для объявления объекта типа `Box` использовалась строка вида:

```
Box mybox = new Box();
```

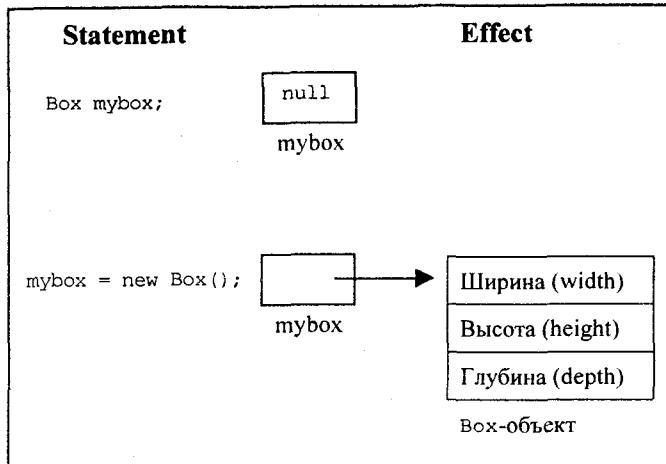
Этот оператор комбинирует два шага, как было только что описано. Чтобы продемонстрировать каждый шаг, его можно переписать так:

```
Box mybox;           // объявить ссылку на объект
mybox = new Box();   // распределить память для Box-объекта
```

Первая строка объявляет `mybox` как ссылку на объект типа `Box`. После того как эта строка выполняется, `mybox` содержит значение `null`, которое означает, что переменная еще не указывает на фактический объект. Любая попытка использовать `mybox` в этой точке приведет к ошибке во время компиляции. Следующая строка распределяет фактический объект и назначает ссылку на него переменной `mybox`. После того, как вторая строка выполняется, вы можете использовать `mybox`, как если бы это был объект `Box`. Но в действительности `mybox` просто содержит адрес ячейки памяти фактического объекта `Box`. Действие этих двух строк программы изображено на рис. 6.1.

### Замечание

Те читатели, кто знаком с C/C++ вероятно заметили, что объектные ссылки кажутся похожими на указатели. Это подозрение, по существу, верно. Объектная ссылка подобна указателю памяти. Главное различие (и ключ к безопасности Java) заключается в том, что вы не можете манипулировать ссылками как с указателями. Вы не можете, например, заставить объектную ссылку указать на произвольное место в памяти или манипулировать ею как целым числом.



**Рис. 6.1.** Объявление объекта типа Box

## Операция *new*

Как только что было указано, операция *new* динамически распределяет память для объекта. Он имеет следующую общую форму:

```
class-var = new classname();
```

Здесь *class-var* — переменная типа "класс", которая создается; *classname* — имя класса, экземпляр которого создается. За именем класса следуют круглые скобки, устанавливающие *конструктор* класса. Конструктор (*constructor*) определяет, что происходит, когда объект класса создается. Это важная часть всех классов. Конструкторы имеют много существенных атрибутов. Большинство классов явно определяют свои собственные конструкторы (внутри своих определений). Однако если явный конструктор не определен, то Java автоматически обеспечит так называемый "конструктор по умолчанию" (*default constructor*) или *умалчивающий конструктор*. Именно так обстоит дело с классом *Box*. Пока мы будем использовать конструктор по умолчанию, но скоро вы увидите, как можно определять ваши собственные конструкторы.

Сейчас вы могли бы задаться вопросом, почему не нужно использовать операцию *new* для таких типов данных, как целые числа или символы. Ответ заключается в том, что простые типы Java не реализованы как объекты. Они реализованы как "нормальные" переменные. Это сделано в интересах эффективности. Объекты имеют много свойств и атрибутов, которые требуют, чтобы Java обращалась с ними иначе, чем с простыми типами. Не применяя к простым типам тот же подход, который используется с объектами, Java может реализовывать простые типы более эффективно. Позже вы увидите

объектные версии простых типов, с которыми можно работать в тех ситуациях, когда необходимы законченные объекты этих типов.

Важно понять, что `new` распределяет память для объекта во время выполнения. Преимущество данного подхода состоит в том, что ваша программа может создавать столько объектов, сколько требуется в течение ее выполнения. Однако, возможно, что операция `new` не будет способна распределить память для объекта, потому что существует недостаток памяти, т. к. она конечна. Если это происходит, возникнет исключительная ситуация при работе программы. (Возможности обработки этих и других исключений вы изучите в главе 10.) Для примеров, предложенных в данной книге, вы не должны беспокоиться относительно нехватки памяти, но нужно обязательно помнить о такой ситуации при написании реальных программ.

Давайте еще раз рассмотрим различия между классом и объектом. Класс создает новый тип данных, который может использоваться для создания объектов. То есть класс создает логическую структуру, которая определяет отношения между его членами. Когда вы объявляете объект класса, вы создаете экземпляр (образец) этого класса. Таким образом, класс — это логическая конструкция, а объект — физическая реальность (т. е. объект занимает место в памяти). Важно постоянно помнить об этом различии.

## Назначение ссыльных переменных объекта

Во время выполнения операции назначения (присваивания) ссылочные переменные объекта (*object reference variables*) действуют иначе, чем можно было бы ожидать. Например, как вы думаете, что делает следующий фрагмент?

```
Box b1 = new Box();
Box b2 = b1;
```

Вы могли бы предположить, что переменной `b2` назначается ссылка на копию объекта, на который ссылается `b1`. То есть вы могли бы подумать, что `b1` и `b2` обращаются к отдельным и различным объектам. Однако это было бы неправильно. На самом деле после того, как этот фрагмент выполнится, обе переменных (`b1` и `b2`) будут ссылаться на один и тот же объект. Назначение `b1` переменной `b2` не распределяет никакой памяти и не копирует какую-либо часть первоначального объекта. Эта операция просто помещает в `b2` ссылку из `b1`. Таким образом, любые изменения, сделанные в объекте через `b2` затронут объект, на который ссылается `b1`, т. к. это один и тот же объект. Данная ситуация изображена на рис. 6.2.

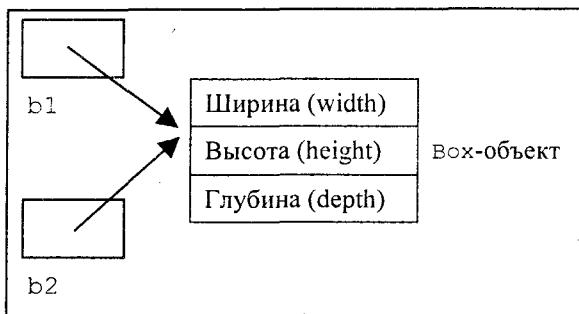
Хотя `b1` и `b2`, обе ссылаются на тот же самый объект, они не связаны каким-то другим способом. Например, следующее назначение на `b1` просто отключит `b1` от исходного объекта без воздействия на объект или переменную `b2`:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Здесь `b1` был установлен в `null`, но `b2` все еще указывает на первоначальный объект.

### Замечание

Когда вы назначаете одну ссылочную переменную объекта другой (ссылочной переменной объекта), вы не создаете копии объекта, а делаете только копию ссылки.



**Рис 6.2.** Две ссылки на один и тот же объект

## Представление методов

Как упоминалось в начале этой главы, классы обычно состоят из двух видов компонентов — переменных экземпляра и методов. Раздел методов весьма обширен, потому что Java наделяет их большой мощью и гибкостью. Немалая часть следующей главы посвящена методам. Однако, чтобы приступить к добавлению методов к классам, некоторые основные принципы нужно изучить сейчас.

Общая форма метода такова:

```
type name(parameter-list) {
    // тело метода
}
```

Здесь `type` определяет тип данных, возвращаемых методом. Это может быть любой допустимый тип, включая типы классов, которые вы создаете. Если метод не возвращает значения, его возвращаемый тип должен быть `void`. `name` определяет имя метода. Это может быть любой допустимый идентификатор, но отличающийся от тех, что уже используются другими элементами в пределах текущей области действия имен. `Parameter-list` — это последовательность пар тип-идентификатор, разделенных запятыми. Параметры —

это, по существу, переменные, которые принимают значения *аргументов*, посылаемых методу во время его вызова. Если метод не имеет параметров, то список параметров будет пуст.

Методы, у которых тип возвращаемого значения отличен от `void`, возвращают значение вызывающей подпрограмме, используя следующую форму оператора `return`:

```
return value;
```

Здесь `value` — возвращаемое значение.

Далее вы увидите, как можно создавать различные типы методов, включая те, что имеют параметры и возвращают значения.

## Добавление метода к классу *Box*

Хорошо, конечно, создавать класс, который содержит только данные, но это случается редко. Большую часть времени приходится использовать методы для доступа к экземплярным переменным, которые определены в классе. Фактически, именно методы определяют интерфейс с большинством классов. Они позволяют разработчику класса скрывать специфическое размещение внутренних структур данных за более ясными абстракциями метода. Кроме методов, которые обеспечивают доступ к данным, можно также определять методы, использующиеся внутри самого класса.

Начнем с добавления метода к *Box*-классу. Глядя на предыдущие программы, вполне может показаться, что вычисление объема блока было бы лучше выполнять в *Box*-классе, а не в классе *BoxDemo*. И, наконец, т. к. объем блока зависит от его размера, то имеет смысл поручить это вычисление *Box*-классу. Для этого нужно добавить метод в класс *Box*, как показано в следующей программе:

```
// Эта программа включает метод внутри класса Box.
class Box {
    double width;
    double height;
    double depth;

    // показать объем блока
    void volume() {
        System.out.print("Объем равен ");
        System.out.println(width * height * depth);
    }
}

class BoxDemo3 {
    public static void main(String args[]) {
```

```
Box mybox1 = new Box();
Box mybox2 = new Box();

// присвоить значения переменным экземпляра mybox1
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;

/* присвоить другие значения
переменным экземпляра mybox2 */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;

// показать объем первого блока
mybox1.volume();

// показать объем второго блока
mybox2.volume();

}

}
```

Эта программа генерирует следующий вывод, который является таким же, как в предыдущей версии.

```
Объем равен 3000.0
Объем равен 162.0
```

Рассмотрим подробнее две следующие строки программы:

```
mybox1.volume();
mybox2.volume();
```

Первая строка включает метод `volume()` объекта `mybox1`. Точнее, она обращается к методу `volume()` объекта `mybox1`, используя имя объекта, за которым следует операция "точка". Таким образом, обращение к `mybox1.volume()` отображает объем блока, определенного переменной `mybox1`, а обращение к `mybox2.volume()` отображает объем блока, определенного переменной `mybox2`.

Если вы не знакомы с концепцией вызова метода, следующее обсуждение поможет прояснить данный вопрос. Когда выполняется `mybox1.volume()`, исполняющая система Java передает управление коду, определенному внутри метода `volume()`. После того как операторы внутри `volume()` выполняются, управление возвращается в вызывающую подпрограмму, и работа продолжается со строки кода, следующей за вызовом. В самом общем смысле метод — это способ реализации подпрограмм в языке Java.

Есть кое-что очень важное, на что нужно обратить внимание внутри метода `volume()`: переменные `width`, `height` и `depth` указаны прямо, без предшествующих им имен объектов и "точечных" операций. Когда метод использует

переменную экземпляра, которая определена в его классе, он указывает ее прямо, без явной ссылки на объект и использования "точечной" операции. Понять это достаточно просто, если немного подумать. Действительно, метод всегда вызывается из некоторого объекта его класса. Раз этот вызов произошел, значит объект известен. Таким образом, внутри метода нет необходимости указывать объект второй раз. Это означает, что переменные `width`, `height` и `depth` внутри метода `volume()` неявно ссылаются на копии переменных, находящихся в объекте, который вызывает этот метод.

Итак, когда к переменной экземпляра обращается код, который не является частью класса, в котором данная переменная определена, это должно быть сделано через объект, при помощи "точечной" операции. Однако, когда к такой переменной обращается код, являющийся частью ее класса, на эту переменную можно ссылаться прямо (т. е. без квалификации ее имени имеем объекта). Этот же принцип относится и к методам.

## Возврат значений

Хотя реализация метода `volume()` перемещает вычисление объема блока внутрь `Box`-класса, которому этот метод принадлежит, это — не лучший способ вычисления. Например, что если другая часть вашей программы захотела просто узнать объем блока, но не отображать его значения? Лучший способ реализации метода `volume()` состоит в том, чтобы он вычислял объем блока и возвращал результат вызывающей программе. Следующий пример (улучшенная версия предшествующей программы) именно это и делает:

```
// Теперь volume() возвращает объем блока.

class Box {
    double width;
    double height;
    double depth;

    // вычислить и возвратить объем
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo4 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // присвоить значения переменным экземпляра mybox1
        mybox1.width = 10;
```

```
mybox1.height = 20;
mybox1.depth = 15;

/* присвоить другие значения
переменным экземпляра mybox2*/
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;

// получить объем первого блока
vol = mybox1.volume();
System.out.println("Объем равен " + vol);

// получить объем второго блока
vol = mybox2.volume();
System.out.println("Объем равен " + vol);
}

}
```

Не трудно заметить, что при вызове метода `volume()` он помещается справа от оператора назначения (присваивания). Слева находится переменная, в данном случае это переменная `vol`, которая примет значение, возвращенное методом `volume()`. Таким образом, после выполнения оператора

```
vol = mybox1.volume();
```

значение `mybox1.volume()` становится равным 3000, и это значение затем сохраняется в `vol`. Еще два важных замечания относительно возвращения значений:

- тип данных, возвращаемых методом, должен быть совместим с типом, указанным в заголовке определения метода. Например, если тип возвращаемого значения некоторого метода — `boolean`, вы не можете возвращать целое число (типа `int`);
- переменная, принимающая значение, возвращаемое методом (такая, как `vol` в нашем случае), должна также быть совместима с типом возвращаемого значения, указанным в определении метода.

Еще одно замечание: предшествующая программа может быть записана немного эффективнее, потому что нет фактически никакой потребности в переменной `vol`. Обращение к `volume()` можно выполнить прямо в утверждении с вызовом `println()`, например:

```
System.out.println("Объем равен " + mybox1.volume());
```

В этом случае при выполнении `println()` метод `mybox1.volume()` будет вызываться автоматически, и его значение будут пересыпаться к `println()`.

## Добавление метода с параметрами

Хотя некоторые методы не нуждаются в параметрах, но большинство из них параметрами все-таки пользуется. Параметры обобщают метод. Параметризованный метод может работать на множестве данных и/или использоваться в ряде похожих ситуаций. Чтобы иллюстрировать это положение, воспользуемся очень простым примером. Имеется метод, который возвращает квадрат числа 10:

```
int square()
{
    return 10 * 10;
}
```

Хотя этот метод действительно осуществляет возврат значения 10, возведенного в квадрат, его использование очень ограничено. Однако если вы измените метод так, чтобы он имел параметр, как показано ниже, тогда вы можете сделать метод `square()` более полезным.

```
int square(int i)
{
    return i * i;
}
```

Теперь `square()` будет возвращать квадрат любого значения, с которым он вызывается. То есть `square()` стал универсальным методом, который может вычислять квадрат любого целого значения, а не только 10. Например:

```
int x, y;
x = square(5);           // x равно 25
x = square(9);           // x равно 81
y = 2;
x = square(y);           // x равно 4
```

В первом обращении к `square()` параметром `i` будет передаваться значение 5. Во втором обращении `i` будет принимать значение 9. Третье обращение передает значение `y`, которое в этом фрагменте равно 2. Как показывают эти примеры, `square()` способен возвращать квадрат любых данных, которые ему пересыпают.

Важно различать два термина *параметр* и *аргумент*. *Параметр* — это переменная, определяемая методом, которая принимает значение во время вызова метода. Например, в методе `square(int i)` определен один параметр `i` типа `int`. *Аргумент* — это значение, которое передается методу, когда тот вызывается. Например, методу `square(100)` в качестве аргумента передается число 100. Внутри метода `square()` это значение принимает параметр `i`.

Вы можете использовать параметризованный метод, чтобы улучшить класс Box. В предыдущих примерах размеры каждого блока должны быть установлены отдельно при помощи последовательности следующих операторов:

```
mybox1.width = 10;  
mybox1.height = 20;  
mybox1.depth = 15;
```

Хотя этот код работает, однако возникает некоторое беспокойство. Во-первых, он кажется каким-то не изящным и склонным к ошибкам. Например, легко забыть установку измерений. Во-вторых, в хорошо разработанных Java-программах к переменным экземпляра нужно обращаться только через методы, определенные их классом. В будущем вы сможете изменять поведение метода, но вы не можете изменять поведение установленной переменной экземпляра.

Таким образом, лучший подход к установке размеров блока состоит в том, чтобы создать метод, который берет измерения блока в свои параметры и походящим образом устанавливает каждую переменную экземпляра. Эта концепция реализована следующей программой:

```
// Эта программа использует параметризованный метод.
```

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // вычислить и возвратить объем  
    double volume() {  
        return width * height * depth;  
    }  
  
    // установить размеры блока  
    void setDim(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
}  
  
class BoxDemo5 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
        double vol;  
  
        // инициализировать каждый блок  
        mybox1.setDim(10, 20, 15);  
        mybox2.setDim(3, 6, 9);  
    }  
}
```

```
// получить объем первого блока  
vol = mybox1.volume();  
System.out.println("Объем равен " + vol);  
  
// получить объем второго блока  
vol = mybox2.volume();  
System.out.println("Объем равен " + vol);  
}  
}
```

Как можно заметить, метод `setDim()` используется, чтобы установить размеры каждого блока. Например, когда

```
mybox1.setDim(10, 20, 15);
```

выполняется, 10 копируется в параметр `w`, 20 копируется в `h` и 15 копируется в `d`. Внутри метода `setDim()` значения `w`, `h` и `d` затем назначаются переменным `width`, `height` и `depth`, соответственно.

Для многих читателей — особенно тех, кто имеет некоторый опыт программирования в C/C++ — концепции, представленные в предшествующих разделах, конечно знакомы. Однако если такие понятия, как вызовы метода, аргументы и параметры являются для вас новыми, то, прежде чем двигаться дальше, желательно потратить некоторое время на эксперименты. Концепции вызова метода, параметры и возврат значений фундаментальны для Java-программирования.

## Конструкторы

При создании экземпляров весьма утомительно инициализировать все переменные в классе. Даже, когда вы добавляете функции для организации удобств, подобные `setDim()`, было бы проще и быстрее делать всю установку во время первоначального создания объекта. Поскольку требования инициализации являются достаточно общими, Java разрешает инициализацию объектов в момент их создания. Эта автоматическая инициализация выполняется с помощью конструктора.

*Конструктор* инициализирует объект после его создания. Он имеет такое же имя, как класс, в котором он постоянно находится и синтаксически подобен методу. Если конструктор определен, то он автоматически вызывается сразу же после того, как объект создается, и прежде, чем завершается выполнение операции `new`. Конструкторы выглядят немного странными, потому что не имеют ни спецификатора возвращаемого типа, ни даже спецификатора `void`. Происходит это от того, что неявным возвращаемым типом конструктора класса является тип самого класса. Работа конструктора заключается в том, чтобы инициализировать внутреннее состояние объекта.

так, что код, создающий экземпляр, будет полностью инициализирован и пригоден для немедленного использования объекта.

Можно переделать пример Box таким образом, чтобы размеры блока были автоматически инициализированы во время построения объекта. Для этого следует заменить метод `setDim()` конструктором. Начнем с определения простого конструктора, который устанавливает одинаковые значения для размеров каждого блока:

```
/* Box использует конструктор для инициализации
размеров блока.

*/
class Box {
    double width;
    double height;
    double depth;

    // Это конструктор класса Box.
    Box() {
        System.out.println("Создание Box");
        width = 10;
        height = 10;
        depth = 10;
    }

    // вычислить и возвратить объем
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo6 {
    public static void main(String args[]) {
        // объявить, разместить в памяти и инициализировать Box-объекты
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        double vol;

        // получить объем первого блока
        vol = mybox1.volume();
        System.out.println("Объем равен " + vol);

        // получить объем второго блока
        vol = mybox2.volume();
        System.out.println("Объем равен " + vol);
    }
}
```

Когда эта программа выполняется, она генерирует следующие результаты:

```
Создание Box
Создание Box
Объем равен 1000.0
Объем равен 1000.0
```

Вы видите, что `mybox1` и `mybox2` были инициализированы конструктором `Box()` во время их создания. Так как конструктор дает всем блокам одинаковые размеры,  $10 \times 10 \times 10$ , `mybox1` и `mybox2` будут иметь один и тот же объем. Предложение `println()` внутри `Box()` приводится только ради иллюстрации. Большинство функций конструктора ничего не будет отображать. Они просто инициализируют объект.

Прежде чем двигаться дальше, давайте повторно рассмотрим операцию `new`. Как вы знаете, при распределении (в памяти) объекта, нужно использовать следующую общую форму:

```
class-var = new classname();
```

Теперь вы можете понять, почему необходимы круглые скобки после имени класса. Фактически выполняется вызов конструктора класса. Таким образом, правая часть операции присваивания в строке

```
Box mybox1 = new Box();
```

есть вызов конструктора `Box()`. Если вы явно конструктор класса не определяете, то Java создает для этого класса *конструктор по умолчанию* (default constructor). Вот почему предшествующая строка программы работала в более ранних версиях `Box`, где конструктор не был определен. Умалчиваемый конструктор автоматически инициализирует все переменные экземпляра нулями. Такой конструктор часто достаточен для простых классов, но не для более сложных. Как только в классе определяется собственный конструктор, умалчиваемый больше не используется.

## Параметризованные конструкторы

Когда конструктор `Box()` в предшествующем примере инициализирует `Box`-объект, это не очень полезно, потому что все блоки имеют одинаковые размеры. Необходим способ создания `Box`-объектов разных размеров. Простое решение состоит в добавлении параметров конструктора. Как вы можете, вероятно, предположить, это сделает его более полезным. Например, следующая версия `Box` определяет параметризованный конструктор, который устанавливает размеры блока с помощью своих параметров. Обратите особое внимание на то, как создаются `Box`-объекты.

```
/* Здесь класс Box использует параметризованный конструктор
   для инициализации размеров блока.
*/
```

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // Это конструктор класса Box.  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
  
    // вычислить и возвратить объем  
    double volume() {  
        return width * height * depth;  
    }  
}  
  
class BoxDemo7 {  
    public static void main(String args[]) {  
        // объявить, распределить и инициализировать Box-объекты  
        Box mybox1 = new Box(10, 20, 15);  
        Box mybox2 = new Box(3, 6, 9);  
  
        double vol;  
  
        // получить объем первого блока  
        vol = mybox1.volume();  
        System.out.println("Объем равен " + vol);  
  
        // получить объем второго блока  
        vol = mybox2.volume();  
        System.out.println("Объем равен " + vol);  
    }  
}
```

Вывод этой программы:

Объем равен 3000.0

Объем равен 162.0

Не трудно обнаружить, что каждый объект инициализирован так, как определено в параметрах его конструктора. Например, в следующей строке

Box mybox1 = new Box(10, 20, 15);

значения 10, 20 и 15 передаются конструктору Box(), когда new создает объект. Таким образом, копии переменных width, height и depth объекта mybox1 будут содержать значения 10, 20 и 15, соответственно.

## Ключевое слово *this*

Иногда у метода возникает необходимость обращаться к объекту, который его вызвал. Для этого Java определяет ключевое слово *this*. Его можно использовать внутри любого метода, чтобы сослаться на *текущий* объект. То есть *this* — это всегда ссылка на объект, метод которого был вызван. Вы можете использовать *this* везде, где разрешается ссылка на объект текущего класса.

Чтобы лучше понять, на что ссылается *this*, рассмотрим следующую версию *Box()*:

```
// Избыточное использование this.  
Box(double w, double h, double d) {  
    this.width = w;  
    this.height = h;  
    this.depth = d;  
}
```

Эта версия *Box()* работает точно так же, как и более ранняя. Использование *this* избыточно, но совершенно корректно. Внутри *Box()* *this* будет всегда ссылаться на вызывающий объект. Хотя и избыточный в данном случае, *this* полезен в других контекстах, один из которых объясняется в следующем разделе.

## Скрытие переменной экземпляра

Как известно, в Java недопустимо объявление двух локальных переменных с одним и тем же именем внутри той же самой или включающей области действия идентификаторов. Заметим, что вы можете иметь локальные переменные, включая формальные параметры для методов, которые перекрываются с именами экземплярных переменных класса. Однако, когда локальная переменная имеет такое же имя, как переменная экземпляра, локальная переменная скрывает переменную экземпляра. Вот почему *width*, *height* и *depth* не использовались как имена параметров конструктора *Box()* внутри класса *Box*. Если бы они были использованы для именования этих параметров, то, скажем *width*, как формальный параметр, скрыл бы переменную экземпляра *width*. Хотя обычно проще указывать различные имена, существует другой способ обойти эту ситуацию. Поскольку *this* позволяет обращаться прямо к объекту, это можно применять для разрешения любых конфликтов пространства имен, которые могли бы происходить между экземплярными и локальными переменными. Ниже представлена другая версия *Box()*, которая использует *width*, *height* и *depth* для имен параметров и затем применяет *this*, чтобы получить доступ к переменным экземпляра с теми же самыми именами:

```
// Используйте этот вариант конструктора
// для разрешения конфликтов пространства имен.
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

### Предупреждение

Использование `this` в указанном контексте иногда может привести к путанице, так что некоторые программисты предпочитают не применять имена локальных переменных и формальных параметров, которые совпадают с переменными экземпляра. Конечно, другие программисты, наоборот, верят, что это хорошее соглашение — использовать одинаковые имена — для ясности, и `this` — чтобы преодолеть скрытие переменной экземпляра. Какой подход принимаете вы — это вопрос вкуса.

Хотя в только что показанных примерах `this` не имеет никакого существенного значения, в некоторых ситуациях он очень полезен.

## Сборка "мусора"

Так как объекты распределяются динамически с помощью операции `new`, можно задать вопрос, как такие объекты ликвидируются и их память освобождается для более позднего перераспределения. В некоторых языках, таких как C++, от динамически распределенных объектов нужно освобождаться вручную — при помощи оператора `delete`. Java использует другой подход: он выполняет освобождение памяти от объекта автоматически. Методика, которая реализует эту процедуру, называется сборкой "мусора". Она работает примерно так: когда никаких ссылок на объект не существует, предполагается, что этот объект больше не нужен, и память, занятая объектом, может быть освобождена. Нет никакой явной потребности уничтожать объекты как в C++. Сборка "мусора" происходит не регулярно (если вообще происходит) во время выполнения программы. Она не будет происходить просто потому, что существует один или более объектов, которые уже не используются. Кроме того, различные реализации исполняющей системы Java имеют разные подходы к сборке "мусора", но вам, по большей части, не придется думать об этом при записи своих программ.

## Метод `finalize()`

Иногда объекту нужно выполнять некоторые действия, когда он разрушается. Например, если объект содержит некоторый не-Java ресурс, такой как дескриптор файла или оконный шрифт, то нужно удостовериться, что до

разрушения объекта эти ресурсы освобождаются. Для обработки таких ситуаций Java использует механизм, называемый *завершением* (finalization). Применяя завершение, можно определять специальные действия, которые будут выполняться примерно тогда, когда объект будет использоваться сборщиком мусора.

Чтобы добавить завершение к классу, вы просто определяете метод `finalize()`. Исполняющая система Java вызывает этот метод всякий раз, когда она собирается ликвидировать объект данного класса. Внутри метода `finalize()` нужно определить те действия, которые должны быть выполнены прежде, чем объект будет разрушен. Сборщик мусора отрабатывает периодически, проверяя объекты, на которые нет больше ссылок ни из выполняющихся процессов, ни косвенных — из других действующих объектов. Непосредственно перед освобождением всех активов исполняющая система Java вызывает для объекта метод `finalize()`.

Метод `finalize()` имеет следующую общую форму:

```
protected void finalize()
{
    // код завершения
}
```

Здесь ключевое слово `protected` — спецификатор, который запрещает доступ к `finalize()` кодам, определенным вне этого класса. Объяснение этого и других спецификаторов приводится в главе 7.

Важно понять, что `finalize()` вызывается только перед самой сборкой "мусора". Он не запускается, когда объект выходит из области действия идентификаторов, например. Это означает, что вы не сможете определить, когда `finalize()` будет выполнен (и даже будет ли он выполнен вообще). Поэтому ваша программа должна обеспечить другие средства освобождения системных ресурсов, используемых объектом. Для нормальной работы программы она не должна полагаться на `finalize()`.

### Замечание

Если вы знакомы с C++, то знаете, что C++ позволяет определять деструктор для класса, который вызывается, когда объект выходит из области действия идентификаторов. Java не поддерживает этой идеи и не использует деструкторов. Метод `finalize()` только аппроксимирует функцию деструктора. Когда вы наберетесь больше опыта в Java, то увидите, что потребность в функциях деструктора минимальна из-за наличия в Java подсистемы сборки "мусора".

## Класс *Stack*

Хотя класс `Box` полезен для иллюстрации существенных элементов класса, он имеет небольшое практическое значение. Чтобы показать действитель-

ную мощь классов, данную главу закончим более сложным примером. Как вы помните из обсуждения объектно-ориентированного программирования (ООР), представленного в главе 2, одним из наиболее важных его преимуществ является инкапсуляция данных и кода, который манипулирует этими данными. Механизмом, с помощью которого достигается инкапсуляция, является класс. Создавая класс, вы организуете новый тип данных, который определяет как характер данных, так и подпрограммы, используемые для манипулирования этими данными. Непротиворечивый и управляемый интерфейс с данными класса определяют методы. Таким образом, вы можете использовать класс через его методы, не беспокоясь о деталях его реализации или о том, как данные фактически управляются внутри класса. В некотором смысле, класс подобен "машине данных". Чтобы использовать машину через ее органы управления, никаких знаний о том, что происходит внутри машины, не требуется. Фактически, поскольку подробности скрыты, ее внутренняя работа может быть изменена так, как это необходимо. Пока ваш код использует класс через его методы, внутренние подробности могут изменяться, не вызывая побочных эффектов вне класса.

Чтобы получить практическое приложение предшествующего обсуждения, давайте разработаем один из типичных примеров инкапсуляции — стек. Стек хранит данные, используя очередь типа LIFO ("Last-In, First-Out") — последним вошел, первым вышел. То есть стек подобен стопке тарелок на столе — последняя тарелка, поставленная на стопку, снимается со стопки первой. Стеки управляются через две операции, традиционно называемые *push* (поместить) и *pop* (извлечь, вытолкнуть). Чтобы поместить элемент в вершину стека, нужно использовать операцию *push*. Чтобы извлечь элемент из стека, нужно использовать операцию *pop*. Заметим, что инкапсуляция полного механизма стека — довольно простая задача.

В следующей программе класс с именем *Stack* реализует стек целых чисел:

```
// Этот класс определяет целый стек для хранения 10 значений.
class Stack {
    int stck[] = new int[10];
    int tos;

    // инициализировать вершину стека
    Stack() {
        tos = -1;
    }

    // поместить элемент в стек
    void push(int item) {
        if(tos==9)
            System.out.println("Стек заполнен.");
        else
```

```

    stck[++tos] = item;
}

// Извлечь элемент из стека
int pop() {
    if(tos < 0) {
        System.out.println("Стек пуст.");
        return 0;
    }
    else
        return stck[tos--];
}
}
}

```

Нетрудно видеть, что класс Stack определяет два элемента данных и три метода. Стек целых чисел содержится в массиве stck. Этот массив индексирован переменной tos, которая всегда содержит индекс вершины стека. Конструктор Stack() инициализирует tos значением -1, которое указывает, что стек пуст. Метод push() помещает элемент в стек. Чтобы извлечь элемент, вызовите метод pop(). Так как доступ к стеку выполняется через push() и pop(), тот факт, что стек содержится в массиве, не мешает использованию стека. Например, стек мог бы храниться в более сложной структуре данных, скажем, типа связного списка, а интерфейс, определенный методами push() и pop(), остался бы тем же самым.

Показанный ниже класс TestStack, демонстрирует работу с классом Stack. Он создает два целочисленных стека, помещает некоторые значения в каждый и затем выталкивает их.

```

class TestStack {
    public static void main(String args[]) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();

        // поместить несколько чисел в стек
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);

        // вытолкнуть эти числа из стека
        System.out.println("Стек в mystack1:");
        for(int i=0; i<10; i++)
            System.out.println(mystack1.pop());

        System.out.println("Стек в mystack2:");
        for(int i=0; i<10; i++)
            System.out.println(mystack2.pop());
    }
}

```

Эта программа генерирует следующий вывод:

Стек в mystack1:

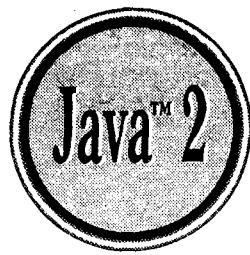
```
9  
8  
7  
6  
5  
4  
3  
2  
1  
0
```

Стек в mystack2:

```
19  
18  
17  
16  
15  
14  
13  
12  
11  
10
```

Нетрудно заметить, что содержимое каждого стека различно.

Наконец, последнее замечание относительно класса Stack: в данной реализации возможно изменение массива `stck`, который содержит стек, кодом, находящимся вне класса Stack, оставляя его открытым для неправильного использования или повреждений. В следующей главе вы увидите, как исправить эту ситуацию.



## ГЛАВА 7

# Методы и классы

Эта глава продолжает обсуждение методов и классов, начатое в предшествующей главе. В ней рассматривается несколько тем, касающихся методов, включая перегрузку, передачу параметров и рекурсию. Затем изложение возвращается к классам, обсуждая управление доступом, использование ключевого слова `static` и один из наиболее важных встроенных Java-классов `String`.

## Перегрузка методов

В языке Java в пределах одного класса можно определить два или более методов, которые совместно используют одно и то же имя, но имеют разное количество параметров. Когда это имеет место, методы называют *перегруженными*, а о процессе говорят как о *перегрузке метода*. Перегрузка методов — один из способов, с помощью которого Java реализует *полиморфизм*. Если вы никогда не пользовались языком, допускающим перегрузку методов, то концепция может сначала показаться странной. Но, как вы увидите, перегрузка метода — одна из наиболее захватывающих и полезных особенностей языка Java.

Чтобы определить при вызове, какую версию перегруженного метода в действительности вызывать, Java руководствуется типом и/или числом его параметров. Таким образом, перегруженные методы должны отличаться по типу и/или числу их параметров. Хотя такие методы могут иметь различные типы возвращаемого значения, однако одного его недостаточно, чтобы различить две версии метода. Когда Java сталкивается с вызовом перегруженного метода, он просто выполняет его (метод) версию, чьи параметры соответствуют параметрам, используемым в вызове.

Простой пример, который иллюстрирует перегруженный метод:

```
// Демонстрация перегруженного метода.
class OverloadDemo {
    void test() {
        System.out.println("Параметры отсутствуют");
    }

    // Перегруженный метод test с одним int-параметром.
    void test(int a) {
        System.out.println("a: " + a);
    }

    // Перегруженный метод test с двумя int-параметрами.
    void test(int a, int b) {
        System.out.println("a и b: " + a + " " + b);
    }

    // Перегруженный метод test с double-параметром.
    double test(double a) {
        System.out.println("Вещественное двойной точности a: " + a);
        return a*a;
    }

    class Overload {
        public static void main(String args[]) {
            OverloadDemo ob = new OverloadDemo();
            double result;

            // вызвать все версии test()
            ob.test();
            ob.test(10);
            ob.test(10, 20);
            result = ob.test(123.2);
            System.out.println("Результат ob.test(123.2): " + result);
        }
    }
}
```

Эта программа генерирует следующий вывод:

Параметры отсутствуют

a: 10

a и b: 10 20

Вещественное двойной точности a: 123.2

Результат ob.test(123.2): 15178.24

Как можно видеть, test() перегружен четыре раза. Первая версия не имеет никаких параметров, вторая имеет один параметр целого типа, третья — два

целочисленных параметра, а четвертая — один double-параметр. Тот факт, что четвертая версия test() еще и возвращает значение, не имеет никакого отношения к перегрузке, так как типы возвращаемых значений не играют никакой роли для выбора перегруженных методов.

Когда вызывается перегруженный метод, Java ищет соответствие между аргументами вызова метода и его параметрами. Однако это соответствие не всегда может быть точным. В некоторых случаях определенную роль в выборе перегруженного метода могут сыграть автоматические преобразования типов Java. Например, рассмотрим следующую программу:

```
// Автоматическое преобразование типов в применении к перегрузке.
class OverloadDemo {
    void test() {
        System.out.println("Параметры отсутствуют");
    }

    // Перегруженный test с двумя int-параметрами.
    void test(int a, int b) {
        System.out.println("a и b: " + a + " " + b);
    }

    // Перегруженный test с double-параметром и возвращаемым типом.
    void test(double a) {
        System.out.println("Внутри test(double) a: " + a);
    }
}

class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        int i = 88;

        ob.test();           // здесь будет вызван test()
        ob.test(10, 20);    // здесь будет вызван test(int, int)

        ob.test(i);         // здесь будет вызван test(double)
        ob.test(123.2);    // здесь будет вызван test(double)
    }
}
```

Эта программа генерирует следующий вывод:

Параметры отсутствуют

а и b: 10 20

Внутри test(double) a: 88

Внутри test(double) a: 123.2

Эта версия OverloadDemo не определяет `test(int)` с одним целым параметром. Поэтому, когда `test()` вызывается с целым аргументом внутри класса `Overload`, никакого согласованного метода не находится. Однако Java может автоматически преобразовывать `int` в `double`, и это преобразование можно использовать для разрешения вызова. Поэтому, после того, как `test(int)` не находится, Java расширяет `i` до `double` и затем вызывает `test(double)`. Конечно, если бы `test(int)` был определен, то он вызывался бы вместо `test(double)`. Java использует эти автоматические преобразования типов только тогда, когда никакого точного соответствия не находится.

Перегрузка методов поддерживает полиморфизм, потому что это один из способов, с помощью которых Java реализует парадигму "один интерфейс, множество методов". Чтобы понять, как это делается, приведем следующие рассуждения. На языках, которые не поддерживают перегрузку методов, каждому методу необходимо давать уникальное имя. Однако часто нужно реализовать, по существу, один и тот же метод для различных типов данных. Рассмотрим функцию абсолютного значения. На языках, которые не поддерживают перегрузку, существует обычно три или более версий этой функции, каждая со слегка отличающимся именем. Например, в C, функция `abs()` возвращает абсолютное значение целого числа, `labs()` возвращает абсолютное значение длинного целого числа, а `fabs()` — абсолютное значение числа с плавающей точкой. Так как C не поддерживает перегрузку, каждая функция должна иметь свое собственное имя, даже при том, что все три функции выполняют, по существу, одно и то же. Это делает ситуацию более сложной, чем она фактически есть на самом деле. Хотя основная концепция каждой функции одна и та же, вам все еще нужно помнить три разных имени. Подобная ситуация отсутствует в Java, потому что метод получения абсолютного значения един для всех типов данных. Действительно, библиотека стандартных классов Java включает метод абсолютного значения, с именем `abs()`. Этот метод перегружен в `Math`-классе Java, чтобы обрабатывать все числовые типы. Java определяет, какую версию `abs()` вызывать, основываясь на типе аргумента. Значение перегрузки заключается в том, что она позволяет осуществлять доступ к связанным методам при помощи общего имени. Таким образом, имя `abs` представляет *общее выполняемое действие*. Право же выбирать правильную специфическую версию для конкретного обстоятельства предоставлено компилятору. Вы же, как программист, должны только помнить общую выполняемую операцию. При использовании полиморфизма несколько имен были сокращены до одного. Хотя этот пример довольно прост, но если расширить концепцию, то можно увидеть, как перегрузка может помочь вам управлять большей сложностью.

Когда вы перегружаете метод, каждая версия этого метода может выполнять любое действие, которое вы пожелаете. Нет никакого правила, заявляющего, как перегруженные методы должны быть связаны друг с другом. Однако, со стилистической точки зрения, перегрузка метода подразумевает некоторую их взаимосвязь. Таким образом, хотя можно использовать то же самое имя,

чтобы перезагрузить несвязанные методы, но этого делать не нужно. Например, можно использовать имя `sqr`, чтобы создать методы, возвращающие *квадрат* целого числа и *квадратный корень* числа с плавающей точкой. Но эти две операции совершенно различны. Такой способ применения перегрузки метода противоречит его первоначальной цели. Практически, следует перегружать только тесно связанные операции.

## Перегрузка конструкторов

В дополнение к перегрузке обычных методов можно также перезагружать методы конструкторов. Фактически, для наиболее реальных классов, которые вы создаете, перезагруженные конструкторы будут нормой, а не исключением. Для объяснения этого утверждения вернемся к классу `Box`, разработанному в предыдущей главе. Самая последняя версия `Box` выглядит так:

```
class Box {
    double width;
    double height;
    double depth;

    // Это конструктор класса Box.
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // вычислить и возвратить объем
    double volume() {
        return width * height * depth;
    }
}
```

Здесь конструктор `Box()` имеет три параметра. Это означает, что все объявления `Box`-объектов должны переслать в конструктор `Box()` три аргумента. Например, следующий оператор недопустим:

```
Box ob = new Box();
```

Так как `Box()` требует трех параметров, то его вызов без них — это ошибка. Это поднимает несколько важных вопросов. Что, если вы просто хотели построить трехмерный блок и не заботились (или не знали), каковы были его начальные размеры? Или, что, если вы хотите инициализировать куб, определяя только одно значение, которое использовалось бы для всех трех размеров? В предложенной выше форме записи класса `Box` эти возможности вам недоступны.

К счастью, решается проблема совсем просто: перегрузите конструктор `Box` так, чтобы он обрабатывал только что описанные ситуации. Ниже показана

программа, содержащая улучшенную версию класса Box, который именно это и делает:

```
/* Класс Box: три конструктора для разных способов
   инициализации размеров блока.

*/
class Box {
    double width;
    double height;
    double depth;

    // конструктор для инициализации всех размеров
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // конструктор для инициализации без указания размеров
    Box() {
        width = -1;           // использовать -1 для указания
        height = -1;          // не инициализированного
        depth = -1;           // блока
    }

    // конструктор для создания куба
    Box(double len) {
        width = height = depth = len;
    }

    // вычислить и возвратить объем
    double volume() {
        return width * height * depth;
    }
}

class OverloadCons {
    public static void main(String args[]) {
        // создать блоки, используя различные конструкторы
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box myscube = new Box(7);

        double vol;

        // получить объем первого блока
        vol = mybox1.volume();
        System.out.println("Объем mybox1 равен " + vol);
    }
}
```

```

// получить объем второго блока
vol = mybox2.volume();
System.out.println("Объем mybox2 равен " + vol);

// получить объем куба
vol = mycube.volume();
System.out.println("Объем mycube равен " + vol);
}
}

```

Вывод, выполненный этой программой:

```

Объем mybox1 равен 3000.0
Объем mybox2 равен -1.0
Объем mycube равен 343.0

```

Как вы видите, подходящий перегруженный конструктор вызывается, основываясь на параметрах, указанных при выполнении операции new.

## Использование объектов в качестве параметров

До сих пор мы использовали в качестве параметров методов только простые типы. Однако существует практика передачи методам объектов. Например, рассмотрим следующую простую программу:

```

// Объекты можно передавать методам в качестве параметров.
class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }

    // возвратить true, если о равно вызывающему объекту
    boolean equals(Test o) {
        if(o.a == a && o.b == b) return true;
        else return false;
    }
}

class PassOb {
    public static void main(String args[]) {
        Test ob1 = new Test(100, 22);
        Test ob2 = new Test(100, 22);
        Test ob3 = new Test(-1, -1);
    }
}

```

```
System.out.println("ob1 == ob2: " + ob1.equals(ob2));  
System.out.println("ob1 == ob3: " + ob1.equals(ob3));  
}  
}
```

Эта программа генерирует следующий вывод:

```
ob1 == ob2: true  
ob1 == ob3: false
```

Не трудно видеть, что метод `equals()` внутри класса `Test` сравнивает два объекта на равенство и возвращает результат. То есть, он сравнивает вызывающий объект с тем, который передан методу. Если они содержат те же значения, то метод возвращает `true`. Иначе, он возвращает `false`. Обратите внимание, что параметр `o` в методе `equals()` указывает `Test` в качестве своего типа. Хотя `Test` — это тип класса, созданный программой, он используется точно таким же способом, как и встроенные типы Java.

Чаще всего в качестве параметров объекта используются конструкторы. Возможно, вы захотите построить новый объект так, чтобы он был первоначально таким же, как некоторый существующий. Чтобы сделать это, вы должны определить конструктор, который имеет объект своего класса в качестве параметра. Например, следующая версия `Box` позволяет одному объекту инициализировать другой:

```
// Здесь Box позволяет одному объекту инициализировать другой.  
  
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // построить клон объекта  
    Box(Box ob) {           // переслать объект конструктору  
        width = ob.width;  
        height = ob.height;  
        depth = ob.depth;  
    }  
  
    // конструктор для всех размеров блока  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
  
    // конструктор блока без размеров  
    Box() {
```

```

width = -1;           // использовать -1 для указания
height = -1;          // неинициализированного
depth = -1;           // блока
}

// конструктор для куба
Box(double len) {
    width = height = depth = len;
}

// создать и возвратить объем
double volume() {
    return width * height * depth;
}

}

class OverloadCons2 {
    public static void main(String args[]) {
        // создать блоки с использованием различных конструкторов
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);

        Box myclone = new Box(mybox1);

        double vol;

        // получить объем первого блока
        vol = mybox1.volume();
        System.out.println("Объем mybox1 равен " + vol);

        // получить объем второго блока
        vol = mybox2.volume();
        System.out.println("Объем mybox2 равен " + vol);

        // получить объем куба
        vol = mycube.volume();
        System.out.println("Объем cube равен " + vol);

        // получить объем клона
        vol = myclone.volume();
        System.out.println("Объем clone равен " + vol);
    }
}

```

Когда вы начинаете создавать собственные классы, обычно нужно организовывать много форм конструкторов, что позволит образовывать объекты удобным и эффективным способом.

## Передача аргументов

Вообще, существуют два способа, с помощью которых машинный язык может передавать аргумент подпрограмме. Первый способ — передача аргумента *по значению*. Этот метод копирует значение аргумента в формальный параметр подпрограммы. Поэтому любые изменения этого параметра подпрограммой не имеют никакого влияния на соответствующий аргумент вызова. Второй способ — передача аргумента *по ссылке*. В этом методе ссылка на параметр (а не его значение) передается параметру. Внутри подпрограммы она используется, чтобы обратиться к фактическому параметру, указанному в вызове. Это означает, что изменения параметра будут влиять на аргумент, использованный для вызова подпрограммы. Java использует оба метода, в зависимости от того, что передается во время вызова подпрограмме.

Когда методу передается простой тип, он передается *по значению*. Таким образом, то, что происходит с параметром, который принимает аргумент, никак не влияет на сам аргумент (т. е. аргумент при этом не изменяется). Например, рассмотрим следующую программу:

```
// Простые типы передаются по значению.
class Test {
    void meth(int i, int j) {
        i *= 2;
        j /= 2;
    }
}

class CallByValue {
    public static void main(String args[]) {
        Test ob = new Test();
        int a = 15, b = 20;

        System.out.println("a и b перед вызовом: " + a + " " + b);
        ob.meth(a, b);

        System.out.println("a и b после вызова: " + a + " " + b);
    }
}
```

Вывод этой программы:

а и b перед вызовом: 15 20

а и b после вызова: 15 20

Как видите, операции, которые происходят внутри `meth()`, никак не влияют на значения `a` и `b`, используемые в вызове (их значения здесь не изменились до 30 и 10, хотя метод `meth()` и выполнил соответствующие вычисления над своими параметрами).

Когда вы передаете методу *объект*, ситуация драматически изменяется, потому что объекты передаются по ссылке. Имейте в виду, что при создании переменной типа "класс", вы создаете только ссылку на объект. Таким образом, когда вы передаете эту ссылку методу, принимающий ее параметр будет ссылаться на тот же самый объект, что и аргумент. Это и означает, что объекты передаются методам по ссылке. Все изменения объекта внутри метода затрагивают объект, используемый как аргумент. Например, рассмотрим следующую программу:

```
// Объекты передаются по ссылке.
class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }

    // передать объект
    void meth(Test o) {
        o.a *= 2;
        o.b /= 2;
    }
}

class CallByRef {
    public static void main(String args[]) {
        Test ob = new Test(15, 20);

        System.out.println("ob.a и ob.b перед вызовом: " + ob.a + " " + ob.b);
        ob.meth(ob);

        System.out.println("ob.a и ob.b после вызова: " + ob.a + " " + ob.b);
    }
}
```

Эта программа генерирует следующий вывод:

```
ob.a и ob.b перед вызовом: 15 20
ob.a и ob.b после вызова: 30 10
```

Не трудно заметить, что в этом случае действия внутри `meth()` воздействовали на объект, используемый как аргумент.

Интересно отметить, что при передаче объектной ссылки методу сама ссылка передается по значению. Однако, т. к. передаваемое значение ссылается на объект, копия этого значения будет ссылаться на тот же объект, что и соответствующий аргумент.

**Замечание**

Простые типы передаются методу по значению, а объекты — по ссылке.

## Возврат объектов

Метод может возвращать любой тип данных, включая типы классов, которые вы создаете. Например, в следующей программе метод incrByTen() возвращает объект, в котором значение a на десять больше, чем в вызывающем объекте.

```
// Возврат объекта.
class Test {
    int a;

    Test(int i) {
        a = i;
    }

    Test incrByTen() {
        Test temp = new Test(a+10);
        return temp;
    }
}

class RetOb {
    public static void main(String args[]) {
        Test ob1 = new Test(2);
        Test ob2;

        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);

        ob2 = ob2.incrByTen();
        System.out.println("ob2.a после повторного увеличения: " + ob2.a);
    }
}
```

Вывод, сгенерированный этой программой:

ob1.a: 2

ob2.a: 12

ob2.a после повторного увеличения: 22

Каждый раз, когда incrByTen() вызывается, создается новый объект, и ссылка на него возвращается вызывающей подпрограмме.

Из предыдущей программы можно сделать следующий важный вывод: т. к. все объекты распределяются динамически с помощью операции new, вас не должен беспокоить выход объекта из области его видимости, потому что метод, в котором он был создан, завершается. Объект продолжает существовать, пока где-то в вашей программе присутствует ссылка на него. Когда же ссылок нет, то в следующем сеансе сборки "мусора" объект будет утилизирован.

## Рекурсия

Java поддерживает *рекурсию*. Рекурсия — это процесс определения чего-то в терминах самого себя. Что касается Java-программирования, то рекурсия — это атрибут, который позволяет методу вызывать самого себя. Такой метод называют *рекурсивным*.

Классический пример рекурсии — вычисление факториала числа. Факториал числа N есть произведение всех целых чисел между 1 и N. Например, факториал 3 равен  $1 \times 2 \times 3$  или 6. Ниже показано, как факториал может быть вычислен при помощи рекурсивного метода:

```
// Простой пример рекурсии.
class Factorial {
    // это рекурсивная функция
    int fact(int n) {
        int result;
        if(n==1) return 1;
        result = fact(n-1) * n;
        return result;
    }
}

class Recursion {
    public static void main(String args[]) {
        Factorial f = new Factorial();
        System.out.println("Факториал 3 равен " + f.fact(3));
        System.out.println("Факториал 4 равен " + f.fact(4));
        System.out.println("Факториал 5 равен " + f.fact(5));
    }
}
```

Вывод этой программы:

Факториал 3 равен 6

Факториал 4 равен 24

Факториал 5 равен 120

Если вы не знакомы с рекурсивными методами, то операция `fact()` может показаться немного запутанной. Вот как она работает. Когда метод `fact()` вызывается с параметром 1, функция возвращает 1, иначе она возвращает произведение `fact(n-1) * n`. Чтобы оценить это выражение, `fact()` вызывается с параметром `n-1`. Этот процесс повторяется до тех пор, пока `n` не станет равным 1, и вызов метода не начнет возврат.

Чтобы лучше понять, как работает метод `fact()`, прокрутим короткий пример. Когда вы вычисляете факториал 3, первый вызов `fact()` приведет ко второму вызову — с аргументом 2. Это обращение, в свою очередь, вызовет `fact()` в третий раз — с аргументом 1, а затем возвратит значение 1, которое потом умножается на 2 (значение `n` во втором обращении). Этот результат (который равен 2) затем возвращается первоначальному вызову `fact()` и умножается на 3 (первоначальное значение `n`). Вся эта процедура приводит к окончательному результату 6. Было бы интересно вставить в `fact()` утверждения с `println()`, которые покажут, на каком уровне каждый вызов находится и каковы промежуточные ответы.

Когда метод вызывает сам себя, новым локальным переменным и параметрам выделяется память в стеке, и код метода выполняется с этими новыми переменными от начала стека. Рекурсивный вызов не делает новой копии метода. Обновляются только параметры. Когда каждый рекурсивный вызов выполняет возврат, старые локальные переменные и параметры удаляются из стека, и выполнение возобновляется в точке вызова внутри метода.

Рекурсивные версии многих подпрограмм могут выполнятся немного медленнее, чем итерационный эквивалент, из-за добавления дополнительных вызовов функций. Частые рекурсивные обращения к методу могут вызывать переполнение стека. Поскольку память для параметров и локальных переменных находится в стеке, и каждый новый вызов создает новую копию этих переменных, возможно, что стек может быть исчерпан. Если это происходит, исполнительная система Java вызовет исключение.

Главное преимущество рекурсивных методов состоит в том, что их можно использовать для создания более ясных и простых версий некоторых алгоритмов (по сравнению с их итерационными "родственниками"). Например, алгоритм быстрой сортировки весьма трудно реализовать итерационным способом. Некоторые проблемы, особенно имеющие отношение к искусственному интеллекту, кажется, удобно решать рекурсивно. Наконец, некоторым людям кажется, что рекурсивное мышление легче, чем итеративное.

При записи рекурсивных методов следует где-то использовать оператор `if`, чтобы вынудить метод осуществить возврат без выполнения рекурсивного вызова. Если вы этого не сделаете, то после вызова метода, он никогда не будет возвращать управления. Это самая распространенная ошибка в работе с рекурсией. Свободно используйте утверждение `println()` во время разработки для наблюдения за происходящим и выполняйте аварийное завершение, если видите, что сделали ошибку.

Ниже приведен еще один пример рекурсии. Рекурсивный метод printArray() печатает 10 первых элементов массива values.

```
// Другой пример использования рекурсии.

class RecTest {
    int values[];

    RecTest(int i) {
        values = new int[i];
    }

    // отобразить массив рекурсивно
    void printArray(int i) {
        if(i==0) return;
        else printArray(i-1);
        System.out.println("[" + (i-1) + "] " + values[i-1]);
    }
}

class Recursion2 {
    public static void main(String args[]) {
        RecTest ob = new RecTest(10);
        int i;
        for(i=0; i<10; i++) ob.values[i] = i;
        ob.printArray(10);
    }
}
```

Эта программа генерирует следующий вывод:

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9
```

## Управление доступом

Напомним, что инкапсуляция связывает данные с кодом, который манипулирует ими. Однако инкапсуляция обеспечивает другой важный атрибут:

*управление доступом.* Через инкапсуляцию можно управлять доступом различных частей программы к членам класса и предотвращать неправильное использование таких членов. Например, разрешая доступ к данным только через хорошо определенный набор методов, есть возможность предотвращения неверного использования этих данных. Таким образом, при правильной реализации класс создает для использования "черный ящик", но внутренняя работа которого не доступна для вмешательства. Однако представленные ранее классы не полностью удовлетворяют этой цели. Например, рассмотрим класс `Stack`, показанный в конце главы 6. Хотя верно, что методы `push()` и `pop()` обеспечивают управляемый интерфейс к стеку, интерфейс этот не предписан. То есть, другая часть программы может обойти указанные методы и получить прямой доступ к стеку. Конечно, в неумелых руках это может привести к неприятностям. В данном разделе вашему вниманию будет представлен механизм, с помощью которого можно точно управлять доступом к различным членам класса.

Способ доступа к элементу определяет *спецификатор доступа*, который модифицирует его объявление, и Java поставляет их богатый набор. Некоторые аспекты управления доступом относятся главным образом к наследованию или пакетам. (*Пакет* — это, по существу, группировка классов.) Данные части механизма управления доступом Java будут обсуждены позже. Здесь начнем с рассмотрения управления доступом в применении к отдельному классу. Как только вы поймете основные принципы управления доступом, остальное будет просто.

Спецификаторы доступа Java: `public` (общий), `private` (частный) и `protected` (защищенный). Java также определяет уровень доступа, заданный по умолчанию (*default access level*). Спецификатор `protected` применяется только при использовании наследования.

Начнем с определения спецификаторов `public` и `private`. Когда элемент класса модифицирован спецификатором `public`, то к этому элементу возможен доступ из любой точки программы. Если член класса определен как `private`, к нему могут обращаться только члены этого класса. Теперь вы можете понять, почему методу `main()` всегда предшествовал спецификатор `public`. Он вызывается кодом, который находится вне программы — т. е. исполняющей системой Java. Когда никакой спецификатор доступа не используется, по умолчанию элемент класса считается `public` в пределах своего собственного пакета, но к нему нельзя обращаться извне этого пакета. (Пакеты обсуждаются в следующей главе.)

В классах, разработанных до этого момента, все их члены использовали заданный по умолчанию режим доступа, который является по существу общим (`public`). Однако это не то, что вам обычно будет нужно. Скорее всего, вы захотите ограничить доступ к компонентам данных класса, разрешая доступ только через методы. Возможно, наступит время, когда вы захотите определить методы, которые являются частными (`private`) для класса.

Спецификатор доступа предшествует остальной части описания типа элемента (члена) класса. То есть он должен начинать предложение объявления такого элемента. Например:

```
public int i;
private double j;

private int myMethod(int a, char b) { //
```

Чтобы понимать специфику общего и частного доступа, рассмотрим следующую программу:

```
/* Эта программа демонстрирует различие между
   методами доступа public и private.
*/
class Test {
    int a;                      // доступ по умолчанию (public)
    public int b;                // общий (public) доступ
    private int c;               // частный (private) доступ

    // методы для доступа к переменной с
    void setc(int i) {          // установить значение с
        c = i;
    }
    int getc() {                 // получить значение с
        return c;
    }
}

class AccessTest {
    public static void main(String args[]) {
        Test ob = new Test();

        // OK, к переменным а и б возможен прямой доступ.
        ob.a = 10;
        ob.b = 20;

        // Не OK и вызовет ошибку.
//      ob.c = 100;                  //Ошибка!

        // Нужен доступ к с через ее методы.
        ob.setc(100);               // OK

        System.out.println("a, b, и c: " + ob.a + " " + ob.b + " " + ob.getc());
    }
}
```

Внутри класса `Test` используется доступ по умолчанию, который для этого примера эквивалентен указанию `public`. Член `b` явно определен как `public`.

Для члена с задан доступ `private`. Это означает, что к нему нельзя обращаться из кода, находящегося вне его класса. Так внутри класса `AccessTest` переменная `s` не может использоваться прямо. К ней нужно обращаться через ее `public`-методы `setS()` и `getS()`. Если бы вы удалили символ комментария в начале следующей строки, то не смогли бы откомпилировать эту программу из-за нарушения правил доступа:

```
// ob.s = 100;           // Ошибка!
```

Чтобы увидеть, как управление доступом может применяться в более практическом примере, рассмотрим следующую улучшенную версию класса `Stack`, показанного в конце главы 6.

```
// Этот класс определяет целый стек, который может содержать 10 значений.
class Stack {
    /* Теперь как stck, так и tos есть private. Это значит,
       что они не могут быть случайно или намеренно
       изменены опасным для стека способом.
    */
    private int stck[] = new int[10];
    private int tos;

    // инициализировать вершину стека
    Stack() {
        tos = -1;
    }

    // поместить элемент в стек
    void push(int item) {
        if(tos==9)
            System.out.println("Стек заполнен.");
        else
            stck[++tos] = item;
    }

    // вытолкнуть элемент из стека
    int pop() {
        if(tos < 0) {
            System.out.println("Стек пуст.");
            return 0;
        }
        else
            return stck[tos--];
    }
}
```

Теперь переменная `stck`, которая содержит стек, и `tos`, которая является индексом вершины стека, определены как `private`. Это означает, что к ним

нельзя обращаться или изменять их, кроме как через методы `push()` и `pop()`. Делая `tos` частной, мы, например, предохраняем другие части программы от неосторожной установки в ней значения, которое находится вне границ массива `stck`.

Следующая программа демонстрирует улучшенный класс `Stack`. Попробуйте удалить закомментированные строки для доказательства (самим себе), что члены `stck` и `tos` действительно недоступны.

```
class TestStack {  
    public static void main(String args[]) {  
        Stack mystack1 = new Stack();  
        Stack mystack2 = new Stack();  
  
        // поместить несколько чисел в стек  
        for(int i=0; i<10; i++) mystack1.push(i);  
        for(int i=10; i<20; i++) mystack2.push(i);  
  
        // вытолкнуть эти числа из стека  
        System.out.println("Стек в mystack1:");  
        for(int i=0; i<10; i++)  
            System.out.println(mystack1.pop());  
  
        System.out.println("Стек в mystack2:");  
        for(int i=0; i<10; i++)  
            System.out.println(mystack2.pop());  
  
        // эти операторы не верны  
        // mystack1.tos = -2;  
        // mystack2.stck[3] = 100;  
    }  
}
```

Доступ к данным, определенным в классе со спецификатором `private`, выполняется только через методы этого класса. Но иногда (и по разным причинам) возникает необходимость обойти эти ограничения. Например, с помощью спецификатора общего доступа `public` можно расширить область действия переменных экземпляра за пределы своего класса и, следовательно, получить возможность работать с ними не только через их собственные методы. Значительная часть классов в этой книге ради простоты создавалась вообще без управления доступом экземплярных переменных. Однако в большинстве реальных классов нужно разрешать операции на данных только через собственные методы. Следующая глава вернется к теме управления доступом. Как вы увидите, это особенно важно, когда нужно пользоваться наследованием.

## Статические элементы

Иногда возникает необходимость определить элемент (член) класса так, чтобы появилась возможность пользоваться им независимо от какого-либо объекта этого класса. Обычно к элементам класса нужно обращаться только через объект этого класса. Однако можно создать элемент для использования без ссылки на определенный объект. Чтобы это сделать, укажите в начале его объявления ключевое слово `static`. Когда элемент объявляется как `static`, к нему можно обращаться до того, как создаются какие-либо объекты его класса, и без ссылки на какой-либо объект. Статическими можно объявлять как методы, так и переменные. Наиболее общим примером `static`-элемента является метод `main()`. Он объявляется как `static`, потому что должен вызываться прежде, чем будут созданы какие-либо объекты.

Переменные экземпляра, объявленные как `static`, это, по существу, глобальные переменные. Когда создаются объекты их класса, никакой копии `static`-переменной не делается. Вместо этого, все экземпляры класса совместно используют (разделяют) одну и ту же `static`-переменную.

Методы, объявленные как `static` имеют несколько ограничений:

- могут вызывать только другие `static`-методы;
- должны обращаться только к `static`-данным;
- никогда не могут ссылаться на `this` или `super`. (Ключевое слово `super` касается наследования и описано в следующей главе.)

Для организации вычислений с целью инициализировать статические переменные можно объявить *статический блок*, который выполняется только один раз, когда класс впервые загружается. Следующий пример демонстрирует класс, который имеет статические методы, несколько переменных и блок инициализации:

```
// Демонстрирует статические переменные, методы и блоки.  
class UseStatic {  
    static int a = 3;  
    static int b;  
  
    static void meth(int x) {  
        System.out.println("x = " + x);  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
    }  
  
    static {  
        System.out.println("Статический блок инициализирован.");  
        b = a * 4;  
    }  
}
```

```

public static void main(String args[]) {
    meth(42);
}
}

```

Все static-инструкции выполняются сразу же после загрузки класса `useStatic`. Сначала, в `a` устанавливается 3, затем выполняется static-блок (печатая сообщения), и наконец, `b` инициализируется значением `a*4` или 12. Затем вызывается метод `main()`, который обращается к `meth()`, передавая 42 параметру `x`. Три оператора `println()` обращаются к двум static-переменным (`a` и `b`) и к локальной переменной `x`.

### Замечание

Внутри статического метода недопустимо обращаться к любым экземплярным переменным.

Вывод этой программы:

Статический блок инициализирован.

`x = 42`  
`a = 3`  
`b = 12`

Вне класса, в котором они определены, статические методы и переменные могут использоваться независимо от любого объекта. В этом случае для обращения к ним используются имена их класса и точечная операция. Например, если нужно вызвать static-метод вне его класса, можно воспользоваться следующей общей формой вызова:

`classname.methodname()`

Здесь `classname` — имя класса, в котором объявлен static-метод; `methodname` — имя статического метода. Нетрудно заметить, что этот формат подобен вызову нестатических методов через переменные, ссылающиеся на объект. К переменной static можно обращаться таким же образом — при помощи точечной операции с именем класса (в качестве левого операнда). Так Java реализует управляемую версию глобальных функций и глобальных переменных.

Например:

```

class StaticDemo {
    static int a = 42;
    static int b = 99;
    static void callme() {
        System.out.println("a = " + a);
    }
}

```

```
class StaticByName {
    public static void main(String args[]) {
        StaticDemo.callme();
        System.out.println("b = " + StaticDemo.b);
    }
}
```

Здесь внутри метода `main()` к статическому методу `callme()` и статической переменной `b` обращаются вне их класса.

Вывод этой программы:

```
a = 42
b = 99
```

## Спецификатор *final*

Переменную можно объявить как `final`, предохраняя ее содержимое от изменения. Это означает, что нужно инициализировать `final`-переменную, когда она объявляется (в таком применении `final` подобен `const` в C/C++).

Например:

```
final int FILE_NEW      = 1;
final int FILE_OPEN     = 2;
final int FILE_SAVE     = 3;
final int FILE_SAVEAS   = 4;
final int FILE_QUIT     = 5;
```

После таких объявлений последующие части программы могут использовать `FILE_OPEN` и т. д., как константы, без опасения, что их значения были изменены.

Общее соглашение кодирования для `final`-переменных — выбирать все идентификаторы в верхнем регистре. Переменные, объявленные как `final`, не занимают память на "поэкземплярной" основе. Таким образом, `final`-переменная — по существу константа.

Ключевое слово `final` может также применяться и с методами, но его назначение в этом случае существенно отличается от применяемого в переменных. Это второе использование `final` демонстрируется в следующей главе, где описано наследование.

## Ревизия массивов

Массивы были представлены в этой книге раньше, чем были рассмотрены классы. Теперь, когда известны классы, для массивов можно сделать одно важное обобщение: их можно реализовать в форме объектов. В связи с этим

имеется специальный атрибут массива, преимуществом которого вы, вероятно, захотите воспользоваться. Речь идет об экземплярной переменной `length`, которая определяет размер (длину) массива, т. е. число элементов, содержащихся в нем. Все массивы-объекты имеют эту переменную, и она всегда содержит размер массива. Следующая программа демонстрирует указанное свойство:

```
// Эта программа демонстрирует элемент длины массива.
class Length {
    public static void main(String args[]) {
        int a1[] = new int[10];
        int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
        int a3[] = {4, 3, 2, 1};

        System.out.println("Размер a1 равен " + a1.length);
        System.out.println("Размер a2 равен " + a2.length);
        System.out.println("Размер a3 равен " + a3.length);
    }
}
```

Эта программа отображает (на экран) следующий вывод:

```
Размер a1 равен 10
Размер a2 равен 8
Размер a3 равен 4
```

Здесь видно, что отображается размер каждого массива. Имейте в виду, что значение `length` не имеет никакого отношения к числу элементов, которые фактически используются. Она отражает только число элементов, на которое массив рассчитан.

Вы можете использовать элемент `length` во многих ситуациях. Например, ниже показана улучшенная версия класса `Stack`. Более ранние версии этого класса всегда создавали стек с десятью элементами. Следующая версия позволяет создавать стеки любого размера. Чтобы предотвратить переполнение стека, используется значение `stckLength`.

```
// Улучшенный Stack-класс, который использует length-элемент массива.
class Stack {
    private int stck[];
    private int tos;

    // выделить память и инициализировать стек
    Stack(int size) {
        stck = new int[size];
        tos = -1;
    }
}
```

```
// поместить элемент в стек
void push(int item) {
    if(tos==stck.length-1), // использовать length-член
        System.out.println("Стек заполнен.");
    else
        stck[++tos] = item;
}

// вытолкнуть элемент из стека
int pop() {
    if(tos < 0) {
        System.out.println("Стек пуст.");
        return 0;
    }
    else
        return stck[tos--];
}

class TestStack2 {
    public static void main(String args[]) {
        Stack mystack1 = new Stack(5);
        Stack mystack2 = new Stack(8);

        // поместить несколько чисел в стек
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);

        // вытолкнуть эти элементы из стека
        System.out.println("Стек в mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());

        System.out.println("Стек в mystack2:");
        for(int i=0; i<8; i++)
            System.out.println(mystack2.pop());
    }
}
```

Обратите внимание, что программа создает два стека: один глубиной в пять элементов, а другой — в восемь элементов. Тот факт, что массивы поддерживают информацию об их собственной длине, облегчает создание стеков произвольного размера.

## Вложенные и внутренние классы

Существует возможность определения одного класса внутри другого. Такие классы известны как *вложенные (nested) классы*. Область видимости вложенных

ного класса ограничивается областью видимости включающего класса. Таким образом, если класс в определен в классе A, то в известен внутри A, но не вне его. Вложенный класс имеет доступ к членам класса (включая private-члены), в который он вложен. Однако включающий класс не имеет доступа к членам вложенного класса.

Существует два типа вложенных классов: *статические* и *нестатические*. Статический вложенный класс — это класс, который имеет модификатор static. Согласно своей характеристике он должен обращаться к членам своего включающего класса через объект. То есть он не может обратиться к членам включающего класса напрямую. Из-за этого ограничения статические вложенные классы используются редко.

Наиболее важный тип вложенного класса — внутренний (inner) класс. Внутренний класс — это нестатический вложенный класс, имеющий доступ ко всем переменным и методам своего внешнего класса и возможность обращаться к ним напрямую, таким же способом, как это делают другие нестатические члены внешнего класса. Итак, внутренний класс находится полностью в пределах видимости своего включающего класса.

Следующая программа показывает, как можно определять и использовать внутренний класс. Класс с именем Outer имеет одну переменную экземпляра с именем outer\_x, один метод экземпляра с именем test() и определяет один внутренний класс с именем Inner.

```
// Демонстрирует внутренний класс.
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // Это внутренний класс
    class Inner {
        void display() {
            System.out.println("В методе display: outer_x = " + outer_x);
        }
    }
}

class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Вывод этой программы:

В методе display: outer\_x = 100

В этой программе внутренний класс с именем Inner определен в пределах области видимости класса Outer. Поэтому любой код в классе Inner может прямо обращаться к переменной outer\_x. Внутри Inner определен экземплярный метод с именем display(). Этот метод отображает outer\_x в поток стандартного вывода. Метод main() класса InnerClassDemo создает экземпляр (объект) класса Outer и вызывает его метод test(). Тот метод создает экземпляр класса Inner и вызывает метод display().

Важно понять, что класс Inner известен только в пределах области действия (т. е. внутри) класса Outer. Java-компилятор генерирует сообщение об ошибке, если какой-то код вне класса Outer делает попытку создать экземпляр (объект) класса Inner. Итак, вложенный класс ничем не отличается от любого другого программного элемента, но он известен только в пределах включающей его области.

Внутренний класс имеет доступ ко всем членам своего включающего класса, но обратное — не верно. Члены внутреннего класса известны только в пределах внутреннего класса и не могут использоваться внешним классом. Например:

```
// Эта программа компилироваться не будет.
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // это внутренний класс
    class Inner {
        int y = 10;                      // y - локальная для Inner
        void display() {
            System.out.println("В методе display: outer_x = " + outer_x);
        }
    }

    void showy() {
        System.out.println(y);           //ошибка, y здесь неизвестна!
    }
}

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

```
    Outer outer = new Outer();
    outer.test();
}
```

Здесь у объявлена как переменная экземпляра класса `Inner`. Таким образом, она не известна вне этого класса, и не может использоваться методом `showy()`.

Хотя мы сосредоточились на вложенных классах, объявленных в области внешнего класса, существует возможность определять внутренние классы в пределах любого блока. Например, можно определить вложенный класс в блоке, установленном в методе или даже в теле цикла `for`, как показывает следующая программа.

// Определение внутреннего класса в цикле for.

```
class Outer {
```

```
int outer x = 100;
```

```
void test() {
```

```
for(int i=0; i<10; i++) {
```

```
class Inner {
```

```
void display() {
```

```
System.out.println("В методе display: outer x = " + outer.x);
```

}

1

```
    Inner inner = new Inner();
```

```
inner.display();
```

1

3

```
class InnerClassDemo {
```

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

```
Outer outer = new Outer();
```

```
outer.test();
```

1

Вывод этой версии программы:

Riassunto di valori costitutivi 106

В методе `display`: `center_x = 100`

В методе display: outer\_x = 100

В методе `display: outer x ≈ 100`

В методе display: outer x = 100

В методе `display`: `outer x = 100`

```
В методе display: outer_x = 100
В методе display: outer_x = 100
В методе display: outer_x = 100
```

В то время как вложенные классы почти не используются в повседневном программировании, они особенно полезны при обработке событий в апплетах. Мы вернемся к теме вложенных классов в главе 20. Там вы увидите, как можно использовать внутренние классы, чтобы упростить код, необходимый для обработки некоторых типов событий. Вы изучите также *анонимные внутренние классы*, являющиеся внутренними, которые не имеют имен.

И последнее замечание: в оригинальной спецификации Java 1.0 вложенные классы не допускались. Они были добавлены в Java 1.1.

## Класс *String*

Хотя класс *String* будет подробно рассмотрен в Части II этой книги, краткое его рассмотрение необходимо уже теперь, потому что в некоторых примерах программ, показанных в конце Части I, мы будем использовать строки. *String* — это, пожалуй, чаще всего используемый класс в библиотеке классов Java. Причина очевидна — строки являются очень важной частью программирования.

Для начала важно понять, что каждая строка, которую вы создаете, в действительности является объектом типа *String*. Даже строчные константы — это фактически *String*-объекты. Например, в утверждении

```
System.out.println("This is a String, too");
```

строка "This is a String, too" является *String*-константой. К счастью, Java обрабатывает *String*-константы так же, как другие машинные языки обрабатывают "нормальные" строки, так что вас это не должно беспокоить.

Кроме того, нужно понять, что объекты типа *String* неизменяемы. Если *String*-объект создан, то его содержимое не может быть изменено. Хотя данное утверждение похоже на серьезное ограничение, однако это не так по двум причинам:

- если нужно изменить строку, то всегда можно создать ее новую модифицированную копию;
- Java определяет *класс просмотра* (*peer class*) для *String*, называемый *StringBuffer*, который позволяет строкам стать изменяемыми, так что все нормальные манипуляции со строками все еще доступны в Java. (*StringBuffer* описан в Части II этой книги.)

Строки можно создать множеством способов. Самый простой — с помощью следующего утверждения:

```
String myString = "Это тест";
```

Как только вы создали String-объект, то можете использовать его везде, где допустима строка. Например, следующее утверждение выводит myString-объект на экран дисплея:

```
System.out.println(myString);
```

Java определяет одну операцию для String-объектов, которая обозначается знаком плюс (+). Она используется для сцепления (конкатенации) двух строк. Например, такой оператор:

```
String myString = "Мне" + " нравится программировать на " + "Java.;"
```

приводит к следующему содержимому объекта myString: "Мне нравится программировать на Java".

Предыдущие концепции демонстрирует следующая программа:

```
// Демонстрация строк.
class StringDemo {
    public static void main(String args[]) {
        String strOb1 = "Первая строка";
        String strOb2 = "Вторая строка";
        String strOb3 = strOb1 + " и " + strOb2;

        System.out.println(strOb1);
        System.out.println(strOb2);
        System.out.println(strOb3);
    }
}
```

Вывод, выполняемый этой программой:

Первая строка

Вторая строка

Первая строка и Вторая строка

Класс String содержит несколько полезных методов. Используя equals(), можно проверять две строки на равенство. Длину строки можно получить, вызывая метод length(). Вызывая charAt(), можно получить символ строки с указанным индексом (номером). Общие формы из этих трех методов:

```
boolean equals(String object)
int length()
char charAt(int index)
```

Ниже приведена программа, которая демонстрирует эти методы:

```
// Демонстрация некоторых методов класса String.
class StringDemo2 {
    public static void main(String args[]) {
```

```

String strOb1 = "Первая строка";
String strOb2 = "Вторая строка";
String strOb3 = strOb1;

System.out.println("Длина strOb1: " + strOb1.length());
System.out.println("Символ с индексом 3 в strOb1: " + strOb1.charAt(3));

if(strOb1.equals(strOb2))
    System.out.println("strOb1 == strOb2");
else
    System.out.println("strOb1 != strOb2");

if(strOb1.equals(strOb3))
    System.out.println("strOb1 == strOb3");
else
    System.out.println("strOb1 != strOb3");
}
}

```

Эта программа генерирует следующий вывод:

```

Длина strOb1: 12
Символ с индексом 3 в strOb1: в
strOb1 != strOb2
strOb1 == strOb3

```

Можно, конечно, создавать и *массивы строк*, аналогично тому, как создаются массивы любого другого типа объектов. Например:

```

// Демонстрирует String-массивы.
class StringDemo3 {
    public static void main(String args[]) {
        String str[] = { "один", "два", "три" };
        for(int i=0; i<str.length; i++)
            System.out.println("str[" + i + "]: " + str[i]);
    }
}

```

Вывод этой программы:

```

str[0]: один
str[1]: два
str[2]: три

```

Как вы увидите в следующем разделе, массивы строк играют важную роль во многих Java-программах.

## Использование аргументов командной строки

Иногда нужно переслать информацию в программу во время ее выполнения. Это делается пересылкой аргументов командной строки методу `main()`. *Аргументы командной строки* — это информация, которая следует непосредственно за именем программы в командной строке, используемой для запуска программы. Обращаться к аргументам командной строки внутри Java-программы весьма просто — они сохраняются как строки в `String`-массиве, передаваемом в `main()`. Например, следующая программа отображает все аргументы командной строки, с которыми она вызывается:

```
// Показать все аргументы командной строки.
class CommandLine {
    public static void main(String args[]) {
        for(int i=0; i<args.length; i++)
            System.out.println("args[" + i + "]: " + args[i]);
    }
}
```

Попытайтесь выполнить эту программу (после ее компиляции), запустив интерпретатор Java, как показано в следующей командной строке:

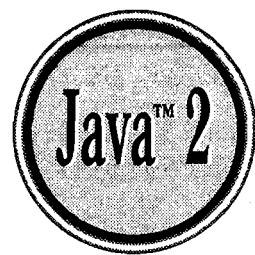
`java CommandLine это тест к программе 100 -1`

После этого должен появиться такой вывод:

```
args[0]: это
args[1]: тест
args[2]: к
args[3]: программе
args[4]: 100
args[5]: -1
```

### Замечание

Все аргументы командной строки пересылаются как строки. Вы должны преобразовать числовые значения к их внутренним формам вручную (см. гл. 14).



## ГЛАВА 8

# Наследование

Наследование — один из краеугольных камней объектно-ориентированного программирования, потому что оно позволяет создавать иерархические классификации. Используя наследование, можно создать главный класс, который определяет свойства, общие для набора связанных элементов. Затем этот класс может быть унаследован другими, более специфическими классами, каждый из которых добавляет те свойства, которые являются уникальными для него. В терминологии Java класс, который унаследован, называется *суперклассом* (superclass). Класс, который выполняет наследование, называется *подклассом* (subclass). Поэтому подкласс — это специализированная версия суперкласса. Он наследует все переменные экземпляра и методы, определенные суперклассом, и прибавляет свои собственные уникальные элементы.

## Основы наследования

Чтобы наследовать класс, нужно просто включить определение одного класса в другое, используя ключевое слово `extends`. Чтобы увидеть, как это делается, начнем с короткого примера. Следующая программа создает суперкласс с именем `A` и подкласс с именем `B`. Обратите внимание, как используется ключевое слово `extends`, чтобы создать подкласс `A`.

```
// Простой пример наследования.  
  
// Создать суперкласс.  
class A {  
    int i, j;  
    void showij() {  
        System.out.println("i и j: " + i + " " + j);  
    }  
}
```

```

// Создать подкласс расширением класса A.
class B extends A {
    int k;

    void showk() {
        System.out.println("k: " + k);
    }
    void sum() {
        System.out.println("i+j+k: " + (i+j+k));
    }
}

class SimpleInheritance {
    public static void main(String args[]) {
        A superOb = new A();
        B subOb = new B();

        // Суперкласс может быть использован сам по себе.
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Содержимое superOb: ");
        superOb.showij();
        System.out.println();

        /* Подкласс имеет доступ ко всем public-членам
         его суперкласса. */
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Содержимое of subOb: ");
        subOb.showij();
        subOb.showk();
        System.out.println();

        System.out.println("Сумма i, j и k в subOb:");
        subOb.sum();
    }
}

```

Подкласс включает все члены его суперкласса A. Вот почему объект subOb может обращаться к i и j и вызывать showij(). Поэтому же внутри sum() можно прямо ссылаться на i и j, как если бы они были частью B.

Хотя A — суперкласс для B, он тоже полностью независимый, автономный класс. Роль суперкласса для некоторого подкласса не означает, что этот суперкласс не может использоваться сам по себе. Более того, подкласс может быть суперклассом для другого подкласса.

Вывод этой программы:

Содержимое superOb:

i и j: 10 20

Содержимое subOb:

i и j: 7 8

k: 9

Сумма i, j и k в subOb:

i+j+k: 24

Ниже показана общая форма объявления класса, который наследует суперкласс:

```
class subclass-name extends superclass-name {
    // тело класса
}
```

где *subclass-name* — имя подкласса, *superclass-name* — имя суперкласса. Можно определять только один суперкласс для любого подкласса, который вы создаете. Java не поддерживает наследования множества суперклассов в одиночном подклассе. (Это отличается от C++, в котором можно наследовать множественные базовые классы.) Можно создавать иерархию наследования, в которой подкласс становится суперклассом другого подкласса. Однако никакой класс не может быть суперклассом самого себя.

## Доступ к элементам и наследование

Хотя подкласс включает все элементы (члены) своего суперкласса, он не может обращаться к тем элементам суперкласса, которые были объявлены как *private*. Например, рассмотрим следующую простую иерархию классов:

```
/* В иерархии классов private члены остаются
   private для ее классов.
```

Эта программа содержит ошибку и не будет компилироваться. \*/

// Создать суперкласс.

```
class A {
    int i;                      // public по умолчанию
    private int j;               // private для A

    void setij(int x, int y) {
        i = x;
        j = y;
    }
}
```

```
// j класса A здесь не доступна.
class B extends A {
    int total;

    void sum() {
        total = i + j; // ОШИБКА, j здесь не доступна
    }
}

class Access {
    public static void main(String args[]) {
        B subOb = new B();

        subOb.setij(10, 12);

        subOb.sum();
        System.out.println("Всего " + subOb.total);
    }
}
```

Эта программа не будет компилироваться, потому что ссылка на `j` внутри метода `sum()` класса `B` вызывает нарушение правил доступа. С тех пор как `j` объявлена с помощью `private`, она доступна только другим членам его собственного класса. Подклассы не имеют доступа к ней.

### Замечание

Член класса, который был объявлен как `private`, остается `private` для этого класса. Он не доступен любым кодам вне его класса, включая подклассы.

## Практический пример

Рассмотрим практический пример, который поможет проиллюстрировать мощь наследования. Последняя версия класса `Box`, разработанного в предшествующей главе, будет расширена, чтобы включить четвертый компонент с именем `weight`. Таким образом, новый класс будет содержать ширину, высоту, глубину и вес блока.

```
// Программа использует наследование для расширения Box.
class Box {
    double width;
    double height;
    double depth;

    // создать клон объекта
    Box(Box ob) {           // передать объект конструктору
        width = ob.width;
```

```
height = ob.height;
depth = ob.depth;
}

// конструктор, используемый, когда указаны все размеры
Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
}

// конструктор, используемый, когда размеры не указаны
Box() {
    width = -1;           // использовать -1 для указания
    height = -1;          // неинициализированного
    depth = -1;           // блока.
}

// конструктор, используемый для создания куба
Box(double len) {
    width = height = depth = len;
}

// вычислить и вернуть объем
double volume() {
    return width * height * depth;
}

// Box расширяется для включения веса.
class BoxWeight extends Box {
    double weight;           // вес блока

    // конструктор для BoxWeight
    BoxWeight(double w, double h, double d, double m) {
        width = w;
        height = h;
        depth = d;
        weight = m;
    }
}

class DemoBoxWeight {
    public static void main(String args[]) {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        double vol;
```

```

vol = mybox1.volume();
System.out.println("Объем mybox1 равен " + vol);
System.out.println("Вес mybox1 равен " + mybox1.weight);
System.out.println();

vol = mybox2.volume();
System.out.println("Объем mybox2 равен " + vol);
System.out.println("Вес mybox2 равен " + mybox2.weight);

}

}

```

Вывод этой программы:

Объем mybox1 равен 3000.0

Вес mybox1 равен 34.3

Объем mybox2 равен 24.0

Вес mybox2 равен 0.076

`BoxWeight` наследует все характеристики `Box` и прибавляет к ним компонент `weight`. Для `BoxWeight` нет необходимости заново создавать все свойства `Box`. Можно просто расширить `Box`, чтобы достичь своих собственных целей.

Главное преимущество наследования состоит в том, что, как только вы создали суперкласс, который определяет общие для набора объектов атрибуты, его можно использовать для создания любого числа более специфичных подклассов. Каждый подкласс может добавить свою собственную классификацию. Например, следующий класс наследует `Box` и прибавляет цветовой атрибут:

```

// Box расширяется для включения цвета.
class ColorBox extends Box {
    int color; // color of box

    ColorBox(double w, double h, double d, int c) {
        width = w;
        height = h;
        depth = d;
        color = c;
    }
}

```

Помните, как только вы создали суперкласс, который определяет общие аспекты объекта, этот суперкласс может наследоваться для формирования специализированных классов. Каждый подкласс просто прибавляет свои собственные, уникальные атрибуты. В этом сущность наследования.

## Переменная суперкласса может ссылаться на объект подкласса

Ссылочной переменной суперкласса может быть назначена ссылка на любой подкласс, производный от того суперкласса. Вы найдете этот аспект наследования весьма полезным в ряде ситуаций. Рассмотрим следующий пример:

```
class RefDemo {  
    public static void main(String args[]) {  
        BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);  
        Box plainbox = new Box();  
        double vol;  
  
        vol = weightbox.volume();  
        System.out.println("Объем weightbox равен " + vol);  
        System.out.println("Вес weightbox равен " + weightbox.weight);  
        System.out.println();  
  
        // назначить ссылку на BoxWeight ссылке на Box  
        plainbox = weightbox;  
  
        vol = plainbox.volume();      // OK, volume() определена в Box  
        System.out.println("Объем plainbox равен " + vol);  
  
        /* Следующее утверждение не верно, потому что plainbox  
         * не определяет член weight. */  
        // System.out.println("Вес plainbox равен " + plainbox.weight);  
    }  
}
```

Здесь `eightbox` — ссылка на объекты `BoxWeight`, а `plainbox` — ссылка на `Box`-объекты. Так как `BoxWeight` — подкласс `Box`, допустимо назначить `plainbox`-ссылку на объект `weightbox`.

Важно понять, что тип ссылочной переменной, а не тип объекта, на который она ссылается, определяет, к каким членам можно обращаться. То есть, когда ссылка на объект подкласса указывает на ссылочную переменную суперкласса, вы будете иметь доступ только к тем частям объекта, которые определены суперклассом. Вот почему `plainbox` не может ссылаться на `weight`, даже когда она ссылается на `BoxWeight`-объект. Если подумать, то данное взаимодействие оправдано, потому что суперкласс не знает, что к нему добавляет подкласс. Вот почему последняя строка кода в предшествующем фрагменте закомментирована. У ссылки нет возможности обратиться к полю `weight`, потому что в `Box` оно не определено.

Хотя предыдущие рассуждения могут показаться немного экзотическими, с ним связаны некоторые важные практические приложения (два таких приложения обсуждаются позже в этой главе).

## Использование ключевого слова *super*

В предшествующих примерах классы, производные от `Box`, не были реализованы так эффективно или устойчиво, как это могло бы быть. Например, конструктор класса `BoxWeight` явно инициализирует поля `width`, `height` и `depth` метода `Box()`. Мало того, что он дублирует код своего суперкласса, что неэффективно, но это означает, что подклассу должен быть предоставлен доступ к этим членам. Однако наступит момент, когда вы захотите создать суперкласс, который сохраняет подробности своей реализации для себя (т. е. хранит свои компоненты данных как `private`). В этом случае у подкласса не будет никакой возможности прямого доступа или инициализации этих переменных как своих собственных. Так как инкапсуляция это первичный атрибут ООР, не удивительно, что Java обеспечивает решение этой проблемы. Всякий раз, когда подкласс должен обратиться к своему непосредственному суперклассу, он может сделать это при помощи ключевого слова `super`.

Ключевое слово `super` имеет две общие формы. Первая вызывает конструктор суперкласса. Вторая используется для доступа к элементу суперкласса, который был скрыт элементом подкласса. Далее рассматриваются обе формы.

### Вызов конструктора суперкласса с помощью первой формы `super`

Подкласс может вызывать метод конструктора, определенный его суперклассом, при помощи следующей формы `super`:

```
super(parameter-list);
```

Здесь `parameter-list` — список параметров, который определяет любые параметры, необходимые конструктору в суперклассе. Похожий по форме на конструктор `super()` должен всегда быть первым оператором, выполняемым внутри конструктора подкласса.

Чтобы посмотреть, как `super()` используется, приведем следующую улучшенную версию класса `BoxWeight`:

```
// BoxWeight теперь использует super для инициализации Box-атрибутов.
class BoxWeight extends Box {
    double weight; // вес блока
    // инициализировать width, height и depth, используя super()
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // вызвать конструктор суперкласса
        weight = m;
    }
}
```

Здесь BoxWeight() вызывает super() с параметрами w, h и d. Используя эти параметры, super() вызывает конструктор Box(), который инициализирует width, height и depth. BoxWeight больше не инициализирует эти значения самостоятельно. Он нуждается в инициализации только уникальной для него переменной — weight. Box, если пожелает, может закрыть свои переменные (сделать их private).

В предыдущем примере, super() вызывался с тремя параметрами. Так как конструкторы могут быть перегружены, super() может вызывать любую их форму, определенную в суперклассе. Выполниться будет тот конструктор, который соответствует аргументам super(). Например, имеется законченная реализация BoxWeight, которая обеспечивает конструкторы для различных способов создания блока. В каждом случае вызывается super(), использующий соответствующие параметры. Обратите внимание, что width, height и depth объявлены частными (private) в классе Box.

```
// Полная реализация BoxWeight.  
class Box {  
    private double width;  
    private double height;  
    private double depth;  
  
    // построить клон объекта  
    Box(Box ob) {                                // передать объект конструктору  
        width = ob.width;  
        height = ob.height;  
        depth = ob.depth;  
    }  
  
    // конструктор, используемый для всех размеров  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
  
    // конструктор, используемый без размеров  
    Box() {  
        width = -1;                            // использовать -1 для указания  
        height = -1;                           // неинициализированного  
        depth = -1;                            // блока  
    }  
  
    // конструктор, используемый для создания куба  
    Box(double len) {  
        width = height = depth = len;  
    }  
}
```

```
// вычислить и возвратить объем
double volume() {
    return width * height * depth;
}

}

// BoxWeight теперь полностью реализует все конструкторы.
class BoxWeight extends Box {
    double weight; // вес блока

    // построить клон объекта
    BoxWeight(BoxWeight ob) { // передать объект конструктору
        super(ob);
        weight = ob.weight;
    }

    // конструктор, используемый для всех размеров
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // вызвать конструктор суперкласса
        weight = m;
    }

    // конструктор по умолчанию
    BoxWeight() {
        super();
        weight = -1;
    }

    // конструктор, используемый для создания куба
    BoxWeight(double len, double m) {
        super(len);
        weight = m;
    }
}

class DemoSuper {
    public static void main(String args[]) {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        BoxWeight mybox3 = new BoxWeight(); // по умолчанию
        BoxWeight mycube = new BoxWeight(3, 2);
        BoxWeight myclone = new BoxWeight(mybox1);
        double vol;

        vol = mybox1.volume();
        System.out.println("Объем mybox1 равен " + vol);
        System.out.println("Вес mybox1 равен " + mybox1.weight);
        System.out.println();
    }
}
```

```
vol = mybox2.volume();
System.out.println("Объем mybox2 равен " + vol);
System.out.println("Вес mybox2 равен " + mybox2.weight);
System.out.println();

vol = mybox3.volume();
System.out.println("Объем mybox3 равен " + vol);
System.out.println("Вес mybox3 равен " + mybox3.weight);
System.out.println();

vol = myclone.volume();
System.out.println("Объем myclone равен " + vol);
System.out.println("Вес myclone равен " + myclone.weight);
System.out.println();

vol = mycube.volume();
System.out.println("Объем mycube равен " + vol);
System.out.println("Вес mycube равен " + mycube.weight);
System.out.println();

}

}

Эта программа генерирует следующий вывод:
```

Объем mybox1 равен 3000.0

Вес mybox1 равен 34.3

Объем mybox2 равен 24.0

Вес mybox2 равен 0.076

Объем mybox3 равен -1.0

Вес mybox3 равен -1.0

Объем myclone равен 3000.0

Вес myclone равен 34.3

Объем mycube равен 27.0

Вес mycube равен 2.0

Обратите особое внимание на следующий конструктор в BoxWeight():

```
// построить клон объекта
BoxWeight(BoxWeight ob) {           // передать объект конструктору
    super(ob);
    weight = ob.weight;
}
```

Заметьте, `super()` вызывается с объектом типа `BoxWeight` — не типа `Box`. Но вызывается при этом все еще конструктор `Box(Box ob)`. Как было упомянуто

ранее, переменная суперкласса может использоваться для ссылки на любой производный объект этого класса. Таким образом, мы способны передать объект BoxWeight конструктору Box. Но, конечно, только Box "знает" свои собственные члены.

Сделаем обзор ключевых концепций `super()`. Когда подкласс вызывает `super()`, он вызывает конструктор своего непосредственного суперкласса. Таким образом, `super()` всегда обращается к непосредственному суперклассу вызывающего класса. Это справедливо даже в многоуровневой иерархии. Кроме того, `super()` всегда должен быть первым оператором, выполняемым внутри конструктора подкласса.

## Использование второй формы `super`

Вторая форма `super` действует в чем-то подобно ссылке `this`, за исключением того, что она всегда обращается к суперклассу подкласса, в котором используется. Общий формат такого использования `super` имеет вид:

`super.member`

где `member` может быть либо методом, либо переменной экземпляра.

Вторая форма `super` больше всего применима к ситуациям, когда имена элементов (членов) подкласса скрывают элементы с тем же именем в суперклассе. Рассмотрим следующую простую иерархию классов:

```
// Использование super для преодоления скрытия имен.
class A {
    int i;
}

// Создание подкласса B расширением класса A.
class B extends A {
    int i;                                // этот i скрывает i в A

    B(int a, int b) {
        super.i = a;                      // i из A
        i = b;                            // i из B
    }

    void show() {
        System.out.println("i из суперкласса: " + super.i);
        System.out.println("i из подкласса: " + i);
    }
}

class UseSuper {
    public static void main(String args[]) {
        B subOb = new B(1, 2);
```

```
    subObj.show();  
}  
}
```

Эта программа выполняет следующий вывод:

```
i из суперкласса: 1  
i из подкласса: 2
```

Хотя экземплярная переменная `i` класса `B` скрывает `i` класса `A`, `super` позволяет получить доступ к `i`, определенной в суперклассе. Кроме того, `super` можно также использовать для вызова методов, скрытых подклассом.

## Создание многоуровневой иерархии

До этого момента мы использовали простые иерархии классов, которые состоят только из суперкласса и подкласса. Однако можно строить иерархии, которые содержат столько уровней наследования, сколько вам нравится. Как уже говорилось, вполне приемлемо использовать подкласс в качестве суперкласса другого класса. Например, при наличии трех классов с именами `A`, `B` и `C`, класс `C` может быть подклассом `B`, который, в свою очередь, является подклассом `A`. В этом случае каждый подкласс наследует все черты всех своих суперклассов. В данном случае `C` наследует все аспекты `B` и `A`. Чтобы продемонстрировать пользу многоуровневой иерархии, рассмотрим следующую программу:

```
// Расширить BoxWeight для включения стоимости отгрузки.  
  
// Начать с класса Box.  
class Box {  
    private double width;  
    private double height;  
    private double depth;  
  
    // создать клон объекта  
    Box(Box ob) {                                // передать объект конструктору  
        width = ob.width;  
        height = ob.height;  
        depth = ob.depth;  
    }  
  
    // конструктор, использующий все размеры  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
}
```

```
// конструктор, не использующий размеров
Box() {
    width = -1;           // использовать -1 для указания
    height = -1;          // неинициализированного
    depth = -1;           // блока
}

// конструктор для создания куба
Box(double len) {
    width = height = depth = len;
}

// вычислить и возвратить объем
double volume() {
    return width * height * depth;
}

// Добавить вес.
class BoxWeight extends Box {
    double weight;         // вес блока

    // создать клон объекта
    BoxWeight(BoxWeight ob) { // передать объект конструктору
        super(ob);
        weight = ob.weight;
    }

    // конструктор, использующий все специфицированные параметры
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d);           // вызвать конструктор суперкласса
        weight = m;
    }

    // конструктор по умолчанию
    BoxWeight() {
        super();
        weight = -1;
    }

    // конструктор для создания куба
    BoxWeight(double len, double m) {
        super(len);
        weight = m;
    }

    // Добавить стоимость отгрузки
    class Shipment extends BoxWeight {
        double cost;
```

```
// построить клон объекта
Shipment(Shipment ob) {           // передать объект конструктору
    super(ob);
    cost = ob.cost;
}

// конструктор для всех специфицированных параметров
Shipment(double w, double h, double d,
          double m, double c) {
    super(w, h, d, m);           // вызвать конструктор суперкласса
    cost = c;
}

// умалчивающий конструктор
Shipment() {
    super();
    cost = -1;
}

// конструктор для создания куба
Shipment(double len, double m, double c) {
    super(len, m);
    cost = c;
}

class DemoShipment {
    public static void main(String args[]) {
        Shipment shipment1 = new Shipment(10, 20, 15, 10, 3.41);
        Shipment shipment2 = new Shipment(2, 3, 4, 0.76, 1.28);

        double vol;

        vol = shipment1.volume();
        System.out.println("Объем shipment1 равен " + vol);
        System.out.println("Вес shipment1 равен " + shipment1.weight);
        System.out.println("Стоимость отгрузки: $" + shipment1.cost);
        System.out.println();

        vol = shipment2.volume();
        System.out.println("Объем shipment2 равен " + vol);
        System.out.println("Вес shipment2 равен " + shipment2.weight);
        System.out.println("Стоимость отгрузки: $" + shipment2.cost);
    }
}
```

Здесь подкласс BoxWeight используется как суперкласс для создания подкласса с именем Shipment. Shipment наследует все черты BoxWeight и Box и

добавляет поле с именем `cost` (стоимость), которое содержит стоимость отгрузки такого пакета.

Вывод этой программы:

Объем `shipment1` равен 3000.0

Вес `shipment1` равен 10.0

Стоимость отгрузки: \$3.41

Объем `shipment2` равен 24.0

Вес `shipment2` равен 0.76

Стоимость отгрузки: \$1.28

Из-за наследования `Shipment` может использовать предварительно определенные классы `Box` и `BoxWeight`, добавляя только дополнительную информацию, которая требуется для своего собственного специфического применения. Одно из преимуществ наследования — возможность повторного использования кода.

Этот пример иллюстрирует другой важный аспект: `super()` всегда обращается к конструктору в самом близком суперклассе. `Super()` в `Shipment` вызывает конструктор класса `BoxWeight`. `Super()` в `BoxWeight` вызывает конструктор класса `Box`. В иерархии классов, если конструктор суперкласса требует наличие параметров, то все подклассы должны передать эти параметры "вверх по линии". И это не зависит от того, нуждается ли подкласс в своих собственных параметрах.

### Замечание

В предыдущей программе полная иерархия классов, включающая `Box`, `BoxWeight` и `Shipment`, показана вся в одном файле. Это только для вашего удобства. В Java все три класса были бы помещены в свои собственные файлы и откомпилированы отдельно. При создании иерархий классов использование отдельных файлов на самом деле — норма, а не исключение.

## Когда вызываются конструкторы

Когда иерархия классов создана, в каком порядке вызываются конструкторы классов, образующих иерархию? Например, если имеется подкласс в и суперкласс А, А-конструктор вызывается перед в-конструкторами, или наоборот? Ответ заключается в том, что в иерархии классов конструкторы вызываются в порядке подчиненности классов — от суперкласса к подклассу. Далее, так как `super()` должен быть первым оператором, выполняемым в конструкторе подкласса, этот порядок всегда одинаков и не зависит от того, используется ли `super()`. Если `super()` не используется, то сначала будет выполнен конструктор по умолчанию (без параметров) каждого суперкласса. Следующая программа иллюстрирует, когда выполняются конструкторы:

```
// Демонстрирует порядок вызова конструкторов.

// Создать суперкласс A.
class A {
    A() {
        System.out.println("Внутри A-конструктора.");
    }
}

// Создать подкласс B расширением A.
class B extends A {
    B() {
        System.out.println("Внутри B-конструктора.");
    }
}

// Создать другой класс (C), расширяющий B.
class C extends B {
    C() {
        System.out.println("Внутри C-конструктора.");
    }
}

class CallingCons {
    public static void main(String args[]) {
        C c = new C();
    }
}
```

Вывод этой программы:

Внутри A-конструктора.

Внутри B-конструктора.

Внутри C-конструктора.

Не трудно видеть, что конструкторы вызываются в порядке подчиненности классов.

Если подумать, то выполнение функций конструкторов в порядке подчиненности классов имеет определенный смысл. Поскольку суперкласс не имеет никакого знания о каком-либо подклассе, любая инициализация, которую ему нужно выполнить, является отдельной и, по возможности, предшествующей любой инициализации, выполняемой подклассом. Поэтому-то она и должна выполняться первой.

## Переопределение методов

В иерархии классов, если метод в подклассе имеет такое же имя и сигнатуру типов (type signature), как метод в его суперклассе, говорят, что метод в под-

классе *переопределяет* (override) метод в суперклассе. Когда переопределенный метод вызывается в подклассе, он будет всегда обращаться к версии этого метода, определенной подклассом. Версия метода, определенная суперклассом, будет скрыта. Рассмотрим следующий фрагмент:

```
// Переопределение методов.
class A {
    int i, j;

    A(int a, int b) {
        i = a;
        j = b;
    }

    // показать i и j на экране
    void show() {
        System.out.println("i и j: " + i + " " + j);
    }
}

class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    // Показать на экране k (этот show() переопределяет show() из A)
    void show() {
        System.out.println("k: " + k);
    }
}

class Override {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);

        subOb.show();           // здесь вызывается show() из B
    }
}
```

Вывод этой программы:

k: 3

Когда show() вызывается для объекта типа B, используется версия show(), определенная в классе B. То есть версия show() внутри в переопределяет (отменяет) версию, объявленную в A.

Если нужно обратиться к версии суперкласса переопределенной функции, то можно сделать это, используя super. Например, в следующей версии под-

класс в вызывается `show()` версия суперкласса A. Это позволяет отобразить все экземплярные переменные.

```
class B extends A {  
    int k;  
  
    B(int a, int b, int c) {  
        super(a, b);  
        k = c;  
    }  
  
    void show() {  
        super.show();           // вызов show() суперкласса A  
        System.out.println("k: " + k);  
    }  
}
```

Если вы замените этой версией класс в предыдущей программы, то получите следующий вывод:

```
i и j: 1 2  
k: 3
```

Здесь `super.show()` вызывает версию `show()` суперкласса.

Переопределение метода происходит только тогда, когда имена и сигнатуры типов этих двух методов идентичны. Если это не так, то оба метода просто перегружены. Например, рассмотрим следующую модифицированную версию предыдущего примера:

```
// Методы с различными сигнатурой типов перегружаются,  
// а не переопределяются.  
class A {  
    int i, j;  
  
    A(int a, int b) {  
        i = a;  
        j = b;  
    }  
  
    // Показать i и j  
    void show() {  
        System.out.println("i и j: " + i + " " + j);  
    }  
}  
  
// Создать подкласс B расширением класса A.  
class B extends A {  
    int k;
```

```

B(int a, int b, int c) {
    super(a, b);
    k = c;
}

// Переопределенный show()
void show(String msg) {
    System.out.println(msg + k);
}

}

class Override {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);

        subOb.show("Это k: ");           // вызов show() класса B
        subOb.show();                  // вызов show() класса A
    }
}

```

Вывод этой программы:

Это k: 3  
i и j: 1 2

Версия show() в классе B имеет строчный параметр. Это делает сигнатуру его типов отличной от класса A, который не имеет никаких параметров. Поэтому никакого переопределения (или скрытия имени) нет.

## Динамическая диспетчеризация методов

В то время как примеры в предшествующем разделе демонстрируют механизму переопределения метода, они не показывают всей мощи этой методики. Действительно, если бы в методике переопределения не имелось бы ничего кроме соглашения о пространстве имен, то это был бы, в лучшем случае, интересный курьез, но он не имел бы большого реального значения. Однако дело обстоит иначе. Методика переопределения формирует основу для одной из наиболее мощных концепций Java — *динамической диспетчеризации методов*. Динамическая диспетчеризация методов это механизм, с помощью которого решение на вызов переопределенной функции принимается во время выполнения, а не во время компиляции. Динамическая диспетчеризация методов важна потому, что таким способом Java реализует полиморфизм времени выполнения.

Начнем с повторения важного принципа: ссылочная переменная суперкласса может обращаться к объекту подкласса. Java использует этот факт, чтобы принимать решения о вызове переопределенных методов во время выполнения. Вот как это делается. Когда переопределенный метод вызывается через

ссылку суперкласса, Java определяет, какую версию этого метода следует выполнять, основываясь на *типе объекта*, на который указывает ссылка в момент вызова. Еще раз подчеркнем, что это определение делается *во время выполнения*. Когда ссылка указывает на различные типы объектов, будут вызываться различные версии переопределенного метода. Другими словами, именно *тип объекта, на который сделана ссылка* (а не тип ссылочной переменной) определяет, какая версия переопределенного метода будет выполнена.

Вот пример, который иллюстрирует динамическую диспетчериизацию методов:

```
// Динамическая диспетчериизация методов.
class A {
    void callme() {
        System.out.println("Внутри А метод callme");
    }
}

class B extends A {
    // переопределить callme()
    void callme() {
        System.out.println("Внутри В метод callme");
    }
}

class C extends A {
    // переопределить callme()
    void callme() {
        System.out.println("Внутри С метод callme");
    }
}

class Dispatch {
    public static void main(String args[]) {
        A a = new A();           // объект типа А
        B b = new B();           // объект типа В
        C c = new C();           // объект типа С
        A r;                    // определить ссылку типа А

        r = a;                  // r на А-объект
        r.callme();              // вызывает А-версию callme

        r = b;                  // r указывает на В-объект
        r.callme();              // вызывает В-версию callme

        r = c;                  // r указывает на С-объект
        r.callme();              // вызывает С-версию callme
    }
}
```

Вывод этой программы:

Внутри А метод callme

Внутри В метод callme

Внутри С метод callme

Эта программа создает один суперкласс с именем A и два его подкласса в и C. Подклассы B и C переопределяют callme(), объявленный в A. Внутри метода main() объявлены объекты типа A, B и C. Объявлена также ссылка типа A с именем r. Затем программа назначает ссылку r на каждый тип объекта и использует эту ссылку, чтобы вызвать соответствующий метод callme(). Как показывает вывод, версия выполняемого callme() определяется типом объекта, на который указывает ссылка во время вызова. Если бы она была определена типом ссылочной переменной r, вы увидели бы три обращения к методу callme() из класса A.

### Замечание

Читатели, знакомые с C++, обнаружат, что переопределенные методы в Java подобны виртуальным функциям в C++.

## Зачем нужны переопределенные методы?

Переопределенные методы позволяют поддерживать полиморфизм времени выполнения. Полиморфизм — весьма существенная черта объектно-ориентированного программирования по одной причине: он позволяет базовому классу определять методы, которые будут общими для всех его производных классов, и, в то же время, разрешает подклассам определять специфические реализации некоторых или всех таких методов. Переопределяемые методы — это еще один<sup>1</sup> способ реализации аспекта полиморфизма "один интерфейс, множественные методы".

Частичным залогом успешного применения полиморфизма является понимание того, что суперклассы и подклассы формируют иерархию, которая движется от меньшей к большей специализации. Суперкласс обеспечивает все элементы, которые подкласс может использовать непосредственно. Он также определяет те методы, которые производный класс должен реализовать на свой собственный манер. Это позволяет подклассу гибко определять свои методы и одновременно предписывает непротиворечивый интерфейс. Таким образом, комбинируя наследование с переопределением методов, суперкласс может определять общую форму методов, которая будет использоваться всеми его подклассами.

Динамический (т. е. реализуемый во время выполнения) полиморфизм — это один из наиболее мощных механизмов, привносимых объектно-ориен-

<sup>1</sup> Первый способ реализации этого аспекта — перегрузка (overloading) методов. — Примеч. пер.

тированным проектированием для обеспечения многократного использования кода и устойчивости к ошибкам. Способность существующих кодовых библиотек вызывать методы на экземплярах новых классов без перекомпиляции и при поддержке ясного абстрактного интерфейса — чрезвычайно мощный инструмент языка Java.

## Применение переопределения методов

Рассмотрим более практический пример, который использует переопределение метода. Следующая программа создает суперкласс с именем `Figure`, который хранит размеры различных двумерных объектов. Он также определяет метод с именем `area()`, который вычисляет площадь объекта. Программа определяет также два подкласса `Figure`. Первый — `Rectangle`, а второй — `Triangle`. Каждый из этих подклассов переопределяет `area()` так, чтобы он возвращал площадь прямоугольника и треугольника, соответственно.

```
// Использование полиморфизма времени выполнения.
class Figure {
    double dim1;
    double dim2;

    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    double area() {
        System.out.println("Площадь Figure не определена.");
        return 0;
    }
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // переопределить area для прямоугольника
    double area() {
        System.out.println("Внутри Area для Rectangle.");
        return dim1 * dim2;
    }
}

class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }
}
```

```

// переопределить area для прямоугольного треугольника
double area() {
    System.out.println("Внутри Area для Triangle.");
    return dim1 * dim2 / 2;
}
}

class FindAreas {
    public static void main(String args[]) {
        Figure f = new Figure(10, 10);
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);

        Figure figref;

        figref = r;
        System.out.println("Площадь равна " + figref.area());

        figref = t;
        System.out.println("Площадь равна " + figref.area());

        figref = f;
        System.out.println("Площадь равна " + figref.area());
    }
}
}

```

Вывод этой программы:

```

Внутри Area для Rectangle.
Площадь равна 45
Внутри Area для Triangle.
Площадь равна 40
Площадь Figure не определена.
Площадь равна 0

```

Через двойные механизмы наследования и полиморфизма времени выполнения можно определить один непротиворечивый интерфейс, который используется несколькими различными, но связанными типами объектов. В этом случае, если объект является производным от Figure-объекта, то его площадь может быть получена с помощью вызова метода area(). Интерфейс этой операции один и тот же, независимо от того, какой тип фигуры применяется.

## Использование абстрактных классов

Существуют ситуации, когда нужно определить суперкласс, который объявляет структуру некоторой абстракции без законченной реализации каждого метода. То есть, иногда необходимо создать суперкласс, определяющий

только обобщенную форму, которая будет раздельно использоваться всеми его подклассами, предостав员я каждому подклассу возможность заполнить ее своими деталями. Такой класс определяет только природу методов, конкретно реализуемых подклассами. Подобная ситуация может возникнуть, например, когда суперкласс не способен создать значимую реализацию метода. В этом случае находится класс `Figure` в предыдущем примере. Определение `area()` можно рассматривать в качестве "хранителя места". Метод не вычисляет и не отображает площадь ни для какого объекта.

При создании собственных библиотек классов вовсе не обязательно иметь сколько-нибудь значимое определение методов в контексте их суперклассов. Управлять этой ситуацией можно двумя способами. Во-первых, как показано в предыдущем примере, можно просто выдать предупреждающее сообщение. Хотя этот подход в некоторых ситуациях, таких как отладка, может быть полезен, обычно он неприемлем. Во-вторых, существует возможность определять методы суперкласса, которые должны быть переопределены подклассом для того, чтобы приобрести для указанного подкласса определенное значение. Рассмотрим класс `Triangle`. В этом случае не имеет никакого значения, определен метод `area()` или нет. Вы просто хотите иметь некоторый способ, гарантирующий, что подкласс действительно переопределяет все необходимые методы. Решением этой проблемы является *абстрактный метод*.

Можно потребовать, чтобы некоторые методы были переопределены подклассами с помощью модификатора типа `abstract`. Эти методы иногда называют методами, "отделанными на ответственность подклассу" (`subclasses responsibility`), потому что для них не определено никакой реализации в суперклассе. Таким образом, подкласс обязательно должен переопределить их — он не может просто использовать версию, определенную в суперклассе. Для объявления абстрактного метода используется следующая общая форма:

```
abstract type(parameter-list);
```

Обратите внимание, что тело метода отсутствует.

Любой класс, который содержит один и более абстрактных методов, должен также быть объявлен абстрактным. Чтобы объявить абстрактный класс, просто используйте ключевое слово `abstract` перед ключевым словом `class` в начале объявления класса. Нельзя создавать никакие объекты абстрактного класса. То есть для абстрактного класса нельзя прямо создать объект с помощью операции `new`. Такие объекты были бы бесполезны, потому что абстрактный класс определен не полностью. Вы не можете также объявлять абстрактные конструкторы или абстрактные статические методы. Любой подкласс абстрактного класса должен или реализовать все абстрактные методы суперкласса, или сам должен быть объявлен как `abstract`.

Далее показан простой пример класса с абстрактным методом, за которым следует класс, реализующий данный метод:

```
// Простая демонстрация абстракций Java.
abstract class A {
    abstract void callme();

    // в абстрактных классах допустимы обычные методы
    void callmetoo() {
        System.out.println("Это конкретный метод.");
    }
}

class B extends A {
    void callme() {
        System.out.println("B – реализация callme.");
    }
}

class AbstractDemo {
    public static void main(String args[]) {
        B b = new B();

        b.callme();
        b.callmetoo();
    }
}
```

Обратите внимание, что никакие объекты класса A не объявлены в программе. Как сказано выше, невозможно определять экземпляры (объекты) абстрактного класса. Кроме того, класс A реализует конкретный метод с именем callmetoo(). Это вполне допустимо. Абстрактные классы могут включать столько реализаций, сколько необходимо.

Хотя абстрактные классы не могут использоваться для создания объектов, их можно использовать для создания объектных ссылок, потому что подход Java к полиморфизму времени выполнения реализован с помощью ссылок суперкласса. Таким образом, возможно создавать ссылку к абстрактному классу так, чтобы она могла использоваться для указания на объект подкласса. Вы увидите использование этого свойства в следующем примере.

Применяя абстрактный класс, можно улучшать класс Figure, показанный ранее. Так как нет никакой значимой концепции (правила) для вычисления площади произвольной двумерной фигуры, следующая версия программы объявляет метод area() как абстрактный внутри Figure. Это, конечно, означает, что все классы, производные от Figure, должны переопределить area().

```
// Использование абстрактных методов и классов.
abstract class Figure {
    double dim1;
    double dim2;
```

```
Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
}

// area теперь не абстрактный метод
abstract double area();
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // переопределить area для прямоугольника
    double area() {
        System.out.println("Внутри Area для Rectangle.");
        return dim1 * dim2;
    }
}

class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // переопределить area для прямоугольного треугольника
    double area() {
        System.out.println("Внутри Area для Triangle.");
        return dim1 * dim2 / 2;
    }
}

class AbstractAreas {
    public static void main(String args[]) {
        // Figure f = new Figure(10, 10);           // теперь незаконно
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);

        Figure figref;                           // OK, объект не создается

        figref = r;
        System.out.println("Площадь равна " + figref.area());

        figref = t;
        System.out.println("Площадь равна " + figref.area());
    }
}
```

Как указывает комментарий внутри `main()`, больше невозможно объявлять объекты типа `Figure`, так как класс теперь абстрактный. Кроме того, все подклассы `Figure` должны переопределять `area()`. Чтобы доказать это, пробуйте создать подкласс, который не переопределяет `area()`. Вы получите ошибку времени компиляции.

Хотя невозможно создать объект типа `Figure`, можно создать ссылочную переменную типа `Figure`. Переменная `figref` объявлена как ссылка на `Figure`, что означает, что она может использоваться для ссылки на объект любого класса, производного от `Figure`. Напомним, что именно через ссылочные переменные суперкласса выбираются переопределенные методы (во время выполнения).

## Использование ключевого слова `final` с наследованием

Ключевое слово `final` имеет три применения. Первое — его можно использовать для создания эквивалента именованной константы. Такое использование было описано в предыдущей главе. Два других применения `final` связаны с наследованием. Оба рассмотрены ниже.

### Использование `final` для отказа от переопределения

Хотя переопределение методов — одно из наиболее мощных свойств Java, может появиться потребность отказаться от него. Чтобы отменить переопределение метода, укажите модификатор `final` в начале его объявления. Методы, объявленные как `final`, не могут переопределяться. Следующий фрагмент иллюстрирует `final` в таком применении:

```
class A {
    final void meth() {
        System.out.println("Это метод final.");
    }
}

class B extends A {
    void meth() { // ОШИБКА! Нельзя переопределять.
        System.out.println("Ошибка!");
    }
}
```

Поскольку `meth()` объявлен как `final`, он не может быть переопределен в классе `B`. Если вы попытаетесь сделать это, то получите ошибку во время компиляции.

Методы, объявленные как `final`, могут иногда улучшать эффективность. Компилятору предоставлено право выполнять *встроенные* (*inline*) вызовы таких методов, потому что он "знает", что те не будут переопределяться подклассом. Когда речь идет о небольшой, например, `final`-функции, то компилятор Java часто может копировать (встраивать) байт-код подпрограммы прямо в точку вызова откомпилированного кода вызывающего метода, устраняя таким образом дополнительные затраты времени, связанные с обычным вызовом (невстроенного метода). Встраивание допустимо только для `final`-методов. Обычно Java организует вызовы методов динамически, во время выполнения. Это называется *поздним связыванием* (вызыва с вызываемой функцией). Однако, т. к. `final`-методы не являются переопределяемыми, их вызов может быть организован во время компиляции. Это называется *ранним связыванием*.

## Использование *final* для отмены наследования

Иногда нужно разорвать наследственную связь классов (отменить наследование одного класса другим). Чтобы сделать это, предварите объявление класса ключевым словом `final`, что позволит неявно объявить и все его методы. Заметим, что недопустимо объявлять класс одновременно как `abstract` и `final`, т. к. абстрактный класс неполон сам по себе и полагается на свои подклассы, чтобы обеспечить полную реализацию. Пример `final`-класса:

```
final class A {  
    // ...  
}  
  
// Следующий класс незаконный.  
class B extends A {           // ОШИБКА! B не может быть подклассом A  
    // ...  
}
```

Комментарий здесь означает, что `B` не может наследовать `A`, т. к. `A` объявлен как `final`.

## Класс *Object*

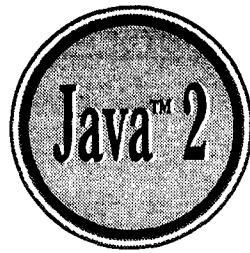
В Java определен один специальный класс — `Object`. Все другие классы являются его подклассами. `Object` — это суперкласс всех других классов. Это означает, что ссылочная переменная типа `Object` может обращаться к объекту любого другого класса. Кроме того, т. к. массивы реализуются как классы, переменная типа `Object` может также обращаться к любому массиву.

`Object` определяет методы (табл. 8.1), что означает, что они доступны в каждом объекте.

Таблица 8.1. Методы `Object`

Метод	Цель
<code>Object clone()</code>	Создает новый объект, который является таким же, как имитируемый объект
<code>boolean equals(Object object)</code>	Определяет, является ли один объект равным другому
<code>void finalize()</code>	Вызывается прежде, чем неиспользованный объект будет переработан (сборщиком мусора)
<code>Class getClass()</code>	Получает класс объекта во время выполнения
<code>int hashCode()</code>	Возвращает хэш-код, связанный с вызовом объекта
<code>void notify()</code>	Возобновляет выполнение потока, ожидающего на объекте вызова
<code>void notifyAll()</code>	Возобновляет выполнение всех потоков, ожидающих на объекте вызова
<code>String toString()</code>	Возвращает строку, которая описывает объект
<code>void wait()</code>	Ждет выполнения на другом потоке
<code>void wait(long millisconds)</code>	
<code>void wait(long millisconds, int nanoseconds)</code>	

Методы `getClass()`, `notify()`, `notifyAll()` и `wait()` объявлены как `final`. Другие можно переопределять. Указанные методы описаны в других местах этой книги. Здесь отметим два метода: `equals()` и `toString()`. Метод `equals()` сравнивает содержимое двух объектов. Он возвращает `true`, если объекты эквивалентны, и `false` — в противном случае. Метод `toString()` возвращает строку, содержащую описание объекта, на котором он вызывается. Кроме того, этот метод вызывается автоматически, когда объект выводится методом `println()`. Многие классов переопределяют данный метод, что позволяет им приспособливать описание специально для типов объектов, которые они создают. Дополнительную информацию о методе `toString()` можно получить в главе 13.



## ГЛАВА 9

# Пакеты и интерфейсы

Эта глава рассматривает два из наиболее инновационных свойств Java — пакеты и интерфейсы. *Пакеты* — это контейнеры для классов, которые используются для сохранения пространства имен классов разделенным на именованные области. Например, вы имеете возможность создать класс с именем `List`, который можно хранить в вашем собственном пакете, не опасаясь, что он столкнется с некоторым другим классом с именем `List`, хранящимся где-то в другом месте. Пакеты сохраняются иерархическим способом и явно импортируются в определения новых классов.

В предыдущих главах вы видели, как методы определяют интерфейс к данным в классе. С помощью ключевого слова `interface` Java позволяет полностью отделить интерфейс от его реализации. Используя интерфейс, можно определить набор методов, которые могут быть реализованы одним или несколькими классами. Сам интерфейс в действительности не определяет никакой реализации. Хотя интерфейсы подобны абстрактным классам, они имеют дополнительную возможность: класс может реализовывать больше одного интерфейса. В противоположность этому класс может наследовать только один суперкласс (абстрактный или другой).

Пакеты и интерфейсы — это два основных компонента Java-программы. В общем случае исходный файл Java может содержать любую (или все) из следующих четырех внутренних частей:

- одиночный `package`-оператор (не обязательно);
- любое число `import`-операторов (не обязательно);
- одиночное объявление общего класса (требуется);
- любое число частных классов пакета (не обязательно).

До сих пор в примерах использовалась только одна из этих частей — одиночное объявление класса `public`. В этой главе рассмотрим остальные части.

## Пакеты

В примерах предшествующих глав имя каждого класса выбиралось из одного и того же пространства имен. Это означает, что во избежании коллизии имен для каждого класса нужно использовать уникальное имя. Кроме того, без некоторого способа управления пространством имен, вы можете быстро исчерпать удобные, дескриптивные имена для индивидуальных классов. Вы также должны быть уверены, что выбранное для класса имя будет разумно уникально и не вступит в противоречие с именами классов, выбранными другими программистами. (Вообразите небольшую группу программистов, борющихся за право использования имени "Foobar" в качестве имени класса. Или вообразите все Internet-сообщество, спорящее, кто первым назвал класс "Espresso".) К счастью, Java обеспечивает специальный механизм для разделения пространства имен классов на управляемые части. Этот механизм называется "пакеты" (packages). Пакет является механизмом как именования, так и управления видимостью. Вы можете определять внутри пакета классы, которые не доступны кодам вне этого пакета. Вы можете также определять члены класса, доступные только другим членам того же самого пакета. Это позволяет вашим классам близко знать друг друга, но не предъявлять это знание остальной части мира.

## Определение пакета

Создать пакет очень легко: просто включите оператор `package` в начало исходного файла Java. Любые классы, объявленные в пределах того файла, будут принадлежать указанному пакету. Оператор `package` определяет пространство имен, в котором сохраняются классы. Если вы опускаете инструкцию `package`, имена класса помещаются в пакет по умолчанию (`default package`), который не имеет никакого имени. (Поэтому-то вы и не должны были волноваться относительно пакетов до настоящего времени.) В то время как пакет по умолчанию хорош для коротких примеров программ, он неадекватен для реальных приложений. В большинстве случаев вы сами будете определять пакет для своего кода.

Общая форма инструкции `package`:

```
package pkg;
```

Здесь `pkg` — имя пакета. Например, следующая инструкция создает пакет с именем `MyPackage`.

```
package MyPackage;
```

Чтобы хранить пакеты, Java использует каталоги файловой системы. Например, `class`-файлы для любых классов, которые вы объявляете как часть пакета `MyPackage`, должны быть сохранены в каталоге с именем `MyPackage`.

Помните, что регистр существенен, и имя каталога должно точно соответствовать имени пакета.

Одну и ту же package-инструкцию могут включать несколько файлов. Она просто указывает, какому пакету принадлежат классы, определенные в файле. Это не исключает принадлежности других классов в других файлах к тому же самому пакету. Большинство реальных пакетов содержат много файлов.

Можно создавать *иерархию пакетов*. Для этого необходимо просто отделить каждое имя пакета от стоящего выше при помощи операции "точка". Общая форма инструкции многоуровневого пакета:

```
package pkg1[.pkg2[.pkg3]];
```

Иерархия пакетов должна быть отражена в файловой системе вашей системы разработки Java-программ. Например, пакет, объявленный как

```
package java.awt.image;
```

должен быть сохранен в каталоге `java/awt/image`, `java\awt\image` или `java:awt:image` файловой системы UNIX, Windows или Macintosh, соответственно. Страйтесь тщательно выбирать имена пакетов. Нельзя переименовывать пакет без переименования каталога, в котором хранятся классы.

## Использование CLASSPATH

Прежде чем будет представлен пример с использованием пакета, необходимо кратко обсудить переменную окружения (environmental variable) `CLASSPATH`. В то время как пакеты решают много серьезных проблем в управлении доступом и разрешении коллизий в пространстве имен, они вызывают некоторые любопытные трудности во время компиляции и выполнения программы. Происходит это потому, что размещением корня<sup>1</sup> любой иерархии пакетов в файловой системе компьютера управляет специальная переменная окружения `CLASSPATH`. До сих пор вы сохраняли все классы в одном и том же неименованном пакете (который используется по умолчанию). Это позволяет просто компилировать исходный код и запускать интерпретатор Java, указывая (в качестве его параметра) имя класса на командной строке. Данный механизм работал, потому что заданный по умолчанию текущий рабочий каталог (`.`) обычно указывается в переменной окружения `CLASSPATH`, определяемой для исполнительной (run-time) системы Java по умолчанию. Однако все становится не так просто, когда включаются пакеты. И вот почему.

Предположим, что вы создаете класс с именем `PackTest` в пакете с именем `test`. Так как ваша структура каталогов должна соответствовать вашим пакетам, вы создаете каталог с именем `test` и размещаете исходный файл

<sup>1</sup> В той форме, в которой его представляет Java-компилятор. — Примеч. пер.

PackTest.java внутри этого каталога. Затем вы назначаете test текущим каталогом и компилируете PackTest.java. Это приводит к сохранению результата компиляции (файла PackTest.class) в каталоге test, как это и должно быть. Когда вы попробуете выполнить этот файл с помощью интерпретатора Java, то он выведет сообщение об ошибке "can't find class PackTest" (не возможно найти класс PackTest). Это происходит потому, что класс теперь сохранен в пакете с именем test. Вы больше не можете обратиться к нему просто как к PackTest. Вы должны обратиться к классу, перечисляя иерархию его пакетов и разделяя пакеты точками. Этот класс должен теперь называться test.PackTest. Однако если вы попробуете использовать test.PackTest, то будете все еще получать сообщение об ошибке "can't find class test/PackTest" (не возможно найти класс test/PackTest).

Причина того, что вы все еще принимаете сообщение об ошибках, скрыта в вашей переменной CLASSPATH. Помните, вершину иерархии классов устанавливает CLASSPATH. Проблема в том, что, хотя вы сами находитесь непосредственно в каталоге test, в CLASSPATH ни он, ни, возможно, вершина иерархии классов, не установлены.

В этот момент у вас имеется две возможности: перейти по каталогам вверх на один уровень и испытать команду java test.PackTest или добавить вершину иерархии классов в переменную окружения CLASSPATH. Тогда вы будете способны использовать команду java test.PackTest из любого каталога, и Java найдет правильный class-файл. Например, если вы работаете над вашим исходным кодом в каталоге C:\myjava, то установите в CLASSPATH следующие пути:

```
.;C:\myjava;C:\java\classes
```

## Короткий пример пакета

Помня о предшествующем обсуждении, протестируйте следующий простой пакет:

```
// Простой пакет
package MyPack;

class Balance {
    String name;
    double bal;

    Balance(String n, double b) {
        name = n;
        bal = b;
    }

    void show() {
        if(bal<0)
```

```
System.out.print("-->> ");
System.out.println(name + ": $" + bal);
}
}

class AccountBalance {
    public static void main(String args[]) {
        Balance current[] = new Balance[3];

        current[0] = new Balance("K. J. Fielding", 123.23);
        current[1] = new Balance("Will Tell", 157.02);
        current[2] = new Balance("Tom Jackson", -12.33);

        for(int i=0; i<3; i++) current[i].show();
    }
}
```

Назовите этот файл AccountBalance.java и поместите его в каталог с именем MyPack.

Далее, откомпилируйте файл и удостоверьтесь, что результирующий .class-файл также находится в каталоге MyPack. Затем пробуйте выполнить класс AccountBalance, используя следующую командную строку:

```
java MyPack.AccountBalance
```

Помните, когда вы выполняете эту команду, то должны или быть на уровень выше каталога MyPack, или иметь подходящую установку переменной CLASSPATH.

Класс AccountBalance теперь является частью пакета MyPack. Это означает, что он не может быть выполнен отдельно. То есть вы не можете использовать следующую командную строку:

```
java AccountBalance
```

AccountBalance должен быть квалифицирован именем своего пакета.

## Защита доступа

В предшествующих главах вы познакомились с различными аспектами механизма управления доступом Java и его спецификаторами доступа. Например, вы уже знаете, что доступ к private-члену класса предоставляется только другим членам того же класса. Пакеты добавляют еще одно измерение к управлению доступом. Java обеспечивает достаточно уровней защиты, чтобы допустить хорошо разветвленный контроль над видимостью переменных и методов в пределах классов, подклассов и пакетов.

Классы и пакеты, с одной стороны, обеспечивают инкапсуляцию, а с другой — поддерживают пространство имен и области видимости переменных и методов. Пакеты действуют как контейнеры для классов и других зависи-

мых пакетов. Классы действуют как контейнеры для данных и кода. Класс — самый мелкий модуль абстракции языка Java. Из-за взаимодействия между классами и пакетами, Java адресует четыре категории видимости для элементов класса:

- подклассы в том же пакете;
- неподклассы в том же пакете;
- подклассы в различных пакетах;
- классы, которые не находятся в том же пакете и не являются подклассами.

Большое разнообразие уровней защиты доступа для этих категорий обеспечивают три спецификатора доступа: `private`, `public` и `protected`. Табл. 9.1 подводит итог соответствующим взаимодействиям.

**Таблица 9.1. Доступ к членам классов**

	Private	Без модификатора	Protected	Public
Тот же класс	Yes	Yes	Yes	Yes
Подкласс того же пакета	No	Yes	Yes	Yes
Неподкласс того же пакета	No	Yes	Yes	Yes
Другой подкласс пакета	No	No	Yes	Yes
Другой неподкласс пакета	No	No	No	Yes

Хотя механизм управления доступом Java может показаться сложным, мы можем упростить его следующим образом. Все, объявленное как `public`, может быть доступно отовсюду. Все, объявленное как `private`, не может быть видимо извне своего класса. Когда элемент не имеет явных спецификаций доступа, он видим в подклассах, также как в других классах в том же самом пакете. Это — доступ, заданный по умолчанию. Если вы хотите позволить элементу быть видимым извне вашего текущего пакета, но только в классах, являющихся прямыми подклассами вашего класса, то объягите этот элемент `protected`.

Табл. 9.1 применяется только к членам классов. Сам класс имеет лишь два возможных уровня доступа — по умолчанию и общий (`public`). Когда класс объявлен как `public`, он доступен из любых других кодов. Если класс имеет доступ по умолчанию, то к нему возможен доступ только из кодов того же пакета.

## Пример управления доступом

Следующий пример показывает все комбинации модификаторов управления доступом, используя два пакета и пять классов. Помните, что классы для

двух различных пакетов нужно сохранять в каталогах, названных именами соответствующих им пакетов — в данном случае p1 и p2.

В первом исходном пакете определено три класса: Protection, Derived и SamePackage. В первом классе определено четыре переменных int в каждом из допустимых режимов защиты. Переменная n объявлена с защитой, заданной по умолчанию, n\_pri объявлена как private, n\_pro — как protected и n\_pub — как public.

Каждый последующий класс в этом примере будет пытаться обратиться к переменным в экземпляре этого класса. Строки, которые не будут компилироваться из-за ограничений доступа, прокомментированы при помощи одностroочного комментария (//...). Перед каждой из этих строк есть комментарий, перечисляющий области, где данный уровень защиты разрешил бы доступ.

Второй класс, Derived, является подклассом Protection в том же пакете p1. Это предоставляет Derived доступ к каждой переменной в Protection за исключением n\_pri, поскольку она private. Третий класс, SamePackage, — не подкласс Protection, но он находится в том же пакете и также имеет доступ ко всем переменным суперкласса, кроме n\_pri.

Итак, файл Protection.java:

```
package p1;

public class Protection {
    int n = 1;
    private int n_pri = 2;
    protected int n_pro = 3;
    public int n_pub = 4;

    public Protection() {
        System.out.println("Основной конструктор");
        System.out.println("n = " + n);
        System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

Файл Derived.java:

```
package p1;

class Derived extends Protection {
    Derived() {
        System.out.println("Производный конструктор");
        System.out.println("n = " + n);
```

```
// Только класс
// System.out.println("n_pri = " + n_pri);

System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}

}
```

Файл SamePackage.java:

```
package p1;
class SamePackage {
    SamePackage() {
        Protection p = new Protection();
        System.out.println("Тот же самый конструктор пакета");
        System.out.println("n = " + p.n);

// только класс
// System.out.println("n_pri = " + p.n_pri);

        System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}
```

Далее следует исходный код (текст) для пакета p2. Два класса, определенных в p2, охватывают два других условия, на которые влияет управление доступом. Первый класс, Protection2, является подклассом p1.Protection. Он предоставляет доступ ко всем переменным класса p1.Protection, кроме n\_pri (потому что она — private) и n (т. к. она объявлена с защитой, заданной по умолчанию). Помните, умолчание разрешает доступ только изнутри класса или пакета, не из подклассов внешних пакетов. Наконец, класс OtherPackage имеет доступ только к одной переменной, n\_pub, которая была объявлена как public.

Файл Protection2.java:

```
package p2;

class Protection2 extends p1.Protection {
    Protection2() {
        System.out.println("Производный конструктор другого пакета");

// только класс или пакет
// System.out.println("n = " + n);

// только класс
// System.out.println("n_pri = " + n_pri);
```

```

        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}

```

### Файл OtherPackage.java:

```

package p2;

class OtherPackage {
    OtherPackage() {
        p1.Protection p = new p1.Protection();
        System.out.println("Конструктор другого пакета");

//    только класс или пакет
//    System.out.println("n = " + p.n);

//    только класс
//    System.out.println("n_pri = " + p.n_pri);

//    только класс, подкласс или пакет
//    System.out.println("n_pro = " + p.n_pro);

        System.out.println("n_pub = " + p.n_pub);
    }
}

```

Если нужно тестировать эти два пакета, то можно использовать два приведенных далее тест-файла. Тест-файл для пакета p1:

```

// demo-пакет p1.
package p1;

// Создает объекты различных классов из p1.
public class Demo {
    public static void main(String args[]) {
        Protection ob1 = new Protection();
        Derived ob2 = new Derived();
        SamePackage ob3 = new SamePackage();
    }
}

```

Тест-файл для пакета p2:

```

// demo-пакет p2.
package p2;

// Создает объекты различных классов из p2.
public class Demo {

```

```

public static void main(String args[]) {
    Protection2 ob1 = new Protection2();
    OtherPackage ob2 = new OtherPackage();
}
}

```

## Импорт пакетов

Учитывая, что пакеты существуют и являются хорошим механизмом для отделения различных классов друг от друга, нетрудно понять, почему все встроенные классы Java хранятся в пакетах. В неименованном пакете умолчания нет классов ядра Java, все стандартные классы хранятся в нескольких *именованных* пакетах. Так как классы в пакетах должны быть полностью квалифицированы именем (или именами) их пакета, весьма утомительно печатать длинное, разделенное точками полное пакетное имя каждого класса, который вы хотите использовать. Чтобы обеспечить видимость некоторых классов или полных пакетов Java, используется оператор `import`. После импортирования на класс можно ссылаться прямо, используя только его имя. Для записи законченной программы в операторе `import` нет технической необходимости, но он удобен для программиста. Однако, если вы собираетесь обращаться к нескольким десяткам классов в вашем приложении, оператор `import` сохранит массу усилий и времени при вводе кода.

В исходном файле Java оператор `import` следует немедленно после оператора `и package` (если он используется) и перед любыми определениями класса. Общая форма оператора `import`:

```
import pkg1[.pkg2].(classname | *);
```

Здесь `pkg1` — имя пакета верхнего уровня, `pkg2` — имя подчиненного пакета внутри внешнего пакета (имена разделяются точкой). Нет никакого практического предела глубине пакетной иерархии, за исключением того, что обусловлен файловой системой. Наконец, вы определяете или явное имя класса, или звездочку (\*), которая указывает, что компилятор Java должен импортировать полный пакет. Следующий кодовый фрагмент показывает использование обеих форм:

```
import java.util.Date;
import java.io.*;
```

### Предупреждение

Форма со звездочкой может увеличить время компиляции — особенно если вы импортируете несколько больших пакетов. По этой причине лучше явно называть классы, которые вы хотите использовать, а не импортировать целые пакеты. Однако такая форма не имеет абсолютно никакого влияния на эффективность времени выполнения или на размер классов.

Все стандартные классы Java хранятся в пакете с именем Java. Основные функции языка размещаются во внутреннем пакете Java с именем java.lang. Обычно нужно импортировать каждый пакет или класс, который вы хотите использовать, но, так как Java бесполезен без многих функциональных возможностей java.lang, указанный пакет неявно импортируется компилятором для всех программ. Это эквивалентно появлению следующей строки в начале всех ваших программ:

```
import java.lang.*;
```

Если класс с тем же именем существует в двух различных пакетах, которые вы импортируете, используя звездочку в качестве имени класса, компилятор никак не отреагирует, если вы не попробуете использовать один из этих классов. В том случае вы получите ошибку времени компиляции и должны явно именовать класс, указывая его пакет.

Везде, где вы используете имя класса, можно применять полное составное имя, включающее всю иерархию его пакетов. Например, следующий фрагмент использует оператор import:

```
import java.util.*;
class MyDate extends Date {
}
```

Тот же пример без оператора import выглядит так:

```
class MyDate extends java.util.Date {
}
```

При импорте пакетов, как показано в табл. 9.1, неподклассам в импортирующем коде будут доступны только те элементы пакета, которые объявляются как public. Например, если нужно, чтобы класс Balance пакета MyPack, показанный ранее, был автономным классом общего пользования вне пакета MyPack, то необходимо объявить его как public и поместить в собственный файл, как показано ниже:

```
package MyPack;

/* Теперь класс Balance, его конструктор и метод show()
   являются общими (public). Это означает, их может
   использовать код неподкласса, находящийся вне их пакета. */
public class Balance {
    String name;
    double bal;
    public Balance(String n, double b) {
        name = n;
        bal = b;
    }
}
```

```

public void show() {
    if(bal<0)
        System.out.print(">>> ");
    System.out.println(name + ": $" + bal);
}
}

```

Как вы видите, класс Balance теперь — с общим (`public`) доступом. То же можно сказать о его конструкторе и методе `show()`. Это означает, что они доступны любым типам кода вне пакета MyPack. Например, в следующей программе класс `TestBalance` импортирует пакет MyPack и получает возможность использовать класс `Balance`:

```

import MyPack.*;

class TestBalance {
    public static void main(String args[]) {
        /* Из-за того, что Balance теперь public, вы можете использовать
         * класс Balance и вызывать его конструктор. */
        Balance test = new Balance("J. J. Jaspers", 99.88);

        test.show();           // можно также вызвать show()
    }
}

```

В качестве эксперимента удалите спецификатор `public` из класса `Balance` и затем попробуйте откомпилировать `TestBalance`. Как объяснялось ранее, в результате вы получите сообщение об ошибке.

## Интерфейсы

Применяя ключевое слово `interface`, вы можете полностью освободить интерфейс класса от его реализации. То есть, используя `interface`, вы можете указать, что класс должен делать, но не как он это делает. Интерфейсы синтаксически подобны классам, но в них нет экземплярных переменных, и их методы объявляются без тела. Практически, это означает, что вы можете определять интерфейсы, которые не делают предположений относительно того, как они реализованы. Как только интерфейс определен, то реализовать его может любое число классов. И наоборот, один класс может реализовать любое число интерфейсов.

Для реализации интерфейса класс должен создать полный набор методов, определенных интерфейсом. Однако каждый класс волен сам определять детали своей реализации. Ключевое слово `interface` позволяет полностью использовать аспект полиморфизма, декларируемый как "один интерфейс, множественные методы".

Интерфейсы разработаны для поддержки динамического вызова методов во время выполнения. Обычно для вызова метода одного класса из другого нужно, чтобы оба класса присутствовали во время компиляции и компилятор Java мог проверить совместимость сигнатур методов. Само по себе это требование предъявляется к статической и нерасширяемой среде классификации. В иерархической же многоуровневой системе, где функциональные возможности обеспечиваются длинными цепочками связанных в иерархию классов, этот механизм неизбежно используется все большим и большим числом подклассов. Интерфейсы разработаны для того, чтобы обойти эту проблему. Они исключают определение метода или набора методов из иерархии наследования. Так как интерфейсы находятся вне иерархии классов, то для несвязанных в этой иерархии классов появляется иная, более эффективная возможность реализации интерфейсов. В этом и заключена действительная их польза.

### Замечание

Именно интерфейсы добавляют во многие приложения те функциональные возможности, которые обычно потребовали бы использования множественного наследования языков типа C++.

## Определение интерфейса

Определение интерфейса во многом подобно определению класса. Общая форма интерфейса выглядит так:

```
access interface name {  
    return-type method-name1(parameter-list);  
    return-type method-name2(parameter-list);  
    type final-varname1 = value;  
    type final-varname2 = value;  
//...  
    return-type method-nameN(parameter-list);  
    type final-varnameN = value;  
}
```

Здесь **access** — спецификатор доступа (или `public` или не используется). Если никакой спецификатор доступа не включен, тогда используется доступ по умолчанию, и интерфейс доступен только другим членам пакета, в котором он объявлен. При объявлении с `public` интерфейс может использоваться любым другим кодом. **name** — имя интерфейса, им может быть любой допустимый идентификатор. Обратите внимание, что объявленные методы не имеют тел. Они заканчиваются точкой с запятой после списка параметров. Это, по существу, абстрактные методы. В пределах интерфейса для них нет никаких умалчиваемых реализаций. Каждый класс, который включает интерфейс, должен реализовать все его методы.

Внутри объявлений интерфейсов можно объявлять переменные. Они неявно считаются `final` и `static` (это означает, что они не могут быть изменены реализующим классом), а также должны быть инициализированы постоянными значениями. Все методы и переменные интерфейсов — неявно общие (`public`), если интерфейс сам не объявлен как `public`.

Пример определения интерфейса:

```
interface Callback {
    void callback(int param);
}
```

Здесь объявлен простой интерфейс, содержащий один метод с именем `callback()`, который имеет единственный целый параметр.

## Реализация интерфейсов

Когда интерфейс определен, он может реализовываться одним или несколькими классами. Для реализации интерфейса в определение класса включают предложение с ключевым словом `implements` и затем создают методы, определенные в интерфейсе. Общая форма класса, который включает `implements` предложение, выглядит примерно так:

```
access class classname [extends superclass]
    implements interface [,interface...]
    // тело-класса
}
```

Здесь `access` — спецификатор доступа (`public` или не используется). Если класс реализует более одного интерфейса, они разделяются запятой. Если класс реализует два интерфейса, которые объявляют один и тот же метод, то клиенты любого интерфейса будут использовать один и тот же метод. Методы, которые реализуют интерфейс, должны быть объявлены как `public`. Кроме того, сигнатура типа реализующего метода должна точно соответствовать сигнатуре типа, указанной в определении интерфейса.

Пример небольшого класса, который реализует интерфейс `Callback`, показанный ранее:

```
class Client implements Callback {
    // Реализация Callback-интерфейса
    public void callback(int p) {
        System.out.println("callback вызван с аргументом " + p);
    }
}
```

Обратите внимание, что `callback()` объявлен со спецификатором доступа `public`.

### Замечание

При реализации метода интерфейса он должен быть объявлен как `public`.

Для классов, которые реализуют интерфейсы, обычно допустимо определять дополнительные собственные члены. Например, следующая версия класса `Client` реализует метод `callback()` и добавляет метод `nonIFaceMeth()`:

```
class Client implements Callback {  
    // Реализация Callback интерфейса  
    public void callback(int p) {  
        System.out.println("callback вызван с аргументом " + p);  
    }  
  
    void nonIFaceMeth() {  
        System.out.println("Классы, реализующие интерфейсы " +  
            "должны также определять другие члены.");  
    }  
}
```

## Реализации доступа через интерфейсные ссылки

Можно объявлять переменные как объектные ссылки, которые используют интерфейсный тип, а не тип класса. В такой переменной можно сохранять всякий экземпляр любого класса, который реализует объявленный интерфейс. Когда вы вызываете метод через ссылку такого рода, будет вызываться его правильная версия, основанная на актуальном экземпляре интерфейса. Это — одно из ключевых свойств интерфейсов. Выполняемый метод отыскивается динамически (во время выполнения), что позволяет создавать классы позже кода, который вызывает их методы. Кодом вызова можно управлять через интерфейс, ничего не зная об объекте вызова. Этот процесс подобен использованию ссылки суперкласса для доступа к объекту подкласса, как описано в главе 8.

### Предупреждение

Поскольку динамический поиск метода во время выполнения приводит к существенной потере производительности по сравнению с нормальным вызовом, следует проявлять осторожность и беспричинно не использовать интерфейсы там, где нужна повышенная эффективность.

В следующем примере вызов метода `callback()` выполняется через ссылочную переменную интерфейса:

```
class TestIface {  
    public static void main(String args[]) {  
        Callback c = new Client();  
        c.callback(42);  
    }  
}
```

Вывод этой программы:

```
callback вызван с аргументом 42
```

Обратите внимание, что переменной `c`, объявленной с типом интерфейса `Callback`, был назначен экземпляр класса `Client`. Хотя разрешается использовать `c` для обращения к методу `callback()`, она не может обращаться к любым другим членам класса `Client`. Переменная интерфейсной ссылки обладает знаниями только методов, объявленных в соответствующей интерфейсной декларации. Таким образом, с нельзя использовать для обращения к методу `nonIfaceMeth()`, т. к. он определен классом `Client`, а не интерфейсом `Callback`.

Хотя предыдущий пример отражает чисто технически, как переменная интерфейсной ссылки может обращаться к объекту реализации, он не демонстрирует полиморфную мощь такой ссылки. Чтобы показать это, сначала создайте другую реализацию `Callback`, как показано ниже:

```
// Другая реализация Callback.
class AnotherClient implements Callback {
    // Implement Callback's interface
    public void callback(int p) {
        System.out.println("Другая версия callback");
        System.out.println("Квадрат p равен " + (p*p));
    }
}
```

Теперь, протестируйте следующий класс:

```
class TestIface2 {
    public static void main(String args[]) {
        Callback c = new Client();
        AnotherClient ob = new AnotherClient();
        c.callback(42);
        c = ob;           // с теперь ссылается на объект AnotherClient
        c.callback(42);
    }
}
```

Вывод этой программы:

```
callback вызван с аргументом 42
```

```
Другая версия callback
```

```
Квадрат p равен 1764
```

Здесь видно, что вызываемая версия `callback()` определяется типом объекта, к которому обращается `c` во время выполнения. Хотя это и очень простой пример, скоро вы увидите другой, более практический.

## Частичные реализации

Если класс включает интерфейс, но полностью не реализует методы, определенные этим интерфейсом, то этот класс должен быть объявлен как `abstract` (абстрактный). Например:

```
abstract class Incomplete implements Callback {  
    int a, b;  
    void show() {  
        System.out.println(a + " " + b);  
    }  
    // ...  
}
```

Здесь класс `Incomplete` не реализует `callback()` и должен быть объявлен как абстрактный. Любой класс, который наследует `Incomplete`, должен реализовать `callback()` или объявить себя как `abstract`.

## Применения интерфейсов

Чтобы понять мощь интерфейсов, рассмотрим более практический пример. В предыдущих главах вы разработали класс с именем `Stack`, который реализовал простой стек фиксированного размера. Однако существуют и другие способы. Например, стек может иметь фиксированный размер, или быть "растущим". Стек может также содержаться в массиве, связном списке, двоичном дереве и т. д. Независимо от того, как стек реализован, интерфейс стека остается тем же самым. То есть методы `push()` и `pop()` определяют интерфейс к стеку независимо от подробностей реализации. Поскольку интерфейс к стеку отделен от его реализации, то легко определить интерфейс стека, оставляя за каждой реализацией определение специфики. Рассмотрим два примера.

Ниже показан интерфейс, определяющий целый стек. Поместите его в файле с именем `IntStack.java`. Этот интерфейс будет использоваться обеими реализациями стека.

```
// Определение интерфейса целого стека.  
interface IntStack {  
    void push(int item);           // запомнить элемент  
    int pop();                    // извлечь элемент  
}
```

Следующая программа создает класс с именем `FixedStack`, который реализует версию целого стека фиксированной длины:

```
// Реализация IntStack, которая использует фиксированную память.  
class FixedStack implements IntStack {
```

```
private int stck[];
private int tos;

// выделить память и инициализировать стек
FixedStack(int size) {
    stck = new int[size];
    tos = -1;
}

// поместить элемент в стек
public void push(int item) {
    if(tos==stck.length-1)      // использовать переменную length
        System.out.println("Стек заполнен.");
    else
        stck[++tos] = item;
}

// извлечь элемент из стека
public int pop() {
    if(tos < 0) {
        System.out.println("Стек пуст.");
        return 0;
    }
    else
        return stck[tos--];
}

}

class IFTest {
    public static void main(String args[]) {
        FixedStack mystack1 = new FixedStack(5);
        FixedStack mystack2 = new FixedStack(8);

        // поместить в стек несколько чисел
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);

        // извлечь из стека эти числа
        System.out.println("Стек в mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());

        System.out.println("Стек в mystack2:");
        for(int i=0; i<8; i++)
            System.out.println(mystack2.pop());
    }
}
```

Далее следует другая реализация `IntStack`, которая создает динамический стек, используя то же самое его определение. В ней каждый стек создается с некоторой исходной длиной. Если эта исходная длина превышена, то стек увеличивается в размере. Каждый раз, когда необходим больший участок памяти, размер стека удваивается.

```
// Реализует "растущий" стек.
class DynStack implements IntStack {
    private int stck[];
    private int tos;

    // выделить память и инициализировать стек
    DynStack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // поместить элемент в стек
    public void push(int item) {
        // если стек заполнен, увеличить его размер
        if(tos==stck.length-1) {
            int temp[] = new int[stck.length * 2];      // двойной размер
            for(int i=0; i<stck.length; i++) temp[i] = stck[i];
            stck = temp;
            stck[++tos] = item;
        }
        else
            stck[++tos] = item;
    }

    // извлечь элемент из стека
    public int pop() {
        if(tos < 0) {
            System.out.println("Стек пуст.");
            return 0;
        }
        else
            return stck[tos--];
    }
}

class IFTest2 {
    public static void main(String args[]) {
        DynStack mystack1 = new DynStack(5);
        DynStack mystack2 = new DynStack(8);

        // эти циклы заставляют каждый стек расти
        for(int i=0; i<12; i++) mystack1.push(i);
        for(int i=0; i<20; i++) mystack2.push(i);
    }
}
```

```

System.out.println("Стек в mystack1:");
for(int i=0; i<12; i++)
    System.out.println(mystack1.pop());

System.out.println("Стек в mystack2:");
for(int i=0; i<20; i++)
    System.out.println(mystack2.pop());
}
}

```

Следующий класс использует обе реализации DynStack и FixedStack через интерфейсную ссылку. Это означает, что обращение к push() и pop() осуществляется во время выполнения, а не во время компиляции.

```

/* Создать интерфейсную переменную
   и обратиться к стекам через нее. */
class IFTest3 {
    public static void main(String args[]) {
        IntStack mystack;           // создать ссылочную переменную интерфейса
        DynStack ds = new DynStack(5);
        FixedStack fs = new FixedStack(8);

        mystack = ds;              // загрузить динамический стек
        // поместить несколько чисел в стек
        for(int i=0; i<12; i++) mystack.push(i);

        mystack = fs;              // загрузить фиксированный стек
        for(int i=0; i<8; i++) mystack.push(i);

        mystack = ds;
        System.out.println("Значения динамического стека:");
        for(int i=0; i<12; i++)
            System.out.println(mystack.pop());

        mystack = fs;
        System.out.println("Значения фиксированного стека:");
        for(int i=0; i<8; i++)
            System.out.println(mystack.pop());
    }
}

```

В этой программе mystack — ссылка на интерфейс IntStack. Таким образом, когда она обращается к ds, то использует версии push() и pop(), определенные реализацией DynStack. Когда она обращается к fs, то использует версии push() и pop(), определенные в FixedStack. Напомним, что эти определения делаются во время выполнения. Доступ к множественным реализациям интерфейса через интерфейсную ссылочную переменную — самое мощное

средство, с помощью которого Java достигает полиморфизма времени выполнения.

## Переменные в интерфейсах

Можно использовать интерфейсы для импорта разделяемых констант во множественные классы просто объявлением интерфейса, который содержит переменные, инициализированные желательными значениями. Когда вы включаете этот интерфейс в класс (т. е. "реализуете" интерфейс), все имена указанных переменных окажутся в области их видимости как константы. Это подобно использованию файла заголовка в C/C++, который создает большое количество констант #defined или const-объявлений. Если интерфейс не содержит методов, то любой класс, который включает такой интерфейс, фактически не реализует ничего. Это выглядит так, как если бы данный класс импортировал постоянные переменные в пространство имен класса как final-переменные. Следующий пример использует эту методику, чтобы реализовать автоматизированное "средство принятия решений":

```
import java.util.Random;

interface SharedConstants {
    int NO = 0;
    int YES = 1;
    int MAYBE = 2;
    int LATER = 3;
    int SOON = 4;
    int NEVER = 5;
}

class Question implements SharedConstants {
    Random rand = new Random();
    int ask() {
        int prob = (int) (100 * rand.nextDouble());
        if (prob < 30)
            return NO;                  // 60%
        else if (prob < 60)
            return YES;                // 30%
        else if (prob < 75)
            return LATER;              // 15%
        else if (prob < 98)
            return SOON;                // 13%
        else
            return NEVER;              // 2%
    }
}
```

```
class AskMe implements SharedConstants {
    static void answer(int result) {
        switch(result) {
            case NO:
                System.out.println("Нет");
                break;
            case YES:
                System.out.println("Да");
                break;
            case MAYBE:
                System.out.println("Возможно");
                break;
            case LATER:
                System.out.println("Позже");
                break;
            case SOON:
                System.out.println("Вскоре");
                break;
            case NEVER:
                System.out.println("Никогда");
                break;
        }
    }

    public static void main(String args[]) {
        Question q = new Question();
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
    }
}
```

Обратите внимание, что данная программа использует один из стандартных классов Java — `Random`. Этот класс генерирует псевдослучайные числа. Он содержит несколько методов, которые позволяют получать случайные числа в форме, необходимой вашей программе. В представленном выше примере используется метод `nextDouble()`. Он возвращает случайные числа в диапазоне от 0.0 до 1.0.

Программа содержит два класса — `Question` и `AskMe`, оба реализуют интерфейс `SharedConstants`, где определены константы `NO`, `YES`, `MAYBE`, `SOON`, `LATER` и `NEVER`. Внутри каждого класса код обращается к ним так, будто каждый класс прямо определил или унаследовал эти константы. Вывод, выполненный этой программой:

Позже  
Вскоре  
Нет  
Да

Заметим, что результаты различны для каждого прогона программы.

## Расширение интерфейсов

Один интерфейс может наследовать другой при помощи ключевого слова `extends`. Синтаксис — тот же самый, что для наследования классов. Когда класс реализует интерфейс, который наследует другой интерфейс, первый должен обеспечить реализацию для всех методов, определенных в цепочке наследования интерфейса. Например:

```
// Один интерфейс расширяет другой.
interface A {
    void meth1();
    void meth2();
}

// В теперь включает meth1() и meth2(), а сам он добавляет meth3().
interface B extends A {
    void meth3();
}

// Этот класс должен реализовать все из А и В.
class MyClass implements B {
    public void meth1() {
        System.out.println("Реализует meth1().");
    }

    public void meth2() {
        System.out.println("Реализует meth2().");
    }

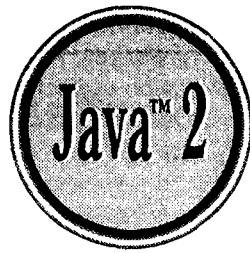
    public void meth3() {
        System.out.println("Реализует meth3().");
    }
}

class IFExtend {
    public static void main(String arg[]) {
        MyClass ob = new MyClass();
        ob.meth1();
        ob.meth2();
    }
}
```

```
    ob.meth3();  
}  
}
```

В качестве эксперимента можно было бы удалить реализацию meth1() из MyClass. Это вызовет ошибку во время компиляции. Как было заявлено ранее, любой класс, который реализует интерфейс, должен реализовать все методы, определенные этим интерфейсом, включая те, что унаследованы от других интерфейсов.

Хотя примеры, включенные в эту книгу, не часто используют пакеты или интерфейсы, оба этих инструмента являются важной частью среды программирования Java. Фактически все реальные программы и апплеты, которые записываются на языке Java, будут содергаться в пакетах. Некоторые из них, вероятно, будут реализовать также и интерфейсы. Это важно, потому что вам удобно их использование.



## ГЛАВА 10

# Обработка исключений

В этой главе рассматривается механизм обработки исключений языка Java. *Исключение* — это аварийное состояние, которое возникает в кодовой последовательности во время выполнения. Другими словами, исключение — это ошибка времени выполнения. В машинных языках, не поддерживающих обработку исключений, ошибки должны быть проверены и обработаны вручную — обычно с помощью кодов ошибки, и т. д. Такой подход довольно сложен и трудоемок. Обработка исключений в Java избегает этих проблем и переносит управление обработкой ошибок времени выполнения в объектно-ориентированное русло.

## Основные принципы обработки исключений

Исключение в языке Java — это объект, который описывает исключительную (т. е. ошибочную) ситуацию, произошедшую в некоторой части кода. Когда исключительная ситуация возникает, создается объект, представляющий это исключение, и "*вбрасывается*" в метод, вызвавший ошибку. В свою очередь, метод может выбрать, обрабатывать ли исключение самому или передать его куда-то еще. В любом случае, в некоторой точке исключение "*захватывается*" и обрабатывается. Исключения могут генерироваться исполнительной системой Java, или ваш код может генерировать их "вручную". Выбрасываемые исключения касаются фундаментальных ошибок, которые нарушают ограничения среды выполнения или правила языка Java. Исключения, сгенерированные вручную, обычно используются, чтобы сообщить вызывающей программе о некоторой аварийной ситуации.

Обработка исключений в Java управляется с помощью пяти ключевых слов — `try`, `catch`, `throw`, `throws` и `finally`. Опишем кратко, как они работают. Программные операторы, которые нужно контролировать относительно

исключений, содержатся в блоке `try`. Если в блоке `try` происходит исключение, говорят, что оно *выброшено* (*thrown*) этим блоком. Ваш код может *перехватить* (*catch*) это исключение (используя оператор `catch`) и обработать его некоторым рациональным способом. Исключения, генерируемые исполнительной (run-time) системой Java, выбрасываются автоматически. Для "ручного" выброса исключения используется ключевое слово `throw`. Любое исключение, которое выброшено из метода, следует определять с помощью ключевого слова `throws`, размещаемого в заголовочном предложении определения метода. Любой код, который обязательно должен быть выполнен перед возвратом из `try`-блока, размещается в `finally`-блоке, указанном в конце блочной конструкции `try{...}-catch{...}-finally{...}`.

Общая форма блока обработки исключений:

```
try{
    // блок кода для контроля над ошибками
}
catch (ExceptionType1 exOb) {
    // обработчик исключений для ExceptionType1
}
catch (ExceptionType2 exOb) {
    // обработчик исключений для ExceptionType2
}
//...
[finally{
    // блок кода для обработки перед возвратом из try блока
}]
```

Здесь `ExceptionType` — тип исключения, которое возникло; `exOb` — объект этого исключения. `finally`-блок — не обязателен. Остальная часть этой главы описывает, как следует применять эту структуру.

## Типы исключений

Все типы исключений являются подклассами встроенного класса `Throwable`. Таким образом, `Throwable` представляет собой вершину иерархии классов исключений. Непосредственно ниже `Throwable` находятся два подкласса, которые разделяют исключения на две различные ветви. Одна ветвь возглавляется классом `Exception`. Этот класс используется для исключительных состояний, которые должны перехватывать программы пользователя. Это также класс, в подклассах которого вы будете создавать ваши собственные заказные типы исключений. У `Exception` имеется важный подкласс, называемый `RuntimeException`. Исключения этого типа для ваших программ определены автоматически и включают такие события, как деление на нуль, недопустимая индексация массива и т. п.

Другую ветвь возглавляет класс `Error`, определяющий исключения, перехват которых вашей программой при нормальных обстоятельствах не ожидается. Исключения типа `Error` применяются исполнительной системой Java для указания ошибок, имеющих отношение непосредственно к среде времени выполнения. Пример подобной ошибки — переполнение стека. Эта глава не содержит описание исключений типа `Error` из-за того, что они обычно создаются в ответ на катастрофические отказы, которые, как правило, не могут обрабатываться вашей программой.

## Неотловленные исключения

Прежде чем узнать, как обрабатывать исключения в своей программе, полезно посмотреть, что произойдет, если их не обрабатывать. Следующая маленькая программа включает выражение, которое преднамеренно вызывает ошибку деления на нуль.

```
class Exc0 {  
    public static void main(String args[]) {  
        int d = 0;  
        int a = 42 / d;  
    }  
}
```

Когда исполнительная система Java обнаруживает попытку деления на нуль, она создает новый объект исключения и затем *выбрасывает* его. Это заставляет выполнение `Exc0` остановиться, потому что, как только исключение окажется выброшенным, оно должно быть *захвачено* обработчиком исключений, и причем — немедленно. В представленном примере мы не снабдили набор имеющихся обработчиков исключений своим собственным вариантом, так что исключение захватывается обработчиком, заданным исполнительной системой Java по умолчанию. Любое исключение, которое не захвачено программой, будет в конечном счете выполнено обработчиком по умолчанию. Этот обработчик отображает (на экран) строку, описывающую исключение, печатает трассу стека от точки, в которой произошло исключение, и завершает программу.

Вот какой вывод генерирует предложенный пример, когда он выполняется стандартным Java-интерпретатором из JDK:

```
java.lang.ArithmaticException: / by zero      // сообщение об ошибке  
at Exc0.main(Exc0.java:4)                      // трасса стека
```

Обратите внимание, как включены в трассу стека следующие элементы: имя класса (`Exc0`), имя метода (`main`), имя файла (`Exc0.java`) и номер строки (4). Кроме того, из первой строки сообщения видно, что тип выброшенного исключения является подклассом `Exception` с именем `ArithmaticException`, которое

более определенно описывает тип произошедшей ошибки. Как будет обсуждено позже в этой главе, Java поставляет несколько встроенных типов исключений, которые соответствуют различным видам ошибок, генерируемых во время выполнения.

Трасса стека всегда показывает последовательность вызовов методов, которые привели к ошибке. Например, вот другая версия предыдущей программы, которая представляет ту же самую ошибку, но в отдельном от `main()` методе:

```
class Excl {
    static void subroutine() {
        int d = 0;
        int a = 10 / d;
    }
    public static void main(String args[]) {
        Excl.subroutine();
    }
}
```

Результирующая трасса стека (полученная от обработчика исключений по умолчанию) показывает, как отображается полный стек вызовов:

```
java.lang.ArithException: / by zero          // (сообщение об ошибке)
    at Excl.subroutine(Excl.java:4)           // (трасса полного стека
    at Excl.main(Excl.java:7)                 // вызовов)
```

Как вы видите, основанием стека является строка 7 метода `main`, которая является обращением к методу `subroutine()`, генерирующему исключение в строке 4. Стек вызовов весьма полезен для отладки, потому что он довольно точно отражает последовательность шагов, которые привели к ошибке.

## Использование операторов `try` и `catch`

Хотя умалчиваемый обработчик исключений исполнительной системы Java полезен для отладки, программисту обычно хочется обрабатывать исключение самостоятельно. Это обеспечивает два преимущества: во-первых, позволяет фиксировать ошибку и, во-вторых, предохраняет программу от автоматического завершения. Большинство пользователей было бы смущено (мягко говоря), если бы ваша программа прекращала выполнение и печатала трассу стека всякий раз, когда произошла ошибка. К счастью, эту ситуацию весьма просто предотвратить.

Для того чтобы отслеживать и обрабатывать ошибку времени выполнения, просто включите код, который нужно контролировать, внутрь блока `try`. Сразу после блока `try` укажите `catch`-блок, определяющий тип исключения, которое нужно перехватить, и его обработчик. Чтобы проиллюстрировать,

как легко это можно сделать, приведем программу, включающую блок `try` и блок `catch`, который обрабатывает исключение типа `ArithmeticeException`, генерируемое ошибкой "деление на нуль":

```
class Exc2 {  
    public static void main(String args[]) {  
        int d, a;  
  
        try {  
            d = 0;  
            a = 42 / d;  
            System.out.println("Это не будет напечатано.");  
        } catch (ArithmeticeException e) {  
            // перехватить ошибку  
            // деления на нуль  
            System.out.println("Деление на нуль.");  
        }  
        System.out.println("После оператора catch.");  
    }  
}
```

Эта программа генерирует следующий вывод:

Деление на нуль.

После оператора catch.

Обратите внимание, что обращение к `println()` внутри блока `try` никогда не выполняется. Как только исключение выброшено, управление программой передается из блока `try` в блок `catch`. Размещенный по-другому блок `catch` не вызывается, так что управление (выполнением) никогда не возвращается из блока `catch` блоку `try`. Таким образом, строка "Это не будет напечатано." никогда не выводится на экран. Сразу после выполнения оператора `catch` программное управление продолжается со строки, следующей за полным механизмом `try/catch`.

Таким образом, `try` и его `catch`-оператор формируют небольшой программный модуль (точнее — пару связанных блоков). Область видимости `catch`-утверждения ограничена ближайшим предшествующим утверждением `try`. Оператор `catch` не может захватывать исключение, выброшенное другим `try`-оператором (кроме случая вложенных `try`-операторов, кратко описанных далее). Операторы, которые контролируются утверждением `try`, должны быть окружены фигурными скобками (т. е. они должны быть внутри блока). Нельзя использовать `try` с одиночным утверждением (без скобок).

Целью хорошо сконструированного `catch`-предложения должно быть разрешение исключительной ситуации с последующим продолжением выполнения программы, как будто ошибка никогда не возникала. Например, в следующей программе каждая итерация цикла `for` получает два случайных целых числа. Они делятся друг на друга, и их частное используется для

деления значения 12 345. Конечный результат помещается в переменную a. Если любая операция деления приводит к ошибке деления на нуль, она перехватывается, значение сбрасывается в нуль, и программа продолжается.

```
// Обработать исключение и продолжить.
import java.util.Random;

class HandleError {
    public static void main(String args[]) {
        int a=0, b=0, c=0;
        Random r = new Random();

        for(int i=0; i<32000; i++) {
            try {
                b = r.nextInt();
                c = r.nextInt();
                a = 12345 / (b/c);
            } catch (ArithmetricException e) {
                System.out.println("Деление на нуль.");
                a = 0;           // сбросить в нуль и продолжить
            }
            System.out.println("a: " + a);
        }
    }
}
```

## Отображение описания исключения

Класс Throwable переопределяет метод `toString()` (определенный в классе `Object`) так, чтобы он возвращал строку, содержащую описание исключения. Вы можете отображать это описание методом `println()`, просто передавая ему исключение как аргумент. Например, блок `catch` в предыдущей программе может быть переписан так:

```
catch (ArithmetricException e) {
    System.out.println("Исключение: " + e);
    a = 0;           // сбросить в нуль и продолжить
}
```

Если этой версией `catch` заменить имеющуюся в программе и выполнить программу стандартным Java-интерпретатором из JDK, каждая ошибка деления на нуль отобразит следующее сообщение:

Исключение: Java.lang.ArithmetricException: / by zero

Хотя в данном контексте это не имеет особого значения, в других обстоятельствах способность отображать описание исключения может оказаться

достаточно ценной — особенно, когда вы экспериментируете с исключениями или для отладки.

## Множественные операторы *catch*

В некоторых случаях на одном участке кода может возникнуть более одного исключения. Чтобы обрабатывать этот тип ситуации, необходимо определить несколько операторов *catch*, каждый — для захвата своего типа исключения. Когда выбрасывается исключение, каждый *catch*-оператор просматривается по порядку и первый, чей тип соответствует типу возникшего исключения, выполняется. После того как этот *catch*-оператор выполнится, другие — обходятся, и выполнение продолжается после блока *try/catch*. Следующий пример отлавливает исключение двух различных типов:

```
// Демонстрация множественных catch-операторов.
class MultiCatch {
    public static void main(String args[]) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = 99;
        } catch(ArithmetcException e) {
            System.out.println("Деление на нуль: " + e);
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Индекс элемента массива oob: " + e);
        }
        System.out.println("После блока try/catch.");
    }
}
```

Эта программа выбросит исключение "деление на нуль", если она будет запускаться без параметров командной строки, так как переменная *a* будет равна нулю. Этой ситуации не возникнет, если вы укажете аргумент в командной строке, устанавливающий *a* что-то большее, чем нуль. Но это вызовет исключение *ArrayIndexOutOfBoundsException*, так как целочисленный массив *c* имеет длину 1, тогда как программа пытается назначить некоторое значение его сорок второму элементу *c[42]*.

Ниже показаны экранные протоколы, показывающие оба варианта выполнения:

```
C:\>java MultiCatch
a = 0
```

Деление на нуль: Java.lang.ArithmetiсException: / by zero

После блока try/catch.

C:\>java MultiCatch TestArg

a = 1

Индекс элемента массива oob: Java.lang.ArrayIndexOutOfBoundsException: 42

После блока try/catch.

Когда вы используете множественные catch-операторы, важно помнить, что в последовательности catch-предложений подклассы исключений должны следовать перед любым из их суперклассов. Это происходит потому, что предложение catch, которое использует суперкласс, будет перехватывать исключения как своего типа, так и любого из своих подклассов. Таким образом, подкласс никогда не был бы достигнут, если бы он следовал после своего суперкласса. Кроме того, в Java недостижимый код — ошибка. Например, рассмотрим следующую программу:

```
/* Эта программа содержит ошибку.
```

Подкласс должен следовать раньше своего суперкласса  
в серии catch-операторов. Если это не так,  
то в результате будет создаваться недостижимый код  
и соответствующий тип ошибки времени выполнения. \*/

```
class SuperSubCatch {
    public static void main(String args[]) {
        try {
            int a = 0;
            int b = 42 / a;
        } catch(Exception e) {
            System.out.println("Генерация исключения catch.");
        }
        /* Этот catch никогда не будет достигнут из-за того, что
           ArithmeticException является подклассом Exception. */
        catch(ArithmeticException e) { // ОШИБКА. Оператор недостижим
            System.out.println("Недостижимый оператор.");
        }
    }
}
```

Если вы попытаетесь откомпилировать данную программу, то примете сообщение об ошибке, заявляющее, что второй catch-оператор недостижим. Так как ArithmeticException — подкласс Exception, первый catch-оператор обработает все ошибки, основанные на Exception, включая и ArithmeticException. Это означает, что второй catch-оператор никогда не будет выполняться. Чтобы устранить проблему, измените порядок операторов catch.

## Вложенные операторы try

Операторы try могут быть вложенными. То есть один try-оператор может находиться внутри блока другого оператора try. При входе в блок try контекст соответствующего исключения помещается в стек. Если внутренний оператор try не имеет catch-обработчика для специфического исключения, стек раскручивается, и просматривается следующий catch-обработчик try-оператора (для поиска соответствия с типом исключения). Процесс продолжается до тех пор, пока не будет достигнут подходящий catch-оператор, или пока все вложенные операторы try не будут исчерпаны. Если соглашения о оператора catch нет, то исключение обработает исполнительная система Java. Пример, который использует вложенные операторы try:

```
// Пример вложенных try-операторов.
class NestTry {
    public static void main(String args[]) {
        try {
            int a = args.length;

            /* Если нет аргументов командной строки,
               следующий оператор будет генерировать
               исключение деления на нуль. */
            int b = 42 / a;

            System.out.println("a = " + a);

            try {                                // вложенный try-блок
                /* Если используется один аргумент командной строки,
                   то следующий код будет генерировать
                   исключение деления на нуль. */
                if(a==1) a = a/(a-a);           // деление на нуль

                /* Если используется два аргумента командной строки,
                   то генерируется исключение выхода за границу массива. */
                if(a==2) {
                    int c[] = { 1 };
                    c[42] = 99;                  // генерировать исключение
                                            // выхода за границу массива
                }
            } catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("Индекс выходит за границу массива: " + e);
            }

        } catch(ArithmeticException e) {
            System.out.println("Деление на нуль: " + e);
        }
    }
}
```

Пример демонстрирует вложение одного try-блока в другой. При выполнении программы без аргументов командной строки исключение деления на нуль генерируется внешним блоком try. Выполнение программы с одним аргументом командной строки генерирует исключение деления на нуль внутри вложенного блока try. Так как внутренний блок не захватывает это исключение, он пересыпает его внешнему блоку try, где оно и обрабатывается. Если вы запустите программу с двумя аргументами командной строки, то во внутреннем блоке try генерируется исключение нарушения границы массива. Следующие протоколы выполнения иллюстрируют каждый из этих случаев:

```
C:\>java NestTry
Деление на нуль: Java.lang.ArithmeticException: / by zero
```

```
C:\>java NestTry One
a = 1
Деление на нуль: Java.lang.ArithmeticException: / by zero
```

```
C:\>java NestTry One Two
a = 2
Индекс выходит за границу массива:
Java.lang.ArrayIndexOutOfBoundsException: 42
```

Вложение инструкций try может происходить менее очевидными способами, когда вложенный try-блок организован в отдельном методе и выполняется через вызов этого метода во внешнем блоке. Ниже показана предыдущая программа, переписанная так, чтобы вложенный блок try был перемещен внутрь метода nesttry():

```
/* Try-операторы можно неявно вкладывать
   через возвоны методов. */
class MethNestTry {
    static void nesttry(int a) {
        try {                                // вложенный try-блок
            /* Если используется один аргумент командной строки,
               то следующий код будет генерировать
               исключение деления на нуль. */
            if(a==1) a = a/(a-a);           // деление на нуль

            /* Если используется два аргумента командной строки,
               то генерируется исключение выхода за границу массива. */
            if(a==2) {
                int c[] = { 1 };
                c[42] = 99;                  // генерировать исключение
            }                                // выхода за границу массива
        } catch(ArrayIndexOutOfBoundsException e) {
```

```
System.out.println("Индекс выходит за границу массива: " + e);
}

}

public static void main(String args[]) {
    try {
        int a = args.length;

        /* Если нет аргументов командной строки,
           следующий оператор будет генерировать
           исключение деления на нуль. */
        int b = 42 / a;

        System.out.println("a = " + a);

        nesttry(a);
    } catch(ArithmetcException e) {
        System.out.println("Деление на нуль: " + e);
    }
}
}
```

Вывод этой программы идентичен выводу предыдущего примера.

## Оператор *throw*

До сих пор вы захватывали только исключения, которые выбрасывались исполнительной системой Java. Однако ваша программа может сама явно выбрасывать исключения, используя оператор *throw*. Общая форма оператора *throw* такова:

**Throw *ThrowableInstance*;**

Здесь *ThrowableInstance* должен быть объектом типа *Throwable* или подкласса *Throwable*. Простые типы, такие как *int* или *char*, а также не-*Throwable*-классы (типа *String* и *Object*) не могут использоваться как исключения. Имеется два способа получения *Throwable*-объекта: использование параметра в предложении *catch* или создание объекта с помощью операции *new*.

После оператора *throw* поток выполнения немедленно останавливается, и любые последующие операторы не выполняются. Затем просматривается ближайший включающий блок *try* с целью поиска оператора *catch*, который соответствует типу исключения. Если соответствие отыскивается, то управление передается этому оператору. Если нет, то просматривается следующее включение оператора *try* и т. д. Если соответствующий *catch* не найден, то программу останавливает обработчик исключений, заданный по умолчанию, и затем выводится трасса стека.

Пример программы, которая создает и выбрасывает исключение (обработчик, который захватывает исключение, перебрасывает его во внешний обработчик):

```
// Демонстрирует throw.
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch(NullPointerException e) {
            System.out.println("Захват внутри demoproc.");
            throw e;           // повторный выброс исключения
        }
    }

    public static void main(String args[]) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("Повторный захват: " + e);
        }
    }
}
```

Эта программа получает две возможности иметь дело с одной и той же ошибкой. Сначала `main()` устанавливает контекст исключения и затем вызывает `demoproc()`. Потом метод `demoproc()` устанавливает другой контекст — для обработки особых ситуаций и немедленно выбрасывает новый экземпляр исключения `NullPointerException`, который захватывается на следующей строке. Далее это исключение выбрасывается повторно. Итоговый вывод этой программы:

Захват внутри demoproc.

Повторный захват: Java.lang.NullPointerException: demo

Программа также иллюстрирует, как можно создавать один из стандартных объектов исключений. Уделите побольше внимания следующей строке:

```
throw new NullPointerException("demo");
```

Здесь `new` используется для создания экземпляра класса `NullPointerException`. Все встроенные исключения времени выполнения имеют два конструктора — один без параметра, а другой — со строчным параметром. Когда используется вторая форма, аргумент определяет строку, описывающую исключение. Данная строка отображается на экран, когда объект указывается в качестве аргумента методами `print()` или `println()`. Ее можно также получить с помощью вызова метода `getMessage()`, который определен в классе `Throwable`.

## Методы с ключевым словом *throws*

Если метод способен к порождению исключения, которое он не обрабатывает, он должен определить свое поведение так, чтобы вызывающие методы могли сами предохранять себя от данного исключения. Это обеспечивается включением предложения *throws* в заголовок объявления метода. Предложение *throws* перечисляет типы исключений, которые метод может выбрасывать. Это необходимо для всех исключений, кроме исключений типа *Error*, *RuntimeException* или любых их подклассов. Все другие исключения, которые метод может выбрасывать, должны быть объявлены в предложении *throws*. Если данное условие не соблюдено, то произойдет ошибка времени компиляции.

Общая форма объявления метода, которое включает предложение *throws*:

```
type method-name(parameter-list) throw exception-list
{
    // тело метода
}
```

Здесь *exception-list* — список разделенных запятыми исключений, которые метод может выбрасывать.

Ниже показан пример неправильной программы, пытающейся выбросить исключение, которое она не перехватывает. Поскольку программа не определяет предложение *throws*, чтобы объявить этот факт, программа не будет компилироваться.

```
// Эта программа содержит ошибку и не будет компилироваться.
class ThrowsDemo {
    static void throwOne() {
        System.out.println("Внутри throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        throwOne();
    }
}
```

Чтобы сделать этот пример компилируемым, требуется внести два изменения. Во-первых, нужно объявить, что *throwOne()* выбрасывает исключение *IllegalAccessException*. Во-вторых, *main()* должен определить оператор *try/catch*, который захватывает исключение. Исправленный пример выглядит так:

```
// Теперь эта программа корректна.
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException {
```

```
System.out.println("Внутри throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
    try {
        throwOne();
    } catch (IllegalAccessException e) {
        System.out.println("Выброс " + e);
    }
}
```

Вывод, сгенерированный выполнением этой программы:

Внутри throwOne.

Выброс java.lang.IllegalAccessException: demo

### Блок *finally*

Когда исключение выбрасывается, выполнение метода имеет довольно неровный, нелинейный путь, который изменяет нормальное прохождение потока через метод. В зависимости от того, как кодирован метод, исключение может вызвать даже преждевременный выход из него. В некоторых случаях это могло бы стать. Например, если метод открывает файл для ввода и закрывает его для вывода, то вы вряд ли захотите, чтобы закрывающий файл код был обойден механизмом обработки исключений. Для реализации этой возможности и предназначено ключевое слово `finally`.

finally определяет блок кода, выполняющийся после того, как блок try/catch завершился и перед кодом, следующим за блоком try/catch. Блок finally будет выполняться независимо от того, было ли выброшено исключение или нет. Если исключение было выброшено, конструкции блока finally будут обрабатываться, даже если нет catch-оператора, соответствующего исключению. Предложение finally выполняется каждый раз, когда метод собирается вернуться к вызывающей программе изнутри блока try/catch (через непойманное исключение или явный оператор return), причем выполнение происходит непосредственно перед возвратом из метода. Это может быть полезно для закрытия дескрипторов файла и освобождения любых других ресурсов, которые могли быть распределены в начале метода, с намерением избавиться от них перед возвратом. Предложение finally необязательно. Однако каждый оператор try требует по крайней мере одного предложения catch или finally.

Пример программы, которая демонстрирует три метода, выполняющие выход различными способами (причем ни один не обходится без выполнения своих `finally`-предложений):

```
// Демонстрирует finally.
class FinallyDemo {
    // Выход из метода через исключение.
    static void procA() {
        try {
            System.out.println("Внутри procA");
            throw new RuntimeException("demo");
        } finally {
            System.out.println("finally для procA ");
        }
    }

    // Возврат изнутри try-блока.
    static void procB() {
        try {
            System.out.println("Внутри procB");
            return;
        } finally {
            System.out.println("finally для procB ");
        }
    }

    // Нормальное выполнение try-блока.
    static void procC() {
        try {
            System.out.println("Внутри procC");
        } finally {
            System.out.println("finally procC");
        }
    }

    public static void main(String args[]) {
        try {
            procA();
        } catch (Exception e) {
            System.out.println("Исключение выброшено");
        }
        procB();
        procC();
    }
}
```

В этом примере, procA() преждевременно выходит из блока try, выбрасывая исключение. Перед самым выходом выполняется предложение finally. Оператор try метода procB() выходит с помощью оператора return. Здесь finally запускается перед возвратом из procB(). В методе procC() оператор

`try` выполняется нормально, без ошибки. Однако блок `finally` все же реализовывается.

### Замечание

Если блок `finally` связывается с блоком `try`, то при возврате из блока `try` блок `finally` будет выполняться всегда.

Вывод, сгенерированный предшествующей программой:

```
Внутри procA
finally для procA
Исключение выброшено
Внутри procB
finally для procB
Внутри procC
finally для procC
```

## Встроенные исключения Java

Внутри стандартного пакета `java.lang` определено несколько классов исключений. Некоторые из них использовались в предыдущих примерах. Наиболее общие из этих исключений — это подклассы стандартного типа `RuntimeException`. Так как `java.lang` неявно импортирован во все Java-программы, большинство исключений, производных от `RuntimeException`, доступны автоматически. Более того, их не нужно включать в `throws`-список любого метода. В языке Java они называются *неконтролируемыми исключениями*, потому что компилятор не проверяет, выбрасывает или обрабатывает метод эти исключения. Неконтролируемые исключения, определенные в `java.lang` (как подклассы `Exception`), перечислены в табл. 10.1. Табл. 10.2 содержит описание исключений, определенных в `java.lang`, которые должны быть включены в `throws`-список метода, если данный метод может генерировать одно из указанных исключений и не обрабатывает его сам. Они называются *контролируемыми исключениями* и являются подклассами класса `RuntimeException`. Кроме того, в Java определены несколько других типов исключений, которые относятся к различным библиотекам его классов.

**Таблица 10.1. Подклассы неконтролируемых исключений Java**

Исключение	Значение
<code>ArithmaticException</code>	Арифметическая ошибка типа деления на нуль
<code>ArrayIndexOutOfBoundsException</code>	Индекс массива находится вне границ

Таблица 10.1 (окончание)

Исключение	Значение
ArrayStoreException	Назначение элементу массива несовместимого типа
ClassCastException	Недопустимое приведение типов
IllegalArgumentException	При вызове метода использован незаконный аргумент
IllegalMonitorStateException	Незаконная операция монитора, типа ожидания на разблокированном потоке
IllegalStateException	Среда или приложение находятся в некорректном состоянии
IllegalThreadStateException	Требуемая операция не совместима с текущим состоянием потока
IndexOutOfBoundsException	Некоторый тип индекса находится вне границ
NegativeArraySizeException	Массив создавался с отрицательным размером
NullPointerException	Недопустимое использование нулевой ссылки
NumberFormatException	Недопустимое преобразование строки в числовой формат
SecurityException	Попытка нарушить защиту
StringIndexOutOfBoundsException	Попытка индексировать вне границ строки
UnsupportedOperationException	Встретилась неподдерживаемая операция

Таблица 10.2. Контролируемые исключения, определенные в java.lang

Исключение	Значение
ClassNotFoundException	Класс не найден
CloneNotSupportedException	Попытка клонировать объект, который не реализует интерфейс Cloneable
IllegalAccessException	Доступ к классу отклонен
InstantiationException	Попытка создавать объект абстрактного класса или интерфейса
InterruptedException	Один поток был прерван другим потоком
NoSuchFieldException	Требуемое поле не существует
NoSuchMethodException	Требуемый метод не существует

## Создание собственных подклассов исключений

Хотя встроенные исключения языка Java обрабатывают наиболее общие ошибки, вам, вероятно, захочется создать свои собственные типы исключений с целью обработки ситуаций, специфических для ваших приложений. Для этого просто определите подкласс `Exception` (который, конечно, является подклассом `Throwable`). На самом деле ваши подклассы не должны ничего реализовывать — само существование их в системе типов позволяет использовать их как исключения.

Класс `Exception` не определяет никаких собственных методов, а наследует эти методы от класса `Throwable`. Таким образом, всем исключениям, даже тем, что вы создаете сами, доступны методы `Throwable`. Список их представлен в табл. 10.3. Кроме того, вы можете переопределить один или несколько этих методов в создаваемых вами классах исключений.

**Таблица 10.3. Методы, определенные в `Throwable`**

Метод	Описание
<code>Throwable fillInStackTrace()</code>	Возвращает <code>Throwable</code> -объект, который содержит полную трассу стека. Этот объект может быть выброшен повторно
<code>String getLocalizedMessage()</code>	Возвращает локализованное описание исключения
<code>String getMessage()</code>	Возвращает описание исключения
<code>void printStackTrace()</code>	Отображает трассу стека
<code>void printStackTrace(PrintStream stream)</code>	Посыпает трассу стека указанному потоку
<code>void printStackTrace(Writer stream)</code>	Посыпает проекцию прямой стека указанному потоку
<code>String toString()</code>	Возвращает <code>String</code> -объект, содержащий описание исключения. Этот метод вызывается из <code>println()</code> при выводе <code>Throwable</code> -объекта

Следующий пример объявляет новый подкласс `Exception` и затем использует его, чтобы сигнализировать об аварийной ситуации в методе. Он переопределяет метод `toString()`, позволяя отобразить описание исключения с помощью `println()`.

```
// Эта программа создает заказной тип исключения.
class MyException extends Exception {
    private int detail;

    MyException(int a) {
        detail = a;
    }

    public String toString() {
        return "MyException[" + detail + "]";
    }
}

class ExceptionDemo {
    static void compute(int a) throws MyException {
        System.out.println("Вызван compute(" + a + ")");
        if(a > 10)
            throw new MyException(a);
        System.out.println("Нормальный выход");
    }

    public static void main(String args[]) {
        try {
            compute(1);
            compute(20);
        } catch (MyException e) {
            System.out.println("Выброшено " + e);
        }
    }
}
```

Этот пример определяет подкласс `Exception` с именем `MyException`. Этот подкласс весьма прост — он имеет только конструктор и перегруженный метод `toString()`, который отображает значение исключения. Класс `ExceptionDemo` определяет метод, названный `compute()`, который выбрасывает объект `MyException`. Исключение выбрасывается, когда целый параметр метода `compute()` больше 10. Метод `main()` устанавливает обработчик исключений для `MyException`, а затем вызывает `compute()` с допустимым значением параметра (меньшим 10) и недопустимым, чтобы показать оба варианта работы программы. Вот результат:

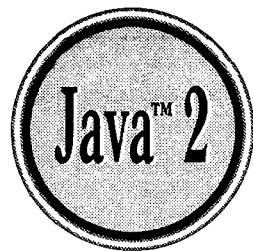
```
Вызван compute(1)
Нормальный выход
Вызван compute(20)
Выброшено MyException[20]
```

## Использование исключений

Обработка исключений обеспечивает мощный механизм управления комплексными программами, обладающими множеством динамических характеристик времени выполнения. Важно представлять механизм `try-throw-catch`, как достаточно ясный способ обработки ошибок и необычных граничных условий в логике программы. Как и большинство программистов, вы, вероятно, привыкли возвращать код ошибки, когда метод терпит неудачу. Если вы программируете на языке Java, то нужно отказаться от этой привычки. Когда произойдет отказ метода, пусть он сам выбросит исключение. Это более ясный способ обработки режимов отказа.

И последнее замечание: операции обработки исключений Java не нужно рассматривать как общий механизм для нелокального ветвления. Если вы так сделаете, это только задутает ваш код и затруднит его поддержку.

# ГЛАВА 11



## Многопоточное программирование

В отличие от большинства других машинных языков, Java обеспечивает встроенную поддержку для *многопоточного программирования*. Многопоточная программа содержит две и более частей, которые могут выполняться одновременно, конкурируя друг с другом. Каждая часть такой программы называется *потоком*, а каждый поток определяет отдельный путь выполнения (в последовательности операторов программы). Таким образом, многопоточность — это специализированная форма многозадачности.

Вы, конечно, почти знакомы с многозадачностью, потому что она поддерживается фактически всеми современными операционными системами. Однако существуют две различные формы многозадачности. Одна основана на процессах, а другая — на потоках. Важно понять различие между ними. Большинству читателей ближе знакома форма многозадачности, основанная на процессах. *Процесс* — это, по-существу, выполняющаяся программа. Таким образом, основанная на процессах многозадачность — это свойство, которое позволяет вашему компьютеру выполнять несколько программ одновременно. Такая многозадачность дает, например, возможность выполнять компилятор Java одновременно с использованием текстового редактора. В многозадачности, основанной на процессах, самой мелкой единицей диспетчеризации планировщика является *программа*.

В многозадачной среде, основанной на потоках, самой мелкой единицей диспетчеризации является *поток*. Это означает, что отдельная программа может выполнять несколько *задач* одновременно. Например, текстовый редактор может форматировать текст одновременно с печатью документа, поскольку эти два действия выполняются двумя отдельными потоками. Таким образом, многозадачность, основанная на процессах, имеет дело со всей картиной, а поточная многозадачность обрабатывает детали.

Многозадачные потоки требуют меньших накладных расходов по сравнению с многозадачными процессами. Процессы — это тяжеловесные задачи, которым требуются отдельные адресные пространства. Связи между процессами

ограничены и стоят не дешево. Переключение контекста от одного процесса к другому также весьма дорогостоящая задача. С другой стороны, потоки достаточно легковесны. Они совместно используют одно и то же адресное пространство и кооперативно оперируют с одним и тем же тяжеловесным процессом, межпоточные связи недороги, а переключение контекста от одного потока к другому имеет низкую стоимость. Хотя Java-программы и используют многозадачное окружение, основанное на процессах, многозадачность такого рода не находится под управлением языка. Однако многопоточной многозадачностью Java все-таки управляет.

Многопоточность дает возможность писать очень эффективные программы, которые максимально используют CPU, потому что время его простоя можно свести к минимуму. Это особенно важно для интерактивной сетевой среды, в которой работает Java, потому что время простоя является общим. Например, скорость передачи данных по сети намного меньше, чем скорость, с которой компьютер может их обрабатывать. Даже локальные ресурсы файловой системы читаются и записываются намного медленнее, чем они могут быть обработаны CPU. И, конечно, ввод пользователя намного медленнее, чем работа компьютера. В традиционной однопоточной среде ваша программа должна ждать окончания каждой своей задачи, прежде чем она сможет перейти к следующей (даже при том, что большую часть времени CPU пристаивает). Многопоточность позволяет получить доступ к этому времени простоя и лучше его использовать.

Если вы программировали для таких операционных систем, как Windows 98 или Windows NT, то вы уже знакомы с многопоточным программированием. Однако факт, что Java управляет потоками, делает многопоточность особенно удобной, потому что многие детали для вас уже обработаны.

## Поточная модель Java

Исполнительная система Java во многом зависит от потоков, и все библиотеки классов разработаны с учетом многопоточности. Java использует потоки для обеспечения асинхронности во всей среде. Это помогает уменьшить неэффективность, предотвращая затраты циклов CPU.

Ценность многопоточной среды лучше понимается по контрасту с ее аналогом. Однопоточные системы используют подход, называемый *циклом событий с опросом* (event loop with polling). В этой модели, единственный поток управления выполняется в бесконечном цикле, опрашивая единственную очередь событий, чтобы решить, что делать дальше. Как только этот механизм опроса возвращает, скажем, сигнал готовности сетевого файла готов для чтения, цикл событий передает управление соответствующему обработчику событий. До возврата из этого обработчика в системе ничего больше случиться не может. Это потеря времени CPU может также привести к доминированию одной части программы над системой и задержке обработ-

ки других событий. И вообще, в однопоточной среде, когда поток блокируется (т. е. приостанавливает выполнение), потому что он ожидает некоторый ресурс, останавливается выполнение всей программы.

Выгода от многопоточности Java заключается в том, что устраняется механизм "главный цикл/опрос". Один поток может делать паузу без остановки других частей программы. Например, время простоя, образующееся, когда поток читает данные из сети или ждет ввод пользователя, может использоваться в другом месте. Многопоточность позволяет циклам мультилипликации бездействовать в течение секунды между каждым фреймом без принуждения делать паузу целой системы. Когда потоки блокируются в Java-программе, паузу "держит" только один поток — тот, который блокирован. Остальные продолжают выполняться.

Потоки существуют в нескольких состояниях. Поток может быть в состоянии *выполнения*. Может находиться в состоянии *готовности к выполнению*, как только он получит время CPU. Выполняющийся поток может быть *приостановлен*, что временно притормаживает его действие. Затем приостановленный поток может быть *продолжен* (возобновлен) с того места, где он был остановлен. Поток может быть *блокирован* в ожидании ресурса. В любой момент выполнение потока может быть завершено, что немедленно останавливает его выполнение. После завершения поток не может быть продолжен.

## Приоритеты потоков

Java назначает каждому потоку приоритет, который определяет порядок обработки этого потока относительно других потоков. Приоритеты потоков — это целые числа, которые определяют относительный приоритет одного потока над другим. Как абсолютное значение, приоритет бессмысленен. Высокоприоритетный поток не выполняется быстрее, чем низкоприоритетный. Приоритет потока используется для того, чтобы решить, когда переключаться от одного выполняющегося потока к следующему. Это называется *переключением контекста*. Правила, которые определяют, когда переключение контекста имеет место, просты.

- Поток может *добровольно отказаться от управления*. Это делается явно, переходом в режим ожидания или блокированием на ожидающем вводе/выводе. В этом сценарии просматриваются все потоки, и CPU передается самому высокоприоритетному потоку, который готов к выполнению.
- Поток может быть *приостановлен более приоритетным потоком*. В этом случае занимающий процессор низкоприоритетный поток временно останавливается (независимо от того, что он делает) потоком с более высоким приоритетом. Иначе говоря, как только поток с более высоким приоритетом хочет выполниться, он сразу же это делает. Данный механизм называется *упреждающей многозадачностью* (preemptive multitasking).

В случаях, где два потока с одинаковым приоритетом конкурируют за циклы CPU, ситуация немного сложнее. Для операционных систем типа Windows 98, потоки равного приоритета квантуются (во времени) автоматически, циклическим способом. Для других типов операционных систем, типа Solaris 2.x, потоки равного приоритета должны добровольно передавать управление другим потокам. Если они этого не делают, другие потоки не будут выполняться.

### Предупреждение

Могут возникнуть проблемы из-за различий в способах, с помощью которых операционные системы переключают контексты потоков равного приоритета.

## Синхронизация

Поскольку многопоточность обеспечивает *асинхронное* поведение ваших программ, должен существовать способ добиться *синхронности*, когда в этом возникает необходимость. Например, если вы хотите, чтобы два потока взаимодействовали и совместно использовали сложную структуру данных типа связного списка, нужно каким-то образом гарантировать отсутствие между ними конфликтов. Вы должны удержать один поток от записи данных, пока другой поток находится в процессе их чтения. Для этой цели Java эксплуатирует старую, но изящную модель синхронизации процессов — монитор. *Монитор* — это механизм управления связью между процессами, первоначально определенный Хором (Hoare, C.A.R). Вы можете представлять монитор, как очень маленький блок, который содержит только один поток. Как только поток входит в монитор, все другие потоки должны ждать, пока данный не выйдет из монитора. Таким образом, монитор можно использовать для защиты совместно используемого (разделяемого) ресурса от управления несколькими потоками одновременно.

Большинство многопоточных систем создает мониторы как объекты, которые ваша программа должна явно получить и использовать. Java обеспечивает более простое решение. В Java-системе нет класса с именем *Monitor*. Вместо этого, каждый объект имеет свой собственный неявный монитор, который вводится автоматически при вызове одного из методов объекта. Как только поток оказывается внутри синхронизированного метода, никакой другой поток не может вызывать иной синхронизированный метод того же объекта. Это дает возможность писать очень ясный и краткий многопоточный код, т. к. поддержка синхронизации встроена в язык.

## Передача сообщений

После того как вы разделите свою программу на отдельные потоки, нужно определить, как они будут взаимодействовать друг с другом. При програм-

мировании на большинстве других языков требуется, в зависимости от операционной системы, установить связь между потоками. Это, конечно, увеличивает издержки (как в программировании, так и в расходовании ресурсов). В противоположность этому, Java обеспечивает ясный, дешевый путь для взаимного общения двух (или нескольких) потоков через вызовы предопределенных методов, которыми обладают все объекты. Система передачи сообщений Java позволяет потоку войти в *синхронизированный* метод на объекте и затем ждать там, пока некоторый другой поток явно не уведомит его о выходе.

## Класс *Thread* и интерфейс *Runnable*

Многопоточная система Java построена на классе *Thread*, его методах и связанным с ним интерфейсе *Runnable*. *Thread* инкапсулирует поток выполнения. Так как вы не можете непосредственно обращаться к внутреннему состоянию потока выполнения, то будете иметь с ним дело через его полномочного представителя — экземпляр (объект) класса *Thread*, который его породил. Чтобы создать новый поток, ваша программа должна будет или расширять класс *Thread* или реализовывать интерфейс *Runnable*.

Класс *Thread* определяет несколько методов, которые помогают управлять потоками. Табл. 11.1 содержит описание методов, которые будут использоваться в этой главе.

**Таблица 11.1. Некоторые методы класса *Thread***

Метод	Значение
<i>GetName()</i>	Получить имя потока
<i>GetPriority()</i>	Получить приоритет потока
<i>IsAlive()</i>	Определить, выполняется ли еще поток
<i>Join()</i>	Ждать завершения потока
<i>Run()</i>	Указать точку входа в поток
<i>Sleep()</i>	Приостановить поток на определенный период времени
<i>Start()</i>	Запустить поток с помощью вызова его метода <i>run()</i>

До настоящего времени все примеры в этой книге использовали одиночный поток выполнения. В остальной части данной главы объясняется, как использовать *Thread* и *Runnable* для создания и управления множественными потоками, начиная с того, которым обладают все Java-программы — главного потока.

## ГлавНЫЙ ПОТОК

Когда Java-программа запускается, один поток начинает выполняться немедленно. Он обычно называется *главным потоком* вашей программы, потому что выполняется в начале программы. Главный поток важен по двум причинам:

- Это поток, из которого будут порождены все другие "дочерние" потоки.
- Это должен быть последний поток, в котором заканчивается выполнение.

Когда главный поток останавливается, программа завершается.

Хотя главный поток создается автоматически после запуска программы, он может управляться через Thread-объект. Для организации управления нужно получить ссылку на него, вызывая метод `currentThread()`, который является

`public static` членом класса `Thread`. Вот его общая форма:

```
static Thread currentThread()
```

Этот метод возвращает ссылку на поток, в котором он вызывается. Как только вы получаете ссылку на главный поток, то можете управлять им точно так же, как любым другим потоком. Начнем со следующего примера:

```
// Управление главным потоком.
class CurrentThreadDemo {
    public static void main(String args[]) {
        Thread t = Thread.currentThread();
        System.out.println("Текущий поток: " + t);

        // изменить имя потока
        t.setName("My Thread");
        System.out.println("После изменения имени: " + t);

        try {
            for(int n = 5; n > 0; n--) {
                System.out.println(n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Главный поток завершен");
        }
    }
}
```

В этой программе ссылка к текущему потоку (главному потоку в этом случае) получена с помощью вызова `currentThread()` и сохранена в локальной переменной `t`. Затем, программа отображает информацию о потоке, вызывая метод `setName()`, чтобы поменять внутреннее имя потока, и вновь выво-

дит информацию о потоке. Далее, запускается обратный (от 5) цикл `for`, приостанавливающий поток на одну секунду на каждом шаге. Пауза выполняется методом `sleep()`. Аргумент `sleep()` определяет время задержки в миллисекундах. Обратите внимание на обрамляющий этот цикл блок `try/catch`. Метод `sleep()` класса `Thread` мог бы выбросить исключение типа `InterruptedException`. Подобная ситуация сложилась, если бы некоторый другой поток хотел прервать это ожидание. Данный пример только печатает сообщение, если он получает прерывание. В реальной программе вам нужно было бы обработать его по-другому. Вывод, сгенерированный этой программой:

```
Текущий поток: Thread[main, 5, main]
После изменения имени: Thread[My Thread, 5, main]
```

```
5
4
3
2
1
```

Обратите внимание на выводы, использующие `t` в качестве аргумента `println()`. Они отображают (по порядку): имя потока, его приоритет и имя его группы. По умолчанию имя главного потока — `main`. Его приоритет равен 5, что тоже является значением по умолчанию и последний идентификатор `main` — имя группы потоков, которой принадлежит данный поток. *Группа потоков* — структура данных, контролирующая состояние совокупности потоков в целом. Этот процесс управляет специальной исполнительной (run-time) средой и подробно здесь не обсуждается. После изменения имени потока переменная `t` снова выводится на экран. На сей раз отображается новое имя потока.

Рассмотрим подробнее методы класса `Thread`, которые используются в программе. Метод `sleep()` заставляет поток, из которого он вызывается, приостановить выполнение на указанное (в аргументе вызова) число миллисекунд. Его общая форма имеет вид:

```
static void sleep(long milliseconds) throws InterruptedException
```

Число миллисекунд интервала приостановки определяется в параметре `milliseconds`. Кроме того, данный метод может выбросить исключение типа `InterruptedException`.

Метод `sleep()` имеет вторую форму, которая позволяет определять период приостановки в долях миллисекунд и наносекунд:

```
static void sleep(long milliseconds, int nanoseconds)
    throws InterruptedException
```

Эта вторая форма полезна только в средах, где допускаются короткие периоды времени, измеряемые в наносекундах.

Как показывает предыдущая программа, используя метод `setName()`, можно установить (записать из программы) новое имя потока. Можно также получить (прочитать) существующее имя потока, вызывая метод `getName()` (однако обратите внимание, что эта процедура не показана в программе). Оба метода являются членами класса `Thread` и объявляются в таких формах:

```
final void setName(String threadName)
final String getName()
```

где `threadName` определяет имя потока.

## Создание потока

В самом общем случае для создания потока строят объект типа `Thread`. В Java это можно выполнить двумя способами:

- реализовать интерфейс `Runnable`;
- расширить класс `Thread`, определив его подкласс.

В следующих двух разделах рассматривается каждый из указанных способов.

## Реализация интерфейса `Runnable`

Самый простой способ создания потока заключается в определении класса, который реализует интерфейс `Runnable`. В `Runnable` определен некоторый абстрактный (без тела) модуль выполняемого кода. Создавать поток можно на любом объекте, который реализует интерфейс `Runnable`. Для реализации `Runnable` в классе нужно определить только один метод с именем `run()`. Форма его объявления:

```
public void run()
```

Внутри `run()` нужно определить код, образующий новый поток. Важно понять, что `run()` может вызывать другие методы, использовать другие классы и объявлять переменные точно так же, как это делает главный (`main`) поток. Единственное различие состоит в том, что `run()` устанавливает в данной программе точку входа для запуска другого, конкурирующего потока выполнения. Этот поток завершит свое выполнение при возврате из `run()`.

После создания класса, который реализует `Runnable`, нужно организовать объект типа `Thread` внутри этого класса. В классе `Thread` определено несколько конструкторов. Мы будем использовать конструктор следующей формы:

```
Thread(Runnable threadOb, String threadName)
```

Здесь `threadOb` — экземпляр (объект) класса, который реализует интерфейс `Runnable`. Он определяет, где начнется выполнение нового потока. Имя нового потока определяет параметр `threadName`.

После того как новый поток создан, он не начнет выполняться, пока вы не вызываете его методом `start()`, который объявлен в `Thread`. В действительности `start()` выполняет вызов `run()`. Формат метода `start()`:

```
void start()
```

Рассмотрим пример, который создает новый поток и начинает его выполнение:

```
// Создание второго потока.
class NewThread implements Runnable {
    Thread t;

    NewThread() {
        // Создать новый, второй поток.
        t = new Thread(this, "Demo Thread");
        System.out.println("Дочерний поток: " + t);
        t.start(); // стартовать поток
    }

    // Это точка входа во второй поток.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Дочерний Поток: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Прерывание дочернего потока.");
        }
        System.out.println("Завершение дочернего потока.");
    }
}

class ThreadDemo {
    public static void main(String args[]) {
        new NewThread(); // создать новый поток

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Главный поток: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Прерывание главного потока.");
        }
        System.out.println("Завершение главного потока.");
    }
}
```

Внутри конструктора NewThread, новый Thread объект создается следующим оператором:

```
t = new Thread(this, "Demo Thread");
```

Передача this в качестве первого параметра указывает, что вы хотите, чтобы новый поток на this-объекте вызвал метод run(). Затем вызывается метод start(), который начинает выполнение потока в методе run(). Это приводит к запуску цикла for дочернего потока. После вызова start() конструктор NewThread возвращается к main(). Когда главный поток возобновляет выполнение, он входит в свой for-цикл. Оба потока продолжают выполнение, используя CPU совместно, до конца своих циклов. Вывод этой программы следующий:

```
Дочерний поток: Thread[Demo Thread,5,main]
```

```
Главный поток: 5
```

```
Дочерний Поток 5
```

```
Дочерний Поток 4
```

```
Главный поток: 4
```

```
Дочерний Поток 3
```

```
Дочерний Поток 2
```

```
Главный поток: 3
```

```
Дочерний Поток 1
```

```
Завершение дочернего потока.
```

```
Главный поток: 2
```

```
Главный поток: 1
```

```
Завершение главного потока.
```

Как уже говорилось, в многопоточной программе главный поток должен заканчивать выполнение последним. Если он заканчивается прежде, чем завершится дочерний поток, то исполнительная система Java может "зависнуть". Предыдущая программа гарантирует, что главный поток заканчивается последним, потому что он бездействует 1000 миллисекунд между итерациями цикла, а дочерний поток — только 500 миллисекунд. Это заставляет дочерний поток завершиться раньше главного. Скоро вы увидите лучший способ предоставления гарантии более позднего завершения главного потока.

## **Расширение Thread**

Для генерации потока вторым способом необходимо создать новый класс, который расширяет Thread, а затем — экземпляр этого класса. Расширяющий класс должен переопределять метод run(), который является точкой входа для нового потока. Он должен также вызывать start(), чтобы начать выполнение нового потока. Ниже приведена предыдущая программа, переписанная так, чтобы расширить Thread:

```
// Создает второй поток расширением класса Thread.
class NewThread extends Thread {

    NewThread() {
        // Создать новый, второй поток
        super("Demo Thread");
        System.out.println("Дочерний поток: " + this);
        start(); // стартовать поток
    }

    // Это точка входа для второго потока,
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Дочерний поток: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Прерывание дочернего потока.");
        }
        System.out.println("Завершение дочернего потока.");
    }
}

class ExtendThread {
    public static void main(String args[]) {
        new NewThread(); // создать новый поток

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Главный поток: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Прерывание главного потока.");
        }
        System.out.println("Завершение главного потока.");
    }
}
```

Эта программа генерирует тот же вывод, что и предшествующая версия. Нетрудно заметить, что дочерний поток создается с помощью объекта NewThread, который получен из Thread.

Обратите внимание на обращение к методу super() внутри NewThread. Оно вызывает следующую форму конструктора Thread:

```
public Thread(String threadName)
```

где **threadName** определяет имя потока.

## Выбор подхода

Вы могли бы задаться вопросом, почему Java имеет два способа создания дочерних потоков, и какой подход лучше? Класс `Thread` определяет несколько методов, которые могут быть переопределены производным классом. Из них только один должен быть переопределен обязательно — метод `run()`. Это, конечно, тот же самый метод, который требовался для реализации интерфейса `Runnable`. Многие Java-программисты чувствуют, что классы должны расширяться только тогда, когда они каким-то образом улучшаются или изменяются. Так, если вы не будете переопределять никакого другого метода класса `Thread`, то, вероятно, лучше всего просто реализовать интерфейс `Runnable` напрямую. Эта, конечно, рекомендация только для вас. Мы же везде в остальной части данной главы будем создавать потоки, используя классы, которые реализуют интерфейс `Runnable`.

## Создание множественных потоков

До сих пор вы использовали только два потока — главный и один дочерний. Однако ваша программа может порождать столько потоков, сколько необходимо. Например, следующая программа создает три дочерних потока:

```
// Создание множественных потоков.
class NewThread implements Runnable {
    String name;                                // имя потока
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("Новый поток: " + t);
        t.start();                                // старт потока
    }

    // Это точка входа для потока.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " прерван.");
        }
        System.out.println(name + " завершен.");
    }
}
```

```
class MultiThreadDemo {  
    public static void main(String args[]) {  
        new NewThread("Первый");           // старт потока  
        new NewThread("Второй");  
        new NewThread("Третий");  
  
        try {  
            // ждать завершения других потоков  
            Thread.sleep(10000);  
        } catch (InterruptedException e) {  
            System.out.println("Прерывание главного потока.");  
        }  
  
        System.out.println("Завершение главного потока.");  
    }  
}
```

Вывод этой программы:

```
Новый поток: Thread[One,5,main]  
Новый поток: Thread[Two,5,main]  
Новый поток: Thread[Three,5,main]  
Первый: 5  
Второй: 5  
Третий: 5  
Первый: 4  
Второй: 4  
Третий: 4  
Первый: 3  
Третий: 3  
Второй: 3  
Первый: 2  
Третий: 2  
Второй: 2  
Первый: 1  
Третий: 1  
Второй: 1  
Первый завершен.  
Второй завершен.  
Третий завершен.  
Завершение главного потока.
```

Как видите, после запуска все три дочерних потока совместно используют CPU. Обратите внимание на вызов `sleep(10000)` в `main()`. Он заставляет главный поток бездействовать в течение десяти секунд и гарантирует, что тот закончится последним.

## Использование методов *isAlive()* и *join()*

Итак, главный поток должен быть последним завершающимся потоком. В предшествующих примерах это выполнялось вызовом метода *sleep()* в *main()* с достаточно длинной задержкой, чтобы гарантировать, что все дочерние потоки закончатся до завершения главного потока. Однако такое решение едва ли удовлетворительно. Кроме того, возникает вопрос: как один поток может знать, закончился ли другой? К счастью, *Thread* обеспечивает средства, с помощью которых вы можете ответить на него.

Существуют два способа определения, закончился ли поток. Один из них позволяет вызывать метод *isAlive()* на потоке. Этот метод определен в *Thread* и его общая форма выглядит так:

```
final boolean isAlive()
```

Метод *isAlive()* возвращает *true*, если поток, на котором он вызывается — все еще выполняется. В противном случае возвращается *false*.

В то время как *isAlive()* полезен только иногда, чаще для ожидания завершения потока вызывается метод *join()* следующего формата:

```
final void join() throws InterruptedException
```

Этот метод ждет завершения потока, на котором он вызван. Его имя происходит из концепции перевода потока в состояние ожидания, пока указанный поток не присоединит его. Дополнительные формы *join()* позволяют определять максимальное время ожидания завершения указанного потока.

Ниже показана улучшенная версия предшествующего примера, использующая *join()* и гарантирующая, что главный поток останавливается последним. Она также демонстрирует метод *isAlive()*.

```
// Использование join() для ожидания окончания потока.
class NewThread implements Runnable {
    String name; // имя потока
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("Новый поток: " + t);
        t.start(); // старт потока
    }

    // Это точка входа потока.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
        }
    }
}
```

```
        Thread.sleep(1000);
    }
} catch (InterruptedException e) {
    System.out.println(name + " прерван.");
}
System.out.println(name + " завершен.");
}

}

class DemoJoin {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("Первый");
        NewThread ob2 = new NewThread("Второй");
        NewThread ob3 = new NewThread("Третий");

        System.out.println("Первый поток выполняется: " + ob1.t.isAlive());
        System.out.println("Второй поток выполняется: " + ob2.t.isAlive());
        System.out.println("Третий поток выполняется: " + ob3.t.isAlive());
        // ждать завершения потоков
        try {
            System.out.println("Ждите завершения потоков.");
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Прерывание главного потока");
        }
        System.out.println("Первый поток выполняется: " + ob1.t.isAlive());
        System.out.println("Второй поток выполняется: " + ob2.t.isAlive());
        System.out.println("Третий поток выполняется: " + ob3.t.isAlive());
        System.out.println("Завершение главного потока.");
    }
}
```

Пример вывода этой программы:

```
Новый поток: Thread[One,5,main]
Новый поток: Thread[Two,5,main]
Новый поток: Thread[Three,5,main]
Первый поток выполняется: true
Второй поток выполняется: true
Третий поток выполняется: true
Ждите завершения потоков.
Первый: 5
Второй: 5
Третий: 5
```

```
Первый: 4
Второй: 4
Третий: 4
Первый: 3
Второй: 3
Третий: 3
Первый: 2
Второй: 2
Третий: 2
Первый: 1
Второй: 1
Третий: 1
Второй завершен.
Третий завершен.
Первый завершен.
Первый поток выполняется: false
Второй поток выполняется: false
Третий поток выполняется: false
Завершение главного потока.
```

Нетрудно видеть, что после возврата из `join()` потоки прекращают выполняться.

## Приоритеты потоков

Планировщик потоков использует их приоритеты для принятия решений о том, когда нужно разрешать выполнение тому или иному потоку. Теоретически высокоприоритетные потоки получают больше времени CPU, чем низкоприоритетные. На практике, однако, количество времени CPU, которое поток получает, часто зависит от нескольких факторов помимо его приоритета. (Например, относительная доступность времени CPU может зависеть от того, как операционная система реализует многозадачный режим.) Высокоприоритетный поток может также упреждать низкоприоритетный (т. е. перехватывать у него управление процессором). Скажем, когда низкоприоритетный поток выполняется, а высокоприоритетный поток возобновляется (от ожидания на вводе/выводе, к примеру), высокоприоритетный поток будет упреждать низкоприоритетный.

Теоретически, потоки равного приоритета должны получить равный доступ к CPU. Но вы должны быть внимательны. Помните, Java спроектирован для работы в широком диапазоне сред. Некоторые из этих сред реализуют многозадачный режим существенно иначе, чем другие. Для безопасности потоки, которые совместно используют один и тот же приоритет, должны время от времени уступать друг другу управление. Это гарантирует, что все потоки имеют шанс выполниться под неприоритетной операционной системой. Практически, даже в неприоритетных средах, большинство потоков все еще

получают шанс выполнятся, потому что большинство из них неизбежно сталкивается с некоторыми блокирующими ситуациями, типа ожидания ввода/вывода. Когда это случается, блокированный поток приостанавливается, а другие могут продолжаться. Но, если вам нужно плавное многопоточное выполнение, лучше не полагаться на подобную ситуацию. К тому же, некоторые типы задач интенсивно используют CPU. Такие потоки доминируют над CPU и, чтобы иные потоки тоже могли выполнятся, необходимо, чтобы эти первые время от времени уступали управление.

Для установки приоритета потока используйте метод `setPriority()`, который является членом класса `Thread`. Вот его общая форма:

```
final void setPriority(int level)
```

где `level` определяет новую установку приоритета для вызывающего потока. Значение параметра `level` должно быть в пределах диапазона `MIN_PRIORITY` и `MAX_PRIORITY`. В настоящее время эти значения равны 1 и 10, соответственно. Чтобы вернуть потоку приоритет, заданный по умолчанию, определите `NORM_PRIORITY`, который в настоящее время равен 5. Эти приоритеты определены в `Thread` как `final`-переменные.

Вы можете получить текущую установку приоритета, вызывая метод `getPriority()` класса `Thread`, чей формат имеет следующий вид:

```
final int getPriority()
```

Реализации Java могут иметь радикально различное поведение, когда они переходят к планированию. Версии Windows 95/98/NT работают примерно так, как вы и ожидаете. Однако другие версии могут работать совершенно иначе. Большинство несоответствий возникает при работе с потоками, которые полагаются на приоритетное поведение, вместо совместного отказа от времени CPU. Самый безопасный способ получить предсказуемое многоплатформенное поведение в Java состоит в том, чтобы использовать потоки, добровольно уступающие управление CPU.

Следующий пример демонстрирует два потока с различными приоритетами, которые не выполняются на приоритетной платформе таким же образом, как они выполняются на неприоритетной платформе. Один поток устанавливает приоритет на два уровня выше нормального, как определено в `Thread.NORM_PRIORITY`, а другой — на два уровня ниже. Потоки запускаются и допускаются к выполнению на десять секунд. Каждый поток осуществляет цикл, отчитывающий несколько итераций. После десяти секунд главный поток останавливает оба потока. Затем на экране отображается количество итераций цикла, сделанных каждым потоком.

```
// Демонстрирует приоритеты потоков.  
class clicker implements Runnable {  
    int click = 0;
```

```
Thread t;
private volatile boolean running = true;

public clicker(int p) {
    t = new Thread(this);
    t.setPriority(p);
}

public void run() {
    while (running) {
        click++;
    }
}

public void stop() {
    running = false;
}

public void start() {
    t.start();
}
}

class HiLoPri {
    public static void main(String args[]) {
        Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
        clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
        clicker lo = new clicker(Thread.NORM_PRIORITY - 2);

        lo.start();
        hi.start();
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Прерывание главного потока.");
        }

        lo.stop();
        hi.stop();

        // Ждать завершения дочерних потоков.
        try {
            hi.t.join();
            lo.t.join();
        } catch (InterruptedException e) {
            System.out.println("InterruptedException caught");
        }
    }
}
```

```
        System.out.println("Поток с низким приоритетом: " + lo.click);
        System.out.println("Поток с высоким приоритетом: " + hi.click);
    }
}
```

Ниже показан вывод этой программы, выполнявшейся в среде Windows 98. Он указывает, что потоки делали контекстное переключение даже без добровольной уступки CPU или блокировки для ввода/вывода. Высокоприоритетный поток получил приблизительно 90 процентов времени CPU.

```
Поток с низким приоритетом: 4408112
Поток с высоким приоритетом: 589626904
```

Конечно, точный вывод этой программы зависит от быстродействия вашего CPU и числа других задач, работающих в системе. Когда эта же программа будет выполняться под неприоритетной системой, результаты окажутся другими.

Еще одно замечание относительно предыдущей программы. Обратите внимание, что переменной `running` (в ее объявлении) предшествует ключевое слово `volatile`. Хотя `volatile` рассматривается более тщательно в следующей главе, здесь оно используется для предоставления гарантии, что значение `running` просматривается на каждой итерации следующего цикла:

```
while (running) {
    click++;
}
```

Без использования `volatile` Java имеет возможность оптимизировать цикл таким способом, что значение `running` будет содержаться в регистре CPU и не обязательно повторно просматриваться в каждой итерации. Использование `volatile` предотвращает эту оптимизацию, сообщая Java, что `running` может изменяться способами, напрямую не очевидными для данного кода.

## Синхронизация

Когда несколько потоков нуждаются в доступе к разделяемому ресурсу, им необходим некоторый способ гарантии того, что ресурс будет использоваться одновременно только одним потоком. Процесс, с помощью которого это достигается, называется *синхронизацией*. Как вы увидите, Java обеспечивает уникальную поддержку синхронизации — на уровне языка.

Ключом к синхронизации является концепция монитора (также называемая *семафором*). Монитор — это объект, который используется для взаимоисключающей блокировки (*mutually exclusive lock*), или *mutex*. Только один поток может иметь *собственный* монитор в заданный момент. Когда поток получает блокировку, говорят, что он *вошел* в монитор. Все другие потоки,

пытающиеся вводить блокированный монитор, будут приостановлены, пока первый не вышел из монитора. Говорят, что другие потоки *ожидают* монитор. При желании поток, владеющий монитором, может повторно вводить тот же самый монитор.

Если вы работали с синхронизацией в других языках, таких как С или С++, то знаете, как сложно ее использовать. Это происходит потому, что большинство языков сами по себе не поддерживают синхронизацию. Вместо этого, чтобы синхронизировать потоки, ваши программы должны применять примитивы операционной системы. К счастью, из-за того, что Java осуществляет синхронизацию через элементы языка, большинство сложностей, связанных с синхронизацией, было устранено.

Синхронизировать код можно двумя способами. Оба используют ключевое слово `synchronized` и рассмотрены ниже.

## Использование синхронизированных методов

Синхронизация в Java проста потому, что каждый объект имеет свой собственный неявный связанный с ним монитор. Чтобы ввести монитор объекта, просто вызывают метод, который был модифицирован ключевым словом `synchronized`. Пока поток находится внутри синхронизированного метода, все другие потоки, пытающиеся вызвать его (или любой другой синхронизированный метод) на том же самом экземпляре, должны ждать. Чтобы выйти из монитора и оставить управление объектом следующему ожидающему потоку, владелец монитора просто возвращается из синхронизированного метода.

Чтобы понять потребность в синхронизации, начнем с простого примера, который ее не использует, хотя должен. Следующая программа имеет три простых класса. Первый, `Callme`, содержит одиночный метод с именем `call()`. Метод `call()` имеет параметр типа `String` с именем `msg`. Этот метод пытается печатать строку `msg` внутри квадратных скобок. Интересно обратить внимание на то, что после того, как `call()` печатает открывающую скобку и строку `msg`, он вызывает `Thread.sleep(1000)`, приостанавливающий текущий поток на одну секунду.

Конструктор следующего класса (с именем `Caller`) имеет в качестве параметров ссылки на экземпляры класса `Callme` и `String`, которые сохраняются в экземплярных переменных `target` и `msg`, соответственно. Кроме того, конструктор создает новый поток, вызывающий метод `run()` его объекта. Поток стартует немедленно. Метод `run()` класса `Caller` вызывает метод `call()` на экземпляре `target` класса `Callme`, передавая ему (в качестве аргумента) строку `msg`. Затем, запускается класс `Synch`, создавая одиночный `Callme`-экземпляр и три `Caller`-экземпляра (каждый — с уникальной строкой сообщения). Каждому `Caller`-объекту (`obj1`, `obj2` и `obj3`) передается один и тот же экземпляр `Callme` (`target`).

```
// Эта программа не синхронизирована.
class Callme {
    void call(String msg) {
        System.out.print("[ " + msg);
        try {
            Thread.sleep(1000);
        } catch(InterruptedException e) {
            System.out.println("Прерывание");
        }
        System.out.println("] ");
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;

    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }

    public void run() {
        target.call(msg);
    }
}

class Synch {
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Привет");
        Caller ob2 = new Caller(target, "Синхронизированный");
        Caller ob3 = new Caller(target, "Мир");

        // ждать завершения потоков
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch(InterruptedException e) {
            System.out.println("Прерывание");
        }
    }
}
```

Вывод этой программы:

```
Привет [Синхронизированный [Мир]
```

```
] 
```

```
] 
```

Вызывая `sleep()`, метод `call()` разрешает переключать выполнение на другой поток. Это приводит к выводу перепутанных строк сообщений. В данной программе нет возможности предохранить три потока от вызова одного и того же метода на одном и том же объекте в одно и то же время. Это известно как *состояние состязаний* (гонок), потому что три потока состязаются друг с другом, чтобы завершить метод. Представленный пример использовал метод `sleep()`, чтобы сделать эффекты повторяемыми и очевидными. В большинстве ситуаций состояние состязания более тонко и менее предсказуемо, потому что вы не можете точно знать, когда произойдет переключение контекста. Это может заставить программу некоторое время выполнять правильно, а затем — не правильно.

Чтобы стабилизировать предшествующую программу, нужно *сериализовать* (преобразовать в последовательную форму) доступ к методу `call()`. То есть требуется организовать доступ к нему только одного потока одновременно. Для этого нужно просто в начало определения метода `call()` вставить ключевое слово `synchronized`, например, как в следующем фрагменте:

```
class Callme {  
    synchronized void call(String msg) {  
        ...  
    }  
}
```

Это предохраняет потоки от ввода `call()`, пока его использует другой поток. После добавления `synchronized` вывод программы становится таким:

```
[Привет]  
[Синхронизированный]  
[Мир]
```

Всякий раз, когда имеется метод (или группа методов), который управляет внутренним состоянием объекта в многопоточной ситуации, нужно использовать ключевое слово `synchronized` для предотвращения состязаний. Помните, как только поток вызывает синхронизированный метод на некотором экземпляре, никакой другой поток не может вызвать никакой другой синхронизированный метод на том же самом экземпляре. Однако несинхронизированные методы на этом экземпляре остаются вызываемыми.

## Оператор `synchronized`

Хотя определения синхронизированных методов внутри классов — это простые и эффективные средства достижения синхронизации, они не будут работать во всех случаях. Чтобы понять, почему так происходит, рассмотрим

следующую ситуацию. Вообразите, что вы хотите синхронизировать доступ к объектам класса, который не был разработан для многопоточного доступа. То есть класс не использует синхронизированные методы. Кроме того, этот класс был создан не вами, а третьим лицом, и вы не имеете доступа к исходному коду. Таким образом, вы не можете добавлять спецификатор `synchronized` к соответствующим методам в классе. Как можно синхронизировать доступ к объекту этого класса? К счастью, решение данной проблемы весьма просто. Нужно поместить вызовы методов, определенных этим классом внутрь синхронизированного блока. Вот общая форма оператора `synchronized`:

```
synchronized(object) {
    // операторы для синхронизации
}
```

где `object` — ссылка на объект, который нужно синхронизировать. Если нужно синхронизировать одиночный оператор, то фигурные скобки можно опустить. Блок гарантирует, что вызов метода, который является членом объекта `object`, происходит только после того, как текущий поток успешно ввел монитор объекта.

Вот альтернативная версия предыдущего примера, использующая синхронизированный блок в методе `run()`:

```
// Эта программа использует синхронизированный блок.
class Callme {
    void call(String msg) {
        System.out.print("[ " + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Прерывание");
        }
        System.out.println("] ");
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;

    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }
}
```

```

// синхронизировать обращения к call()
public void run() {
    synchronized(target) { // синхронизированный блок
        target.call(msg);
    }
}

class Synch1 {
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Привет");
        Caller ob2 = new Caller(target, "Синхронизированный");
        Caller ob3 = new Caller(target, "Мир");

        // ждать завершения потоков
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch(InterruptedException e) {
            System.out.println("Прерывание");
        }
    }
}

```

Здесь метод `call()` не модифицируется ключевым словом `synchronized`. Вместо этого внутри класса `Caller` метода `run()` используется оператор `synchronized`. Он выполняет такой же правильный вывод, как в предыдущем примере, потому что каждый поток ожидает завершения предшествующего потока перед своим продолжением.

## Межпоточные связи

Предыдущие примеры безоговорочно блокировали другие потоки от асинхронного доступа к некоторым методам. Использование неявных мониторов в объектах Java — довольно мощное средство, но вы можете достичь более тонкого уровня управления через *связь между процессами*. В Java такая связь организуется особенно просто.

Как обсуждалось ранее, многопоточность заменяет программирование цикла событий, делением задач на дискретные и логические модули. Потоки также обеспечивают и второе преимущество — они отменяют опрос. Опрос обычно реализуется циклом, который используется для повторяющейся проверки некоторого условия. Как только условие становится истинным, предпринимается соответствующее действие. На этом теряется время CPU. Например,

рассмотрим классическую проблему организации очереди, где один поток производит некоторые данные, а другой — их потребляет. Чтобы сделать задачу более интересной, предположим, что, прежде чем генерировать большее количество данных, производитель должен ждать, пока потребитель не закончит свою работу. В системе же опроса, потребитель тратил бы впустую много циклов CPU на ожидание конца работы производителя. Как только производитель закончил свою работу, он вынужден начать опрос, затрачивая много циклов CPU на ожидание конца работы потребителя. Ясно, что такая ситуация нежелательна.

Чтобы устраниТЬ опросы, Java содержит изящный механизм межпроцессовой связи через методы `wait()`, `notify()` и `notifyAll()`. Они реализованы как `final`-методы в классе `Object`, поэтому доступны всем классам. Все три метода можно вызывать только внутри синхронизированного метода. Хотя сама концепция построена на перспективных методиках информатики, правила для использования этих методов в действительности весьма просты:

- `wait()` сообщаетзывающему потоку, что нужно уступить монитор и переходить в режим ожидания ("спячки"), пока некоторый другой поток не введет тот же монитор и не вызовет `notify()`;
- `notify()` "пробуждает" первый поток (который вызвал `wait()`) на том же самом объекте;
- `notifyAll()` пробуждает все потоки, которые вызывали `wait()`, на том же самом объекте. Первым будет выполняться самый высокоприоритетный поток.

Эти методы объявляются в классе `Object` в следующей форме:

```
final void wait() throws InterruptedException  
final void notify()  
final void notifyAll()
```

Существуют дополнительные формы `wait()`, которые позволяют указать период времени ожидания.

Следующий пример программы неправильно реализует простую форму задачи производитель/потребитель. Программа состоит из четырех классов: `Q` — очередь, которую вы пробуете синхронизировать; `Producer` — поточный объект, который производит элементы ввода в очередь; `Consumer` — поточный объект, который потребляет элементы ввода очереди; и `PC` — крошечный класс, который создает одиночные классы `Q`, `Producer` и `Consumer`.

```
// Некорректная реализация производителя и потребителя.  
class Q {  
    int n;  
  
    synchronized int get() {  
        System.out.println("Получено: " + n);  
        return n;  
    }  
}
```

```
    return n;
}

synchronized void put(int n) {
    this.n = n;
    System.out.println("Отдано: " + n);
}
}

class Producer implements Runnable {
    Q q;

    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        int i = 0;

        while(true) {
            q.put(i++);
        }
    }
}

class Consumer implements Runnable {
    Q q;

    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }

    public void run() {
        while(true) {
            q.get();
        }
    }
}

class PC {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);

        System.out.println("Для прерывания нажмите Control-C.");
    }
}
```

Хотя методы `put()` и `get()` класса `Q` синхронизированы, ничто не останавливает производителя от переполнения потребителя, так же, как ничто не будет останавливать потребителя от потребления одного и того же значения очереди дважды. Таким образом, вы получаете следующий ошибочный вывод (точный вывод будет меняться в зависимости от быстродействия процессора и загруженности задачами):

```
Отдано: 1
Получено: 1
Получено: 1
Получено: 1
Получено: 1
Получено: 1
Отдано: 2
Отдано: 3
Отдано: 4
Отдано: 5
Отдано: 6
Отдано: 7
Получено: 7
```

Заметьте, что после выдачи производителем первого элемента (1) потребитель запускается и получает ту же 1 пять раз (это видно в последовательных строчках вывода). Затем производитель возобновляет работу и производит элементы от 2 до 7, не оставляя потребителю шансов для их потребления.

Для получения корректной версии этой программы необходимо использовать методы `wait()` и `notify()`, чтобы сигнализировать в обоих направлениях, как показано ниже:

```
// Корректная реализация поставщика и потребителя.
class Q {
    int n;
    boolean valueSet = false;

    synchronized int get() {
        if(!valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException выброшено");
            }
        System.out.println("Получено: " + n);
        valueSet = false;
        notify();
    }
```

```
    return n;
}

synchronized void put(int n) {
    if(valueSet)
        try {
            wait();
        } catch(InterruptedException e) {
            System.out.println("InterruptedException выброшено");
        }

    this.n = n;
    valueSet = true;
    System.out.println("Отдано: " + n);
    notify();
}
}

class Producer implements Runnable {
    Q q;

    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        int i = 0;

        while(true) {
            q.put(i++);
        }
    }
}

class Consumer implements Runnable {
    Q q;

    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }

    public void run() {
        while(true) {
            q.get();
        }
    }
}
```

```
class PCFixed {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);

        System.out.println("Для прерывания нажмите Control-C.");
    }
}
```

Внутри `get()` вызывается `wait()`. Это приостанавливает выполнение `get()`, пока `Producer` не уведомит его, что некоторые данные готовы. Когда это случается, выполнение внутри `get()` возобновляется. После того как данные были получены, `get()` вызывает `notify()`. Тот сообщает объекту `Producer`, что можно дальше размещать данные в очереди. Внутри `put()` метод `wait()` приостанавливает выполнение, пока `Consumer` не удалил элемент из очереди. Когда выполнение возобновляется, в очередь помещается следующий элемент данных и вызывается `notify()`. Это информирует `Consumer`, что тот должен теперь удалить элемент.

Вывод этой программы показывает ее чисто синхронное поведение:

```
Отдано 1
Получено 1
Отдано 2
Получено 2
Отдано 3
Получено 3
Отдано 4
Получено 4
Отдано 5
Получено 5
```

## Блокировка

Специальный тип ошибки, которую вам нужно избегать и которая специально относится к многозадачности, это — (взаимная) **блокировка**. Она проходит, когда два потока имеют циклическую зависимость от пары синхронизированных объектов. Например, предположим, что один поток вводит монитор в объект `x`, а другой поток вводит монитор в объект `y`. Если поток в `x` пробует вызвать любой синхронизированный метод объекта `y`, это приведет к блокировке, как и ожидается. Однако если поток в `y`, в свою очередь, пробует вызвать любой синхронизированный метод объекта `x`, то он будет всегда ждать, т. к. для получения доступа к `x`, он был бы должен снять свою собственную блокировку с `y`, чтобы первый поток мог завершиться. Взаимоблокировка — трудная ошибка для отладки по двум причинам:

- Вообще говоря, она происходит очень редко, когда интервалы временного квантования двух потоков находятся в определенном соотношении.
- Она может включать больше двух потоков и синхронизированных объектов. (То есть блокировка может происходить через более замысловатую последовательность событий, чем только что описано.)

Для полного понимания блокировки полезно увидеть ее в действии. Следующий пример создает два класса (A и B) с методами foo() и bar(), соответственно, которые делают краткую паузу перед попыткой вызвать метод другого класса. Главный класс (с именем Deadlock) создает экземпляры типа A и B, и затем стартует второй поток для установки состояния блокировки. Оба метода используют sleep() для вызова состояния блокировки.

```
// Пример блокировки.
class A {
    synchronized void foo(B b) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " вошел в A.foo");
        try {
            Thread.sleep(1000);
        } catch(Exception e) {
            System.out.println("A прерван");
        }
        System.out.println(name + " пытается вызвать B.last()");
        b.last();
    }
    synchronized void last() {
        System.out.println("Внутри A.last");
    }
}

class B {
    synchronized void bar(A a) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " вошел в B.bar");
        try {
            Thread.sleep(1000);
        } catch(Exception e) {
            System.out.println("B прерван");
        }
        System.out.println(name + " пытается вызвать A.last()");
        a.last();
    }
}
```

```
synchronized void last() {
    System.out.println("Внутри A.last");
}
}

class Deadlock implements Runnable {
    A a = new A();
    B b = new B();

    Deadlock() {
        Thread.currentThread().setName("MainThread");
        Thread t = new Thread(this, "RacingThread");
        t.start();

        a.foo(b);           // получить блокировку на а в этом потоке
        System.out.println("Возврат в главный поток");
    }

    public void run() {
        b.bar(a);           // получить блокировку на б в другом потоке
        System.out.println("Возврат в другой поток");
    }
}

public static void main(String args[]) {
    new Deadlock();
}
```

Когда вы выполните эту программу, то увидите следующий вывод:

```
MainThread вошел в A.foo
RacingThread вошел в B.bar
MainThread пытается вызвать B.last()
RacingThread пытается вызвать A.last()
```

Поскольку программа была блокирована, вы должны нажать клавиши **<Ctrl>+<C>**, чтобы закончить программу. Вы можете увидеть полный кэш-дамп потока и монитора, если нажмете клавиши **<Ctrl>+<Break>** на PC (или **<Ctrl>+<\>** на Solaris). Вы увидите, что RacingThread имеет монитор на **б**, в то время как он ожидает монитор на **а**. В то же самое время, MainThread имеет **а** и ожидает получения **б**. Эта программа никогда не будет завершена. Как показывает данный пример, если ваша многопоточная программа иногда блокируется, то блокировка — это одно из первых состояний, которое вы должны проверить.

## Приостановка, возобновление и остановка потоков

Приостановка выполнения потока иногда полезна. Например, отдельные потоки могут использоваться, чтобы отображать время дня. Если пользователь не хочет видеть отображения часов, то их поток может быть приостановлен. В любом случае приостановка потока — простое дело. После приостановки перезапуск потока также не сложен.

Механизмы приостановки, остановки и возобновления потоков различны для Java 2 и более ранних версий Java. Хотя для всего нового кода следует использовать подход Java 2, необходимо все-таки понять, как подобные операции выполнялись в более ранних средах Java. Например, вам требуется модифицировать или поддерживать программы старых версий. Вы также должны понимать, почему были сделаны изменения для Java 2. По этим причинам, следующий раздел описывает первоначальный способ управления выполнением потока, а следующий — новый подход, используемый в Java 2.

## Приостановка, возобновление и остановка потоков в Java 1.1 и более ранних версиях

До Java 2 для приостановки и перезапуска выполнения потока программа использовала методы `suspend()` и `resume()`, которые определены в классе `Thread`. Они имеют такую форму:

```
final void suspend()  
final void resume()
```

Представленная далее программа демонстрирует эти методы:

```
// Использование suspend() и resume().  
class NewThread implements Runnable {  
    String name; // Имя потока  
    Thread t;  
  
    NewThread(String threadname) {  
        name = threadname;  
        t = new Thread(this, name);  
        System.out.println("Новый поток: " + t);  
        t.start(); // запуск (старт) потока  
    }  
  
    // Это точка входа для потока.  
    public void run() {  
        try {  
            for(int i = 15; i > 0; i--) {
```

```
        System.out.println(name + ": " + i);
        Thread.sleep(200);
    }
} catch (InterruptedException e) {
    System.out.println(name + " прерван.");
}
System.out.println(name + " завершен.");
}

class SuspendResume {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("Первый");
        NewThread ob2 = new NewThread("Второй");

        try {
            Thread.sleep(1000);
            ob1.t.suspend();
            System.out.println("Приостановка Первого потока");
            Thread.sleep(1000);
            ob1.t.resume();
            System.out.println("Перезапуск Первого потока");
            ob2.t.suspend();
            System.out.println("Приостановка Второго потока");
            Thread.sleep(1000);
            ob2.t.resume();
            System.out.println("Перезапуск Второго потока");
        } catch (InterruptedException e) {
            System.out.println("Прерывание главного потока");
        }

        // ждать завершения потока
        try {
            System.out.println("Ждите завершения потоков.");
            ob1.t.join();
            ob2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Прерывание главного потока.");
        }
        System.out.println("Завершение главного потока.");
    }
}
```

Пример вывода этой программы:

Новый поток: Thread[One/5,main]

Первый: 15

Новый поток: Thread[Two,5,main]

Второй: 15

Первый: 14

Второй: 14

Первый: 13

Второй: 13

Первый: 12

Второй: 12

Первый: 11

Второй: 11

Приостановка Первого потока

Второй: 10

Второй: 9

Второй: 8

Второй: 7

Второй: 6

Перезапуск Первого потока

Приостановка Второго потока

Первый: 10

Первый: 9

Первый: 8

Первый: 7

Первый: 6

Перезапуск Второго потока

Ждите завершения потоков.

Второй: 5

Первый: 5

Второй: 4

Первый: 4

Второй: 3

Первый: 3

Второй: 2

Первый: 2

Второй: 1

Первый: 1

Второй завершен.

Первый завершен.

Завершение главного потока.

Класс `Thread` также определяет метод с именем `stop()`, который останавливает поток. Его сигнатура имеет следующий вид:

`void stop()`

Если поток был остановлен, то его нельзя перезапускать с помощью метода `resume()`.

## Приостановка, возобновление и остановка потока в Java 2

Хотя применение методов `suspend()`, `resume()` и `stop()`, определенных в классе `Thread`, кажется совершенно разумным и удобным подходом к управлению выполнением потоков, они не должны использоваться в новых Java-программах. И вот почему. Метод `suspend()` класса `Thread` исключен из Java 2. Это было сделано, потому что `suspend()` может иногда вызывать серьезные системные отказы. Предположим, что поток получил блокировки на критических структурах данных. Если этот поток приостанавливается в такой точке, то указанные блокировки не отменяются. Так что другие потоки, ожидающие эти ресурсы, могут быть заблокированы.

Метод `resume()` также исключен. Он не вызывает проблемы, но не может использоваться без своего напарника — метода `suspend()`.

Метод `stop()` класса `Thread` также исключен из Java 2. Это было сделано, потому что данный метод может иногда вызывать серьезные системные отказы. Предположим, что поток пишет в критически важную структуру данных и завершил только часть ее изменений. Если он останавливается в такой точке, то структура данных может оказаться в разрушенном состоянии.

Поскольку в Java 2 запрещено использовать методы `suspend()`, `resume()` или `stop()` для управления потоком, можно подумать, что нет никакого способа для приостановки, перезапуска или завершения потока. Но, к счастью, это не верно. Вместо этого, поток должен быть спроектирован так, чтобы метод `run()` периодически проверял, должен ли этот поток приостанавливать, возобновлять или останавливать свое собственное выполнение. Это, как правило, выполняется применением флаговой переменной, которая указывает состояние выполнения потока. Пока флагок установлен на "выполнение", метод `run()` должен продолжать позволять потоку выполняться. Если эта переменная установлена на "приостановить", поток должен сделать паузу. Если она установлена на "стоп", поток должен завершиться. Конечно, существует много способов записи такого кода, но центральная тема будет одна и та же для всех программ.

Следующий пример иллюстрирует, как методы `wait()` и `notify()`, унаследованные от `Object`, можно использовать для управления выполнением потока. Данный пример подобен программе в предыдущем разделе. Однако вызовы исключенных методов были удалены. Рассмотрим работу этой программы.

Класс `NewThread` содержит `boolean`-переменную экземпляра с именем `suspendFlag`, которая используется для управления выполнением потока. Она инициализирована (с помощью конструктора) значением `false`. Метод `run()` содержит синхронизированный блочный оператор, который проверяет `suspendFlag`. Если эта переменная — `true`, то вызывается метод `wait()`, что-

бы приостановить выполнение потока. Метод `mysuspend()` устанавливает в `suspendFlag` значение `true`. Метод `myresume()` устанавливает в `suspendFlag` значение `false` и вызывает метод `notify()`, чтобы пробудить поток. Наконец, метод `main()` был модифицирован для вызова методов `mysuspend()` и `myresume()`.

```
// Приостанавливает и возобновляет поток для Java 2
class NewThread implements Runnable {
    String name;                                // имя потока
    Thread t;
    boolean suspendFlag;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("Новый поток: " + t);
        suspendFlag = false;
        t.start();                                // старт потока
    }

    // Это точка входа для потока.
    public void run() {
        try {
            for(int i = 15; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(200);
                synchronized(this) {
                    while(suspendFlag) {
                        wait();
                    }
                }
            }
        } catch (InterruptedException e) {
            System.out.println(name + " прерван.");
        }
        System.out.println(name + " завершен.");
    }

    void mysuspend() {
        suspendFlag = true;
    }

    synchronized void myresume() {
        suspendFlag = false;
        notify();
    }
}
```

```
class SuspendResume {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("Первый");
        NewThread ob2 = new NewThread("Второй");

        try {
            Thread.sleep(1000);
            ob1.mysuspend();
            System.out.println("Приостановка Первого потока");
            Thread.sleep(1000);
            ob1.myresume();
            System.out.println("Перезапуск Первого потока");
            ob2.mysuspend();
            System.out.println("Приостановка Второго потока");
            Thread.sleep(1000);
            ob2.myresume();
            System.out.println("Перезапуск Второго потока");
        } catch (InterruptedException e) {
            System.out.println("Прерывание главного потока");
        }

        // ждать завершения потока
        try {
            System.out.println("Ждите завершения потоков.");
            ob1.t.join();
            ob2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Прерывание главного потока.");
        }

        System.out.println("Завершение главного потока.");
    }
}
```

Вывод этой программы идентичен показанному в предыдущем разделе. Позже вы увидите больше примеров, которые используют новый (принаследующий языку Java 2) механизм управления потоками. Хотя этот механизм не столь же "чист" как предыдущий способ, однако, данная методика способна гарантировать отсутствие ошибок времени выполнения. В этом заключается подход, который *должен* использоваться для всех новых программ.

## Использование многопоточности

Как и для большинства программистов,строенная в язык многопоточная поддержка для вас будет новостью. Ключом к эффективному использованию

указанной поддержки является скорее параллельное (одновременное), чем последовательное мышление. Например, когда вы имеете две подсистемы внутри программы, которые могут выполняться одновременно, организуйте их индивидуальными потоками. С осторожным использованием многопоточности вы можете создавать очень эффективные программы. Однако необходимы и слова предостережения. Если вы создадите слишком много потоков, то, напротив, ухудшите эффективность программы. Помните, что с контекстным переключением связаны некоторые издержки. Если вы создаете слишком много потоков, больше времени CPU будет потрачено на изменения контекстов, чем на выполнение вашей программы.



## ГЛАВА 12

# Ввод/вывод, апплеты и другие темы

В этой главе представлено два самых важных пакета `java.io` и `java.applet`. Пакет `java.io` поддерживает основную систему ввода/вывода Java, включая файловый ввод/вывод. Пакет `java.applet` поддерживает *апплеты*. Поддержка обеих подсистем опирается на библиотеки ядра API<sup>1</sup> Java (а не на ключевые слова языка). Поэтому подробности данных тем обсуждаются в Части II этой книги, где исследуются классы Java API. В текущей главе рассматриваются основные средства этих двух подсистем, чтобы вы смогли увидеть, как они интегрированы в язык Java и вписываются в более широкий контекст Java-программирования и среды выполнения. Здесь также рассматриваются следующие ключевые слова языка Java: `transient`, `volatile`, `instanceof`, `native` и `strictfp`.

## Основы ввода/вывода

Как вы, возможно, заметили при чтении предшествующих одиннадцати глав, в примерах программ до сих пор ввод/вывод использовался мало. Фактически, кроме `print()` и `println()`, ни один из методов ввода/вывода не применялся. Причина проста: реальные приложения Java не основаны на консольных *текстовых* программах. Скорее они являются *графическими* апплетами, которые для взаимодействия с пользователем полагаются на систему классов AWT (Abstract Window Toolkit, инструментарий абстрактного окна) языка Java. Хотя текстовые программы превосходны как примеры для обучения, они не очень полезны для языка Java в его реальных приложениях. Поддержка Java для консольного ввода/вывода ограничена и не очень удобна в использовании — даже в простых примерах программ. Текстовый

<sup>1</sup> API (Application Programming Interface) — интерфейс прикладного программирования. — Примеч. пер.

консольный ввод/вывод в действительности не очень важен для Java-программирования.

Несмотря на это, Java обеспечивает сильную, гибкую поддержку по существу *текстового ввода/вывода* для файлов и сетей. Система ввода/вывода Java связана и непротиворечива. Фактически, как только вы поняли ее основные принципы, овладеть остальной частью системы ввода/вывода очень просто.

## Потоки

Java-программы выполняют ввод/вывод через потоки. *Поток* является абстракцией, которая или производит, или потребляет информацию. Поток связывается с физическим устройством с помощью системы ввода/вывода Java (Java I/O system). Все потоки ведут себя одинаковым образом, хотя фактические физические устройства, с которыми они связаны, могут сильно различаться. Таким образом, одни и те же классы и методы ввода/вывода можно применять к устройствам любого типа. Это означает, что *поток ввода* может извлекать много различных видов входных данных: из дискового файла, с клавиатуры или сетевого разъема. Аналогично, *поток вывода* может обращаться к консоли, дисковому файлу или сетевому соединению (сокету). Благодаря потокам ваша программа выполняет ввод/вывод, не понимая различий между клавиатурой и сетью, например. Java реализует потоки с помощью иерархии классов, определенных в пакете `java.io`.

### Замечание

Если вы знаете C/C++, то уже знакомы с концепцией потока. Подход Java к потокам почти такой же, как в C/C++.

## Байтовые и символьные потоки

Java 2 определяет два типа потоков<sup>1</sup>: байтовый и символьный. *Байтовые потоки* предоставляют удобные средства для обработки ввода и вывода байтов. Байтовые потоки используются, например, при чтении или записи данных в двоичном коде. *Символьные потоки* предоставляют удобные средства для обработки ввода и вывода символов. Они используют Unicode и поэтому могут быть интернационализированы<sup>2</sup>. Кроме того, в некоторых случаях символьные потоки более эффективны, чем байтовые.

Первоначальная версия Java (Java 1.0) не включала символьные потоки и, таким образом, весь ввод/вывод был байтовым. Символьные потоки были добавлены в Java 1.1, а некоторые байтовые классы и методы были исключены.

<sup>1</sup> Точнее — поточных классов и объектов. — Примеч. пер.

<sup>2</sup> Речь, по-видимому, идет о возможности ввода/вывода широкого спектра национальных алфавитов (с помощью Unicode). — Примеч. пер.

чены. Поэтому, чтобы воспользоваться их преимуществами (там, где это уместно, конечно), старый код, не использующий символьные потоки, должен быть модифицирован.

На самом низком уровне весь ввод/вывод все еще байтовый. Символьно-ориентированные потоки обеспечивают удобные и эффективные средства для обработки символов.

В следующих разделах представлен краткий обзор как байтовых, так и символьных потоков.

## Классы байтовых потоков

Байтовые потоки определяются в двух иерархиях классов. Наверху этой иерархии — два абстрактных класса: `InputStream` и `OutputStream`. Каждый из этих абстрактных классов имеет несколько конкретных подклассов, которые обрабатывают различия между разными устройствами, такими как дисковые файлы, сетевые соединения и даже буферы памяти. Классы байтовых потоков показаны в табл. 12.1. Некоторые из этих классов обсуждаются далее в этом разделе. Другие описаны в Части II данной книги. Помните, чтобы использовать поточные классы, нужно импортировать пакет `java.io`.

Абстрактные классы `InputStream` и `OutputStream` определяют несколько ключевых методов, которые реализуются другими поточными классами. Два наиболее важных — `read()` и `write()`, которые, соответственно, читают и записывают байты данных. Оба метода объявлены как абстрактные внутри классов `InputStream` и `OutputStream` и переопределяются производными поточными классами.

## Классы символьных потоков

Символьные потоки определены в двух иерархиях классов. Наверху этой иерархии два абстрактных класса: `Reader` и `Writer`. Они обрабатывают потоки символов Unicode. В Java существуют несколько конкретных подклассов каждого из них. Классы символьных потоков показаны в табл. 12.2.

Абстрактные классы `Reader` и `Writer` определяют несколько ключевых методов, которые реализуются другими поточными классами. Два самых важных метода — `read()` и `write()`, которые читают и записывают символы данных, соответственно. Они переопределяются производными поточными классами.

**Таблица 12.1. Классы байтовых потоков**

Поточный класс	Значение
<code>BufferedInputStream</code>	Буферизированный поток ввода
<code>BufferedOutputStream</code>	Буферизированный поток вывода

**Таблица 12.1 (окончание)**

<b>Поточный класс</b>	<b>Значение</b>
ByteArrayInputStream	Поток ввода, который читает из байт-массива
ByteArrayOutputStream	Поток вывода, который записывает в байт-массив
DataInputStream	Поток ввода, который содержит методы для чтения данных стандартных типов Java
DataOutputStream	Поток вывода, который содержит методы для записи данных стандартных типов Java
FileInputStream	Поток ввода, который читает из файла
FileOutputStream	Поток вывода, который записывает в файл
FilterInputStream	Реализует InputStream
FilterOutputStream	Реализует OutputStream
InputStream	Абстрактный класс, который описывает поточный ввод
OutputStream	Абстрактный класс, который описывает поточный вывод
PipedInputStream	Канал ввода
PipedOutputStream	Канал вывода
PrintStream	Поток вывода, который поддерживает print() и println()
PushbackInputStream	Поток (ввода), который поддерживает однобайтовую операцию "unget", возвращающую байт в поток ввода
RandomAccessFile	Поддерживает ввод/вывод файла произвольного доступа
SequenceInputStream	Поток ввода, который является комбинацией двух или нескольких потоков ввода, которые будут читаться последовательно, один за другим

**Таблица 12.2. Классы ввода/вывода символьных потоков**

<b>Поточный класс</b>	<b>Значение</b>
BufferedReader	Буферизированный символьный поток ввода
BufferedWriter	Буферизированный символьный поток вывода
CharArrayReader	Поток ввода, который читает из символьного массива
CharArrayWriter	Выходной поток, который записывает в символьный массив
FileReader	Поток ввода, который читает из файла

Таблица 12.2 (окончание)

Поточный класс	Значение
FileWriter	Выходной поток, который записывает в файл
FilterReader	Отфильтрованный поток ввода
FilterWriter	Отфильтрованный поток вывода
InputStreamReader	Поток ввода, который переводит байты в символы
LineNumberReader	Поток ввода, который считает строки
OutputStreamWriter	Поток ввода, который переводит символы в байты
PipedReader	Канал ввода
PipedWriter	Канал вывода
PrintWriter	Поток вывода, который поддерживает <code>print()</code> и <code>println()</code>
PushbackReader	Поток ввода, возвращающий символы в поток ввода
Reader	Абстрактный класс, который описывает символьный поток ввода
StringReader	Поток ввода, который читает из строки
StringWriter	Поток вывода, который записывает в строку
Writer	Абстрактный класс, который описывает символьный поток вывода

## Предопределенные потоки

Как известно, все программы Java автоматически импортируют пакет `java.lang`. Этот пакет определяет класс с именем `System`, инкапсулирующий некоторые аспекты исполнительной среды Java. Например, используя некоторые из его методов, вы можете получить текущее время и параметры настройки различных свойств, связанных с системой. Класс `System` содержит также три предопределенные поточные переменные `in`, `out` и `err`. Эти поля объявлены в `System` со спецификаторами `public` и `static`. Это означает, что они могут использоваться любой другой частью вашей программы, и причем без ссылки на конкретный `System`-объект.

Объект `System.out` называют *потоком стандартного вывода*. По умолчанию с ним связана консоль. На объект `System.in` ссылаются как на *стандартный ввод*, который по умолчанию связан с клавиатурой. К объекту `System.err` обращаются как к *стандартному потоку ошибок*, который по умолчанию также связан с консолью. Однако эти потоки могут быть переназначены на любое совместимое устройство ввода/вывода.

`System.in` — это объект типа `InputStream`; `System.out` и `System.err` — объекты типа `PrintStream`. Это байтовые потоки, хотя они обычно используются, чтобы читать и записывать символы с консоли и на консоль. Однако вы можете упаковать их в символьные потоки, если пожелаете.

В примерах предыдущих глав использовался объект `System.out`. Аналогичным образом вы можете использовать и `System.err`. Как объясняется в следующем разделе, использование `System.in` немного сложнее.

## Чтение консольного ввода

В Java 1.0 единственный способ выполнения консольного ввода состоял в применении байтового потока, и старые коды, которые использует этот подход, еще существуют. В настоящее время использование байтового потока для чтения консольного ввода все еще технически возможно, но это может потребовать применения исключенного метода, и такой подход не рекомендуется. Предпочтительный способ чтения консольного ввода для Java 2 заключается в использовании символьного потока, который упрощает интернационализацию и поддержку программы.

### Замечание

Java не имеет обобщенного метода консольного ввода, который соответствует стандартной С-функции `scanf()` или операциям ввода C++.

Консольный ввод в Java выполняется с помощью считывания из объекта `System.in`. Чтобы получить символьный поток, который присоединен к консоли, вы переносите ("упаковываете") `System.in` в объект типа `BufferedReader`. Класс `BufferedReader` поддерживает буферизованный входной поток. Обычно используется следующий его конструктор:

`BufferedReader(Reader inputReader)`

где `inputReader` — поток, который связан с создающимся экземпляром класса `BufferedReader`. `Reader` — абстрактный класс. Один из его конкретных подклассов — это `InputStreamReader`, который преобразовывает байты в символы. Чтобы получить `InputStreamReader`-объект, который связан с `System.in`, используйте следующий конструктор:

`InputStreamReader(InputStream inputStream)`

Поскольку `System.in` ссылается на объект типа `InputStream`, его можно использовать в качестве параметра `inputStream`. Объединив все это вместе, следующая строка кода создает объект класса `BufferedReader`, который связан с клавиатурой:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

После того как этот оператор выполнится, объектная переменная `br` станет символьным потоком, связанным с консолью через `System.in`.

## Чтение символов

Для чтения символа из `BufferedReader` используйте метод `read()`. Версия `read()`, которую мы будем применять, такова:

```
int read() throws IOException
```

При каждом вызове `read()` читает символ из входного потока и возвращает его в виде целочисленного значения. Когда `read()` сталкивается с концом потока, то возвращает `-1`. Как вы видите, он может выбрасывать исключение ввода/вывода (`I/O-исключение — IOException`).

Следующая программа демонстрирует `read()`, читая символы с консоли, пока пользователь не напечатает "q":

```
// Использует BufferedReader для чтения символов с консоли.
import java.io.*;

class BRRead {
    public static void main(String args[])
        throws IOException
    {
        char c;
        BufferedReader br = new
            BufferedReader(new InputStreamReader(System.in));
        System.out.println("Введите символы, 'q' - для завершения.");

        // чтение символов
        do {
            c = (char) br.read();
            System.out.println(c);
        } while(c != 'q');
    }
}
```

Результат выполнения этого примера:

Введите символы, 'q' - для завершения.

123abcq

1

2

3

a

b

c

q

Этот вывод может выглядеть немного иначе, чем вы ожидали, потому что `System.in` по умолчанию — буферизированная строка. Это означает, что никакой ввод фактически не пересыпается программе, пока вы не нажмете клавишу <Enter>. Не трудно предположить, что это не придает методу `read()` особенной ценности для интерактивного консольного ввода.

## Чтение строк

Для чтения строки, вводимой с клавиатуры, используйте версию метода `readLine()`, который является элементом класса `BufferedReader`. Его общая форма:

```
String readLine() throws IOException
```

Как видно, он возвращает `String`-объект.

Следующая программа демонстрирует `BufferedReader` и метод `readLine()`. Она читает и отображает строки текста, пока вы не введете слово "stop":

```
// Читает строки с консоли, используя BufferedReader.
import java.io.*;

class BRReadLines {
    public static void main(String args[])
        throws IOException
    {
        // создать BufferedReader, используя System.in
        BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in));
        String str;

        System.out.println("Введите строки текста.");
        System.out.println("Введите 'stop' для завершения.");
        do {
            str = br.readLine();
            System.out.println(str);
        } while(!str.equals("stop"));
    }
}
```

Следующий пример демонстрирует крошечный текстовый редактор. Сначала он создает массив `String`-объектов и затем считывает строки текста, сохраняя каждую из них в массиве. Он будет читать до сотой строки или до тех пор, пока вы не введете строку "stop". Для чтения с консоли используется объект класса `BufferedReader` (переменная `pr`).

```
// Крошечный редактор.
import java.io.*;
```

```

class TinyEdit {
    public static void main(String args[])
        throws IOException
    {
        // Создать BufferedReader-объект, используя System.in
        BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in));
        String str[] = new String[100];

        System.out.println("Введите строки текста.");
        System.out.println("Введите 'stop' для завершения.");
        for(int i=0; i<100; i++) {
            str[i] = br.readLine();
            if(str[i].equals("stop")) break;
        }

        System.out.println("\nВот ваш файл:");
        // Вывести строки на экран.
        for(int i=0; i<100; i++) {
            if(str[i].equals("stop")) break;
            System.out.println(str[i]);
        }
    }
}

```

Пример вывода этой программы:

```

Введите строки текста.
Введите 'stop' для завершения.
Это строка 1.
Это строка 2.
Java облегчает работу со строками.
Создать String-объекты.
stop
Вот ваш файл:
Это строка 1.
Это строка 2.
Java облегчает работу со строками.
Создать String-объекты.

```

## Запись консольного вывода

Консольный вывод легче всего выполнить с помощью описанных ранее методов `print()` и `println()`, которые используются в большинстве примеров данной книги. Эти методы определены классом `PrintStream` (который явля-

ется типом (классом) объекта `System.out`). Хотя `System.out` — *байтовый* поток, его использование для вывода в простых программах все еще допустимо. Его *символьная* альтернатива описана в следующем разделе.

Поскольку `PrintStream` — выходной поток, производный от `OutputStream`, он также реализует метод нижнего уровня `write()`. Его можно использовать для записи на консоль. Самая простая форма `write()`, определенная в `PrintStream`, имеет вид:

```
void write(int byteval) throws IOException
```

Этот метод записывает в файл байт, указанный в параметре `byteval`. Хотя `byteval` объявлен как целое число, записываются только младшие восемь битов. Ниже показан короткий пример, который использует `write()` для вывода на экран символа "A", за которым следует символ `newline`:

```
// Демонстрирует System.out.write().
class WriteDemo {
    public static void main(String args[]) {
        int b;

        b = 'A';
        System.out.write(b);
        System.out.write('\n');
    }
}
```

Вы не часто будете применять `write()` для выполнения консольного вывода (хотя это может быть полезно в некоторых ситуациях), потому что использовать `print()` и `println()` намного легче.

## Класс `PrintWriter`

Хотя использование объекта `System.out` для записи на консоль все еще допустимо в Java, его применение рекомендуется главным образом для отладочных целей или для демонстрационных программ, типа тех, которые показаны в этой книге. Для реальных Java-программ для записи на консоль рекомендуется работать с потоком типа `PrintWriter`. `PrintWriter` — это один из классов символьного ввода/вывода. Использование подобного класса для консольного вывода облегчает интернационализацию вашей программы.

`PrintWriter` определяет несколько конструкторов. Мы будем использовать следующий:

```
PrintWriter(OutputStream outputStream, boolean flushOnNewline)
```

Здесь `outputStream` — объект типа `OutputStream`; `flushOnNewline` — булевский параметр, используемый как средство управления сбросыванием выходного

потока в буфер вывода (на диск) каждый раз, когда выводится символ newline (\n). Если `flushOnNewline` — true, поток сбрасывается автоматически, если — false, то не автоматически.

`PrintWriter` поддерживает методы `print()` и `println()` для всех типов, включая `Object`. Поэтому эти методы можно применять так же, как они использовались с объектом `System.out`. Если аргумент не является простым типом, то методы класса `PrintWriter` вызывают объектный метод `toString()` и затем печатают результат.

Чтобы записывать на консоль, используя класс `PrintWriter`, создайте объект `System.out` для выходного потока, и сбрасывайте поток после каждого символа newline. Например, следующая строка кода создает объект типа `PrintWriter`, который соединен с консольным выводом:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

Очередное приложение иллюстрирует использование `PrintWriter` для управления консольным выводом:

```
// Демонстрирует PrintWriter.  
import java.io.*;  
  
public class PrintWriterDemo {  
    public static void main(String args[]) {  
        PrintWriter pw = new PrintWriter(System.out, true);  
        pw.println("Это строка:");  
        int i = -7;  
        pw.println(i);  
        double d = 4.5e-7;  
        pw.println(d);  
    }  
}
```

Вывод этой программы:

Это строка:

-7

4.5E-7

Помните, что во время изучения Java или отладки программы нет ничего не правильного в использовании `System.out` для записи простого текстового вывода на консоль. Однако с использованием `PrintWriter` ваши реальные приложения будет проще интернационализировать. Поскольку никакого преимущества от использования `PrintWriter` в демонстрационных программах этой книги не проявляется, для записи на консоль мы продолжим использовать объект `System.out`.

## Чтение и запись файлов

Java обеспечивает ряд классов и методов, которые позволяют читать и записывать файлы. Для Java все файлы имеют байтовую структуру, а Java обеспечивает методы для чтения и записи байтов в файл. Кроме того, Java позволяет упаковывать *байтовый* файловый поток в символьно-ориентированный объект. Эта методика описана в Части II. В данной главе рассматриваются основы файлового ввода/вывода.

Для создания байтовых потоков, связанных с файлами, чаще всего используются два поточных класса — `FileInputStream` и `OutputStream`. Для открытия файла вы просто создаете объект одного из этих классов, указывая имя файла как аргумент конструктора. Хотя оба класса поддерживают несколько переопределенных конструкторов, мы будем использовать только следующие формы:

```
FileInputStream(String fileName) throws FileNotFoundException
OutputStream(String fileName) throws FileNotFoundException
```

где `fileName` определяет имя файла, который вы хотите открыть. Когда вы создаете входной поток при отсутствующем файле, выбрасывается исключение `FileNotFoundException`. Для выходных потоков, если файл не может быть создан, выбрасывается такое же исключение (`FileNotFoundException`). Когда выходной файл открывается, любой файл, существовавший ранее с тем же самым именем, разрушается.

### Замечание

В ранних версиях Java `OutputStream()` выбрасывал исключение `IOException`, когда выходной файл не мог быть создан. В Java 2 это было изменено.

После завершения работы с файлом, его нужно закрыть, вызвав метод `close()`. Он определен как в `FileInputStream`, так и в `OutputStream` в следующей форме:

```
void close() throws IOException
```

Для чтения файла можно использовать версию метода `read()`, который определен в `FileInputStream`. Мы будем использовать такую версию:

```
int read() throws IOException
```

При каждом вызове он (метод) читает один байт из файла и возвращает его в форме целочисленного значения. Когда `read()` встречает символ конца файла (EOF), то возвращает `-1`. Метод `read()` может выбрасывать исключение `IOException`.

Следующая программа использует `read()` для ввода и отображения содержимого текстового файла, имя которого указывается как параметр команд-

ной строки. Обратите внимание на блоки `try/catch`, обрабатывающие две ошибки, которые могут произойти во время использования программы: указанный файл не найден, или пользователь забыл включить в командную строку имя файла.

```
/* Выведет на экран текстовый файл.
```

При запуске программы укажите (в параметре команды запуска) имя файла, который вы хотите просмотреть.

Например, чтобы просмотреть файл с именем TEST.TXT,  
используйте следующую командную строку

```
java ShowFile TEST.TXT
*/
import java.io.*;

class ShowFile {
    public static void main(String args[])
        throws IOException
    {
        int i;
        FileInputStream fin;

        try {
            fin = new FileInputStream(args[0]);
        } catch(FileNotFoundException e) {
            System.out.println("Файл не найден");
            return;
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Используйте: ShowFile имя_файла");
            return;
        }

        // читать символы файла, пока не встретится символ EOF
        do {
            i = fin.read();
            if(i != -1) System.out.print((char)i);
        } while(i != -1);

        fin.close();
    }
}
```

Для записи в файл используется метод `write()`, определенный в классе `FileOutputStream`. Его самая простая форма имеет вид:

```
void write(int byteval) throws IOException
```

Данный метод записывает в файл байт, указанный в параметре `byteval`. Хотя `byteval` объявлен как целое число, в файл записывается только восемь младших битов. Если во время записи происходит ошибка, выбрасывается исключение `IOException`. В следующем примере метод `write()` применяется для копирования текстового файла:

```
/* Копирование текстового файла.
```

При запуске этой программы укажите имя исходного файла и целевого файла(в который будет выполняться копирование).

Например, чтобы скопировать файл с именем FIRST.TXT

в файл с именем SECOND.TXT, используйте следующую командную строку

```
java CopyFile FIRST.TXT SECOND.TXT
```

```
*/
```

```
import java.io.*;
```

```
class CopyFile {
```

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

```
        int i;
```

```
        FileInputStream fin;
```

```
        FileOutputStream fout;
```

```
        try {
```

```
            // открыть файл для ввода
```

```
            try {
```

```
                fin = new FileInputStream(args[0]);
```

```
            } catch(FileNotFoundException e) {
```

```
                System.out.println("Исходный файл не найден");
```

```
                return;
```

```
}
```

```
            // открыть файл для вывода
```

```
            try {
```

```
                fout = new FileOutputStream(args[1]);
```

```
            } catch(FileNotFoundException e) {
```

```
                System.out.println("Ошибка открытия выходного файла");
```

```
                return;
```

```
}
```

```
} catch(ArrayIndexOutOfBoundsException e) {
```

```
    System.out.println("CopyFile копирует исходный файл в выходной");
```

```
    return;
```

```
}
```

```
// копировать файл
```

```
try {
```

```
do {
    i = fin.read();
    if(i != -1) fout.write(i);
} while(i != -1);
} catch(IOException e) {
    System.out.println("Файловая ошибка");
}

fin.close();
fout.close();
}
}
```

Обратите внимание на способ обработки потенциальных ошибок ввода/вывода в этой и в предыдущей программе. В отличие от большинства других машинных языков, включая С и С++, которые используют коды ошибки, чтобы сообщать о файловых ошибках, Java используют собственный механизм обработки особых ситуаций (исключений). Это не только делает обработку файла более ясной, но и позволяет во время ввода легко отличить состояние конца файла от файловых ошибок. В С/C++ многие функции ввода возвращают одно и то же значение, когда происходит ошибка и когда достигнут конец файла (т. е. в С/C++ признак конца файла (EOF) часто отображается в то же значение, что и ошибка ввода). Это обычно означает, что программист должен включить в код дополнительные операторы для того, чтобы определить, что же фактически произошло. В Java ошибки передаются в вашу программу через исключения, а не через значения, возвращаемые методом `read()`. Таким образом, когда `read()` возвращает `-1`, это означает только одно — при чтении встретился конец файла.

## Апплеты. Основы программирования

Все предшествующие примеры в этой книге были Java-приложениями. Однако *приложение* — это только один тип Java-программ. Другой тип программ представлен *апплетом*. В главе 1 *апплеты* определялись как небольшие приложения, которые доступны на Internet-сервере, транспортируются по Internet, автоматически устанавливаются и выполняются как часть Web-документа. После того, как апплет прибывает к клиенту, он имеет ограниченный доступ к ресурсам системы, которые использует для создания произвольного мультимедийного интерфейса пользователя и выполнения комплексных вычислений без риска заражения вирусами или нарушения целостности данных.

Многие проблемы, связанные с созданием и использованием апплетов, обсуждаются в Части II, где рассматривается пакет `applet`. Здесь же представлены основные принципы, связанные с созданием апплетов, потому что они

не структурированы таким же образом, как используемые до настоящего времени программы. Как вы увидите, апплеты отличаются от приложений в нескольких ключевых областях.

Начнем с примера простого апплета:

```
import java.awt.*;
import java.applet.*;

public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("A Simple Applet", 20, 20);
    }
}
```

Апплет начинается с двух операторов `import`. Первый импортирует AWT классы (из большой иерархии awt-пакетов Java). Таким образом, апплеты взаимодействуют с пользователем через AWT, а не через классы консольного ввода/вывода. AWT осуществляет поддержку графического оконного интерфейса. Система AWT классов достаточно велика и сложна, ее полное обсуждение занимает несколько глав в Части II этой книги. К счастью, в нашем простом апплете использование AWT очень ограничено. Второй оператор `import` импортирует пакет `java.applet`, который содержит класс `Applet`. Каждый апплет, который вы создаете, должен быть подклассом этого класса.

Следующая строка в программе объявляет класс `SimpleApplet`. Он должен быть объявлен как `public`, потому что к нему необходимо обеспечить доступ из кодов, которые находятся вне программы.

Внутри `SimpleApplet` объявлен метод `paint()`. Этот метод определен в AWT и должен быть переопределен апплетом. Метод `paint()` вызывается каждый раз, когда апплет должен восстанавливать изображение своего вывода. Данная ситуация может возникать в нескольких случаях. Например, окно, в котором выполняется апплет, может быть перекрыто другим окном, которое затем закрывается. Или окно апплета может быть свернуто и затем восстановлено. Метод `paint()` вызывается также, когда апплет начинает выполнение. Безотносительной причины, всякий раз, когда апплет должен перерисовать свой вывод, вызывается метод `paint()`. Метод имеет один параметр типа `Graphics`, через который получает графический контекст, описывающий графическую среду выполнения апплета. Этот контекст используется всякий раз, когда апплету требуется вывод.

Внутри `paint()` находится обращение к методу `drawString()`, который является членом класса `Graphics`. Этот метод выводит строку, начиная с указан-

<sup>1</sup> AWT (Abstract Windowing Toolkit) — абстрактный оконный интерфейс. — Примеч. перев.

ных его аргументами  $(x, y)$ -координат в окне апплета. Он имеет следующую общую форму:

```
void drawString(String message, int x, int y)
```

Здесь *message* — строка, которую нужно вывести. В окне апплета левый верхний угол имеет координаты  $(0, 0)$ . Обращение к *drawString()* в апплете отображает сообщение "A Simple Applet", начиная с координат  $(20, 20)$ <sup>1</sup>.

Заметим, что апплет не содержит метода *main()*. В отличие от Java-программ, апплеты не начинают выполнение в *main()*. Фактически, большинство апплетов даже не имеют этого метода. Апплет начинает выполнение, когда имя его класса передается программе просмотра апплетов или браузеру.

После ввода исходного кода *SimpleApplet* откомпилируйте его так же, как вы компилировали программы. Существует два способа выполнения апплета:

- Выполнение апплета Java-совместимом Web-браузером, типа *Netscape Navigator* или *Microsoft Internet Explorer*.
- Использование программы просмотра апплетов, типа стандартной утилиты *JDK appletviewer*. Программа просмотра апплетов выполняет апплет в его окне. Это, вообще, самый быстрый и простой способ проверки работы апплета.

Для выполнения апплета в Web-браузере нужно записать короткий текстовый файл в формате языка HTML<sup>2</sup>, который содержит специальный тег *<applet>*. HTML-файл, который выполняет *SimpleApplet*, совсем прост:

```
<applet code="SimpleApplet" width=200 height=60>
</applet>
```

Внутри тега *<applet>* его параметры *width* и *height* определяют размеры окна апплета. (Тег *<applet>* содержит несколько других параметров, которые рассматриваются в Части II.) После создания файла, вы можете запустить свой браузер и затем загрузить этот файл, что приведет к выполнению *SimpleApplet*.

Для выполнения *SimpleApplet* с помощью программы просмотра апплетов можно использовать этот же HTML-файл. Например, если предшествующий HTML-файл назвать *RunApp.html*, то следующая командная строка выполнит *SimpleApplet*:

```
C:\>appletviewer RunApp.html
```

Однако существует и более удобный метод, с помощью которого можно ускорить тестирование. Для этого нужно просто включить в заголовок файла

<sup>1</sup> Координаты окна апплета измеряются в пикселях. — Примеч. пер.

<sup>2</sup> HTML (HyperText Markup Language) — язык разметки гипертекста. — Примеч. пер.

исходного кода Java комментарий, который содержит HTML-тег <applet>. Это документирует исходный код прототипом необходимых инструкций HTML, и вы можете тестировать откомпилированный аплет, просто запуская программу просмотра аплетов с файлом исходного кода Java (в качестве операнда). При использовании этого метода исходный файл SimpleApplet.java выглядит так:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/
public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("A Simple Applet", 20, 20);
    }
}
```

Процедура быстрой разработки аплета по этой методике включает три шага:

1. Редактирование исходного файла Java.
2. Компиляция программы.
3. Запуск программы просмотра аплетов со спецификацией имени исходного файла аплета в ее аргументе. Встретив в комментарии тег <applet>, утилита просмотра выполнит его.

На рис. 12.1 показано окно SimpleApplet, в отображении утилиты просмотра аплетов.

Хотя тематика аплетов более полно обсуждается в следующих главах данной книги, несколько ключевых моментов нужно запомнить уже теперь.

- Аплеты не нуждаются в методе main().
- Аплеты должны выполняться программой просмотра аплетов или браузером, поддерживающим Java.
- Пользовательский ввод/вывод в аплетах не выполняется с помощью Java-классов поточного ввода/вывода. Вместо этого аплеты используют интерфейс, обеспеченный системой AWT.

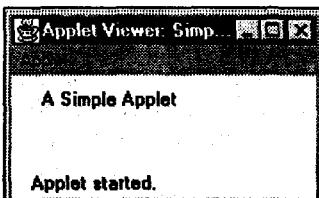


Рис. 12.1. Окно аплета SimpleApplet

## Модификаторы *transient* и *volatile*

В Java определено два интересных модификатора типа: *transient* и *volatile*. Эти модификаторы используются для обработки некоторых специальных ситуаций.

Когда экземплярная переменная объявлена как *transient*, то ее значение не будет запомнено при сохранении объекта. Например:

```
class T {  
    transient int a;           // не будет сохраняться  
    int b;                   // будет сохраняться  
}
```

Если бы объект типа *T* записывался в постоянную область памяти, содержимое переменной *a* не было бы сохранено, в то время как содержимое *b* — было бы.

Модификатор *volatile* сообщает компилятору, что переменная, модифицированная с его помощью, может быть неожиданно изменена другими частями вашей программы. Одна из этих ситуаций включает многопоточные программы. (Вы видели пример этого в главе 11.) В многопоточной программе два или несколько потоков иногда совместно используют одну и ту же экземплярную переменную. По соображениям эффективности, каждый поток может хранить свою собственную, частную копию такой разделяемой переменной. Реальная (или главная — *master*-) копия переменной модифицируется в разные моменты времени, например, при входе в синхронизированный метод. Такой подход обычно работает прекрасно, но время от времени может быть неэффективным. В некоторых случаях, все, что действительно имеет значение, так это то, что *master*-копия переменной всегда отражает ее текущее состояние. Для гарантии подобной ситуации просто специфицируют переменную как *volatile*, что сообщает компилятору, что он должен всегда использовать *master*-копию *volatile*-переменной (или, по крайней мере, всегда сохранять любые частные копии, соответствующие текущим значениям главной копии, и наоборот).

### Замечание

Модификатор *volatile* в Java имеет примерно то же значение, что и в C/C++.

## Использование *instanceof*

Иногда полезно распознавать тип объекта во время выполнения. Например, можно иметь один поток выполнения для генерации различных типов объектов, а другой — для их обработки. В этой ситуации обрабатывающему процессу полезно было бы знать тип каждого объекта, принимаемого на обработку. Другая ситуация, в которой знание типа объекта во время выпол-

нения очень важно, — это приведение типов (cast). В Java недопустимое приведение вызывает ошибку времени выполнения. Много недопустимых приведений можно перехватить во время компиляции. Однако операции приведения, связанные с типами объектов (т. е. с иерархиями классов), могут оказаться недопустимыми и могут быть обнаружены только во время выполнения. Например, суперкласс с именем A может иметь два подкласса: B и C. Приведение объектов B или C к типу A — законно, а приведение B-объекта к типу C (или, наоборот, C-объекта к типу B) — нет. Поскольку объект типа A может ссылаться как на объект B, так и на объект C, то каким образом вы можете определить (во время выполнения), на объект какого типа фактически ссылается A-объект перед попыткой приведения его типа к C? Это может быть объект любого типа — A, B или C. Если это — объект типа B, будет выброшено исключение (времени выполнения). Для ответа на этот вопрос Java использует специальную операцию — instanceof. Операция instanceof имеет следующую общую форму:

**object instanceof type**

где **object** — экземпляр класса; **type** — класс (как тип). Если **object**-операнд имеет тип или его тип может быть приведен к типу, указанный в **type**-операнде, то результат операции instanceof имеет значение **true**. Иначе, ее результат — **false**. Таким образом, instanceof — это средство, с помощью которого ваша программа может получить информацию о типе объекта во время выполнения. Следующая программа демонстрирует операцию instanceof:

```
// Демонстрирует операцию instanceof.
```

```
class A {
    int i, j;
}

class B {
    int i, j;
}

class C extends A {
    int k;
}

class D extends A {
    int k;
}

class InstanceOf {
    public static void main(String args[]) {
        A a = new A();
        B b = new B();
        C c = new C();
        D d = new D();

        System.out.println("a instanceof A: " + (a instanceof A));
        System.out.println("a instanceof B: " + (a instanceof B));
        System.out.println("a instanceof C: " + (a instanceof C));
        System.out.println("a instanceof D: " + (a instanceof D));
        System.out.println("b instanceof A: " + (b instanceof A));
        System.out.println("b instanceof B: " + (b instanceof B));
        System.out.println("b instanceof C: " + (b instanceof C));
        System.out.println("b instanceof D: " + (b instanceof D));
        System.out.println("c instanceof A: " + (c instanceof A));
        System.out.println("c instanceof B: " + (c instanceof B));
        System.out.println("c instanceof C: " + (c instanceof C));
        System.out.println("c instanceof D: " + (c instanceof D));
        System.out.println("d instanceof A: " + (d instanceof A));
        System.out.println("d instanceof B: " + (d instanceof B));
        System.out.println("d instanceof C: " + (d instanceof C));
        System.out.println("d instanceof D: " + (d instanceof D));
    }
}
```

<sup>1</sup> Обратите внимание, что instanceof — это двухместная инфиксная логическая операция с двумя operandами. — Примеч. пер.

```
A a = new A();
B b = new B();
C c = new C();
D d = new D();

if(a instanceof A)
    System.out.println("а это экземпляр класса А");
if(b instanceof B)
    System.out.println("б это экземпляр класса В");
if(c instanceof C)
    System.out.println("с это экземпляр класса С");
if(c instanceof A)
    System.out.println("тип с можно привести к типу А");

if(a instanceof C)
    System.out.println("тип а можно привести к типу С");

System.out.println();

// сравнивать производные типы
A ob;

ob = d;           // ссылка на d
System.out.println("об теперь ссылается на d");
if(ob instanceof D)
    System.out.println("об теперь экземпляр класса D");

System.out.println();

ob = c;           // ссылка на с
System.out.println("об теперь ссылается на с");

if(ob instanceof D)
    System.out.println("тип об можно привести к типу D");
else
    System.out.println("тип об нельзя привести к типу D");

if(ob instanceof A)
    System.out.println("тип об можно привести к типу А");

System.out.println();

// все объекты можно привести к типу Object
if(a instanceof Object)
    System.out.println("тип а можно привести к типу Object");
if(b instanceof Object)
    System.out.println("тип б можно привести к типу Object");
if(c instanceof Object)
    System.out.println("тип с можно привести к типу Object");
```

```

if(d instanceof Object)
    System.out.println("тип d можно привести к типу Object");
}
}
}

```

Вывод этой программы:

а теперь экземпляр класса A  
 б теперь экземпляр класса B  
 с теперь экземпляр класса C  
 тип с можно привести к типу A

об теперь ссылается на d  
 об теперь экземпляр класса D

об теперь ссылается на с  
 тип об нельзя привести к типу D  
 тип об можно привести к типу A

тип а можно привести к типу Object  
 тип b можно привести к типу Object  
 тип с можно привести к типу Object  
 тип d можно привести к типу Object

Операция `instanceof` не нужна для большинства программ, потому что, вообще-то, вы знаете тип объекта, с которым работаете. Однако она может быть очень полезна, когда вы пишете обобщенные подпрограммы, работающие на объектах из сложной иерархии классов.

## Ключевое слово `strictfp`

Java 2 добавляет к языку Java новое ключевое слово `strictfp`. С созданием Java 2, модель вычисления с плавающей точкой была слегка ослаблена, чтобы для некоторых процессоров, таких как Pentium, увеличить скорость вычислений с плавающими числами. Новая модель не требует усечения некоторых промежуточных значений, которые появляются во время вычислений. Изменяя класс или метод с помощью `strictfp`, вы гарантируете, что вычисления с плавающей точкой (и таким образом все усечения) выполняются точно так же, как в более ранних версиях Java. Усечение воздействует только на экспоненту некоторых операций. Когда некоторый класс модифицируется с помощью `strictfp`, все методы в этом классе автоматически модифицируются с помощью `strictfp`.

Например, следующий фрагмент сообщает Java, что нужно использовать первоначальную модель с плавающей точкой для вычислений во всех методах, определенных в `MyClass`:

```
strictfp class MyClass { //...
```

Большинство программистов никогда не использует `strictfp`, т. к. эта конструкция решает небольшой круг проблем.

## Native-методы

Хотя это бывает редко и совершенно случайно, но иногда может возникнуть желание вызвать подпрограмму, которая написана на другом языке, а не на Java. Как правило, такая подпрограмма существует как выполняемый код для CPU и среды, в которой вы работаете — то есть как "родной" (native) код. Например, нужно вызвать подпрограмму native-кода для достижения более быстрого времени выполнения. Или нужно использовать специализированную библиотеку типа статистического пакета. Однако из-за того, что Java-программы компилируются в байт-код, который затем интерпретируется (или компилируется "на лету") исполнительной системой Java, казалось бы, невозможно вызвать подпрограмму native-кода изнутри Java-программы. К счастью, это не так. В Java существует ключевое слово `native`, которое используется для объявления методов native-кода. После объявления эти методы можно вызывать внутри Java-программы точно так же, как вызывается любой другой метод Java.

Для объявления native-метода нужно предварить его заголовок модификатором `native`, при этом, однако, не следует определять никакого тела. Например:

```
public native int meth();
```

После объявления native-метода, следует записать сам родной метод и выполнить довольно сложную процедуру для связи его с кодом Java.

Большинство родных методов записываются на С. Механизм, используемый для интеграции С-кода с Java-программой, называется JNI<sup>1</sup>-интерфейсом. Эта методология была создана для Java 1.1 и затем расширена и улучшена в Java 2. (Java 1.0 использовал иной подход, который является теперь полностью устаревшим.) Детальное описание JNI находится за пределами данной книги, но следующее описание обеспечивает достаточную информацию для большинства приложений.

### Замечание

Точные процедуры native-технологии зависят от операционной среды и версий Java. Они также зависят от языка, используемого для реализации native-метода. Следующее обсуждение предполагает среду Windows 95/98/NT. Для реализации native-метода используется язык С.

Лучше всего процесс воспринимается на примере. Для начала, введем следующую короткую программу, которая использует метод `native` с именем `test()`:

<sup>1</sup> JNI (Java Native Interface) — native-интерфейс Java. — Примеч. пер.

```

// Простой пример, который использует native-метод.
public class NativeDemo {
    int i;
    public static void main(String args[]) {
        NativeDemo ob = new NativeDemo();
        ob.i = 10;
        System.out.println("Этот об.i перед native-методом:" + ob.i);
        ob.test();                                // вызов native-метода
        System.out.println("Этот об.i после native-метода:" + ob.i);
    }
    // объявить native-метод
    public native void test();
    // загрузить DLL, который содержит static-метод
    static {
        System.loadLibrary("NativeDemo");
    }
}

```

Заметим, что метод `test()` объявлен как `native` и не имеет тела. Он будет реализован на С. Обратите также внимание на блок `static`. Как объяснялось ранее, `static`-блок выполняется только один раз, когда программа начинает выполняться (или, более точно, когда его класс впервые загружается). В данном случае он используется для загрузки DLL<sup>1</sup>-библиотеки, которая содержит native-реализацию метода `test()`. (Далее вы увидите, как можно создать такую библиотеку.)

Библиотека загружается методом `loadLibrary()`, который является частью класса `System`. Вот его общая форма:

```
static void loadLibrary(String filename)
```

Здесь `filename` — строка, которая специфицирует имя файла, содержащего библиотеку. Для среды Windows 95/98/NT предполагается, что этот файл имеет расширение `.dll`.

После ввода программы, откомпилируйте ее, чтобы получить файл `NativeDemo.class`. Затем, вы должны использовать JDK-утилиту `javadoc.exe` для получения файла С/C++ заголовка `NativeDemo.h`. Файл `NativeDemo.h` нужно включить в реализацию метода `test()`. Для построения `NativeDemo.h` используйте следующую команду:

```
javadoc -jni NativeDemo
```

---

<sup>1</sup> DLL (Dynamic Link Library) — библиотека программ с динамической загрузкой. — *Примеч. перев.*

Данная команда производит файл заголовка с именем NativeDemo.h. Этот файл должен быть включен в С-файл, который реализует test(). Вывод указанной команды:

```
/* НЕ РЕДАКТИРУЙ ЭТОТ ФАЙЛ – он сгенерирован машиной */
#include <jni.h>
/* Заголовок класса NativeDemo */
#ifndef _Included_NativeDemo
#define _Included_NativeDemo
#ifndef __cplusplus
extern "C" {
#endif /* */
* Class:      NativeDemo
* Method:     test
* Signature:  ()V
*/
JNIEXPORT void JNICALL Java_NativeDemo_test
    (JNIEnv *, jobject);

#ifndef __cplusplus
}
#endif
#endif
```

Обратите особое внимание на следующую строку, определяющую прототип функции test(), которую вы будете создавать:

```
JNIEXPORT void JNICALL Java_NativeDemo_test (JNIEnv *, jobject);
```

Заметим, что имя функции — Java\_NativeDemo\_test(). Его нужно использовать как имя native-функции, которую вы реализуете. То есть вместо создания С-функции, названной test(), вы будете создавать функцию с именем Java\_NativeDemo\_test(). Компонент NativeDemo префикса добавляется потому, что он идентифицирует метод test() как часть класса NativeDemo. Помните, что другой класс может определить свой собственный native-метод test(), который полностью отличается от того, что объявлен в NativeDemo. Включение в префикс имени класса обеспечивает возможность дифференцировать различные версии. Общее правило: native-функциям нужно давать имя, чей префикс включает имя класса, в котором они объявлены.

После создания необходимого файла заголовка вы можете написать свою реализацию test() и сохранить его в файле с именем NativeDemo.c:

```
/* Этот файл содержит С-версию метода test(). */

#include <jni.h>
#include "NativeDemo.h"
#include <stdio.h>
```

```

JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *env, jobject obj)
{
    jclass cls;
    jfieldID fid;
    jint i;

    printf("Запуск native-метода.\n");
    cls = (*env)->GetObjectClass(env, obj);
    fid = (*env)->GetFieldID(env, cls, "i", "I");

    if(fid == 0) {
        printf("Не возможно получить id поля.\n");
        return;
    }
    i = (*env)->GetIntField(env, obj, fid);
    printf("i = %d\n", i);
    (*env)->SetIntField(env, obj, fid, 2*i);
    printf("Завершение native-метода.\n");
}

```

Заметим, что файл включает jni.h, который содержит интерфейсную информацию. Этот файл обеспечивается Java-компилятором. Файл заголовка NativeDemo.h был создан ранее с помощью утилиты javah.

В этой функции метод GetObjectClass() используется для получения С-структуры, которая содержит информацию о классе NativeDemo. Метод GetFieldID() возвращает С-структуру с информацией о поле класса с именем "i". Метод GetIntField() извлекает первоначальное значение этого поля. Метод SetIntField() хранит обновленное значение в данном поле. (Дополнительные методы, которые обрабатывают другие типы данных, см. в файле jni.h.)

После создания NativeDemo.c нужно откомпилировать его и создать DLL-файл. Для этого используется Microsoft-компилятор C/C++ со следующей командной строкой:

```
c1 /LD NativeDemo.c
```

Это создает файл с именем NativeDemo.dll. Как только указанная процедура проделана, можно выполнять Java-программу, которая генерирует следующий вывод:

```

Этот ob.i перед native-методом: 10
Запуск native-метода.
i = 10
Завершение native-метода.
Этот ob.i после native-метода: 20

```

### Замечание

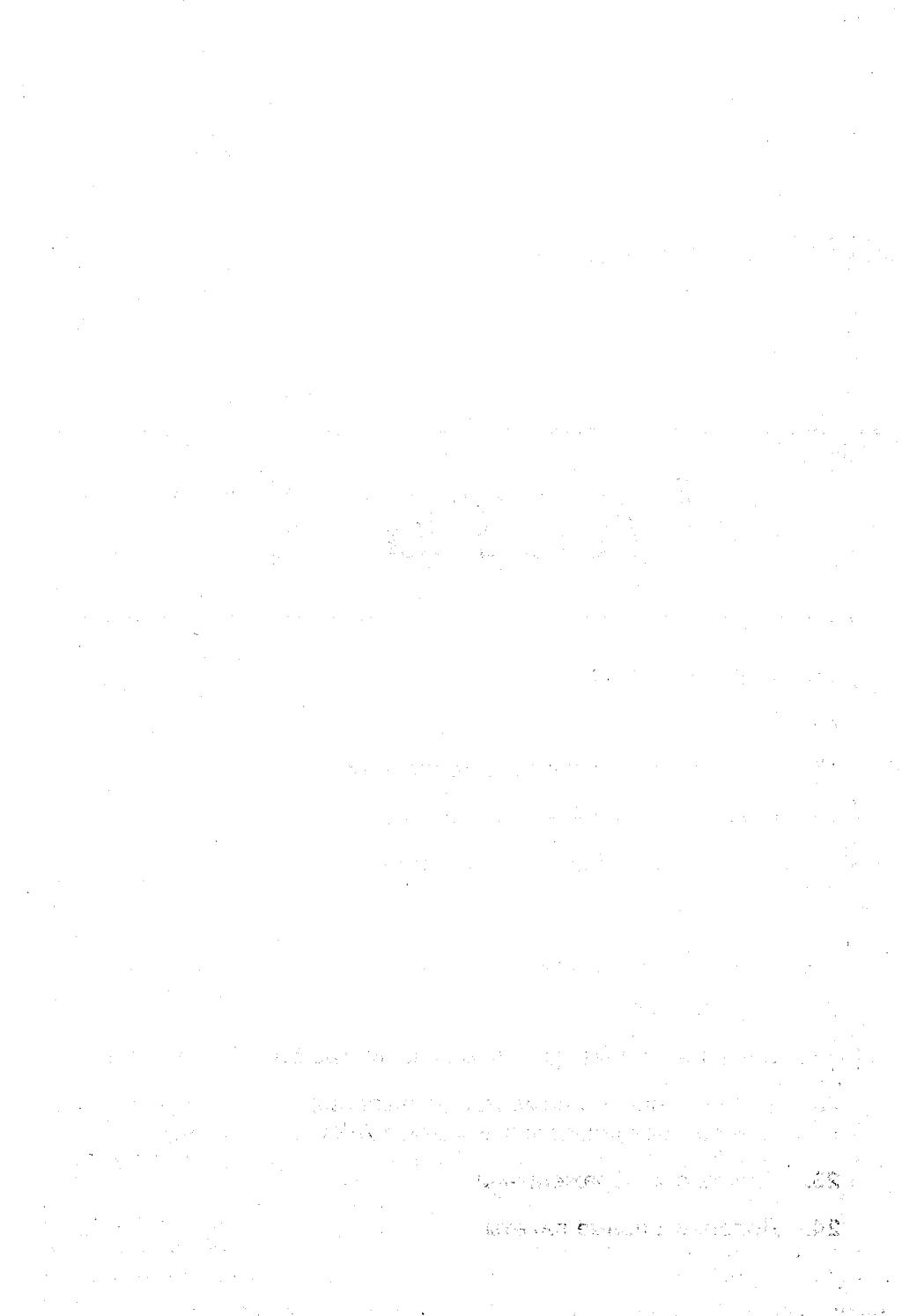
Специфика использования native зависит от реализации и среды. Кроме того, специфика способа взаимодействия с кодом Java подвергается изменениям. По деталям работы с native-методами нужно обращаться к документации системы разработки Java-программ.

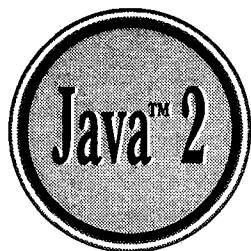
## Проблемы native-методов

Native-методы порождают большие надежды, потому что обеспечивают доступ к существующей базе библиотечных подпрограмм и более быструю работу во время выполнения. Но они порождают также две существенные проблемы:

- **Потенциальный риск безопасности.** Поскольку native-метод выполняет фактический машинный код, он может получать доступ к любой части хост-системы. То есть native-код не ограничен средой выполнения Java. Это может привести к заражению вирусом, например. По этой же причине native-методы не могут использовать апплеты. Загрузка DLL-файлов может быть ограничена и подчинена одобрению руководителя службы безопасности.
- **Потеря мобильности.** Поскольку native-код содержится в DLL-файле, он должен присутствовать на машине, выполняющей программу Java. Далее, так как любой native-метод зависит от CPU и операционной системы, каждый такой DLL-файл неизбежно непереносим. Таким образом, приложение Java, которое использует native-методы, будет способно выполниться только на машине, где был установлен соответствующий DLL-файл.

Использование native-методов должно быть ограничено, потому что они делают ваши Java-программы непереносными и вносят существенный риск защиты.





## БИБЛИОТЕКА JAVA

# ЧАСТЬ II

13. Обработка строк
14. Пакет *java.lang*
15. Пакет *java.util*: структура коллекций
16. Пакет *java.util*: сервисные классы
17. Ввод/вывод: обзор пакета *java.io*
18. Работа в сети
19. Класс *Applet*
20. Обработка событий
21. Введение в AWT: работа с окнами, графикой и текстом
22. Использование элементов управления, менеджеров компоновки и меню AWT
23. Работа с изображениями
24. Дополнительные пакеты





## ГЛАВА 13

# Обработка строк

Краткий обзор этой темы был представлен в главе 7. Здесь приводится ее детальное описание. Как в большинстве других языков программирования, *строка* в Java — это последовательность символов. Но, в отличие от многих языков, которые реализуют строки как *символьные массивы*, в Java строки реализуются как *объекты* типа *String*.

Реализация строк в виде встроенных объектов обеспечивает полный комплект свойств, которые делают обработку строк очень удобной. Например, в Java существуют методы для сравнения двух строк, поиска подстроки, конкатенации (сцепления) двух строк и изменения регистра символов строки. Имеется несколько способов создания *String*-объектов, что облегчает их построение.

Несколько неожиданно, что после создания *String*-объекта символы, входящие в строку, нельзя изменять. Сначала это кажется серьезным ограничением. Однако это не так. Над этим объектом можно выполнять все типы строковых операций. Для того чтобы изменить версию существующей строки, нужно создать новый объект типа *String*, который содержит необходимую модификацию. Исходная строка остается неизменной. Этот подход используется, потому что фиксированные, неизменяемые строки можно реализовать более эффективно, чем изменяемые. Для тех случаев, когда желательна изменяемая строка, существует компаньон класса *String* с именем *StringBuffer*, чьи объекты содержат строки, которые могут изменяться после их создания.

Классы *String* и *StringBuffer* определены в пакете *java.lang*. Таким образом, они доступны всем программам автоматически. Оба объявлены как *final*, что означает, что ни один из этих классов не может иметь подклассов. При этом допускаются некоторые оптимизации, которые увеличивают эффективность строковых операций.

И последнее. Когда говорят, что строки в объектах типа `String` являются неизменяемыми, это означает, что содержимое `String`-объекта не может быть модифицировано после того, как он был создан. Однако переменную, объявленную как `String`-ссылка, можно в любое время изменить так, чтобы она указывала на другой `String`-объект.

## **String-конструкторы**

Класс `String` поддерживает несколько конструкторов. Чтобы создать пустой объект типа `String`, нужно вызвать умалчивающий конструктор. Например, следующий оператор

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

создает экземпляр класса `String`, не содержащий символов (т. е. пустую строку).

Часто необходимо создавать строки с начальными значениями. Для этого в классе `String` определен ряд конструкторов. Чтобы создать `String`-объект, инициализированный массивом символов, используйте следующий конструктор:

```
String(char chars[])
```

Например:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

Этот конструктор инициализирует (объектную) переменную `s` строкой "abc".

В качестве инициализатора можно указать поддиапазон символьного массива, для чего используется следующий конструктор:

```
String(char chars[], int startIndex, int numChars)
```

где `startIndex` определяет индекс<sup>1</sup>, с которого начинается поддиапазон; `numChars` определяет число символов в диапазоне. Например:

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
```

что инициализирует строчный объект `s` символами cde.

<sup>1</sup> Поскольку строки в Java представляются массивами, то для символьных компонентов строки используется терминология массивов, где `индекс` — это просто порядковый номер символа (точнее его позиции) в строке, причем нумерация выполняется как в массиве — с нуля. — Примеч. пер.

## С помощью конструктора

`String(String strObj)`

можно создать `String`-объект, который содержит такую же символьную последовательность, как другой `String`-объект. Здесь `strObj` — объект типа `String`. Рассмотрим следующий пример:

```
// Построение одного String-объекта из другого.
class MakeString {
    public static void main(String args[]) {
        char c[] = {'J', 'a', 'v', 'a'};
        String s1 = new String(c);
        String s2 = new String(s1);

        System.out.println(s1);
        System.out.println(s2);
    }
}
```

Вывод этой программы:

```
Java
Java
```

Как вы видите, `s1` и `s2` содержат одну и ту же строку.

Тип `char` в Java использует 16-разрядное представление символов (из набора Unicode), тогда как в Internet для представления строчных символов используется 8-разрядный набор символов ASCII. Поскольку 8-разрядные строки ASCII используются достаточно широко, в классе `String` имеются конструкторы, которые инициализируют строку 8-разрядными `byte`-массивами. Формы этих конструкторов таковы:

```
String(byte asciiChars[])
String(byte asciiChars[], int startIndex, int numChars)
```

где `asciiChars` указывает байтовый массив. Вторая форма позволяет указать поддиапазон. В каждом из этих конструкторов преобразование байтов в символы использует кодовый набор символов платформы, заданный по умолчанию. Следующая программа иллюстрирует использование этих конструкторов:

```
// Создание строки из подмножества символьного массива.
class SubStringCons {
    public static void main(String args[]) {
        byte ascii[] = {65, 66, 67, 68, 69, 70};

        String s1 = new String(ascii);
        System.out.println(s1);
```

```

String s2 = new String(ascii, 2, 3);
System.out.println(s2);
}
}

```

Эта программа генерирует следующий вывод:

ABCDEF  
CDE

Определены также расширенные версии преобразующих конструкторов, где можно указать иные кодовые наборы для кодирования символов. Однако большую часть времени вы будете использовать кодирование, применяемое платформой по умолчанию.

### Замечание

Всякий раз, когда вы создаете `String`-объект из массива, содержимое массива копируется. Если вы изменяете содержимое массива после того, как создали строку, `String`-объект останется неизменным.

## Длина строки

Длина строки определяется количеством содержащихся в ней символов. Для получения этого значения вызовите метод `length()` в форме:

`int length()`

Следующий фрагмент выведет число 3, т. к. в строке `s` имеется три символа:

```

char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());

```

## Специальные строковые операции

Поскольку работа со строками — обычная и очень важная часть программирования, в синтаксис языка Java добавлена поддержка для некоторых специальных строковых операций. К этим операциям относятся автоматическое создание новых `String`-объектов из строковых литералов, конкатенация множественных `String`-объектов при помощи операции + и преобразование других типов данных в строковое представление. Существуют явные методы для реализации всех этих функций, но Java выполняет их автоматически как для удобства программиста, так и для того, чтобы сделать запись программы более ясной.

## Строковые литералы

Предыдущие примеры показали, как можно явно создавать `String`-объекты из массива символов, используя операцию `new`. Однако есть более простой способ — можно для этой же цели использовать *строковый* литерал. Для каждого строкового литерала в программе Java автоматически создает `String`-объект. Например, следующий кодовый фрагмент создает две эквивалентные строки:

```
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);
String s2 = "abc";           // использование строкового литерала
```

Поскольку объект типа `String` создается для каждого строкового литерала, то этот литерал можно применять в любом месте, где указывается `String`-объект. Например, разрешается вызывать методы, используя строку в кавычках прямо на месте объектной ссылки, как показано в следующем операторе:

```
System.out.println("abc".length());
```

Здесь строчный литерал ("abc") указан на месте, где должна бы была стоять объектная ссылка `s2` из предыдущего фрагмента. Аргумент `"abc".length()` вызывает метод `length()` прямо для строки "abc" (вместо того, чтобы вызвать его для объекта `s2`). Как и ожидается, оператор напечатает число "3".

## Конкатенация строк

Вообще, Java не разрешает применять операции к `String`-объектам. Однако в этом правиле есть одно исключение. Это операция `+`, которая связывает две строки, строя в результате `String`-объект с объединенной последовательностью символов. Можно также организовать цепочку из нескольких `+` операций. Например, следующий фрагмент связывает три строки:

```
String age = "9";
String s = "Ему " + age + " лет.";
System.out.println(s);
```

Здесь происходит конкатенация (сцепление) трех строк, в результате которой на экран выводится строка "Ему 9 лет."

Еще одно практическое использование конкатенации — это создание очень длинных строк. Вместо ввода длинных последовательностей символов в исходный код можно разбить их на меньшие части и использовать цепочку `+` операций для их сцепления. Например:

```
// Использование конкатенации для создания длинных строк.
class ConCat {
    public static void main(String args[]) {
```

```

String longStr = "Это была бы очень длинная строка, " +
    "не удобная ни для ввода, ни для вывода. " +
    "Но конкатенация строк " +
    "устраняет этот недостаток. ";

System.out.println(longStr);
}
}

```

## Конкатенация других типов данных

Операцию конкатенации строк можно использовать с другими типами данных. Например, рассмотрим следующую, немного измененную, версию предыдущего примера:

```

int age = 9;
String s = "Emy " + age + " лет.";
System.out.println(s);

```

В этом случае переменная `age` — имеет `int`-тип, а не `String`, как в предыдущем фрагменте, но вывод — такой же, как прежде. Это потому, что значение переменной `age` автоматически преобразуется в ее строчное представление внутри `String`-объекта и затем склеивается аналогичным способом. Компилятор преобразует операнд операции конкатенации к его строчному эквиваленту всякий раз, когда другой операнд этой операции является экземпляром (объектом) типа `String`.

Однако будьте внимательны, когда смешиваете другие типы операций в выражениях конкатенации строк. Вы можете получить неожиданные результаты. Рассмотрим следующий фрагмент:

```

String s = "четыре: " + 2 + 2;
System.out.println(s);

```

Этот фрагмент выводит на экран:

```

четыре: 22
а не
четыре: 4

```

как вы, вероятно, ожидали. И вот, почему. С учетом старшинства операций сначала выполняется конкатенация первого операнда ("четыре:") со строчным эквивалентом второго ("2"). Этот результат затем склеивается со строчным эквивалентом третьего операнда (тоже "2"). Чтобы выполнить сначала целочисленное сложение, нужно использовать круглые скобки:

```

String s = "четыре: " + (2+2);

```

Теперь `s` содержит строку "четыре: 4".

## Преобразование строк и метод *toString()*

Когда во время конкатенации данные преобразуются в их строчное представление, вызывается одна из перегруженных версий метода преобразования строк `valueOf()`, определенного в классе `String`. `ValueOf()` перегружен для всех простых типов и для типа `Object`. Для простых типов `valueOf()` возвращает строку, которая содержит удобочитаемый эквивалент значения, для которого он вызывается. Для объектов `valueOf()` вызывает метод `toString()`. Мы рассмотрим `valueOf()` подробнее позже (в этой главе). Здесь же познакомимся с методом `toString()`, потому что это средство, с помощью которого вы можете определять строчное представление для объектов тех классов, которые вы создаете в своей программе.

Каждый класс реализует `toString()`, поскольку данный метод определен в классе `Object`. Однако реализации `toString()`, заданной по умолчанию, редко достаточно. Для наиболее важных классов, которые вы создаете сами, нужно переопределять `toString()` и обеспечивать тем самым свои собственные строчные представления объектов. К счастью, это делается достаточно просто. Метод `toString()` имеет следующую общую форму:

```
String toString()
```

Реализация `toString()` просто возвращает объект типа `String`, который содержит удобочитаемую строку, описывающую объект вашего класса.

Переопределяя `toString()` для создаваемых вами классов, вы получаете строчные представления объектов, полностью интегрированные в среду программирования Java. Например, они могут использоваться в операторах `print()` и `println()` и в выражениях конкатенации. Следующая программа демонстрирует это, переопределяя `toString()` для `Box`-классов:

```
// Переопределение toString() для Box-классов.
class Box {
    double width;
    double height;
    double depth;

    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    public String toString() {
        return "Размеры Box-объекта: " + width + " x " +
               depth + " x " + height + ".";
    }
}
```

```

class toStringDemo {
    public static void main(String args[]) {
        Box b = new Box(10, 12, 14);
        String s = "Box b: " + b;           // конкатенация Box-объекта
        System.out.println(b);            // преобразование Box-объекта в строку
        System.out.println(s);
    }
}

```

Вывод этой программы:

```

Размеры Box-объекта: 10 × 14 × 12.
Box b: Размеры Box-объекта: 10 × 14 × 12.

```

Обратите внимание, что метод `toString()` вызывается автоматически, когда Box-объект используется в выражении конкатенации или в обращении к `println()`.

## Извлечение символов

Класс `String` предоставляет несколько способов извлечения символов из объекта типа `String`. Все они рассматриваются в этом разделе. Хотя символы, которые составляют строку `String`-объекта, не могут быть индексированы, как в символьном массиве, многие из `String`-методов используют индекс (порядковый номер или позицию) символа в строке для выполнения своих операций. Подобно массивам, индекс строки начинается с нуля.

### Метод `charAt()`

Для извлечения одиночного символа из `String`-объекта вы можете прямо сослаться на индивидуальный символ через метод `charAt()`. Он имеет следующую общую форму:

```
char charAt(int where)
```

где параметр `where` — индекс (номер) символа, который вы хотите получить. Значение `where` должно определять позицию искомого символа в строке и не может быть отрицательным. `charAt()` возвращает символ, находящийся в указанной позиции строки. Например, фрагмент:

```
char ch;
ch = "abc".charAt(1);
```

назначает символьное значение "b" переменной `ch`.

## Метод *getChars()*

Если нужно извлечь больше одного символа, то можно использовать метод *getChars()*. Он имеет следующую общую форму:

```
void getChars(int sourceStart, int sourceEnd, char target[],  
             int targetStart)
```

Здесь *sourceStart* указывает индекс начала подстроки; *sourceEnd* указывает индекс, который на 1 больше индекса конца желательной подстроки. Таким образом, подстрока содержит символы в позициях от *sourceStart* до *sourceEnd* - 1. Массив, который будет принимать символы, указывается параметром *target[]*. Позиция в *target*, начиная с которой будет скопирована подстрока, передается через параметр *targetStart*. Позаботьтесь, чтобы размер массива *target* был достаточно большим, чтобы вместить все символы указанной подстроки. Следующая программа демонстрирует *getChars()*:

```
class getCharsDemo {  
    public static void main(String args[]) {  
        String s = "This is a demo of the getChars method.";  
        int start = 10;  
        int end = 14;  
        char buf[] = new char[end - start];  
  
        s.getChars(start, end, buf, 0);  
        System.out.println(buf);  
    }  
}
```

Выход этой программы:

```
demo
```

## Метод *getBytes()*

Имеется альтернатива *getChars()*, которая сохраняет символы в массиве байтов. Этот метод называется *getBytes()*. Он выполняет преобразование символов в байты заданное по умолчанию на используемой платформе. Самая простая форма этого метода:

```
byte[] getBytes()
```

Имеются и другие формы *getBytes()*. *getBytes()* наиболее полезен, когда вы экспортируете *String*-значение в среду, которая не поддерживает 16-разрядные символы Unicode. Например, большинство протоколов Internet и форматов текстовых файлов использует 8-разрядную кодировку ASCII для всего текстового обмена.

## Метод `toCharArray()`

Если вы хотите преобразовать все символы в объекте типа `String` в символьный массив, самый простой способ состоит в вызове метода `toCharArray()`. Он возвращает массив символов всей строки и имеет следующую общую форму:

```
char[ ] toCharArray()
```

Эта функция обеспечивает определенные удобства, так как достичь того же результата можно и с помощью метода `getChars()`.

## Сравнение строк

Класс `String` включает несколько методов, которые сравнивают строки или подстроки внутри строк. Все они рассматриваются в данном разделе.

### Методы `equals()` и `equalsIgnoreCase()`

Чтобы сравнивать две строки на равенство, нужно использовать метод `equals()`. Он имеет следующую общую форму:

```
boolean equals(Object str)
```

где `str` — `String`-объект, который сравнивается с вызывающим `String`-объектом. Метод возвращает значение `true`, если строки содержат одни и те же символы в одинаковом порядке, иначе возвращается `false`. Сравнение чувствительно к регистру.

Чтобы выполнить сравнение, которое игнорирует различия в регистре, вызывается метод `equalsIgnoreCase()`. При сравнении двух строк он предполагает, что символы `A-Z` и `a-z` не различаются. Общий формат этого метода:

```
boolean equalsIgnoreCase(String str)
```

где `str` — `String`-объект, который сравнивается с вызывающим `String`-объектом. Он тоже возвращает `true`, если строки содержат одни и те же символы в одном и том же порядке, иначе возвращает `false`.

Пример, который демонстрирует `equals()` и `IgnoreCase()`:

```
// Демонстрирует equals() и equalsIgnoreCase().
class equalsDemo {
    public static void main(String args[]) {
        String s1 = "Hello";
        String s2 = "Hello";
        String s3 = "Good-bye";
        String s4 = "HELLO";
        System.out.println(s1 + " равно " + s2 + " -> " + s1.equals(s2));
    }
}
```

```

        System.out.println(s1 + " равно " + s3 + " -> " + s1.equals(s3));
        System.out.println(s1 + " равно " + s4 + " -> " + s1.equals(s4));
        System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
                           s1.equalsIgnoreCase(s4));
    }
}

```

Вывод этой программы:

```

Hello равно Hello -> true
Hello равно equals Good-bye -> false
Hello равно equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true

```

## Метод *regionMatches()*

Метод *regionMatches()* сравнивает некоторую область внутри строчного объекта с другой некоторой областью в другом строчном объекте. Имеется перегруженная форма, которая позволяет игнорировать регистр при таких сравнениях. Общие формы этих двух методов:

```

boolean regionMatches(int startIndex, String str2,
                      int str2startIndex, int numChars)

boolean regionMatches(boolean ignoreCase,
                      int startIndex, String str2,
                      int str2startIndex, int numChars)

```

Для обеих версий *startIndex* определяет индекс, с которого область начинается в вызывающем *String*-объекте. Сравниваемый *String*-объект указывается параметром *str2*. Индекс, в котором сравнение начнется внутри *str2*, определяется параметром *str2startIndex*. Длина сравниваемой подстроки пересыпается через *numChars*. Во второй версии, если *ignoreCase* — *true*, регистр символов игнорируется. Иначе, регистр учитывается.

## Методы *startsWith()* и *endsWith()*

В классе *String* определены две подпрограммы, которые являются специализированными формами метода *regionMatches()*. Метод *startsWith()* определяет, начинается ли данный *String*-объект с указанной строки. Наоборот, метод *endsWith()* определяет, заканчивается ли *String*-объект указанной строкой. Они имеют следующие общие формы:

```

boolean startsWith(String str)
boolean endsWith(String str)

```

где `str` — проверяемый `String`-объект. Если строки согласованы, возвращается `true`, иначе — `false`. Например,

```
"Foobar".endsWith("bar")
```

и

```
"Foobar".startsWith("Foo")
```

оба возвращают `true`.

Вторая форма `startsWith()` с помощью своего второго параметра (`startIndex`) позволяет определить начальную точку области сравнения (в вызывающем объекте):

```
boolean startsWith(String str, int startIndex)
```

где `startIndex` определяет индекс символа в вызывающей строке, с которого начинается поиск символов для операции сравнения. Например:

```
"Foobar".startsWith("bar", 3)
```

возвращает `true` (потому что строка первого аргумента вызова точно совпадает с подстрокой "Foobar", начинающейся с четвертой<sup>1</sup> позиции в исходной строке).

## Сравнение `equals()` и операции `==`

Важно понять, что метод `equals()` и оператор `==` выполняют две различных операции. Как только что было объяснено, метод `equals()` сравнивает символы внутри `String`-объекта, а оператор `==` — две объектные ссылки, чтобы видеть, обращаются ли они к одному и тому же экземпляру (объекту). Следующая программа показывает, что два различных `String`-объекта могут содержать одни и те же символы, но ссылки на эти объекты не будут оцениваться как равные:

```
// equals() в сравнении с ==
class EqualsNotEqualTo {
    public static void main(String args[]) {
        String s1 = "Hello";
        String s2 = new String(s1);

        System.out.println(s1 + " равен " + s2 + " -> " + s1.equals(s2));
        System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
    }
}
```

Переменная `s1` ссылается на `String`-экземпляр, созданный строкой "Hello". Объект, на который указывает `s2`, создается с объектом `s1` в качестве ини-

---

<sup>1</sup> Не забывайте, что символы в строке индексируются как в массивах — с нуля. -- Примеч. пер.

циализатора. Таким образом, содержимое двух `String`-объектов идентично, но это — разные объекты. Это означает, что `s1` и `s2` не ссылаются на один и тот же объект и, поэтому, при сравнении с помощью операции `==` оказываются не равными, как показывает вывод предыдущего примера:

```
Hello равен Hello -> true
```

```
Hello == Hello -> false
```

## Метод `compareTo()`

Часто, не достаточно просто знать, идентичны ли две строки. Для приложений сортировки нужно знать, какая из них меньше, равна, или больше чем другая. Одна строка считается меньше чем другая, если она расположена *перед* другой в словарном (упорядоченном по алфавиту) списке. Стока считается больше чем другая, если она расположена *после* другой в словарном списке. Такое сравнение и выполняет `String`-метод `compareTo()`. Он имеет следующую общую форму:

```
int compareTo(String str)
```

Здесь `str` — `String`-объект, сравниваемый с вызывающим `String`-объектом. Результат сравнения возвращается (в вызывающую программу) и интерпретируется так:

- Меньше нуля: строка вызова — меньше, чем `str`.
- Больше нуля: строка вызова — больше, чем `str`.
- Нуль: две строки равны.

Ниже показан пример программы, которая сортирует массив строк. Программа использует метод `compareTo()` для упорядочивания строк по алгоритму "пузырьковой сортировки":

```
// Пузырьковая сортировка строк.
class SortString {
    static String arr[] = {
        "Now", "is", "the", "time", "for", "all", "good", "men",
        "to", "come", "to", "the", "aid", "of", "their", "country"
    };
    public static void main(String args[]) {
        for(int j = 0; j < arr.length; j++) {
            for(int i = j + 1; i < arr.length; i++) {
                if(arr[i].compareTo(arr[j]) < 0) {
                    String t = arr[j];
                    arr[j] = arr[i];
                    arr[i] = t;
                }
            }
        }
    }
}
```

```
        System.out.println(arr[j]);
    }
}
}
```

Вывод этой программы — упорядоченный список слов:

```
Now
aid
all
come
country
for
good
is
men
of
the
the
their
time
to
to
```

Как вы видите, `compareTo()` принимает во внимание символы нижнего и верхнего регистра. Слово "Now" идет перед всеми другими словами, потому что оно начинается с прописной буквы, а это означает, что оно имеет более низкое значение в наборе символов ASCII (где коды прописных букв меньше, чем коды строчных).

Если вы хотите игнорировать различия в регистре при сравнении двух строк, используйте метод `compareToIgnoreCase()`, формат которого имеет вид:

```
int compareToIgnoreCase(String str)
```

Данный метод возвращает те же самые результаты, что и `compareTo()`, исключая, однако, различия в регистре. Этот метод был добавлен в Java 2. Если вы попробуете подставить его в предыдущую программу (вместо `compareTo()`), то после выполнения слово "Now" больше не будет первым в списке вывода.

## Поиск строк

Класс `String` предоставляет два метода, которые позволяют выполнять поиск указанного символа или подстроки внутри строки:

- `indexOf()`. Поиск первого вхождения символа или подстроки.
- `lastIndexOf()`. Поиск последнего вхождения символа или подстроки.

Эти два метода имеют несколько различных перегруженных вариантов. Во всех случаях методы возвращают индекс того элемента строки, в котором символ или подстрока были найдены. При неудачном поиске возвращается `-1`.

Для поиска *первого* вхождения символа используйте

```
int indexOf(int ch)
```

Для поиска *последнего* вхождения символа используйте

```
int lastIndexOf(int ch)
```

Здесь `ch` — разыскиваемый символ.

Для поиска первого или последнего вхождения подстроки используйте

```
int indexOf(String str)  
int lastIndexOf(String str)
```

Здесь `str` определяет подстроку.

Можно определить начальную точку поиска, применяя следующие формы:

```
int indexOf(int ch, int startIndex)  
int lastIndexOf(int ch, int startIndex)  
  
int indexOf(String str, int startIndex)  
int lastIndexOf (String str, int startIndex)
```

Здесь `startIndex` указывает индекс (номер) символа, с которого начинается поиск. Для `indexOf()` поиск выполняется от символа с индексом `startIndex` до конца строки. Для `lastIndexOf()` поиск выполняется от символа с индексом `startIndex` до нуля.

Следующий пример показывает, как можно использовать различные индексные методы для поиска внутри `Strings`-объекта:

```
// Демонстрирует indexOf() и lastIndexOf().  
class indexOfDemo {  
    public static void main(String args[]) {  
        String s = "Now is the time for all good men " +  
                  "to come to the aid of their country.";  
  
        System.out.println(s);  
        System.out.println("indexOf(t) = " + s.indexOf('t'));  
        System.out.println("lastIndexOf(t) = " + s.lastIndexOf('t'));  
        System.out.println("indexOf(the) = " + s.indexOf("the"));  
        System.out.println("lastIndexOf(the) = " + s.lastIndexOf("the"));  
        System.out.println("indexOf(t, 10) = " + s.indexOf('t', 10));  
        System.out.println("lastIndexOf(t, 60) = " + s.lastIndexOf('t', 60));  
        System.out.println("indexOf(the, 10) = " + s.indexOf("the", 10));  
    }  
}
```

```

        System.out.println("lastIndexOf(the, 60) = " +
                           s.lastIndexOf("the", 60));
    }
}

```

Вывод этой программы:

```

Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) =11
lastIndexOf(t, 60) =55
indexOf(the, 10) =44
lastIndexOf (the, 60) =55

```

## Изменение строки

Поскольку `String`-объекты неизменяемы, всякий раз, когда вы хотите изменить `String`-объект, нужно или копировать его в `StringBuffer`, или использовать один из следующих `String`-методов, которые создадут новую копию строки с вашими модификациями.

### Метод `substring()`

Вы можете извлечь подстроку с помощью метода `substring()`. Он имеет две формы. Первая:

```
String substring(int startIndex)
```

Здесь `startIndex` специфицирует индекс символа, с которого начнется подстрока. Эта форма возвращает копию подстроки, которая начинается с номера `startIndex` и простирается до конца строки вызова.

Вторая форма `substring()` позволяет указывать как начальный, так и конечный индексы подстроки:

```
String substring(int startIndex, int endIndex)
```

Здесь `startIndex` указывает начальный индекс; `endIndex` определяет индекс последнего символа подстроки. Возвращаемая строка содержит все символы от начального до конечного индекса (но не включая символ с конечным индексом).

Следующая программа использует метод `substring()` для замены всех экземпляров одной подстроки на другую строку:

```
// Замена подстроки.
class StringReplace {
    public static void main(String args[]) {
        String org = "This is a test. This is, too.";
        String search = "is";
        String sub = "was";
        String result = "";
        int i;

        do {                                // заменить все совпадшие подстроки
            System.out.println(org);
            i = org.indexOf(search);
            if(i != -1) {
                result = org.substring(0, i);
                result = result + sub;
                result = result + org.substring(i + search.length());
                org = result;
            }
        } while(i != -1);

    }
}
```

Вывод этой программы:

```
This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.
```

## Метод concat()

Можно склеивать две строки, используя метод concat(), с такой сигнатурой:

```
String concat(String str)
```

Данный метод создает новый объект, включающий строку вызова с содержимым объекта *str*, добавленным в конец этой строки. concat() выполняет ту же функцию, что и операция конкатенации +. Например, фрагмент

```
String s1 = "one";
String s2 = s1.concat("two");
```

Помещает строку "onetwo" в *s2*. Он генерирует тот же результат, что следующая последовательность:

```
String s1 = "one";
String s2 = s1 + "two";
```

## Метод *replace()*

Метод *replace()* заменяет все вхождения одного символа в строке вызова другим символом. Он имеет следующую общую форму:

```
String replace(char original, char replacement)
```

Здесь *original* определяет символ, который будет заменен символом, указанным в *replacement*. Стока, полученная в результате замены, возвращается в вызывающую программу. Например,

```
String s = "Hello".replace('l', 'w');
```

помещает в *s* строку "Hewwo".

## Метод *trim()*

Метод *trim()* возвращает копию строки вызова, из которой удалены любые ведущие и завершающие пробелы. Он имеет следующую общую форму:

```
String trim()
```

Пример:

```
String s = "Hello World ".trim();
```

Этот оператор помещает в строку *s* "Hello World".

Метод *trim()* весьма полезен, когда вы обрабатываете команды пользователя. Например, следующая программа запрашивает у пользователя название штата и затем отображает столицу этого штата. Она использует *trim()* для удаления любых ведущих и завершающих пробелов, которые, возможно, по неосторожности были введены пользователем.

```
// Использование trim() для обработки команд.
import java.io.*;

class UseTrim {
    public static void main(String args[])
        throws IOException
    {
        // создать BufferedReader, использующий System.in
        BufferedReader br = new
            BufferedReader(new InputStreamReader(System.in));
        String str;

        System.out.println("Enter 'stop' to quit.");
        System.out.println("Enter State: ");
        do {
            str = br.readLine();
            str = str.trim();           // удалить пробелы
```

```
if(str.equals("Illinois"))
    System.out.println("Capital is Springfield.");
else if(str.equals("Missouri"))
    System.out.println("Capital is Jefferson City.");
else if(str.equals("California"))
    System.out.println("Capital is Sacramento.");
else if(str.equals("Washington"))
    System.out.println("Capital is Olympia.");
// ...
} while(!str.equals("stop"));
}
```

## Преобразование данных, использующее метод *valueOf()*

Метод *valueOf()* преобразует данные из их внутреннего формата в удобную для чтения форму. Это статический метод, который перегружен в классе *String* для всех встроенных типов Java так, что каждый тип можно преобразовать в строку. *valueOf()* также перегружен для типа *Object*, и любой объект класс-типа, который вы создаете, также может использоваться как аргумент при обращении к этому методу. (Напомним, что *Object* является суперклассом для всех классов.) Имеется несколько форм метода *valueOf()*:

```
static String valueOf(double num)
static String valueOf(long num)
static String valueOf (Object ob)
static String valueOf(char chars[ ])
```

Как мы обсуждали ранее, *valueOf()* вызывается, когда необходимо строчное представление некоторого другого типа данных — например, во время операций конкатенации. Вы можете вызывать этот метод прямо, с любым типом данных в аргументе, и получать разумное строчное представление. Все простые типы преобразуются к их обычному *String*-представлению. Любой объект, который вы передаете в метод *valueOf()*, возвращает результат обращения к методу *toString()*. На самом деле, вы можете прямо вызывать метод *toString()* и получать тот же результат.

Для большинства массивов *valueOf()* возвращает довольно загадочную строку, которая указывает, что это — массив некоторого типа. Для массивов типа *char*, однако, создается *String*-объект, который содержит символы этого массива. Существует специальная версия *valueOf()*, которая позволяет указывать подмножество *char*-массива. Она имеет следующую общую форму:

```
static String valueOf(char chars[ ], int startIndex, int numChars)
```

Здесь `chars` — массив, который содержит символы; `startIndex` — индекс в массиве символов, в котором начинается желательная подстрока; `numChars` указывает длину подстроки.

## Изменение регистра символов в строке

Метод `toLowerCase()` преобразует все символы в строке с верхнего регистра на нижний. Метод `toUpperCase()` преобразует все символы в строке с нижнего регистра на верхний. Неалфавитные символы, типа цифр, остаются незатронутыми. Общие формы этих методов:

```
String toLowerCase()  
String toUpperCase()
```

Оба метода возвращают объект типа `String`, который содержит верхне- или нижнерегистровый эквивалент вызывающего `String`-объекта.

Пример, который использует `toLowerCase()` и `toUpperCase()`:

```
// Демонстрирует toUpperCase() и toLowerCase().
```

```
class ChangeCase {  
    public static void main(String args[])  
    {  
        String s = "Это тест.";  
  
        System.out.println("Оригинал: " + s);  
  
        String upper = s.toUpperCase();  
        String lower = s.toLowerCase();  
  
        System.out.println("Uppercase: " + upper);  
        System.out.println("Lowercase: " + lower);  
    }  
}
```

Вывод, выполненный этой программой:

Оригинал: Это тест.

Uppercase: ЭТО ТЕСТ.

Lowercase: это тест.

## Класс `StringBuffer`

`StringBuffer` — это класс, равный по положению классу `String`. Он обеспечивает много функциональных возможностей для строк. Как вы знаете, `String` представляет неизменяемые символьные последовательности фикси-

рованной длины. Напротив, `StringBuffer` представляет возрастающие и перезаписываемые символьные последовательности. `StringBuffer` может вставлять символы и подстроки в середину строки или добавлять их в конец строки. `StringBuffer` растет автоматически, чтобы создать место для таких добавлений, и часто имеет больше предварительно выделенной памяти, чем фактически необходимо для роста. Java интенсивно использует оба класса, но многие программисты имеют дело только с классом `String` и позволяют Java манипулировать с `StringBuffers` "за сценой", используя перегруженную операцию `+`.

## Конструкторы `StringBuffer`

`StringBuffer` определяет три конструктора:

```
StringBuffer()
StringBuffer(int size)
StringBuffer(String str)
```

Заданный по умолчанию конструктор (без параметров) резервирует участок памяти для шестнадцати дополнительных символов, не участвующих в распределении. Вторая версия принимает целочисленный аргумент, который явно устанавливает размер буфера. Третья версия принимает `String`-аргумент, который устанавливает начальное содержимое объекта типа `StringBuffer` и резервирует участок памяти для еще шестнадцати дополнительных символов. `StringBuffer` распределяет участок памяти для шестнадцати дополнительных символов, когда не затребовано никакой определенной длины буфера, из-за того что перераспределение — довольно дорогостоящий процесс (по затратам времени). Кроме того, частые перераспределения могут фрагментировать память. Распределяя участок памяти для нескольких дополнительных символов, `StringBuffer` сокращает число необходимых перераспределений.

## Методы `length()` и `capacity()`

Текущую длину объекта типа `StringBuffer` можно найти с помощью метода `length()`, а общий распределенный объем — с помощью метода `capacity()`. Они имеют следующие общие формы:

```
int length()
int capacity()
```

Пример:

```
// StringBuffer length() в сравнении с capacity().
class StringBufferDemo {
```

```

public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Hello");

    System.out.println("buffer = " + sb);
    System.out.println("length = " + sb.length());
    System.out.println("capacity = " + sb.capacity());
}
}

```

Вывод этой программы, который показывает, как `StringBuffer` резервирует дополнительное пространство для возможных манипуляций:

```

buffer = Hello
length = 5
capacity = 21

```

Объект `sb` инициализируется строкой "Hello" (в момент создания), его длина равна 5. В то же время полный размер объекта (вместимость, емкость) — 21, т. к. автоматически добавлен участок памяти для шестнадцати дополнительных символов.

## Метод `ensureCapacity()`

Если вы хотите предварительно выделить участок памяти для некоторого числа символов после того, как `StringBuffer` был создан, то для установки размера буфера можно использовать метод `ensureCapacity()`. Это полезно, если вы знаете заранее, что будете добавлять (в конец буфера `StringBuffer`) большое количество маленьких строк. `ensureCapacity()` имеет следующую общую форму:

```
void ensureCapacity(int capacity)
```

где `capacity` определяет общий размер (емкость) буфера.

## Метод `setLength()`

Чтобы устанавливать длину буфера в пределах объекта типа `StringBuffer`, используйте метод `setLength()`. Его общая форма:

```
void setLength(int len)
```

где `len` определяет длину буфера. Это значение должно быть неотрицательным.

Когда вы увеличиваете размер буфера, к концу существующего буфера добавляются нулевые символы. Если вы вызываете `setLength()` со значением меньше чем текущее значение, возвращенное методом `length()`, то символы, хранящиеся вне новой длины, будут потеряны. Программа `setCharAtDemo` следующего раздела использует `setLength()` для сокращения `StringBuffer`.

## Методы `charAt()` и `setCharAt()`

Значение одиночного символа можно получить из `StringBuffer` с помощью метода `charAt()`. Устанавливать значение символа в `StringBuffer` может метод `setCharAt()`. Их общие форматы:

```
char charAt(int where)
void setCharAt(int where, char ch)
```

Параметр `where` в `charAt()` указывает индекс получаемого символа. В `setCharAt()` `where` указывает индекс устанавливаемого символа, а `ch` — новое значение этого символа. Для обоих методов параметр `where` должен быть неотрицательным и не должен определять позицию вне конца буфера.

Следующий пример демонстрирует `charAt()` и `setCharAt()`:

```
// Демонстрирует charAt() и setCharAt().
class setCharAtDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        System.out.println("buffer before = " + sb);
        System.out.println("charAt(1) before = " + sb.charAt(1));
        sb.setCharAt(1, 'i');
        sb.setLength(2);
        System.out.println("buffer after = " + sb);
        System.out.println("charAt(1) after = " + sb.charAt(1));
    }
}
```

Вывод, сгенерированный этой программой:

```
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i
```

## Метод `getChars()`

Для копирования подстроки `StringBuffer` в массив можно использовать метод `getChars()`. Его общая форма:

```
void getChars(int sourceStart, int sourceEnd, char target[],
              int targetStart)
```

где `sourceStart` указывает индекс начала подстроки; `sourceEnd` определяет индекс, на 1 больший, чем индекс конца подстроки. Это означает, что подстрока содержит символы от `sourceStart` до `sourceEnd` - 1. Массив символов указывается параметром `target`. Индекс (номер) позиции в `target`, с кото-

рой будет скопирована подстрока, передается в `targetStart`. Необходимо позаботиться о том, чтобы массив `target` был достаточно большим для размещения всех символов указанной подстроки.

## Метод `append()`

Метод `append()` добавляет строчные представления любого другого типа данных в конец вызывающего объекта типа `StringBuffer`. Он имеет перегруженные версии для всех встроенных типов и для типа `Object`. Имеется несколько его форм:

```
StringBuffer append(String str)
StringBuffer append(int num)
StringBuffer append(Object obj)
```

Чтобы получить строчное представление каждого параметра, вызывается метод `String.valueOf()`. Результат добавляется в конец текущего `StringBuffer`-объекта. Сам буфер возвращается каждой версией `append()`. Это позволяет соединять в цепочку все последовательные вызовы `append()`, как показано в следующем примере:

```
// Демонстрирует append().
class appendDemo {
    public static void main(String args[]) {
        String s;
        int a = 42;
        StringBuffer sb = new StringBuffer(40);

        s = sb.append("a = ").append(a).append("!").toString();
        System.out.println(s);
    }
}
```

Вывод этого примера:

```
a = 42!
```

Чаще всего метод `append()` вызывается, когда на `String`-объектах используется операция `+`. Java автоматически заменяет модификации `String`-объекта на аналогичные операции в объекте типа `StringBuffer`. Таким образом, конкатенация вызывает метод `append()` на `StringBuffer`-объекте. После выполнения конкатенации компилятор вставляет обращение к `toString()`, чтобы вернуть модифицируемый `StringBuffer`-объект обратно в неизменяемый `String`-объект. Все это может показаться необоснованно сложным. Почему нельзя иметь только один строчный класс и сделать его поведение более или менее подобным классу `StringBuffer`? Ответ таков — из-за эффективности. Существует много оптимизационных процедур, которые может

выполнять исполнительная система Java, только зная, что `String`-объекты являются неизменяемыми. К счастью, Java скрывает большую часть сложности преобразования между объектами типа `Strings` и `StringBuffers`. На самом деле, многие программисты никогда не ощутят потребности в прямом использовании `StringBuffer` и будут выражать большинство операций в терминах операции `+` на `String`-переменных.

## Метод `insert()`

Метод `insert()` вставляет одну строку в другую. Он перегружен для приема значений всех простых типов плюс `Strings` и `Objects`. Подобно `append()`, он вызывает метод `String.valueOf()`, чтобы получить строчное представление значения, с которым он вызывается. Затем эта строка вставляется в вызывающий `StringBuffer`-объект. Вот несколько форм этого метода:

```
StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)
```

Параметр `index` указывает индекс (номер позиции) (в вызывающем `StringBuffer`-объекте), с которого будет вставлена строка, символ или объект, указанные во втором параметре.

Следующая программа вставляет "нравится" между "Мне" и "Java":

```
// Демонстрирует insert().
class insertDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Мне Java!");
        sb.insert(2, "нравится ");
        System.out.println(sb);
    }
}
```

Вывод этого примера:

Мне нравится Java!

## Метод `reverse()`

Можно изменить порядок символов в объекте типа `StringBuffer`, используя метод `reverse()` с форматом:

```
StringBuffer reverse()
```

Данный метод возвращает реверсированный (с обратным расположением символов) объект вызова. Следующая программа демонстрирует `reverse()`:

```
// Использование reverse() для реверса StringBuffer-объекта.
class ReverseDemo {
    public static void main(String args[]) {
        StringBuffer s = new StringBuffer("abcdef");
        System.out.println(s);
        s.reverse();
        System.out.println(s);
    }
}
```

Вывод этой программы:

```
abcdef
fedcba
```

## Методы *delete()* и *deleteCharAt()*

В Java 2 классу StringBuffer добавлена способность удалять символы с помощью методов *delete()* и *deleteCharAt()*. Форматы этих методов:

```
StringBuffer delete(int startIndex, int endIndex)
StringBuffer deleteCharAt(int loc)
```

Метод *delete()* удаляет последовательность символов из объекта вызова. Параметр *startIndex* указывает индекс первого удаляемого символа; *endIndex* определяет индекс, на 1 больший, чем у последнего удаляемого. Таким образом, удаляемая подстрока простирается от *startIndex* до *endIndex* - 1. Возвращается StringBuffer-объект, полученный в результате удаления.

Метод *deleteCharAt()* удаляет символ с индексом, указанным в *loc*, и возвращает результирующий StringBuffer-объект.

Программа, которая демонстрирует методы *delete()* и *deleteCharAt()*:

```
// Демонстрирует delete() и deleteCharAt()
class deleteDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Это проверка.");
        System.out.println("До delete: " + sb);
        sb.delete(4, 7);
        System.out.println("После delete: " + sb);
        sb.deleteCharAt(0);
        System.out.println("После deleteCharAt: " + sb);
    }
}
```

Вывод этой программы:

До delete: Это проверка.  
После delete: Это верка.  
После deleteCharAt: то верка.

## Метод *replace()*

Другой новый метод, добавленный к *StringBuffer* в Java 2 — *replace()*. Он заменяет внутри *StringBuffer*-объекта один набор символов другим. Его сигнатура:

```
StringBuffer replace(int startIndex, int endIndex, String str)
```

Заменяемая подстрока указывается индексами *startIndex* и *endIndex*. Таким образом, заменяемая подстрока занимает позиции от *startIndex* до *endIndex* – 1. Заменяющая строка передается через параметр *str*. Модифицированный таким образом *StringBuffer*-объект возвращается в вызывающую программу. Следующая программа демонстрирует *replace()*:

```
// Демонстрирует replace()
class replaceDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Это есть тест.");
        sb.replace(4, 8, "был");
        System.out.println("После replace: " + sb);
    }
}
```

Вывод этой программы:

После replace: Это был тест.

## Метод *substring()*

В Java 2 также добавлен метод *substring()*, который возвращает часть *StringBuffer*-объекта. Он имеет две формы:

```
String substring(int startIndex)
String substring(int startIndex, int endIndex)
```

Первая форма возвращает подстроку, которая начинается в позиции *startIndex* и простирается до конца вызывающего *StringBuffer*-объекта. Вторая форма возвращает подстроку, которая начинается в позиции *startIndex* и простирается до позиции *endIndex* – 1. Перечисленные методы работают точно так же, как аналогичные методы, определенные в классе *String*, которые были описаны ранее.

# ГЛАВА 14



## Пакет *java.lang*

В этой главе обсуждаются классы и интерфейсы пакета `java.lang`. Как вы знаете, `java.lang` автоматически импортируется во все программы. Он содержит классы и интерфейсы, которые являются фундаментальными фактически для всего Java-программирования. Это наиболее широко используемый пакет Java.

`Java.lang` включает следующие классы:

- Boolean
- Byte
- Character
- Class
- ClassLoader
- Compiler
- Double
- Float
- InheritableThreadLocal (Java 2)
- Integer
- Long
- Math
- Number
- Object
- Package (Java 2)
- Process
- Runtime
- RuntimePermission (Java 2)
- SecurityManager
- Short
- String
- StringBuffer
- System
- Thread
- ThreadGroup
- ThreadLocal (Java 2)
- Throwable
- Void

Кроме того, существует два класса, определенных в классе `Character`: `Character.Subset` и `Character.UnicodeBlock`. Они были добавлены в Java 2.

В `Java.lang` также определены следующие интерфейсы:

- Cloneable
- Comparable
- Runnable

Интерфейс `Comparable` был добавлен в Java 2.

Несколько классов `java.lang` содержат исключенные методы, большинство из которых относится к Java 1.0. Эти методы все еще используются в Java 2 для поддержки старых программ, но не рекомендуются для нового кода. Большинство же исключений было сделано еще до Java 2. Подобные методы здесь не обсуждаются. Упоминаются только те исключения, которые произошли в Java 2.

Java 2 добавляет в пакет `java.lang` несколько новых классов и методов (все они в дальнейшем имеют специальные отметки).

## Оболочки простых типов

Как мы упоминали в Части I книги, в Java широко применяются (по причинам их эффективности) простые типы, такие как `int` и `char`. Эти типы данных не являются частью объектной иерархии. Они передаются методам по значению, и их нельзя прямо передавать по ссылке. Для двух методов нет никакого способа сослаться на один и тот же экземпляр типа `int`. Время от времени вам необходимо будет создавать объектное представление одного из этих простых типов. Например, существуют классы перечисления (`enum`eration), обсуждающиеся в главе 15, которые имеют дело только с объектами. Чтобы сохранить простой тип в одном из этих классов, вы должны заключить его в этот класс. Для удовлетворения данной потребности в Java существуют классы, которые соответствуют каждому из простых типов. По существу эти классы инкапсулируют или "обволакивают" (`wrap`) простые типы внутри класса. Их обычно называют *оболочками* простых типов (`type wrappers`).

### Класс `Number`

Абстрактный класс `Number` определяет суперкласс, который реализуется классами-оболочками числовых типов `byte`, `short`, `int`, `long`, `float` и `double`. В классе `Number` существуют абстрактные методы, которые возвращают значения в форме *объекта* каждого числового формата. Например, `doubleValue()` — возвращает значение в форме объекта типа `double`, `floatValue()` — возвращает значение в форме объекта типа `float`, и т. д. Вот сигнатуры этих методов:

```
byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
short shortValue()
```

Значения, возвращаемые перечисленными методами, можно округлять.

Number имеет шесть конкретных подклассов, которые содержат явные значения каждого числового типа, — Double, Float, Byte, Short, Integer и Long.

## Оболочки **Double** и **Float**

Double и Float — это оболочки (т. е. классы) для значений с плавающей точкой типа double и float, соответственно. Конструкторы Float таковы:

```
Float(double num)
Float(float num)
Float(String str) throws NumberFormatException
```

Как вы видите, можно создавать Float-объекты со значениями типа float или double. Их можно также конструировать из строкового представления чисел с плавающей точкой.

Конструкторы Double следующие:

```
Double(double num)
Double(String str) throws NumberFormatException
```

Double-объекты можно создавать либо с числовым значением типа double, либо со строковым значением, содержащим символьное представление числа с плавающей точкой.

Методы оболочки Float представлены в табл. 14.1, а методы оболочки Double — в табл. 14.2. Обе оболочки определяют следующие константы:

- MAX\_VALUE. Максимальное положительное значение.
- MIN\_VALUE. Минимальное положительное значение.
- NaN. Нечисловое значение (Not a Number).
- POSITIVE\_INFINITY. Положительная бесконечность.
- NEGATIVE\_INFINITY. Отрицательная бесконечность.
- TYPE. Объект типа Class для числовых типов float или double.

Таблица 14.1. Методы оболочки *Float*

Метод	Описание
<code>byte byteValue()</code>	Возвращает значение вызывающего объекта, соответствующее типу byte
<code>int compareTo(Float f)</code>	Сравнивает числовое значение вызывающего объекта со значением объекта <i>f</i> . Возвращает 0, если значения равны; отрицательное значение, если вызывающий объект имеет меньшее значение; положительное значение, если вызывающий объект имеет большее значение. (Добавлен в Java 2)

Таблица 14.1 (продолжение)

Метод	Описание
<code>int compareTo(Object obj)</code>	Действует аналогично <code>compareTo(Float)</code> , если <i>obj</i> имеет класс <code>Float</code> . Иначе выбрасывает исключение типа <code>ClassCastException</code> . (Добавлен в Java 2)
<code>double doubleValue()</code>	Возвращает значение вызывающего объекта, соответствующее типу <code>double</code>
<code>boolean equals(Object FloatObj)</code>	Возвращает <code>true</code> , если вызывающий <code>Float</code> -объект эквивалентен объекту, указанному в <i>FloatObj</i> . Иначе — <code>false</code>
<code>static int floatToIntBits(float num)</code>	Возвращает двоичное представление <code>float</code> -значения, указанного в <i>num</i> . Представление совместимо со стандартом IEEE для чисел с одинарной точностью
<code>float floatValue()</code>	Возвращает значение вызывающего объекта, соответствующее типу <code>float</code>
<code>int hashCode()</code>	Возвращает хэш-код <sup>1</sup> вызывающего объекта
<code>static float intBitsToFloat(int num)</code>	Возвращает <code>float</code> -эквивалент двоичного представления <code>int</code> -значения, указанного в <i>num</i> . Представление совместимо со стандартом IEEE для чисел с одинарной точностью
<code>int intValue()</code>	Возвращает значение вызывающего объекта, соответствующее типу <code>int</code>
<code>boolean isInfinite()</code>	Возвращает <code>true</code> , если вызывающий объект содержит бесконечное значение. Иначе — <code>false</code>
<code>static boolean isInfinite(float num)</code>	Возвращает <code>true</code> , если в <i>num</i> указано бесконечное значение. Иначе — <code>false</code>
<code>boolean isNaN()</code>	Возвращает <code>true</code> , если вызывающий объект содержит не числовое значение. Иначе — <code>false</code>

<sup>1</sup> Хэш-код — кодирование методом рандомизации (случайное кодирование). — Примеч. пер.

Таблица 14.1 (окончание)

Метод	Описание
<code>static boolean isNaN(float num)</code>	Возвращает <code>true</code> , если <code>num</code> указывает значение, которое не является числом. Иначе — <code>false</code>
<code>long longValue()</code>	Возвращает значение вызывающего объекта, соответствующее типу <code>long</code>
<code>static float parseFloat(String str) throws NumberFormatException</code>	Возвращает <code>float</code> -эквивалент числа, содержащегося в строке <code>str</code> , используя десятичную систему счисления. (Добавлен в Java 2)
<code>short shortValue()</code>	Возвращает значение вызывающего объекта, соответствующее типу <code>short</code>
<code>String toString()</code>	Возвращает строковый эквивалент вызывающего объекта
<code>static String toString(float num)</code>	Возвращает строковый эквивалент значения, указанного в <code>num</code>
<code>static Float valueOf(String str) throws NumberFormatException</code>	Возвращается <code>Float</code> -объект, который содержит значение, указанное в строке <code>str</code>

Таблица 14.2. Методы оболочки `Double`

Метод	Описание
<code>byte byteValue()</code>	Возвращает значение вызывающего объекта с типом <code>byte</code>
<code>int compareTo(Double d)</code>	Сравнивает числовое значение вызывающего объекта с числовым значением <code>Double</code> объекта <code>d</code> . Возвращает 0, если значения равны; отрицательное значение, если вызывающий объект имеет меньшее значение; положительное значение, если вызывающий объект имеет большее значение. (Добавлен в Java 2)
<code>int compareTo(Object obj)</code>	Действует аналогично <code>compareTo(Double)</code> , если <code>obj</code> имеет класс <code>Double</code> . Иначе, выбрасывает исключение типа <code>ClassCastException</code> . (Добавлен в Java 2)

Таблица 14.2 (продолжение)

Метод	Описание
<code>static long doubleToLongBits(double num)</code>	Возвращает двоичное представление <code>double</code> -значения, указанного в <code>num</code> . Представление совместимо со стандартом IEEE для чисел с двойной точностью
<code>double doubleValue()</code>	Возвращает значение вызывающего объекта с типом <code>double</code>
<code>boolean equals(Object DoubleObj)</code>	Возвращает <code>true</code> , если вызывающий <code>Double</code> объект эквивалентен указанному в <code>DoubleObj</code> . Иначе — <code>false</code>
<code>float floatValue()</code>	Возвращает значение вызывающего объекта с типом <code>float</code>
<code>int hashCode()</code>	Возвращает хэш-код вызывающего объекта
<code>int intValue()</code>	Возвращает значение вызывающего объекта с типом <code>int</code>
<code>boolean isInfinite()</code>	Возвращает <code>true</code> , если вызывающий объект содержит бесконечное значение. Иначе — <code>false</code>
<code>static boolean isInfinite(double num)</code>	Возвращает <code>true</code> , если <code>num</code> определяет бесконечное значение. Иначе — <code>false</code>
<code>boolean isNaN()</code>	Возвращает <code>true</code> , если вызывающий объект содержит не числовое значение. Иначе возвращает <code>false</code>
<code>static boolean isNaN(double num)</code>	Возвращает <code>true</code> , если <code>num</code> указывает значение, которое не является числом. Иначе возвращает <code>false</code>
<code>static double longBitsToDouble(long num)</code>	Возвращает <code>double</code> -эквивалент двоичного представления <code>long</code> -значения, указанного в <code>num</code> . Представление совместимо со стандартом IEEE для чисел с двойной точностью

Таблица 14.2 (окончание)

Метод	Описание
<code>long longValue()</code>	Возвращает значение вызывающего объекта с типом <code>long</code>
<code>static double parseDouble(String str) throws NumberFormatException</code>	Возвращает <code>double</code> -эквивалент числа, содержащегося в строке <code>str</code> , используя десятичную систему счисления. (Добавлен в Java 2)
<code>short shortValue()</code>	Возвращает значение вызывающего объекта с типом <code>short</code>
<code>String toString()</code>	Возвращает строковый эквивалент вызывающего объекта
<code>static String toString(double num)</code>	Возвращает строчный эквивалент <code>num</code>
<code>static Double valueOf(String str) throws NumberFormatException</code>	Возвращается <code>Double</code> -объект, который содержит значение, указанное в строке <code>str</code>

В следующем примере создается два `Double`-объекта: один инициализируется числовым `double`-значением (3.14159), а другой — строчным представлением этого же числа ("314159E-5").

```
class DoubleDemo {
    public static void main(String args[]) {
        Double d1 = new Double(3.14159);
        Double d2 = new Double("314159E-5");

        System.out.println(d1 + " = " + d2 + " -> " + d1.equals(d2));
    }
}
```

Заметьте, что оба конструктора создали идентичные `Double`-экземпляры, как показывает метод `equals()`, возвращающий `true`:

`3.14159 = 3.14159 -> true`

## Методы `isInfinite()` и `isNaN()`

В оболочках `Float` и `Double` определены методы `isInfinite()` и `isNaN()`, которые помогают при управлении двумя специальными значениями типа `double` и `float`. Оба метода проверяют два уникальных значения, определен-

ных техническими спецификациями IEEE<sup>1</sup> для числовых данных с плавающей точкой: бесконечность и NaN (Not a Number — не число). Метод `isInfinite()` возвращает `true`, если проверяемое значение бесконечно велико или мало по величине. Метод `isNaN()` возвращает `true`, если проверяемое значение — не число.

Следующий пример создает два `Double`-объекта; один — бесконечен, а другой — не число:

```
// Демонстрирует методы isInfinite() и isNaN().
class InfNaN {
    public static void main(String args[]) {
        Double d1 = new Double(1/0.);
        Double d2 = new Double(0/0.);

        System.out.println(d1 + ": " + d1.isInfinite() + ", " + d1isNaN());
        System.out.println(d2 + ": " + d2.isInfinite() + ", " + d2.isnan());
    }
}
```

Программа генерирует следующий вывод:

```
Infinity: true, false
NaN: false, true
```

## Оболочки `Byte`, `Short`, `Integer` и `Long`

Классы `Byte`, `Short`, `Integer` и `Long` — оболочки для типов `byte`, `short`, `int` и `long`, соответственно. Их конструкторы таковы:

```
Byte(byte num)
Byte(String str) throws NumberFormatException
Short(short num)
Short(String str) throws NumberFormatException
Integer(int num)
Integer(String str) throws NumberFormatException
Long(long num)
Long(String str) throws NumberFormatException
```

Не трудно видеть, что объекты можно создавать из числовых значений или из строк, которые содержат допустимые полные числовые значения.

Методы, определенные в указанных классах, показаны в табл. 14.3—14.6 и предназначены для синтаксического анализа целых чисел в их строчном представлении и преобразования строк обратно в целые числа. Варианты

---

<sup>1</sup> IEEE (Institute of Electrical and Electronic Engineers) — Институт инженеров по электротехнике и радиоэлектронике. — Примеч. пер.

методов позволяют вам указывать при преобразовании основание системы счисления (иногда называемое *числовой базой*). Обычно используются следующие основания систем счисления: 2 — для двоичных чисел, 8 — для восьмеричных, 10 — для десятичных и 16 — для шестнадцатеричных.

Определены следующие константы:

- `MIN_VALUE`. Минимальное значение.
- `MAX_VALUE`. Максимальное значение.
- `TYPE`. Объект типа `Class byte, short, int или long`.

**Таблица 14.3. Методы оболочки `Byte`**

Метод	Описание
<code>byte byteValue()</code>	Возвращает значение вызывающего объекта с типом <code>byte</code>
<code>int compareTo(Byte b)</code>	Сравнивает числовое значение вызывающего объекта со значением объекта <code>b</code> . Возвращает 0, если значения равны; отрицательное значение, если вызывающий объект имеет меньшее значение; положительное значение, если вызывающий объект имеет большее значение. (Добавлен в Java 2)
<code>int compareTo(Object obj)</code>	Действует аналогично <code>compareTo(Byte)</code> , если <code>obj</code> имеет класс <code>Byte</code> . Иначе, выбрасывает исключение типа <code>ClassCastException</code> . (Добавлен в Java 2)
<code>static Byte decode(String str) throws NumberFormatException</code>	Возвращает <code>Byte</code> -объект, который содержит значение, указанное в строке <code>str</code>
<code>double doubleValue()</code>	Возвращает значение вызывающего объекта типом <code>double</code>
<code>boolean equals(Object ByteObj)</code>	Возвращает <code>true</code> , если вызывающий <code>Byte</code> -объект эквивалентен объекту, указанному в <code>ByteObj</code> . Иначе возвращает <code>false</code>
<code>float floatValue()</code>	Возвращает значение вызывающего объекта с типом <code>float</code>
<code>int hashCode()</code>	Возвращает хэш-код вызывающего объекта

**Таблица 14.3 (окончание)**

Метод	Описание
<code>int intValue()</code>	Возвращает значение вызывающего объекта с типом <code>int</code>
<code>long longValue()</code>	Возвращает значение вызывающего объекта с типом <code>long</code>
<code>static byte parseByte(String str) throws NumberFormatException</code>	Возвращает <code>byte</code> -эквивалент числа, содержащегося в строке <code>str</code> , используя десятичную систему счисления
<code>static byte parseByte(String str, int radix) throws NumberFormatException</code>	Возвращает <code>byte</code> -эквивалент числа, содержащегося в строке <code>str</code> , используя систему счисления <code>radix</code>
<code>short shortValue()</code>	Возвращает значение вызывающего объекта с типом <code>short</code>
<code>String toString()</code>	Возвращает строку, которая содержит десятичный эквивалент вызывающего объекта
<code>static String toString(byte num)</code>	Возвращает строку, которая содержит десятичный эквивалент <code>num</code>
<code>static Byte valueOf(String str) throws NumberFormatException</code>	Возвращается <code>Byte</code> -объект, который содержит значение, указанное в строке <code>str</code> , используя десятичную систему счисления
<code>static Byte valueOf(String str, int radix) throws NumberFormatException</code>	Возвращается <code>Byte</code> -объект, который содержит значение, указанное в строке <code>str</code> , используя основание системы счисления <code>radix</code>

**Таблица 14.4. Методы оболочки `Short`**

Метод	Описание
<code>byte byteValue()</code>	Возвращает значение вызывающего объекта с типом <code>byte</code>
<code>int compareTo(Short s)</code>	Сравнивает числовое значение вызывающего объекта со значением объекта <code>s</code> . Возвращает 0, если значения равны; отрицательное значение, если вызывающий объект имеет меньшее значение; положительное значение, если вызывающий объект имеет большее значение. (Добавлен в Java 2)

Таблица 14.4 (продолжение)

Метод	Описание
<code>int compareTo(Object obj)</code>	Действует аналогично <code>compareTo(Short)</code> , если <code>obj</code> имеет класс <code>Short</code> . Иначе, выбрасывает исключение типа <code>ClassCastException</code> . (Добавлен в Java 2)
<code>static Short decode(String str) throws NumberFormatException</code>	Возвращает <code>Short</code> -объект, который содержит значение, указанное в строке <code>str</code>
<code>double doubleValue()</code>	Возвращает значение вызывающего объекта с типом <code>double</code>
<code>boolean equals(Object ShortObj)</code>	Возвращает <code>true</code> , если вызывающий <code>Byte</code> -объект эквивалентен объекту, указанному в <code>ShortObj</code> . Иначе возвращает <code>false</code>
<code>float floatValue()</code>	Возвращает значение вызывающего объекта с типом <code>float</code> .
<code>int hashCode()</code>	Возвращает хэш-код вызывающего объекта
<code>int intValue()</code>	Возвращает значение вызывающего объекта с типом <code>int</code>
<code>long longValue()</code>	Возвращает значение вызывающего объекта с типом <code>long</code>
<code>static short parseShort(String str) throws NumberFormatException</code>	Возвращает <code>short</code> -эквивалент числа, содержащегося в строке <code>str</code> , используя десятичную систему счисления
<code>static short parseShort(String str,                           int radix) throws NumberFormatException</code>	Возвращает <code>short</code> -эквивалент числа, содержащегося в строке <code>str</code> , используя систему счисления <code>radix</code>
<code>short shortValue()</code>	Возвращает значение вызывающего объекта с типом <code>short</code>
<code>String toString()</code>	Возвращает строку, которая содержит десятичный эквивалент вызывающего объекта
<code>static String toString(short num)</code>	Возвращает строку, которая содержит десятичный эквивалент <code>num</code>
<code>static Short valueOf(String str) throws NumberFormatException</code>	Возвращается <code>Short</code> -объект, который содержит значение, указанное в строке <code>str</code> , используя основание системы счисления 10

Таблица 14.4 (окончание)

Метод	Описание
<pre>static Short valueOf(String str,                      int radix) throws NumberFormatException</pre>	Возвращается Short-объект, который содержит значение, указанное в строке <i>str</i> , используя основание системы счисления <i>radix</i>

Таблица 14.5. Методы оболочки *Integer*

Метод	Описание
<code>byte byteValue()</code>	Возвращает значение вызывающего объекта с типом <code>byte</code>
<code>int compareTo(Integer i)</code>	Сравнивает числовое значение вызывающего объекта со значением объекта <i>i</i> . Возвращает 0, если значения равны; отрицательное значение, если вызывающий объект имеет меньшее значение; положительное значение, если вызывающий объект имеет большее значение. (Добавлен в Java 2)
<code>int compareTo(Object obj)</code>	Действует аналогично <code>compareTo(Integer)</code> , если <i>obj</i> имеет класс <code>Integer</code> . Иначе, выбрасывает исключение типа <code>ClassCastException</code> . (Добавлен в Java 2)
<code>static Integer decode(String str) throws NumberFormatException</code>	Возвращает <code>Integer</code> -объект, который содержит значение, указанное в строке <i>str</i>
<code>double doubleValue()</code>	Возвращает значение вызывающего объекта с типом <code>double</code>
<code>boolean equals(Object IntegerObj)</code>	Возвращает <code>true</code> , если вызывающий <code>Byte</code> -объект эквивалентен объекту, указанному в <i>IntegerObj</i> . Иначе возвращает <code>false</code>
<code>float floatValue()</code>	Возвращает значение вызывающего объекта с типом <code>float</code>
<code>static Integer getInteger (String propertyName)</code>	Возвращает значение, связанное со свойством среди <i>propertyName</i> . При неудаче возвращается <code>null</code>

Таблица 14.5 (продолжение)

Метод	Описание
<code>static Integer getInteger (String propertyName, int default)</code>	Возвращает значение, связанное со свойством среди <i>propertyName</i> . При неудаче возвращается <i>default</i>
<code>static Integer getInteger (String propertyName, Integer default)</code>	Возвращает значение, связанное со свойством среди <i>propertyName</i> . При неудаче возвращается <i>default</i>
<code>int hashCode()</code>	Возвращает хэш-код вызывающего объекта
<code>int intValue()</code>	Возвращает значение вызывающего объекта с типом <code>int</code>
<code>long longValue()</code>	Возвращает значение вызывающего объекта с типом <code>long</code>
<code>static int parseInt(String str) throws NumberFormatException</code>	Возвращает <code>short</code> -эквивалент числа, содержащегося в строке <i>str</i> , используя десятичную систему счисления
<code>static int parseInt(String str, int radix) throws NumberFormatException</code>	Возвращает <code>short</code> -эквивалент числа, содержащегося в строке <i>str</i> , используя систему счисления <i>radix</i>
<code>short shortValue()</code>	Возвращает значение вызывающего объекта с типом <code>short</code>
<code>static String toBinaryString(int num)</code>	Возвращает строку, которая содержит двоичный эквивалент <i>num</i>
<code>static String toHexString(int num)</code>	Возвращает строку, которая содержит шестнадцатеричный эквивалент <i>num</i>
<code>static String toOctalString(int num)</code>	Возвращает строку, которая содержит восьмеричный эквивалент <i>num</i>
<code>String toString()</code>	Возвращает строку, которая содержит десятичный эквивалент вызывающего объекта
<code>static String toString(int num)</code>	Возвращает строку, которая содержит десятичный эквивалент <i>num</i>
<code>static String toString(int num, int radix)</code>	Возвращает строку, которая содержит десятичный эквивалент <i>num</i> , используя основание системы счисления <i>radix</i>

Таблица 14.5 (окончание)

Метод	Описание
<code>static Integer valueOf(String str) throws NumberFormatException</code>	Возвращается <code>Integer</code> -объект, который содержит значение, указанное в строке <code>str</code>
<code>static Integer valueOf(String str,                       int radix) throws NumberFormatException</code>	Возвращается <code>Integer</code> -объект, который содержит значение, указанное в строке <code>str</code> , используя основание системы счисления <code>radix</code>

Таблица 14.6. Методы оболочки `Long`

Метод	Описание
<code>byte byteValue()</code>	Возвращает значение вызывающего объекта с типом <code>byte</code>
<code>int compareTo(Long l)</code>	Сравнивает числовое значение вызывающего объекта со значением объекта <code>l</code> . Возвращает 0, если значения равны; отрицательное значение, если вызывающий объект имеет меньшее значение; положительное значение, если вызывающий объект имеет большее значение. (Добавлен в Java 2)
<code>int compareTo(Object obj)</code>	Действует аналогично <code>compareTo(Long)</code> , если <code>obj</code> имеет класс <code>Long</code> . Иначе, выбрасывает исключение типа <code>ClassCastException</code> . (Добавлен в Java 2)
<code>static Long decode(String str) throws NumberFormatException</code>	Возвращает <code>Long</code> -объект, который содержит значение, указанное в строке <code>str</code>
<code>double doubleValue()</code>	Возвращает значение вызывающего объекта с типом <code>double</code>
<code>boolean equals(Object LongObj)</code>	Возвращает <code>true</code> , если вызывающий <code>long</code> -объект эквивалентен объекту <code>LongObj</code> . Иначе возвращает <code>false</code>
<code>float floatValue()</code>	Возвращает значение вызывающего объекта с типом <code>float</code>
<code>static Long getLong (String propertyName)</code>	Возвращает значение, связанное со свойством среди <code>propertyName</code> . При неудаче возвращает <code>null</code>

Таблица 14.6 (продолжение)

Метод	Описание
<code>static Long getLong (String propertyName, long default)</code>	Возвращает значение, связанное со свойством среди <code>propertyName</code> . При неудаче возвращает <code>default</code>
<code>static Long getLong(String propertyName, Long default)</code>	Возвращает значение, связанное со свойством среди <code>propertyName</code> . При неудаче возвращает <code>default</code>
<code>int hashCode()</code>	Возвращает хэш-код вызывающего объекта
<code>int intValue()</code>	Возвращает значение вызывающего объекта с типом <code>int</code>
<code>long longValue()</code>	Возвращает значение вызывающего объекта с типом <code>long</code>
<code>static long parseLong(String str) throws NumberFormatException</code>	Возвращает <code>long</code> -эквивалент числа, содержащегося в строке <code>str</code> , используя десятичную систему счисления
<code>static long parseLong(String str, int radix) throws NumberFormatException</code>	Возвращает <code>long</code> -эквивалент числа, содержащегося в строке <code>str</code> , используя систему счисления <code>radix</code>
<code>short shortValue()</code>	Возвращает значение вызывающего объекта с типом <code>short</code>
<code>static String toBinaryString (long num)</code>	Возвращает строку, которая содержит двоичный эквивалент <code>num</code>
<code>static String toHexString (long num)</code>	Возвращает строку, которая содержит шестнадцатеричный эквивалент <code>num</code>
<code>static String toOctalString (long num)</code>	Возвращает строку, которая содержит восьмеричный эквивалент <code>num</code>
<code>String toString()</code>	Возвращает строку, которая содержит десятичный эквивалент вызывающего объекта
<code>static String toString(long num)</code>	Возвращает строку, которая содержит десятичный эквивалент <code>num</code>
<code>static String toString(long num, int radix)</code>	Возвращает строку, которая содержит десятичный эквивалент <code>num</code> , используя основание системы счисления <code>radix</code>
<code>static Long valueOf(String str) throws NumberFormatException</code>	Возвращается <code>Long</code> -объект, который содержит значение, указанное в строке <code>str</code>

Таблица 14.6 (окончание)

Метод	Описание
<code>static Long valueOf(String str,                       int radix) throws NumberFormatException</code>	Возвращается Long-объект, который содержит значение, указанное в строке <code>str</code> , используя основание системы счисления <code>radix</code>

## Преобразование чисел в строки и обратно

Одной из наиболее обычных работ в программировании является преобразование строкового представления числа в его внутренний, двоичный формат. К счастью, Java обеспечивает простой способ ее выполнения. В классах `Byte`, `Short`, `Integer` и `Long` определены методы `parseByte()`, `parseShort()`, `parseInt()` и `parseLong()`. Эти методы возвращают `byte`-, `short`-, `int`- или `long`-эквивалент числовой строки, с которой они вызываются. (Подобные методы также существуют для классов `Double` и `Float`.)

Следующая программа демонстрирует метод `parseInt()`. Она суммирует список целых чисел, введенных пользователем. Для чтения целых чисел используется метод `readLine()`, а метод `parseInt()` применяется для конвертирования этих строк в их `int`-эквиваленты.

```
/* Эта программа суммирует список чисел, введенных
пользователем. Она преобразует строчное представление
каждого числа в int-значение, используя метод parseInt(). */

import java.io.*;

class ParseDemo {
    public static void main(String args[])
        throws IOException
    {
        // создать объект BufferedReader, используя System.in
        BufferedReader br = new
            BufferedReader(new InputStreamReader(System.in));
        String str;
        int i;
        int sum=0;

        System.out.println("Введите числа от 0 до quit.");
        do {
            str = br.readLine();
            try {
                i = Integer.parseInt(str);
                sum+=i;
            } catch(NumberFormatException e) {

```

```

        System.out.println("Неправильный формат");
        i = 0;
    }
    sum += i;
    System.out.println("Текущая сумма: " + sum);
} while(i != 0);
}
}

```

Чтобы конвертировать полное число в десятичную строку, используйте версии метода `toString()`, определенные в классах `Byte`, `Short`, `Integer` или `Long`. В классах `Integer` и `Long` определяются также методы `toBinaryString()`, `toHexString()` и `toOctalString()`, которые переводят внутримашинное значение в двоичную, шестнадцатеричную или восьмеричную строку, соответственно.

Следующая программа демонстрирует двоичное, шестнадцатеричное и восьмеричное преобразование:

```

/* Преобразует целые числа в двоичные,
шестнадцатеричные и восьмеричные. */

class StringConversions {
    public static void main(String args[]) {
        int num = 19648;

        System.out.println(num + " в двоичной форме: " +
                           Integer.toBinaryString(num));
        System.out.println(num + " в восьмеричной форме: " +
                           Integer.toOctalString(num));
        System.out.println(num + " в шестнадцатеричной форме: " +
                           Integer.toHexString(num));
    }
}

```

Вывод этой программы:

```

19648 в двоичной форме: 100110011000000
19648 в восьмеричной форме: 46300
19648 в шестнадцатеричной форме: 4cc0

```

## Оболочка *Character*

Класс `Character` — простая оболочка для типа `char`. Конструктор класса `Character` имеет следующую форму:

`Character(char ch)`

где `ch` — символ, который будет "обернут" в создаваемый объект типа `Character`.

Чтобы получить char-значение, содержащееся в Character-объекте, вызовите метод `charValue()`. Сигнатура этого метода:

```
char charValue()
```

Метод возвращает char-значение, т. е. символ.

Класс `Character` определяет несколько констант, включая следующее:

- `MAX_RADIX`. Наибольшее основание.
- `MIN_RADIX`. Наименьшее основание.
- `MAX_VALUE`. Наибольшее значение символа.
- `MIN_VALUE`. Наименьшее значение символа.
- `TYPE`. Class-объект для char-типа.

`Character` включает несколько статических методов, которые указывают категорию символов и изменяют их регистр. Они представлены в табл. 14.7.

**Таблица 14.7. Различные методы оболочки *Character***

Метод	Описание
<code>static boolean isDefined(char ch)</code>	Возвращает true, если <code>ch</code> — символ Unicode. Иначе — false
<code>static boolean isDigit(char ch)</code>	Возвращает true, если <code>ch</code> — цифра. Иначе возвращает false
<code>static boolean isIdentifierIgnorable(char ch)</code>	Возвращает true, если <code>ch</code> должен быть проигнорирован в идентификаторе. Иначе — false
<code>static boolean isISOControl(char ch)</code>	Возвращает true, если <code>ch</code> — управляющий символ ISO <sup>1</sup> . Иначе — false
<code>static boolean isJavaIdentifierPart(char ch)</code>	Возвращает true, если <code>ch</code> является частью Java-идентификатора (но не первым символом). Иначе — false
<code>static boolean isJavaIdentifierStart(char ch)</code>	Возвращает true, если <code>ch</code> является первым символом Java-идентификатора. Иначе — false
<code>static boolean isLetter(char ch)</code>	Возвращает true, если <code>ch</code> — буква. Иначе — false

<sup>1</sup> ISO (International Standards Organization) — Международная организация по стандартам. — Примеч. пер.

Таблица 14.7 (окончание)

Метод	Описание
<code>static boolean isLetterOrDigit(char ch)</code>	Возвращает <code>true</code> , если <code>ch</code> – буква или цифра. Иначе – <code>false</code>
<code>static boolean isLowerCase(char ch)</code>	Возвращает <code>true</code> , если <code>ch</code> – символ нижнего регистра. Иначе – <code>false</code>
<code>static boolean isSpaceChar(char ch)</code>	Возвращает <code>true</code> , если <code>ch</code> – пробельный символ Unicode. Иначе – <code>false</code>
<code>static boolean isTitleCase(char ch)</code>	Возвращает <code>true</code> , если <code>ch</code> – заголовочный символ Unicode. Иначе – <code>false</code>
<code>static boolean isUnicodeIdentifierPart(char ch)</code>	Возвращает <code>true</code> , если <code>ch</code> является частью идентификатора Unicode (но не первым символом). Иначе – <code>false</code>
<code>static boolean isUnicodeIdentifierStart(char ch)</code>	Возвращает <code>true</code> , если <code>ch</code> является первым символом идентификатора Unicode. Иначе – <code>false</code>
<code>static boolean isUpperCase(char ch)</code>	Возвращает <code>true</code> , если <code>ch</code> – прописная буква. Иначе – <code>false</code>
<code>static boolean isWhitespace(char ch)</code>	Возвращает <code>true</code> , если <code>ch</code> – пробельный (whitespace) символ. Иначе – <code>false</code>
<code>static char toLowerCase(char ch)</code>	Возвращает эквивалент <code>ch</code> в нижнем регистре
<code>static char toTitleCase(char ch)</code>	Возвращает заголовочный эквивалент <code>ch</code>
<code>static char toUpperCase(char ch)</code>	Возвращает эквивалент <code>ch</code> в верхнем регистре

Следующий пример демонстрирует некоторые из этих методов:

```
// Демонстрирует некоторые методы Is...
class IsDemo {
    public static void main(String args[]) {
        char a[] = {'a', 'b', '5', '?', 'A', ' '};
        for(int i=0; i<a.length; i++) {
            if(Character.isDigit(a[i]))
                System.out.println(a[i] + " - цифра.");
        }
    }
}
```

```
if(Character.isLetter(a[i]))
    System.out.println(a[i] + " - буква.");
if(Character.isWhitespace(a[i]))
    System.out.println(a[i] + " - пробельный символ.");
if(Character.isUpperCase(a[i]))
    System.out.println(a[i] + " - верхний регистр.");
if(Character.isLowerCase(a[i]))
    System.out.println(a[i] + " - нижний регистр.");
}
}
}
```

Вывод из этой программы:

```
a - буква.
a - нижний регистр.
b - буква.
b - нижний регистр.
5 - цифра.
A - буква.
A - верхний регистр.
- пробельный символ.
```

В *Character* определены методы *forDigit()* и *digit()* со следующим синтаксисом:

```
static char forDigit(int num, int radix)
static int digit(char digit, int radix)
```

Метод *forDigit()* возвращает цифровой символ, связанный со значением *num*. Основание системы счисления преобразования определено в *radix*. Метод *digit()* возвращает целое значение, связанное с указанным символом (который, предположительно, является цифрой) с указанным в *radix* основанием системы счисления.

Другой метод, определенный в классе *Character* — *compareTo()*, который имеет две следующие формы:

```
int compareTo(Character c)
int compareTo(Object obj)
```

Первая форма возвращает 0, если вызывающий объект и *c* имеют одно и то же значение; отрицательное значение, если вызывающий объект имеет более низкое значение; положительное значение — в любом другом случае. Вторая форма работает точно так же, как первая, если *obj* — ссылка на *Character*-объект. Иначе выбрасывается исключение *ClassCastException*. Перечисленные методы были добавлены в Java 2.

В классе *Character* также определены методы *equals()* и *hashCode()*.

Оставшиеся символьные классы — это `Character.Subset`, который используется для описания подмножества `Unicode`, и `Character.UnicodeBlock`, содержащий символьные блоки `Unicode` 2.0.

## Оболочка `Boolean`

`Boolean` — очень тонкая оболочка вокруг `boolean`-значений, которая является полезной главным образом тогда, когда вы хотите передать переменную `boolean` по ссылке. Она содержит константы `TRUE` и `FALSE`, которые обозначают истинные и ложные `Boolean`-объекты. Класс `Boolean` определяет также поле `TYPE`, которое является объектом типа `Class` для `boolean`. `Boolean` определяет следующие конструкторы:

```
Boolean(boolean boolValue)
Boolean(String boolString)
```

В первой версии параметр `boolValue` должен быть или `true`, или `false`. Во второй версии, если параметр `boolString` содержит строку "true" (в верхнем или нижнем регистре), то новый `Boolean`-объект будет `true`. Иначе — `false`.

В `Boolean` определены методы, показанные в табл. 14.8.

Таблица 14.8. Методы оболочки `Boolean`

Метод	Описание
<code>boolean booleanValue()</code>	Возвращает <code>boolean</code> -эквивалент
<code>boolean equals(Object boolObj)</code>	Возвращает <code>true</code> , если вызывающий объект эквивалентен <code>boolObj</code> . Иначе — <code>false</code>
<code>static Boolean getBoolean(String propertyName)</code>	Возвращает <code>true</code> , если системное свойство, указанное в <code>propertyName</code> — <code>true</code> . Иначе — <code>false</code>
<code>int hashCode()</code>	Возвращает хэш-код вызывающего объекта
<code>String toString()</code>	Возвращает эквивалент строки вызывающего объекта
<code>static Boolean valueOf(String boolString)</code>	Возвращает <code>true</code> , если <code>boolString</code> содержит строку "true" (в верхнем или нижнем регистре). Иначе — <code>false</code>

## Класс `Void`

Класс `Void` имеет единственное поле `TYPE`, которое содержит ссылку к объекту типа `Class` для типа `void`. Экземпляры этого класса не создаются.

## Класс *Process*

Абстрактный класс *Process* инкапсулирует *процесс*, т. е. выполняющуюся программу. Он используется прежде всего как суперкласс для типа объектов, создаваемых методами *exec()* в классе *Runtime*, который описан в следующем разделе. *Process* содержит абстрактные методы, показанные в табл. 14.9.

**Таблица 14.9. Абстрактные методы класса *Process***

Метод	Описание
<code>void destroy()</code>	Заканчивает процесс
<code>int exitValue()</code>	Возвращает код завершения, полученный от подпроцесса
<code>InputStream getErrorStream()</code>	Возвращает входной поток, который читает входные элементы <i>err</i> -потока (выходного потока сообщений об ошибках)
<code>InputStream getInputStream()</code>	Возвращает входной поток, который читает входные элементы <i>out</i> -потока (потока вывода)
<code>OutputStream getOutputStream()</code>	Возвращает выходной поток, который записывает вывод в <i>in</i> -поток (поток ввода)
<code>int waitFor() throws InterruptedException</code>	Возвращает код завершения, возвращенный процессом. Данный метод не возвращает своего значения, пока не завершится процесс, на котором он вызывается

## Класс *Runtime*

Класс *Runtime* инкапсулирует исполнительную среду Java (среду времени выполнения). Вы не имеете возможности создавать объект типа *Runtime*. Однако вы можете получить ссылку на текущий *Runtime*-объект, вызывая статический метод *Runtime.getRuntime()*. Получив ссылку, вы сможете вызывать некоторые методы, которые управляют состоянием и поведением виртуальной машины Java (JVM). Апллеты и другие ненадежные коды обычно не могут вызывать какой-либо *Runtime*-метод без того, чтобы возникло исключение типа *SecurityException*.

Методы, определенные в *Runtime*, показаны в табл. 14.10. В Java 2 метод *runFinalizersOnExit()* исключен. Он был добавлен в Java 1.1, но считался нестабильным.

Таблица 14.10. Методы класса Runtime

Метод	Описание
<code>Process exec(String progName) throws IOException</code>	Выполняет программу <i>progName</i> как отдельный процесс. Возвращает объект типа <code>Process</code> , который описывает новый процесс
<code>Process exec(String progName,               String environment[ ]) throws IOException</code>	Выполняет программу <i>progName</i> как отдельный процесс со средой <sup>1</sup> <i>environment</i> . Возвращает объект типа <code>Process</code> , который описывает новый процесс
<code>Process exec(String comLineArray[ ]) throws IOException</code>	Выполняет командные строки <i>comLineArray</i> как отдельный процесс. Возвращает объект типа <code>Process</code> , который описывает новый процесс
<code>Process exec(String comLineArray[ ],               String environment[ ]) throws IOException</code>	Выполняет командные строки <i>comLineArray</i> как отдельный процесс со средой <i>environment</i> . Возвращает объект типа <code>Process</code> , который описывает новый процесс
<code>void exit(int exitCode)</code>	Останавливает выполнение и возвращает значение <i>exitCode</i> (код завершения) родительскому процессу. По соглашению, 0 обозначает нормальное завершение. Все другие значения указывают некоторую форму ошибки
<code>long freeMemory()</code>	Возвращает приблизительное число байтов свободной памяти, доступной исполнительной системе Java
<code>void gc()</code>	Инициализирует сборку "мусора"
<code>static Runtime getRuntime()</code>	Возвращает текущий <code>Runtime</code> -объект
<code>void load(String libraryFileName)</code>	Загружает динамическую библиотеку, чей файл указан в <i>libraryFileName</i> (должно быть указано его полное имя, вместе со спецификацией полного пути в каталоге)

<sup>1</sup> Поскольку параметр *environment[]* определен как массив, под *средой* здесь понимается набор строк, представляющих необходимые переменные среды исполнения. — Примеч. пер.

Таблица 14.10 (окончание)

Метод	Описание
<code>void loadLibrary(String libraryName)</code>	Загружает динамическую библиотеку, чье имя указывается в <code>libraryName</code>
<code>void runFinalization()</code>	Инициализирует вызов методов <code>finalize()</code> для неиспользуемых, но еще обрабатываемых объектов
<code>long totalMemory()</code>	Возвращает общее количество байтов памяти, доступной программе
<code>void traceInstructions (boolean traceOn)</code>	Включает или выключает трассировку команд, в зависимости от значения <code>traceOn</code> . Если <code>traceOn</code> — <code>true</code> , трасса отображается. Если <code>false</code> — трассировка выключена
<code>void traceMethodCalls (boolean traceOn)</code>	Включает или выключает трассировку метода, в зависимости от значения <code>traceOn</code> . Если <code>traceOn</code> — <code>true</code> , трасса отображается. Если <code>false</code> — трассировка выключена

Рассмотрим два наиболее общих применения класса `Runtime`: управление памятью и выполнение дополнительных процессов.

## Управление памятью

Хотя Java обеспечивает автоматическую сборку "мусора", иногда нужно знать, как велик объем объектной динамической памяти (`object heap`), и сколько ее осталось свободной. Вы можете использовать эту информацию, например, для проверки эффективности вашего кода или приблизительной оценки количества создаваемых объектов (некоторого типа). Чтобы получить эти значения, применяйте методы `totalMemory()` и `freeMemory()`.

Как уже упоминалось в Части I, для того чтобы убрать неиспользованные объекты, периодически выполняется сборка мусора Java. Однако иногда необходимо собрать сброшенные объекты до следующих запланированных запусков сборщика. Вы можете запускать сборщик мусора по потребности, вызывая метод `gc()`. Чтобы получить наглядное представление о фактическом использовании динамической памяти, полезно выполнить следующую процедуру. Вызовите метод `gc()`, а затем `freeMemory()`. Потом выполните свой код и снова вызовите `freeMemory()`, чтобы увидеть, сколько памяти ему распределяется. Следующая программа иллюстрирует эту идею.

```
// Демонстрирует использование totalMemory(), freeMemory() и gc().
class MemoryDemo {
    public static void main(String args[]) {
        Runtime r = Runtime.getRuntime();
        long mem1, mem2;
        Integer someints[] = new Integer[1000];

        System.out.println("Полный объем памяти: " + r.totalMemory());

        mem1 = r.freeMemory();
        System.out.println("Начальная свободная память: " + mem1);
        r.gc();
        mem1 = r.freeMemory();
        System.out.println("Свободная память после сборки мусора: " + mem1);

        for(int i=0; i<1000; i++)
            someints[i] = new Integer(i);           // выделение int-памяти

        mem2 = r.freeMemory();
        System.out.println("Свободная память после выделения: " + mem2);
        System.out.println("Выделено памяти: " + (mem1-mem2));

        // сбросить указатели Integer-элементов
        for(int i=0; i<1000; i++) someints[i] = null;

        r.gc();                                // вызов сборщика мусора
        mem2 = r.freeMemory();
        System.out.println("Свободная память после " +
                           "сборки выделенной области: " + mem2);
    }
}
```

Пример вывода этой программы (конечно, ваши фактические результаты могут быть иными):

```
Полный объем памяти: 1048568
Начальная свободная память: 751392
Свободная память после сборки мусора: 841424
Свободная память после выделения: 824000
Выделено памяти: 17424
Свободная память после сборки выделенной области: 842640
```

## Выполнение других программ

В безопасных средах вы можете использовать Java, чтобы выполнять другие "тяжеловесные" процессы (т. е. программы) в вашей многозадачной операционной системе. Некоторые формы метода `exec()` позволяют именовать как программу, которую вы хотите выполнить, так и ее входные параметры. Указанный метод возвращает объект типа `Process`, который можно затем

использовать для контроля за взаимодействием вашей Java-программы с этим новым выполняющимся процессом. Поскольку Java может выполнятьсь на различных платформах и под разными операционными системами, exec() неизбежно зависит от среды.

Следующий пример использует exec() для запуска программы notepad.exe (Блокнота). Этот пример, очевидно, должен выполняться в среде операционной системы Windows.

```
// Демонстрирует метод exec().  
class ExecDemo {  
    public static void main(String args[]) {  
        Runtime r = Runtime.getRuntime();  
        Process p = null;  
  
        try {  
            p = r.exec("notepad");  
        } catch (Exception e) {  
            System.out.println("Ошибка выполнения notepad.");  
        }  
    }  
}
```

Существует несколько альтернативных форм метода exec(), но показанная в примере — наиболее общая. Объектом типа Process, возвращенным из exec(), можно управлять (с помощью Process-методов) уже после того, как новая программа начинает выполняться. Вы можете уничтожить подпроцесс с помощью метода destroy(). Метод waitFor() заставляет вашу программу ждать, пока подпроцесс не закончится. Метод exitValue() возвращает значение, переданное подпроцессом при его завершении. Это обычно 0, если не возникает никаких проблем. Ниже представлен предыдущий пример с exec(), модифицированный так, чтобы ожидать завершения выполняющегося процесса (прежде, чем завершить свою собственную работу):

```
// Ждет, пока notepad не завершится.  
class ExecDemoFini {  
    public static void main(String args[]) {  
        Runtime r = Runtime.getRuntime();  
        Process p = null;  
  
        try {  
            p = r.exec("notepad");  
            p.waitFor();  
        } catch (Exception e) {  
            System.out.println("Ошибка выполнения notepad.");  
        }  
        System.out.println("Notepad завершился " + p.exitValue());  
    }  
}
```

Во время выполнения подпроцесса вы можете писать в его стандартный ввод и читать его стандартный вывод. Методы `getOutputStream()` и `getInputStream()` возвращают дескрипторы стандартных `in`- и `out`-подпроцессов. (Ввод/вывод рассмотрен подробно в главе 17.)

## Класс `System`

Класс `System` содержит совокупность статических методов и переменных. Стандартный ввод, вывод и вывод ошибок исполнительной системы Java хранятся в переменных `in`, `out` и `err`, соответственно. Методы, определенные в `System`, показаны в табл. 14.11. Обратите внимание, что многие из них выбрасывают исключение типа `SecurityException`, если операция не разрешается менеджером безопасности (security manager). Из Java 2 исключен метод `runFinalizersOnExit()`. Этот метод был добавлен в Java 1.1, но определялся как нестабильный.

**Таблица 14.11. Методы класса `System`**

Метод	Описание
<code>static void arraycopy(Object source,                       int sourceStart,                      Object target,                      int targetStart,                      int size)</code>	Копирует массив. Параметры: <code>source</code> – исходный массив, который будет скопирован; <code>sourceStart</code> – индекс (номер) элемента, с которого начнется копирование источника; <code>target</code> – целевой массив, который примет копию; <code>targetStart</code> – индекс элемента целевого массива, с которого начнется копия; <code>size</code> – число копируемых элементов (размер копии)
<code>static long currentTimeMillis()</code>	Возвращает текущее время (в форме количества милли- секунд, прошедших с полуночи 1 января 1970 г.)
<code>static void exit(int exitCode)</code>	Останавливает выполнение и воз- вращает родительскому процессу (обычно операционной системе) значение <code>exitCode</code> (код завер- шения). По соглашению, 0 обоз- начает нормальное завершение. Все другие значения указывают некоторую форму ошибки
<code>static void gc()</code>	Инициализирует сборку "мусора"

Таблица 14.11 (продолжение)

Метод	Описание
<code>static Properties getProperties()</code>	Возвращает свойства, связанные с исполнительной (run-time) системой Java. (Класс <code>Properties</code> описан в главе 15)
<code>static String getProperty(String which)</code>	Возвращает свойство, указанное в <code>which</code> . Возвращает <code>null</code> -объект, если желательное свойство не найдено
<code>static String getProperty(String which, String default)</code>	Возвращает свойство, указанное в <code>which</code> . Возвращает <code>default</code> , если желательное свойство не найдено
<code>static SecurityManager getSecurityManager()</code>	Возвращает текущий менеджер безопасности или <code>null</code> -объект, если никакой менеджер безопасности не установлен
<code>static native int identityHashCode(Object obj)</code>	Возвращает хэш-код тождества (identity hash code) для <code>obj</code>
<code>static void load(String libraryFileName)</code>	Загружает динамическую библиотеку, чей файл указан в <code>libraryFileName</code> (имя файла должно быть указано с полным путем в дереве каталога)
<code>static void loadLibrary (String libraryName)</code>	Загружает динамическую библиотеку, чье имя указывает <code>libraryName</code>
<code>static String mapLibraryName(String lib)</code>	Возвращает специфическое для используемой платформы имя библиотеки, указанной в <code>lib</code> . (Добавлен в Java 2)
<code>static void runFinalization()</code>	Инициализирует вызовы методов <code>finalize()</code> для неиспользуемых, но еще не переработанных сборщиком мусора объектов
<code>static void setErr(PrintStream eStream)</code>	Устанавливает <code>eStream</code> в качестве потока ошибок (err-потока)

Таблица 14.11 (окончание)

Метод	Описание
<code>static void setIn(InputStream iStream)</code>	Устанавливает <code>iStream</code> в качестве стандартного входного потока ( <code>in</code> -потока)
<code>static void setOut(PrintStream oStream)</code>	Устанавливает <code>oStream</code> в качестве стандартного выходного потока ( <code>out</code> -потока)
<code>static void setProperties(Properties sysProperties)</code>	Устанавливает <code>sysProperties</code> в качестве текущих системных свойств
<code>static String getProperty(String which, String v)</code>	Назначает значение <code>v</code> свойству <code>which</code> . (Добавлен в Java 2)
<code>static void setSecurityManager(SecurityManager secMan)</code>	Устанавливает <code>secMan</code> в качестве менеджера безопасности

Рассмотрим некоторые общие возможности использования `System`.

## Использование метода `currentTimeMillis()`

Особый интерес может предоставить использование метода `currentTimeMillis()` для подсчета времени выполнения различных частей вашей программы. Метод `currentTimeMillis()` возвращает текущее время в терминах миллисекунд, прошедших с полуночи 1 января 1970 г. Для подсчета времени выполнения секций вашей программы, сохраните это значение перед самым началом выполнения рассматриваемой секции, а сразу же после завершения ее выполнения вызовите `currentTimeMillis()` снова. Время выполнения секции вычисляется как разность между временем завершения и временем начала ее выполнения. Все это демонстрирует следующая программа.

```
// Вычисляет время выполнения программы.
class Elapsed {
    public static void main(String args[]) {
        long start, end;

        System.out.println("Цикла с шагом от 0 до 1,000,000");
        // время выполнения цикла с шагом от 0 до 1,000,000
        start = System.currentTimeMillis(); // получить время начала
        for(int i=0; i < 1000000; i++);
        end = System.currentTimeMillis(); // получить время конца

        System.out.println("Время выполнения: " + (end-start));
    }
}
```

Пример вывода (помните, что ваши результаты, вероятно, будут отличаться от этих):

Цикл с шагом от 0 до 1,000,000

Время выполнения: 10

## Использование метода *arraycopy()*

Метод *arraycopy()* можно использовать для быстрого копирования массивов любого типа из одного места в другое. Он работает намного быстрее, чем эквивалентный обычный цикл Java. Ниже приведен пример копирования двух массивов с помощью указанного метода. Сначала, а копируется в b, затем все элементы a смещаются вниз на один элемент. Потом b смещается вверх на один элемент.

```
// Использование arraycopy().
class ACDEmo {
    static byte a[] = { 65, 66, 67, 68, 69, 70, 71, 72, 73, 74 };
    static byte b[] = { 77, 77, 77, 77, 77, 77, 77, 77, 77 };

    public static void main(String args[]) {
        System.out.println("a = " + new String(a));
        System.out.println("b = " + new String(b));
        System.arraycopy(a, 0, b, 0, a.length);
        System.out.println("a = " + new String(a));
        System.out.println("b = " + new String(b));
        System.arraycopy(a, 0, a, 1, a.length - 1);
        System.arraycopy(b, 1, b, 0, b.length - 1);
        System.out.println("a = " + new String(a));
        System.out.println("b = " + new String(b));
    }
}
```

Как видно из следующего вывода, можно копировать один и тот же источник и адресат в любом направлении:

```
a = ABCDEFGHIJ
b = MMMMMMM
a = ABCDEFGHIJ
b = ABCDEFGHIJ
a = AABCDEFIGHI
b = BCDEFGHIJJ
```

## Свойства среды

Для любой среды Java 2 доступны следующие свойства:

- |  |   |
|--|---|
| <input type="checkbox"/> file.separator  | <input type="checkbox"/> java.class.version |
| <input type="checkbox"/> java.class.path | <input type="checkbox"/> java.home          |

- java.specification.name
- java.specification.vendor
- java.specification.version
- java.vendor
- java.vendor.url
- java.version
- java.vm.name
- java.vm.specification.name
- java.vm.specification.vendor
- java.vm.specification.version
- java.vm.vendor
- java.vm.version
- line.separator
- os.arch
- os.name
- os.version
- path.separator
- user.dir
- user.home
- user.name

Вы можете получить значения различных переменных среды, вызывая метод `System.getProperty()`. Например, следующая программа отображает путь к текущему каталогу пользователя.

```
class ShowUserDir {
    public static void main(String args[]) {
        System.out.println(System.getProperty("user.dir"));
    }
}
```

## Класс *Object*

Как мы упомянули в Части I, класс `Object` является суперклассом всех других классов. В `Object` определены методы, показанные в табл. 14.12, которые доступны любому объекту.

Таблица 14.12. Методы класса *Object*

Метод	Описание
<code>Object clone() throws CloneNotSupportedException</code>	Создает новый объект, который является точной копией (клоном) вызывающего объекта
<code>boolean equals(Object object)</code>	Возвращает <code>true</code> , если вызывающий объект эквивалентен объекту <code>object</code>
<code>void finalize() throws Throwable</code>	Метод <code>finalize()</code> , действующий по умолчанию. Он обычно переопределется подклассами
<code>final Class getClass()</code>	Получает объект типа <code>Class</code> , который описывает вызывающий объект

Таблица 14.12 (окончание)

Метод	Описание
<code>int hashCode()</code>	Возвращает хэш-код, связанный с вызывающим объектом
<code>final void notify()</code>	Возобновляет выполнение потока, ожидающего на вызывающем объекте
<code>final void notifyAll()</code>	Возобновляет выполнение всех потоков, ожидающих на вызывающем объекте
<code>String toString()</code>	Возвращает строку, которая описывает объект
<code>final void wait() throws InterruptedException</code>	Ждет выполнения на другом потоке
<code>final void wait(long milliseconds) throws InterruptedException</code>	Ждет выполнения на другом потоке указанное число миллисекунд
<code>final void wait(long milliseconds, int nanoseconds) throws InterruptedException</code>	Ждет выполнения на другом потоке указанное число миллисекунд и наносекунд

## Использование метода `clone()` и интерфейса `Cloneable`

Большинство методов, определенных в `Object`, уже обсуждались в других местах этой книги. Однако один из них, `clone()`, заслуживает специального внимания. Метод `clone()` генерирует двойную копию объекта, на котором он вызывается. Клонироваться могут только классы, которые реализуют интерфейс `Cloneable`.

В интерфейсе `Cloneable` никакие члены не определены. Он используется для указания, что для класса разрешено сделать поразрядную копию объекта (т. е. клон). Если вы попробуете вызвать `clone()` для класса, который не реализует интерфейс `Cloneable`, то будет выброшено исключение типа `CloneNotSupportedException`. Когда клон создан, конструктор для клонируемого объекта не вызывается. Клон — просто точная копия оригинала.

Клонирование — потенциально опасное действие, потому что оно может вызывать непреднамеренные побочные эффекты. Например, если имитируемый объект содержит ссылочную переменную с именем `abRef`, то, когда клон сделан, `abRef` в клоне обратится к тому же объекту, что и `abRef` в оригинале. Если клон выполняет изменение содержимого объекта, на кото-

рый ссылается `objRef`, то будет также изменен и оригинальный объект. Другой пример: если объект открывает поток ввода/вывода и затем клонируется, то оба объекта будут способны действовать на одном и том же потоке. Далее, если один из этих объектов закрывает поток, другой мог бы все еще пытаться записывать в него, вызывая ошибку.

Поскольку клонирование может вызывать проблемы, `clone()` объявлен со спецификатором `protected` (защищенный) внутри класса `Object`. Это означает, что он должен или вызваться изнутри метода, определенного классом, который реализует интерфейс `Cloneable`, или он должен быть явно переопределен этим классом так, чтобы стать `public` (общим). Рассмотрим пример каждого подхода.

Следующая программа реализует интерфейс `Cloneable` и определяет метод `cloneTest()`, который вызывает метод `clone()` класса `Object`:

```
// Демонстрирует метод clone().
class TestClone implements Cloneable {
    int a;
    double b;

    // Этот метод вызывает метод clone() класса Object.
    TestClone cloneTest() {
        try {
            // вызвать клон класса Object.
            return (TestClone) super.clone();
        } catch(CloneNotSupportedException e) {
            System.out.println("Клонирование не разрешено.");
            return this;
        }
    }
}

class CloneDemo {
    public static void main(String args[]) {
        TestClone x1 = new TestClone();
        TestClone x2;

        x1.a = 10;
        x1.b = 20.98;

        x2 = x1.cloneTest(); // clone x1

        System.out.println("x1: " + x1.a + " " + x1.b);
        System.out.println("x2: " + x2.a + " " + x2.b);
    }
}
```

Здесь, метод `cloneTest()` вызывает `clone()` класса `Object` и возвращает результат. Обратите внимание, что объект, возвращенный методом `clone()`

должен быть приведен к подходящему типу, как, например, сделано в следующем фрагменте:

```
x2 = (TestClone) x1.clone();
```

Представленный ниже пример программы переопределяет метод `clone()` так, чтобы он мог вызываться извне кода его класса. Для этого его спецификатор доступа должен быть `public`, как показано в следующей программе:

```
// Переопределение метода clone().  
class TestClone implements Cloneable {  
    int a;  
    double b;  
  
    // clone() переопределен как public.  
    public Object clone() {  
        try {  
            // вызов метода clone() класса Object.  
            return super.clone();  
        } catch(CloneNotSupportedException e) {  
            System.out.println("Клонирование не разрешено.");  
            return this;  
        }  
    }  
}  
class CloneDemo2 {  
    public static void main(String args[]) {  
        TestClone x1 = new TestClone();  
        TestClone x2;  
  
        x1.a = 10;  
        x1.b = 20.98;  
  
        // здесь clone() вызывается прямо (не из родительского класса)  
        x2 = (TestClone) x1.clone();  
  
        System.out.println("x1: " + x1.a + " " + x1.b);  
        System.out.println("x2: " + x2.a + " " + x2.b);  
    }  
}
```

Побочные эффекты, вызванные клонированием, иногда трудно заметить с самого начала. Можно подумать, что класс безопасен для клонирования, когда это, фактически, не так. В общем случае, вы не должны реализовывать `Cloneable` для произвольного класса без серьезного основания.

## Класс `Class`

`Class` инкапсулирует состояние объекта или интерфейса во время выполнения программы (так называемое *run-time-состояние*). Объекты типа `Class`

создаются автоматически во время загрузки классов. Вы не можете явно объявлять объект типа `Class`. Но вам позволено получить объект типа `Class`, вызывая метод `getClass()` класса `Object`. Некоторые из обычно используемых методов класса `Class` показаны в табл. 14.13.

**Таблица 14.13. Методы класса `Class`**

Метод	Описание
<code>static Class forName(String name) throws ClassNotFoundException</code>	Возвращает <code>Class</code> -объект по его полному имени, которое метод получает через параметр <code>name</code>
<code>static Class forName(String name, boolean how, ClassLoader ldr) throws ClassNotFoundException</code>	Возвращает <code>Class</code> -объект по его полному имени, которое метод получает через параметр <code>name</code> . Объект загружается, используя загрузчик, указанный в <code>ldr</code> . Если <code>how</code> — <code>true</code> , объект инициализируется, иначе — нет. (Добавлен в Java 2)
<code>Class[ ] getClasses()</code>	Возвращает <code>Class</code> -объект для каждого из общих ( <code>public</code> ) классов и интерфейсов, которые являются членами вызывающего объекта
<code>ClassLoader getClassLoader()</code>	Возвращается <code>ClassLoader</code> -объект, который загрузил класс или интерфейс (речь идет об объекте, используемом для создания вызывающего объекта)
<code>Constructor[ ] getConstructors() throws SecurityException</code>	Возвращает <code>Constructor</code> -объект для всех общих ( <code>public</code> ) конструкторов данного класса
<code>Constructor[ ] getDeclaredConstructors() throws SecurityException</code>	Возвращает <code>Constructor</code> -объект для всех конструкторов, которые объявлены в данном классе
<code>Field[ ] getDeclaredFields() throws SecurityException</code>	Возвращает <code>Field</code> -объект для всех полей, которые объявлены в данном классе
<code>Method[ ] getDeclaredMethods() throws SecurityException</code>	Возвращает <code>Method</code> -объект для всех методов, которые объявлены в данном классе или интерфейсе
<code>Field[ ] getFields() throws SecurityException</code>	Возвращает <code>Field</code> -объект для всех общих ( <code>public</code> ) полей данного класса

Таблица 14.13 (окончание)

Метод	Описание
<code>Class[ ] getInterfaces()</code>	При вызове на объекте этот метод возвращает массив интерфейсов, реализованных класс-типовом данного объекта. При вызове на интерфейсе этот метод возвращает массив интерфейсов, добавленных данным интерфейсом
<code>Method[ ] getMethods() throws SecurityException</code>	Возвращает <code>Method</code> -объект для всех общих ( <code>public</code> ) методов данного класса
<code>String getName()</code>	Возвращает полное имя класса или интерфейса вызывающего объекта
<code>ProtectionDomain getProtectionDomain()</code>	Возвращает домен защиты ( <code>protection domain</code> ), связанный с вызывающим объектом. (Добавлен в Java 2)
<code>Class getSuperclass()</code>	Возвращает суперкласс вызывающего объекта. Возвращает <code>null</code> -значение, если вызывающий объект имеет тип <code>Object</code>
<code>boolean isInterface()</code>	Возвращает <code>true</code> , если вызывающий объект является интерфейсом. Иначе возвращает <code>false</code>
<code>Object newInstance() throws IllegalAccessException, InstantiationException</code>	Создает новый экземпляр (т. е. новый объект), который имеет тот же тип, что и вызывающий объект. Это эквивалентно использованию операции <code>new</code> с заданным по умолчанию конструктором класса. Возвращает новый объект
<code>String toString()</code>	Возвращает строковое представление вызывающего объекта или интерфейса

Методы, определенные в классе `Class`, часто полезны в тех ситуациях, где требуется специальный тип информации об объекте — информация времени выполнения. В табл. 14.13 показаны методы, которые позволяют определять дополнительную информацию относительно специфического класса, такую как его `public`-конструкторы, поля и методы. Она важна для функционирования Java Bean-компонентов, которые обсуждаются позже в этой книге.

Представленная ниже программа демонстрирует методы `getClass()` (наследуется из класса `Object`) и `getSuperclass()` (из `Class`):

```
// Демонстрирует информацию времени выполнения.
class X {
    int a;
    float b;
}
class Y extends X {
    double c;
}
class RTTI {
    public static void main(String args[]) {
        X x = new X();
        Y y = new Y();
        Class clObj;

        clObj = x.getClass();           // получить ссылку на Class-объект
        System.out.println("x это объект типа: " + clObj.getName());
        clObj = y.getClass();           // получить ссылку на Class-объект
        System.out.println("y это объект типа: " + clObj.getName());
        clObj = clObj.getSuperclass();
        System.out.println("Суперклассом для у является " + clObj.getName());
    }
}
```

Вывод из этой программы:

```
x это объект типа: X
y это объект типа: Y
Суперклассом для у является X
```

## Класс *ClassLoader*

Абстрактный класс *ClassLoader* определяет, как загружаются классы. Ваше приложение может создавать подклассы, расширяющие *ClassLoader*, реализуя его методы. Это позволяет загружать ваши классы несколько иным способом, чем при обычной их загрузке исполнительной системой Java. Некоторые методы класса *ClassLoader* показаны в табл. 14.14.

**Таблица 14.14.** Некоторые методы класса *ClassLoader*

Метод	Описание
<code>final Class defineClass(String str,                            byte b[],                            int index,                            int numBytes) throws ClassFormatError</code>	Возвращает Class-объект. Параметры: <i>str</i> – имя класса; <i>b</i> – объект (в форме byte-массива); <i>index</i> – индекс (номер) элемента в массиве, с которого начинается объект;

Таблица 14.14 (окончание)

Метод	Описание
(прод.)	<i>numBytes</i> — количество байт в объектной части массива. Данные в <i>b</i> должны представлять правильный объект
<code>final Class findSystemClass (String name) throws ClassNotFoundException</code>	Возвращает объект типа <i>Class</i> ( <i>name</i> — задает его имя)
<code>abstract Class loadClass (String name, boolean callResolveClass) throws ClassNotFoundException</code>	Реализация этого абстрактного метода должна загрузить класс, имя которого задано в <i>name</i> , и вызвать метод <i>resolveClass()</i> , если <i>callResolveClass</i> указан как <i>true</i>
<code>final void resolveClass(Class obj)</code>	Класс, на который ссылается объект <i>obj</i> , разрешается <sup>1</sup> (т. е. его имя вводится в пространство имен классов)

## Класс *Math*

Класс *Math* содержит все функции с плавающей точкой, которые применяются в геометрии и тригонометрии, а также несколько универсальных методов. В *Math* определены две константы типа *double*: *E* (приблизительно 2.72) и *PI* (приблизительно 3.14).

## Трансцендентные функции

В табл. 14.15 представлены три метода, которые принимают параметр типа *double* (угол в радианах) и возвращают результат соответствующей трансцендентной функции.

Таблица 14.15. Методы трансцендентных функций

Метод	Описание
<code>static double sin(double arg)</code>	Возвращает синус угла, указанного параметром <i>arg</i> (в радианах)

<sup>1</sup> От англ. глагола *resolve* — разрешать, сводить к чему-либо. — Примеч. пер.

Таблица 14.15 (окончание)

Метод	Описание
<code>static double cos(double arg)</code>	Возвращает косинус угла, указанного параметром <code>arg</code> (в радианах)
<code>static double tan(double arg)</code>	Возвращает тангенс угла, указанного параметром <code>arg</code> (в радианах)

В табл. 14.16 показаны методы, получающие (через параметр) результат трансцендентной функции и возвращающие угол (в радианах), который произвел бы этот результат. Они являются инверсией предыдущих методов.

Таблица 14.16. Дополнительные методы трансцендентных функций

Метод	Описание
<code>static double asin(double arg)</code>	Возвращает угол, чей синус указан параметром <code>arg</code>
<code>static double acos(double arg)</code>	Возвращает угол, чей косинус указан параметром <code>arg</code>
<code>static double atan(double arg)</code>	Возвращает угол, чей тангенс указан параметром <code>arg</code>
<code>static double atan2(double x, double y)</code>	Возвращает угол, чей тангенс равен <code>x/y</code> .

## Экспоненциальные функции

В Math определены экспоненциальные (показательные) методы, представленные в табл. 14.17.

Таблица 14.17. Экспоненциальные методы

Метод	Описание
<code>static double exp(double arg)</code>	Возвращает значение функции $e^{arg}$ (где $e = 2.72$ , основание натурального логарифма)
<code>static double log(double arg)</code>	Возвращает значение натурального логарифма $\ln(arg)$

Таблица 14.17 (окончание)

Метод	Описание
<code>static double pow(double y, double x)</code>	Возвращается значение степени $y^x$ , например, <code>pow(2.0, 3.0)</code> возвращает 8.0
<code>static double sqrt(double arg)</code>	Возвращает значение квадратного корня из <code>arg</code> (т. е. $arg^{1/2}$ )

## Округление функций

Класс Math определяет несколько методов, которые обеспечивают различные типы операций округления. Они перечислены в табл. 14.18.

Таблица 14.18. Округляющие методы класса Math

Методы	Описание
<code>static int abs(int arg)</code>	Возвращает абсолютное значение <code>arg</code>
<code>static long abs(long arg)</code>	Возвращает абсолютное значение <code>arg</code>
<code>static float abs(float arg)</code>	Возвращает абсолютное значение <code>arg</code>
<code>static double abs(double arg)</code>	Возвращает абсолютное значение <code>arg</code>
<code>static double ceil(double arg)</code>	Возвращает наименьшее целое число, большее или равное <code>arg</code>
<code>static double floor(double arg)</code>	Возвращает наибольшее целое число, меньше или равное <code>arg</code>
<code>static int max(int x, int y)</code>	Возвращает максимум из <code>x</code> и <code>y</code>
<code>static long max(long x, long y)</code>	Возвращает максимум из <code>x</code> и <code>y</code>
<code>static float max(float x, float y)</code>	Возвращает максимум из <code>x</code> и <code>y</code>
<code>static double max(double x, double y)</code>	Возвращает максимум из <code>x</code> и <code>y</code>
<code>static int min(int x, int y)</code>	Возвращает минимум из <code>x</code> и <code>y</code>
<code>static long min(long x, long y)</code>	Возвращает минимум из <code>x</code> и <code>y</code>
<code>static float min(float x, float y)</code>	Возвращает минимум из <code>x</code> и <code>y</code>
<code>static double min(double x, double y)</code>	Возвращает минимум из <code>x</code> и <code>y</code>

Таблица 14.18 (окончание)

Методы	Описание
<code>static double rint(double arg)</code>	Возвращает ближайшее целое значение <code>arg</code>
<code>static int round(float arg)</code>	Возвращает <code>arg</code> , округленный до ближайшего целого <code>int</code> -значения
<code>static long round(double arg)</code>	Возвращает <code>arg</code> , округленный до ближайшего целого <code>long</code> -значения

## Разные методы класса Math

В дополнение к только что показанным, в `Math` определены следующие методы:

```
static double IEEEremainder(double dividend, double divisor)
static double random()
static double toRadians(double angle)
static double toDegrees(double angle)
```

Метод `IEEEremainder()` возвращает остаток от деления `dividend/divisor`. Метод `random()` предназначен для получения псевдослучайного числа из промежутка от 0 до 1. Класс `Random` используется для генерации случайных чисел. Метод `toRadians()` конвертирует градусы в радианы. `toDegrees()` переводит радианы в градусы. Два последних метода были добавлены в Java 2.

Программа, демонстрирующая методы `toRadians()` и `toDegrees()`, такова:

```
// Демонстрирует методы toDegrees() и toRadians().
class Angles {
    public static void main(String args[]) {
        double theta = 120.0;

        System.out.println(theta + " градусов это " +
                           Math.toRadians(theta) + " радиан.");
        theta = 1.312;
        System.out.println(theta + " радиан это " +
                           Math.toDegrees(theta) + " градусов.");
    }
}
```

Вывод этой программы:

120.0 градусов это 2.0943951023931953 радиан.

1.312 радиан это 75.17206272116401 градусов.

## Компилятор

Класс *Compiler* поддерживает создание Java-сред, в которых байт-код Java компилируется (не интерпретируется) в выполняемый код. Для нормального программирования он не используется.

## Классы *Thread*, *ThreadGroup* и интерфейс *Runnable*

Интерфейс *Runnable* и классы *Thread* и *ThreadGroup* поддерживают многопоточное программирование. Рассмотрим каждый из них.

### Замечание

Краткий обзор методик управления потоками, реализации интерфейса *Runnable* и создания многопоточных программ представлен в главе 11.

### Интерфейс *Runnable*

Интерфейс *Runnable* должен быть реализован любым классом, который будет инициализировать отдельный поток выполнения. *Runnable* определяет только один абстрактный метод — *run()*, который является точкой входа в поток. Он определен так:

```
abstract void run()
```

Создаваемые вами потоки должны реализовать данный метод.

### Класс *Thread*

Класс *Thread* создает новый поток выполнения. В нем определены следующие конструкторы:

```
Thread()
Thread(Runnable threadOb)
Thread(Runnable threadOb, String threadName)
Thread(String threadName)
Thread(ThreadGroup groupOb, Runnable threadOb)
Thread(ThreadGroup groupOb, Runnable threadOb, String threadName)
Thread(ThreadGroup groupOb, String threadName)
```

где *threadOb* — экземпляр класса, который реализует интерфейс *Runnable* и определяет, где будет начинаться выполнение потока. Имя потока передается через *threadName*. Если имя не указывается, то его создает JVM (виртуальная Java-машина). *GroupOb* обозначает группу, к которой будет принад-

лежать новый поток. Когда поточная группа не определена, новый поток принадлежит той же самой группе, что и порождающий поток.

В Thread определены следующие константы:

- MAX\_PRIORITY;
- MIN\_PRIORITY;
- NORM\_PRIORITY.

Эти константы указывают максимальный, минимальный и умалчиваемый приоритеты потоков.

Методы класса Thread перечислены в табл. 14.19. В прошлых версиях Java (до 2) Thread включал также методы stop(), suspend() и resume(). Однако, как объяснялось в главе 11, они были исключены в Java 2, потому что оказались неизбежно нестабильными. В Java 2 исключен также и метод countStackFrames(), т. к. он вызывает suspend().

**Таблица 14.19. Методы класса Thread**

Метод	Описание
<code>static int activeCount()</code>	Возвращает число потоков в группе, которой принадлежит поток
<code>void checkAccess()</code>	Заставляет менеджер безопасности проверять, может ли текущий поток иметь доступ и/или изменять поток, на котором вызывается checkAccess()
<code>static Thread currentThread()</code>	Возвращает Thread-объект, который инкапсулирует поток, вызывающий этот метод
<code>void destroy()</code>	Завершает поток
<code>static void dumpStack()</code>	Отображает стек вызовов для потока
<code>static int enumerate (Thread threads[ ])</code>	Помещает копии всех Thread-объектов в текущей группе потоков в массив <code>threads</code> . Возвращает число потоков
<code>ClassLoader getContextClassLoader()</code>	Возвращает загрузчик класса, который используется для загрузки классов и ресурсов для данного потока. (Добавлен в Java 2)
<code>final String getName()</code>	Возвращает имя потока
<code>final int getPriority()</code>	Возвращает установку поточного приоритета

Таблица 14.19 (продолжение)

Метод	Описание
<code>final ThreadGroup getThreadGroup()</code>	Возвращается ThreadGroup-объект, членом которого является вызывающий поток
<code>void interrupt()</code>	Прерывает поток
<code>static boolean interrupted()</code>	Возвращает true, если выполняющийся в настоящее время поток был спланирован для прерывания. Иначе — false
<code>final boolean isAlive()</code>	Возвращает true, если поток все еще активен. Иначе — false
<code>final boolean isDaemon()</code>	Возвращает true, если поток является "демоническим" (daemon thread), т. е. потоком, который является частью исполнительной системы Java. Иначе возвращает false
<code>boolean isInterrupted()</code>	Возвращает true, если поток прерван. Иначе возвращает false
<code>final void join() throws InterruptedException</code>	Ждет завершения потока
<code>final void join(long milliseconds) throws InterruptedException</code>	Ждет указанное число миллисекунд завершения потока, на котором он вызывается
<code>final void join(long milliseconds, int nanoseconds) throws InterruptedException</code>	Ждет указанное число миллисекунд плюс наносекунд завершения потока, на котором он вызывается
<code>void run()</code>	Начинает выполнение потока
<code>void setContextClassLoader (ClassLoader cl)</code>	Устанавливает <i>cl</i> в качестве загрузчика класса, который будет использоваться вызывающим потоком. (Добавлен в Java 2)
<code>final void setDaemon(boolean state)</code>	Устанавливает daemon-состояние потока (если <i>state</i> — true, то поток становится "демоническим", т. е. частью исполнительной системы Java)
<code>final void setName (String threadName)</code>	Устанавливает <i>threadName</i> в качестве имени потока

Таблица 14.19 (окончание)

Метод	Описание
<code>final void setPriority(int priority)</code>	Устанавливает приоритет потока ( <code>priority</code> указывает значение приоритета, которое вы хотите установить)
<code>static void sleep(long milliseconds) throws InterruptedException</code>	Приостанавливает выполнение потока на указанное (параметром <code>milliseconds</code> ) число миллисекунд
<code>static void sleep(long milliseconds, int nanoseconds) throws InterruptedException</code>	Приостанавливает выполнение потока на указанное (параметром <code>milliseconds</code> ) число миллисекунд и наносекунд (указанных в <code>nanoseconds</code> )
<code>void start()</code>	Начинает выполнение потока
<code>String toString()</code>	Возвращает строчный эквивалент потока
<code>static void yield()</code>	Вызывающий поток уступает CPU другому потоку

## Класс *ThreadGroup*

*ThreadGroup* создает группу потоков. В нем определяются два конструктора:

`ThreadGroup(String groupName)`

`ThreadGroup(ThreadGroup parentObj, String groupName)`

Для обеих форм `groupName` указывает имя группы потоков. Первая версия создает новую группу, которая имеет текущий поток в качестве ее родителя. Во второй форме родитель указывается через параметр `parentObj`.

Методы класса *ThreadGroup* представлены в табл. 14.20. В старых версиях Java (до 2) *ThreadGroup* также включал методы `stop()`, `ususpend()` и `resume()`. В Java 2 они исключены, т. к. оказались нестабильными.

Таблица 14.20. Методы класса *ThreadGroup*

Метод	Описание
<code>int activeCount()</code>	Возвращает число потоков в группе плюс любые группы, для которых данный поток является родителем

Таблица 14.20 (продолжение)

Метод	Описание
<code>int activeGroupCount()</code>	Возвращает число групп, для которых вызывающий поток является родителем
<code>final void checkAccess()</code>	Заставляет менеджер безопасности проверять, может ли текущий поток иметь доступ и/или изменять группу потоков, на которой вызывается <code>checkAccess()</code>
<code>final void destroy()</code>	Уничтожает группу потоков (и любые порожденные группы), на которых он вызывается
<code>int enumerate(Thread group[ ])</code>	Потоки, которые составляют группу вызывающих потоков, помещаются в массив <code>group</code>
<code>int enumerate(Thread group[ ], boolean all)</code>	Потоки, которые составляют группу вызывающих потоков, помещаются в массив <code>group</code> . Если <code>all</code> — true, то потоки во всех подгруппах потока также помещаются в <code>group</code>
<code>int enumerate(ThreadGroup group[ ])</code>	Подгруппы вызывающего потока помещаются в массив <code>group</code>
<code>int enumerate(ThreadGroup group[ ], boolean all)</code>	Подгруппы вызывающего потока помещаются в массив <code>group</code> . Если <code>all</code> — true, то все подгруппы подгрупп (и т. д.) также помещаются в <code>group</code>
<code>final int getMaxPriority()</code>	Возвращает установку максимального приоритета для группы
<code>final String getName()</code>	Возвращает имя группы
<code>final ThreadGroup getParent()</code>	Возвращает null, если вызывающий ThreadGroup-объект не имеет родителя. Иначе возвращает родителя вызывающего объекта
<code>final void interrupt()</code>	Вызывает методы <code>interrupt()</code> всех потоков в группе. (Добавлен в Java 2)
<code>final boolean isDaemon()</code>	Возвращает true, если группа является daemon-группой. Иначе возвращает false

Таблица 14.20 (окончание)

Метод	Описание
<code>boolean isDestroyed()</code>	Возвращает <code>true</code> , если группа была разрушена. Иначе возвращает <code>false</code>
<code>void list()</code>	Отображает информацию о группе
<code>final boolean parentOf (ThreadGroup group)</code>	Возвращает <code>true</code> , если вызывающий поток является родителем группы <code>group</code> (или самой группой <code>group</code> ). Иначе возвращает <code>false</code>
<code>final void setDaemon(boolean isDaemon)</code>	Если <code>isDaemon</code> — <code>true</code> , то вызывающая группа отмечается как <code>daemon</code> -группа
<code>final void setMaxPriority (int priority)</code>	Устанавливает значение, полученное через <code>priority</code> , в качестве максимального приоритета вызывающей группы
<code>String toString()</code>	Возвращает строчный эквивалент группы
<code>void uncaughtException(Thread thread, Throwable e)</code>	Этот метод вызывается, когда возникает непойманное исключение

Поточные группы предлагают удобный способ управления группами потоков как одним устройством. Это особенно ценно в ситуациях, когда вы хотите приостановить и возобновить несколько связанных потоков. Например, вообразите программу, в которой один набор потоков используется для печати документа, другой — для отображения документа на экране, и третий — для сохранения документа в файле на диске. Если печать прервана, вы захотите иметь простой способ для остановки всех потоков, связанных с печатью. Такое удобство и предлагают поточные группы. Следующая программа, которая создает две поточные группы (по два потока в каждой) иллюстрирует эту ситуацию:

```
// Демонстрация поточных групп.
class NewThread extends Thread {
    boolean suspendFlag;

    NewThread(String threadname, ThreadGroup tgOb) {
        super(tgOb, threadname);
        System.out.println("Новый поток: " + this);
        suspendFlag = false;
        start(); // запустить поток
    }
}
```

```
// Это точка входа для потока.
public void run() {
    try {
        for(int i = 5; i > 0; i--) {
            System.out.println(getName() + ":" + i);
            Thread.sleep(1000);
            synchronized(this) {
                while(suspendFlag) {
                    wait();
                }
            }
        }
    } catch (Exception e) {
        System.out.println("Исключение в" + getName());
    }
    System.out.println(getName() + " завершен.");
}

void mysuspend() {
    suspendFlag = true;
}

synchronized void myresume() {
    suspendFlag = false;
    notify();
}
}

class ThreadGroupDemo {
    public static void main(String args[]) {
        ThreadGroup groupA = new ThreadGroup("Group A");
        ThreadGroup groupB = new ThreadGroup("Group B");

        NewThread ob1 = new NewThread("One", groupA);
        NewThread ob2 = new NewThread("Two", groupA);
        NewThread ob3 = new NewThread("Three", groupB);
        NewThread ob4 = new NewThread("Four", groupB);

        System.out.println("\nВывод метода list():");
        groupA.list();
        groupB.list();
        System.out.println();

        System.out.println("Приостановка группы A");
        Thread tga[] = new Thread[groupA.activeCount()];
        groupA.enumerate(tga);           // получить массив потоков группы
        for(int i = 0; i < tga.length; i++) {
            ((NewThread)tga[i]).mysuspend(); // приостановить каждый поток
        }
    }
}
```

```
try {
    Thread.sleep(4000);
} catch (InterruptedException e) {
    System.out.println("Главный поток прерван.");
}

System.out.println("Продолжение выполнения потоков группы A");
for(int i = 0; i < tga.length; i++) {
    ((NewThread)tga[i]).myresume(); // продолжить потоки группы
}

// ждать завершения потоков
try {
    System.out.println("Ждать завершения потоков.");
    ob1.join();
    cb2.join();
    ob3.join();
    ob4.join();
} catch (Exception e) {
    System.out.println("Исключение в главном потоке");
}

System.out.println("Главный поток завершен.");
}
}
```

Вывод этой программы:

```
Новый поток: Thread[One,5,Group A]
Новый поток: Thread[Two,5,Group A]
Новый поток: Thread[Three,5,Group B]
Новый поток: Thread[Four,5,Group B]
Вывод метода list():
java.lang.ThreadGroup[name=Group A,maxpri=10]
    Thread[One,5,Group A]
    Thread[Two,5,Group A]
java.lang.ThreadGroup[name=Group B,maxpri=10]
    Thread[Three,5,Group B]
    Thread[Four,5,Group B]
Приостановка группы A
Three: 5
Four: 5
Three: 4
Four: 4
Three: 3
Four: 3
Three: 2
Four: 2
```

Продолжение выполнения потоков группы A

Ждать завершения потоков.

One: 5

Two: 5

Three: 1

Four: 1

One: 4

Two: 4

Three завершен.

Four завершен.

One: 3

Two: 3

One: 2

Two: 2

One: 1

Two: 1

One завершен.

Two завершен.

Главный поток завершен.

Обратите внимание: поточная группа A приостанавливается на четыре секунды. Как подтверждает вывод, потоки One и Two при этом делают паузу, тогда как потоки Three и Four продолжают выполняться. После этих четырех секунд потоки One и Two продолжают свое выполнение. Обратите также внимание, как поточная группа A приостанавливается и продолжается. Сначала потоки в группе A получаются с помощью вызова метода `enumerate()` для группы A. Затем каждый поток приостанавливается внутри цикла обработки массива потоков, полученного в результате выполнения `enumerate()`. Для возобновления выполнения потоков группы A снова организуется цикл просмотра списка потоков, где каждый из них и продолжает свое выполнение. В этом примере используется подход к приостановке и возобновлению потоков, который рекомендуется в Java 2. Он не использует исключенные методы `suspend()` и `resume()`.

## Классы *ThreadLocal* и *InheritableThreadLocal*

В Java 2 к пакету *java.lang* добавлены два новых класса, связанных с потоками:

- ThreadLocal*. Используется для создания поточных локальных переменных. Каждый поток будет иметь свою собственную копию поточной локальной переменной.
- InheritableThreadLocal*. Создает поточные локальные переменные, которые можно наследовать.

## Класс Package

В Java 2 добавлен класс с именем `Package`, который инкапсулирует данные, связанные с версией пакета. Информация о версии пакета становится все более важной из-за быстрого роста и специализации пакетов и потому, что Java-программа должна знать, какая версия пакета является доступной. Методы класса `Package` перечислены в табл. 14.21.

**Таблица 14.21. Методы класса `Package`**

Метод	Описание
<code>String getImplementationTitle()</code>	Возвращает заголовок вызывающего пакета
<code>String getImplementationVendor()</code>	Возвращает имя поставщика вызывающего пакета
<code>String getImplementationVersion()</code>	Возвращает номер версии вызывающего пакета
<code>String getName()</code>	Возвращает имя вызывающего пакета
<code>static Package getPackage(String pkgName)</code>	Возвращает <code>Package</code> -объект с именем, указанным в <code>pkgName</code>
<code>static Package[ ] getPackages()</code>	Возвращает все пакеты, которые в текущий момент известны вызывающей программе
<code>String getSpecificationTitle()</code>	Возвращает заголовок спецификаций вызывающего пакета
<code>String getSpecificationVendor()</code>	Возвращает имя владельца спецификаций вызывающего пакета
<code>String getSpecificationVersion()</code>	Возвращает номер версии спецификаций вызывающего пакета
<code>int hashCode()</code>	Возвращает хэш-код вызывающего пакета
<code>boolean isCompatibleWith(String verNum) throws NumberFormatException</code>	Возвращает <code>true</code> , если <code>verNum</code> — меньше или равен номеру версии вызывающего пакета
<code>boolean isSealed()</code>	Возвращает <code>true</code> , если пакет блокирован. Иначе — <code>false</code>

Таблица 14.21 (окончание)

Метод	Описание
<code>boolean isSealed(URL url)</code>	Возвращает <code>true</code> , если вызывающий пакет блокирован по отношению к <code>url</code> . Иначе возвращает <code>false</code>
<code>String toString()</code>	Возвращает строковый эквивалент вызывающего пакета

Следующая программа демонстрирует класс `Package` и отображает пакеты, которые в настоящий момент известны программе:

```
// Демонстрирует класс Package.
class PkgTest {
    public static void main(String args[]) {
        Package pkgs[];
        pkgs = Package.getPackages();
        for(int i=0; i < pkgs.length; i++)
            System.out.println(
                pkgs[i].getName() + " " +
                pkgs[i].getImplementationTitle() + " " +
                pkgs[i].getImplementationVendor() + " " +
                pkgs[i].getImplementationVersion()
            );
    }
}
```

## Класс `RuntimePermission`

`RuntimePermission` был добавлен к `java.lang` в Java 2. Он связан с механизмом защиты Java и подробнее не рассматривается в данной книге.

## Класс `Throwable`

Класс `Throwable` поддерживает систему обработки исключений в Java и является классом, от которого производятся все классы исключений. Он обсуждался в главе 10.

## Класс *SecurityManager*

*SecurityManager* — это абстрактный класс, который ваши подклассы могут реализовать для создания менеджера безопасности. В общем случае вы не должны реализовывать собственный менеджер безопасности. Однако если вы это делаете, то должны обратиться к документации, которая поставляется с вашей системой разработки Java-программ.

## Интерфейс *Comparable*

В Java 2 к *java.lang* добавлен новый интерфейс — *Comparable*. Объекты классов, которые реализуют *Comparable*, могут быть упорядоченными. Другими словами, классы, реализующие *Comparable*, содержат объекты, которые можно сравнивать некоторым значимым способом. Интерфейс *Comparable* объявляет единственный метод, предназначенный для определения того, что Java 2 называет *естественным упорядочение* экземпляров (объектов) класса. Сигнатура этого метода:

```
int compareTo(Object obj)
```

Метод сравнивает вызывающий объект с *obj*. Он возвращает 0, если значения равны; отрицательное значение, если вызывающий объект меньше параметра. В любом другом случае возвращается положительное значение.

Указанный интерфейс реализован в некоторых, уже рассмотренных в этой книге, классах. Например, метод *compareTo()* определен в классах *Byte*, *Character*, *Double*, *Float*, *Long*, *Short*, *String* и *Integer*. Кроме того, как показано в следующей главе, объекты, которые реализуют этот интерфейс, можно использовать в различных коллекциях.

## Пакеты *java.lang.ref* и *java.lang.reflect*

В Java определено два подпакета *java.lang* — *java.lang.ref* и *java.lang.reflect*. Далее приводится их краткое описание.

### Пакет *java.lang.ref*

Вы узнали ранее, что средства сборки "мусора" в Java автоматически определяют, когда ссылки к объекту отсутствуют. В этом случае предполагается, что объект больше не нужен, и его память регенерируется (восстанавливается). Классы в пакете *java.lang.ref*, которые были добавлены в Java 2, обеспечивают более гибкий контроль над процессом сборки "мусора". Например, предположим, что ваша программа создала многочисленные объекты, которые вы хотите многократно использовать в более позднее время. Вы

можете продолжать поддерживать ссылки к этим объектам, но это может потребовать слишком много памяти.

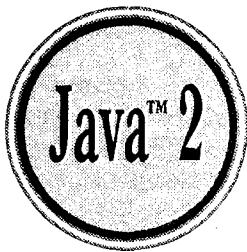
Вместо этого можно определить "мягкие" ссылки к объектам. Объект, который является "мягко доступным", может быть восстановлен сборщиком мусора, если достаточно ниже памяти. В этом случае сборщик мусора устанавливает "мягкие" ссылки к тому объекту на `null`. В противном случае он сохраняет объект для возможного будущего использования.

Программист имеет возможность определить, был ли "мягко доступный" объект регенерирован. Если да, то он может быть создан повторно. Иначе объект все еще доступен для многократного использования. Вы можете также создавать "слабые" и "фантомные" ссылки к объектам. Обсуждение подобных и других особенностей пакета `java.lang.ref` выходит за рамки возможностей этой книги.

## **Пакет *java.lang.reflect***

*Отражение* (*reflection*) — это способность программы анализировать саму себя. Пакет `java.lang.reflect` обеспечивает способность получить информацию о полях, конструкторах, методах и модификаторах класса. Вы нуждаетесь в этой информации, чтобы строить программные инструментальные средства, которые дают возможность работать с компонентами Java Beans. Инструментальные средства используют отражение, чтобы определять динамически характеристики компонента. Эта тема рассматривается в главе 25.

Кроме того, пакет `java.lang.reflect` включает класс, который дает возможность создавать и обращаться с массивами *динамически*.



## ГЛАВА 15

# Пакет *java.util*: структура коллекций

Пакет *java.util* содержит одно из наиболее захватывающих расширений Java 2 — коллекции. *Коллекция* — это группа объектов. Добавление коллекций вызвало фундаментальные изменения в структуре и архитектуре многих элементов пакета *java.util*. Они также расширили круг задач, к которым может применяться пакет. Коллекции — современная технология, которая заслуживает внимания всех программистов Java.

В дополнение к коллекциям, *java.util* содержит большой выбор классов и интерфейсов, которые поддерживают широкий диапазон функциональных возможностей. Эти классы и интерфейсы используются повсюду в пакетах ядра Java, а также доступны для применения в пользовательских программах. Подобные приложения включают генерацию псевдослучайных чисел, манипулирование датой и временем, наблюдение за событиями, манипулирование наборами битов и синтаксический анализ строк. Из-за обширного набора свойств *java.util* — один из наиболее широко используемых пакетов Java.

Ниже перечислены классы *java.util* (в скобках отмечены классы, добавленные в Java 2):

- AbstractCollection* (Java 2)
- AbstractList* (Java 2)
- AbstractMap* (Java 2)
- AbstractSequentialList* (Java 2)
- AbstractSet* (Java 2)
- ArrayList* (Java 2)
- Arrays* (Java 2)
- BitSet*
- Calendar*
- Collections* (Java 2)
- Date*
- Dictionary*
- EventObject*
- GregorianCalendar*
- HashMap* (Java 2)
- HashSet* (Java 2)

- |  |   |
|--|---|
| <input type="checkbox"/> Hashtable                   | <input type="checkbox"/> ResourceBundle       |
| <input type="checkbox"/> LinkedList (Java 2)         | <input type="checkbox"/> SimpleTimeZone       |
| <input type="checkbox"/> ListResourceBundle          | <input type="checkbox"/> Stack                |
| <input type="checkbox"/> Locale                      | <input type="checkbox"/> StringTokenizer      |
| <input type="checkbox"/> Observable                  | <input type="checkbox"/> TimeZone             |
| <input type="checkbox"/> Properties                  | <input type="checkbox"/> TreeMap (Java 2)     |
| <input type="checkbox"/> PropertyPermission (Java 2) | <input type="checkbox"/> TreeSet (Java 2)     |
| <input type="checkbox"/> PropertyResourceBundle      | <input type="checkbox"/> Vector               |
| <input type="checkbox"/> Random                      | <input type="checkbox"/> WeakHashMap (Java 2) |

Java.util определяет следующие интерфейсы (обратите внимание, что большинство из них были добавлены в Java 2):

- |  |  |   |
|--|--|---|
| <input type="checkbox"/> Collection (Java 2) | <input type="checkbox"/> List (Java 2)         | <input type="checkbox"/> Set (Java 2)       |
| <input type="checkbox"/> Comparator (Java 2) | <input type="checkbox"/> ListIterator (Java 2) | <input type="checkbox"/> SortedMap (Java 2) |
| <input type="checkbox"/> Enumeration         | <input type="checkbox"/> Map (Java 2)          | <input type="checkbox"/> SortedSet (Java 2) |
| <input type="checkbox"/> EventListener       | <input type="checkbox"/> Map.Entry (Java 2)    |   |
| <input type="checkbox"/> Iterator (Java 2)   | <input type="checkbox"/> Observer              |   |

Классы ResourceBundle, ListResourceBundle и PropertyResourceBundle помогают в интернационализации больших программ, обладающих значительным количеством региональных ресурсов. Эти классы здесь не рассматриваются. Обсуждение класса PropertyPermission, который предоставляет разрешение для чтения/записи системных свойств, также лежит вне рамок этой книги. Классы EventObject и EventListener описываются в главе 20. Остальные классы и интерфейсы рассматриваются в данной и следующей главе подробнее.

Поскольку пакет java.util весьма большой, его описание разбито на две части. Данная глава исследует те члены java.util, которые касаются коллекцией объектов. В главе 16 обсуждаются остальные классы и интерфейсы.

## Краткий обзор коллекций

*Структура коллекций* (collections framework) Java стандартизирует способ, с помощью которого ваши программы обрабатывают *группы объектов*. В прошлых версиях Java обеспечивал специальные классы типа Dictionary, Vector, Stack и Properties для хранения и манипулирования группами объектов. Хотя перечисленные классы были весьма полезны, они испытывали недостаток в централизованном, унифицирующем подходе при работе с группами

объектов. Таким образом, способ использования класса `vector`, например, отличался от способа применения класса `Properties`. Кроме того, предыдущий специальный (*ad hoc*<sup>1</sup>) подход не был предназначен для легкого расширения и адаптации. Коллекции как раз явились ответом на эти (и другие) проблемы.

Структура коллекций была разработана для нескольких целей. Во-первых, структура должна была быть высокоэффективной. Действительно, реализации фундаментальных коллекций (динамических массивов, связанных списков, деревьев и хэш-таблиц) в структуре коллекций Java 2 высоко эффективны. Вам редко (а может и никогда) потребуется кодировать одну из этих "машин данных" вручную. Во-вторых, структура коллекций должна была позволить различным типам коллекций работать похожим друг на друга образом и с высокой степенью способности к взаимодействию. В-третьих, расширение и/или адаптация коллекции должна была быть простой. В довершение ко всему, полная структура коллекций разработана в окружении набора стандартных интерфейсов. Несколько стандартных реализаций этих интерфейсов (такие как `LinkedList`, `HashSet` и `TreeSet`) построены так, что вы можете использовать их непосредственно в том виде, как они есть. Вы также можете реализовать собственную коллекцию по вашему выбору. Для удобства созданы различные реализации специального назначения, а также некоторые частичные реализации, которые упрощают создание собственного коллекционного класса. Наконец, были добавлены механизмы, которые позволяют интегрировать в структуру коллекций стандартные массивы.

*Алгоритмы* — другая важная часть механизма коллекций. Алгоритмы работают на коллекциях и определены как статические методы в классе `Collections`. Таким образом, они доступны для всех коллекций. Каждый коллекционный класс не нуждается в реализации своих собственных версий. Алгоритмы обеспечивают стандартные средства манипулирования коллекциями.

Другой элемент, созданный структурой коллекций, — это интерфейс `Iterator`. *Итератор* обеспечивает универсальный, стандартизованный способ доступа к элементам коллекции — по одному. Таким образом, итератор обеспечивает средства *перечисления содержимого коллекции*. Поскольку каждая коллекция реализует интерфейс `Iterator`, к элементам любого коллекционного класса можно обращаться через методы этого интерфейса. Поэтому (и только с мелкими изменениями) код, который циклически проходит, скажем, через *набор*, может также использоваться для циклического прохода *списка*, например.

---

<sup>1</sup> *Ad hoc* (лат.) — на данный случай. В нашем контексте речь идет об отсутствии унифицированного подхода к работе со специальными классами, когда для работы с каждым из них использовался отдельный, не унифицированный подход. — Примеч. пер.

В дополнение к коллекциям рассматриваемая структура определяет несколько интерфейсов и классов отображений. *Карта отображений* (map) хранит пары ключ/значение. Хотя карты (отображений) — не "коллекции" (в используемом смысле термина "коллекция"), они полностью интегрированы с коллекциями. На языке структуры коллекций, вы можете получить *коллекционный вид* или *представление* (collection-view) карты отображений. Подобное представление содержит элементы карты отображений, хранящиеся в виде коллекции. Таким образом, вы можете обрабатывать содержимое отображения как коллекцию.

Механизм коллекций был приспособлен к некоторым из первоначальных классов, определенных в `java.util` так, чтобы они также могли быть интегрированы в новую систему. Важно понять, что, хотя добавление коллекций изменило архитектуру многих из первоначальных сервисных классов, это не привело к их исключению. Коллекции просто обеспечивают лучший способ исполнения некоторых уже существующих компонентов пакета.

И последнее, если вы знакомы с C++, то полезно знать, что технология коллекций Java подобна (по духу) Стандартной Библиотеке Шаблонов (STL, Standard Template Library), определенной в C++. То, что C++ называет *контейнером*, Java называет *коллекцией*.

## Интерфейсы коллекций

В структуре коллекций определено несколько интерфейсов. В этом разделе приводится краткий обзор каждого из них. Начать с интерфейсов коллекций необходимо потому, что они определяют фундаментальный характер коллекционных классов. Иначе, конкретные классы просто являются различными реализациями стандартных интерфейсов. Интерфейсы, которые подкрепляют коллекции, кратко описаны в табл. 15.1.

**Таблица 15.1. Интерфейсы коллекций**

Интерфейс	Описание
<code>Collection</code>	Дает возможность работать с группами объектов; он находится на верху всей иерархии коллекций
<code>List</code>	Расширяет <code>Collection</code> для обработки последовательностей (списков) объектов
<code>Set</code>	Расширяет <code>Collection</code> для обработки наборов (sets), которые должны содержать уникальные элементы
<code>SortedSet</code>	Расширяет <code>Set</code> для обработки сортированных наборов (sorted sets)

В дополнение к этим интерфейсам, коллекции используют также интерфейсы `Comparator`, `Iterator` и `ListIterator`, которые подробнее описываются

далее в этой главе. Comparator определяет, как сравниваются два объекта, а Iterator и ListIterator перечисляют объекты в коллекции.

Для обеспечения максимальной гибкости в использовании, некоторые методы в интерфейсах коллекций могут быть необязательными. Необязательные методы дают возможность изменить содержимое коллекции. Коллекции, которые поддерживают эти методы, называются *изменяемыми* (modifiable). Коллекции, которые не допускают изменения их содержимого, называются *неизменяемыми* (unmodifiable). Если сделана попытка использования одного из необязательных методов на неизменяемой коллекции, выбрасывается исключение UnsupportedOperationException. Все встроенные коллекции являются изменяемыми.

В следующих разделах рассматриваются интерфейсы коллекций.

## Интерфейс Collection

Интерфейс Collection — основа, на которой сформирована структура коллекций. В нем объявляются основные методы, которые будут наследоваться всеми коллекциями. Эти методы описаны в табл. 15.2. Поскольку все коллекции реализуют интерфейс Collection, знакомство с его методами необходимо для четкого понимания структуры коллекций. Некоторые из методов могут выбрасывать исключение UnsupportedOperationException. Это происходит, если коллекция не может быть изменена. Исключение ClassCastException генерируется тогда, когда один объект несовместим с другим, т. е. когда осуществляется попытка добавить к коллекции несовместимый объект.

**Таблица 15.2. Методы класса Collection**

Метод	Описание
<code>boolean add(Object obj)</code>	Прибавляет <i>obj</i> к вызывающей коллекции. Возвращает true, если <i>obj</i> был добавлен к коллекции, и false, если <i>obj</i> уже является элементом коллекции или если коллекция не допускает дубликатов
<code>boolean addAll(Collection c)</code>	Добавляет все элементы <i>c</i> к вызывающей коллекции. Возвращает true, если операция закончилась успешно (т. е. все элементы были добавлены). Иначе возвращает false
<code>void clear()</code>	Удаляет все элементы из вызывающей коллекции

Таблица 15.2 (продолжение)

Метод	Описание
<code>boolean contains (Object obj)</code>	Возвращает <code>true</code> , если вызывающая коллекция содержит элемент <code>obj</code> . Иначе — <code>false</code>
<code>boolean containsAll (Collection c)</code>	Возвращает <code>true</code> , если вызывающая коллекция содержит все элементы <code>c</code> . Иначе — <code>false</code>
<code>boolean equals (Object obj)</code>	Возвращает <code>true</code> , если объект вызывающей коллекции и объект <code>obj</code> равны. Иначе — <code>false</code>
<code>int hashCode ()</code>	Возвращает хэш-код вызывающей коллекции
<code>boolean isEmpty ()</code>	Возвращает <code>true</code> , если вызывающая коллекция пуста. Иначе — <code>false</code>
<code>Iterator iterator ()</code>	Возвращает итератор ( <code>iterator</code> ) для вызывающей коллекции
<code>boolean remove (Object obj)</code>	Удаляет один экземпляр <code>obj</code> из вызывающей коллекции. Возвращает <code>true</code> , если элемент был удален. Иначе — <code>false</code>
<code>boolean removeAll (Collection c)</code>	Удаляет все элементы <code>c</code> из вызывающей коллекции. Возвращает <code>true</code> , если коллекция изменена (т. е. элементы были удалены). Иначе возвращает <code>false</code>
<code>boolean retainAll (Collection c)</code>	Удаляет все элементы из вызывающей коллекции, кроме элементов <code>c</code> . Возвращает <code>true</code> , если коллекция изменена (т. е. элементы были удалены). Иначе возвращает <code>false</code>
<code>int size ()</code>	Возвращает число элементов, содержащихся в вызывающей коллекции
<code>Object[ ] toArray ()</code>	Возвращает массив, который содержит все элементы, хранящиеся в вызывающей коллекции. Элементы массива являются копиями элементов коллекции
<code>Object[ ] toArray (Object array[ ])</code>	Возвращает массив, содержащий только те элементы коллекции, чей тип согласуется с типом элементов массива <code>array</code> . Элементы этого массива являются копиями элементов коллекции.

Таблица 15.2 (окончание)

Метод	Описание
(прод.)	Если размер массива <code>array</code> равен количеству согласованных элементов, они возвращаются прямо в массиве <code>array</code> . Если размер <code>array</code> меньше, чем количество согласованных элементов, то распределяется и возвращается новый массив необходимого размера. Если размер <code>array</code> больше, чем количество согласованных элементов, элемент массива, следующий за последним элементом коллекции, устанавливается в <code>null</code> . Если любой элемент коллекции имеет тип, который не является подтипов массива <code>array</code> , то выбрасывается исключение <code>ArrayStoreException</code>

Объекты добавляются в коллекцию вызовом метода `add()`. Обратите внимание, что `add()` получает аргумент типа `Object`. Поскольку `Object` — суперкласс всех классов, в коллекции можно сохранять объект любого типа. Однако примитивные типы сохранять нельзя. Например, коллекция не может непосредственно хранить значения типа `int`, `char`, `double` и т. д. Конечно, если вы хотите хранить такие объекты, то можете использовать одну из оболочек примитивных типов, описанных в главе 14. Можно добавить полное содержимое одной коллекции к другой, вызывая метод `addAll()`.

Для удаления объекта служит метод `remove()`. Чтобы удалить группу объектов, вызывайте метод `removeAll()`. Вызывая метод `retainAll()` можно удалить все элементы, кроме определенной их группы. Для чистки коллекции вызовите метод `clear()`.

Существует возможность определять, содержит ли коллекция определенный объект с помощью вызова метода `contains()`. Чтобы выяснить, содержит ли одна коллекция все члены другой, вызывают метод `containsAll()`. Вызывая метод `isEmpty()`, можно определить, является ли коллекция пустой. Количество элементов, содержащихся в настоящем время в коллекции, можно вычислить с помощью метода `size()`.

Метод `toArray()` возвращает массив, который содержит элементы, хранящиеся в вызывающей коллекции. Данный метод более важен, чем это могло бы показаться сначала. Часто обработка содержимого коллекции с помощью синтаксиса, похожего на синтаксис массивов, обеспечивает определенные преимущества. Прокладывая связующие пути между коллекциями и массивами, можно извлечь лучшее из этих представлений данных.

Есть возможность сравнивать две коллекции на равенство, вызывая метод `equals()`. Точное значение "равенства" может отличаться от коллекции к коллекции. Например, можно реализовать `equals()` так, чтобы он сравнивал значения элементов, хранимых в коллекции. Кроме того, `equals()` позволяет сравнивать ссылки к этим элементам.

Еще один очень важный метод — `iterator()`, который возвращает итератор (`Iterator`) коллекции. Как вы скоро убедитесь, итераторы имеют решающее значение для успешного программирования при использовании структуры коллекций.

## Интерфейс *List*

Интерфейс `List` расширяет `Collection` и объявляет поведение коллекции, которая хранит последовательность элементов. Элементы могут быть вставлены или извлечены с помощью их позиций в списке через отсчитываемый от нуля индекс. Список может содержать дублированные элементы.

В дополнение к методам, определенным в `Collection`, интерфейс `List` определяет собственные методы, которые показаны в табл. 15.3. Снова обратите внимание, что некоторые из этих методов выбрасывают исключения `UnsupportedOperationException`, если коллекция не может изменяться, и `ClassCastException`, когда один объект несовместим с другим, например, когда делается попытка добавить к коллекции несовместимый объект.

К версиям методов `add()` и `addAll()`, определенным в `Collection`, интерфейс `List` добавляет методы `add(int, Object)` и `addAll(int, Collection)`. Эти методы вставляют элементы по указанному индексу. Кроме того, семантика `add(Object)` и `addAll(Collection)`, определенная в `Collection`, изменена в `List` так, чтобы они добавляли элементы в конец списка.

Чтобы получить объект, хранящийся в определенном месте, вызывайте метод `get()` с индексом (номером) объекта. Для присвоения значения элементу в списке обратитесь к методу `set()`, указывая индекс (номер) объекта, который будет изменен. Чтобы найти индекс объекта, используйте метод `indexOf()` или `lastIndexOf()`.

**Таблица 15.3. Методы класса *List***

Метод	Описание
<code>void add(int index, Object obj)</code>	Вставляет <code>obj</code> в вызывающий список в позицию с индексом <code>index</code> . Любые элементы, существовавшие ранее в точке вставки или за ней, сдвигаются (т. е. никакие элементы не перезаписываются поверх уже существующих)

Таблица 15.3 (окончание)

Метод	Описание
<code>boolean addAll(int index, Collection c)</code>	Вставляет все элементы <i>c</i> в вызывающий список с позиции (индекса) <i>index</i> . Любые элементы, существовавшие ранее в точке вставки или за ней, сдвигаются (т. е. никакие элементы не перезаписываются поверх уже существующих). Возвращает <code>true</code> , если вызывающий список изменяется, и <code>false</code> — иначе
<code>Object get(int index)</code>	Возвращает объект, хранящийся в индексной позиции <i>index</i> вызывающей коллекции
<code>int indexOf(Object obj)</code>	Возвращает индекс (номер) первого экземпляра объекта <i>obj</i> в вызывающем списке. Если <i>obj</i> — не элемент списка, то возвращается <code>-1</code>
<code>int lastIndexOf(Object obj)</code>	Возвращает индекс последнего экземпляра <i>obj</i> в вызывающем списке. Если <i>obj</i> — не элемент списка, то возвращается <code>-1</code>
<code>ListIterator listIterator()</code>	Возвращает итератор, установленный к началу вызывающего списка
<code>ListIterator listIterator(int index)</code>	Возвращает итератор, установленный к позиции <i>index</i> вызывающего списка
<code>Object remove(int index)</code>	Удаляет элемент в позиции <i>index</i> из вызывающего списка и возвращает удаленный элемент. Результатирующий список уплотняется (т. е. индексы последующих элементов уменьшаются на 1)
<code>Object set(int index, Object obj)</code>	Устанавливает объект <i>obj</i> в позицию, указанную в <i>index</i> , в вызывающем списке
<code>List subList(int start, int end)</code>	Возвращает список, который включает элементы от номера <i>start</i> до <i>end</i> —1 вызывающего списка. После этого вызывающий объект будет ссылаться на элементы возвращенного (а не исходного) списка

Вызывая метод `subList()`, можно получить подсписок списка, указывая индексы (номера позиций) начала и окончания подсписка (в исходном списке). Использование этого метода обеспечивает очень удобную обработку списков.

## Интерфейс **Set**

Интерфейс `Set` расширяет интерфейс `Collection` и объявляет поведение коллекции, не допускающей дублирования элементов. Поэтому метод `add()` возвращает `false`, если осуществляется попытка добавить в набор дублирующие элементы. В `Set` не определяются никаких дополнительных собственных методов.

## Интерфейс **SortedSet**

Интерфейс `SortedSet` расширяет `Set` и объявляет поведение набора, отсортированного в возрастающем порядке. В дополнение к методам, определенным в `Set`, интерфейс `SortedSet` объявляет методы, представленные в табл. 15.4. Некоторые методы выбрасывают исключение `NoSuchElementException`, когда элементы не содержатся в вызывающем наборе. Исключение `ClassCastException` выбрасывается, когда объект несовместим с элементами набора, а исключение `NullPointerException` — если сделана попытка использовать null-объект<sup>1</sup> (null-объекты недопустимы в наборе).

В `SortedSet` определено несколько методов, которые делают обработку набора более удобной. Для получения первого объекта набора вызовите метод `first()`, а для получения последнего — метод `last()`. Вы можете получить подмножество сортированного набора, вызывая метод `subSet()`, указывая (в аргументах вызова) позиции (индексы) его первого и последнего объекта (в исходном наборе). Если вам нужен поднабор, который начинается с первого элемента исходного набора, используйте метод `headSet()`. Если вам нужен поднабор, который заканчивает исходный набор, используйте метод `tailSet()`.

**Таблица 15.4. Методы класса `SortedSet`**

Метод	Описание
<code>Comparator comparator()</code>	Возвращает компаратор вызывающего сортированного набора. Если для этого набора используется естественное упорядочение, то возвращается null (пустая ссылка)

<sup>1</sup> Null-объект — это объект, ссылка на который имеет значение null (пустая ссылка). — Примеч. пер.

Таблица 15.4 (окончание)

Метод	Описание
<code>Object first()</code>	Возвращает первый элемент вызывающего сортированного набора
<code>SortedSet headSet(Object end)</code>	Возвращает <code>SortedSet</code> -объект, содержащий те элементы вызывающего сортированного набора, которые меньше, чем <code>end</code> . На эти элементы ссылается также и вызывающий объект
<code>Object last()</code>	Возвращает последний элемент вызывающего отсортированного набора
<code>SortedSet subSet(Object start, Object end)</code>	Возвращает объект <code>SortedSet</code> , который содержит элементы, находящиеся между объектами <code>start</code> и <code>end - 1</code> вызывающего отсортированного списка. На эти элементы ссылается также и вызывающий объект
<code>SortedSet tailSet(Object start)</code>	Возвращает объект <code>SortedSet</code> , который содержит элементы сортированного набора, больше чем или равные <code>start</code> -объекту. На элементы возвращенного набора ссылается также и вызывающий объект

## Классы *Collection*

После ознакомления с интерфейсами коллекций, вы теперь готовы к рассмотрению стандартных классов, которые их реализуют. Для некоторых классов обеспечены полные реализации, которые можно использовать непосредственно (т. е. сразу же приступая к созданию их экземпляров). Другие классы являются абстрактными и обеспечивают только схематические реализации, которые служат отправными точками для создания конкретных коллекций. Ни один из коллекционных классов не синхронизирован, но, как вы увидите позже в этой главе, можно получить и синхронные версии.

Стандартные классы коллекций перечислены в табл. 15.5.

Таблица 15.5. Стандартные классы коллекций

Класс	Описание
<code>AbstractCollection</code>	Реализует большую часть интерфейса <code>Collection</code>
<code>AbstractList</code>	Расширяет <code>AbstractCollection</code> и реализует большую часть интерфейса <code>List</code>

**Таблица 15.5 (окончание)**

Класс	Описание
<code>AbstractSequentialList</code>	Расширяет класс <code>AbstractList</code> для реализации коллекции, которая использует последовательный, а не произвольный доступ к ее элементам
<code>LinkedList</code>	Реализует связный список, расширяя класс <code>AbstractSequentialList</code>
<code>ArrayList</code>	Реализует динамический массив, расширяя класс <code>AbstractList</code>
<code>AbstractSet</code>	Расширяет класс <code>AbstractCollection</code> и реализует большую часть интерфейса <code>Set</code>
<code>HashSet</code>	Расширяет класс <code>AbstractSet</code> для реализации хеш-таблиц
<code>TreeSet</code>	Реализует набор, хранящийся в виде дерева (древовидный набор). Расширяет класс <code>AbstractSet</code>

### Замечание

В дополнение к коллекционным классам для поддержки коллекций были пере-проектированы некоторые наследуемые классы, такие как `Vector`, `Stack` и `Hashtable`. Они будут обсуждаться позже в этой главе.

Следующие разделы рассматривают конкретные классы коллекций и иллюстрируют их использование.

## Класс `ArrayList`

Класс `ArrayList` расширяет `AbstractList` и реализует интерфейс `List`. `ArrayList` поддерживает динамические массивы, которые могут расти по мере необходимости. В Java стандартные массивы имеют фиксированную длину. После того как массивы созданы, они не могут расширяться или сжиматься, что означает, что вы должны знать заранее, сколько элементов массив будет содержать. Но, иногда, вы не можете знать до момента выполнения точного размера массива. Для обработки подобных ситуаций в структуре коллекций определен класс `ArrayList`. По существу, он является массивом объектных ссылок переменной длины. То есть `ArrayList` можно динамически увеличивать или уменьшать в размере. Списки массивов создаются с некоторым начальным размером. Когда этот размер превышается, коллекция автоматически расширяется. Когда объекты удаляются, массив может быть сокращен.

### Замечание

Динамические массивы также поддерживаются наследуемым классом `Vector`, который описан далее в этой главе.

`ArrayList` имеет следующие конструкторы:

```
ArrayList()  
ArrayList(Collection c)  
ArrayList(int capacity)
```

Первый конструктор строит пустой список массива. Второй — список массива, который инициализирован элементами коллекции `c`. Третий конструктор формирует список массива с указанной начальной *емкостью* (`capacity`). Емкость — это размер (т. е. количество элементов) рассматриваемого массива, который служит для хранения указанного (в параметре `capacity`) количества элементов. Когда элементы добавляются к списку массива, емкость растет автоматически.

Следующая программа демонстрирует простое использование класса `ArrayList`. Сначала создается список массива, и затем к нему добавляются объекты типа `String`. (Вспомните, что цитируемая строка транслируется в `String`-объект.) Далее список отображается на экране. Потом некоторые элементы удаляются, и список отображается снова.

```
// Демонстрирует класс ArrayList.  
import java.util.*;  
  
class ArrayListDemo {  
    public static void main(String args[]) {  
        // создать список массива  
        ArrayList al = new ArrayList();  
  
        System.out.println("Начальный размер al: " + al.size());  
  
        // добавить элементы в список массива  
        al.add("C");  
        al.add("A");  
        al.add("E");  
        al.add("B");  
        al.add("D");  
        al.add("F");  
        al.add(1, "A2");  
  
        System.out.println("Размер al после добавлений: " + al.size());  
  
        // показать на экране список массива  
        System.out.println("Contents of al: " + al);  
  
        // удалить элементы из списка массива  
        al.remove("F");  
        al.remove(2);
```

```
System.out.println("Размер al после удалений: " + al.size());
System.out.println("Содержимое al: " + al);
}
}
```

Вывод этой программы:

```
Начальный размер al: 0
Размер al после добавлений: 7
Содержимое al: [C, A2, A, E, B, D, F]
Размер al после удалений: 5
Содержимое al: [C, A2, E, B, D]
```

Обратите внимание, что объект *al* создается пустым и растет по мере добавления в него элементов. При удалении элементов его размер уменьшается.

В предыдущем примере содержимое коллекции отображается с использованием преобразования, заданного по умолчанию и обеспеченного методом *toString()*, который был унаследован от *AbstractCollection*. Хотя этого достаточно для коротких примеров программ, но вы редко будете использовать данный метод для отображения содержимого реальной коллекции. Обычно, вы будете применять свои собственные подпрограммы вывода. Тем не менее для следующих нескольких примеров метод *toString()* все еще будет использоваться для вывода.

Хотя емкость объекта *ArrayList* увеличивается автоматически по мере сохранения в нем объектов, вы можете увеличивать емкость *ArrayList* объекта вручную, вызывая метод *ensureCapacity()*. Это нужно делать, если вы заранее знаете, что в коллекции потребуется хранить намного больше элементов, чем она может содержать в настоящее время. Увеличивая емкость коллекции один раз в самом начале, вы можете предотвратить несколько последующих ее перераспределений (в памяти). Поскольку перераспределения — это довольно дорогостоящие операции (в смысле временных затрат), их сокращение улучшает эффективность. Сигнатура метода *ensureCapacity()*:

```
void ensureCapacity(int cap)
```

Здесь *cap* — новая емкость коллекции.

Наоборот, если вы хотите уменьшить размер массива, лежащего в основе *ArrayList*-объекта, чтобы его размер точно соответствовал числу содержащихся в нем в настоящее время элементов, то вызывайте метод *trimToSize()* со следующей сигнатурой:

```
void trimToSize()
```

## Получение массива из *ArrayList*-объекта

При работе с *ArrayList*-объектами иногда нужно получить фактический массив, который содержит элементы этой коллекции. Как объяснялось ра-

нее, это можно сделать, вызывая метод `toArray()`. Существуют несколько причин, по которым нужно конвертировать коллекцию в массив, например:

- чтобы ускорить время выполнения некоторых операций;
- чтобы передать массив методу, который не перегружен для работы с коллекциями;
- чтобы интегрировать ваши более новый код, использующий коллекции, со старым кодом, который не понимает коллекций.

Независимо от причины, преобразование `ArrayList`-объекта в массив — тривиальный вопрос, как показывает следующая программа:

```
// Преобразование объекта типа ArrayList в массив.
import java.util.*;

class ArrayListToArray {
    public static void main(String args[]) {
        // создать объект типа ArrayList
        ArrayList al = new ArrayList();

        // добавить элементы в список массива
        al.add(new Integer(1));
        al.add(new Integer(2));
        al.add(new Integer(3));
        al.add(new Integer(4));

        System.out.println("Содержимое объекта al: " + al);
        // получить массив
        Object ia[] = al.toArray();
        int sum = 0;

        // суммировать элементы массива
        for(int i=0; i<ia.length; i++)
            sum += ((Integer) ia[i]).intValue();

        System.out.println("Сумма равна: " + sum);
    }
}
```

Вывод этой программы:

```
Содержимое объекта al: [1, 2, 3, 4]
Сумма равна: 10
```

Программа начинается с создания коллекции целых чисел. Как объяснялось ранее, вы не можете хранить примитивные типы (в данном случае `int`) в коллекции, так что создаются и сохраняются *объекты* типа `Integer`. Затем вызывается метод `toArray()`, и с его помощью создается массив `Object`-элементов. Содержимое массива приводится к типу `Integer`, и затем значения суммируются.

## Класс *LinkedList*

Класс *LinkedList* расширяет *AbstractSequentialList* и реализует интерфейс *List*. Он обеспечивает структуру данных связного списка и имеет два следующих конструктора:

```
LinkedList()  
LinkedList(Collection c)
```

Первый конструктор строит пустой связный список, а второй создает связный список, инициализированный элементами коллекции *c*.

В дополнение к методам, которые он наследует, класс *LinkedList* определяет некоторые собственные полезные методы для манипулирования и доступа к спискам. Чтобы добавить элемент в начало списка, используйте метод *addFirst()*, а для добавления элементов в конец списка вызывайте метод *addLast()*. Их сигнатуры:

```
void addFirst(Object obj)  
void addLast(Object obj)
```

Здесь *obj* — добавляемый элемент.

Чтобы получить первый элемент, вызовите метод *getFirst()*, а для извлечения последнего элемента — метод *getLast()*. Они имеют следующие сигнатуры:

```
Object getFirst()  
Object getLast()
```

Для удаления первого элемента служит метод *removeFirst()*, а для удаления последнего — *removeLast()*. Вот их форматы:

```
Object removeFirst()  
Object removeLast()
```

Следующая программа иллюстрирует некоторые методы класса *LinkedList*:

```
// Демонстрирует класс LinkedList.  
import java.util.*;  
  
class LinkedListDemo {  
    public static void main(String args[]) {  
        // создать связный список  
        LinkedList ll = new LinkedList();  
  
        // добавить элементы в связный список  
        ll.add("F");  
        ll.add("B");  
        ll.add("D");  
        ll.add("E");
```

```

ll.add("C");
ll.addLast("Z");
ll.addFirst("A");

ll.add(1, "A2");

System.out.println("Исходное содержимое ll: " + ll);

// удалить элементы из связного списка
ll.remove("F");
ll.remove(2);

System.out.println("Содержимое ll после удаления: " + ll);

// удалить первый и последний элементы
ll.removeFirst();
ll.removeLast();

System.out.println("ll после удаления первого и последнего: " + ll);

// получить и установить значение
Object val = ll.get(2);
ll.set(2, (String) val + " Изменен");

System.out.println("ll после изменения: " + ll);
}
}

```

Вывод этой программы:

```

Исходное содержимое ll: [A, A2, F, B, D, E, C, Z]
Содержимое ll после удаления: [A, A2, D, E, C, Z]
ll after после удаления первого и последнего: [A2, D, E, C]
ll после изменения: [A2, D, E Изменен, C]

```

Поскольку `LinkedList` реализует интерфейс `List`, обращения к `add(Object)` добавляют элементы в конец списка, как это делает `addLast()`. Для вставки элемента в определенную позицию списка используйте метод формата `add(int, Object)`, как иллюстрирует обращение `add(1, "A2")` в предыдущем примере.

Обратите внимание, как изменяется третий элемент объекта `ll` — с помощью вызовов `get()` и `set()`. Для получения текущего значения элемента передайте в `get()` индекс (номер) хранимого элемента, а для назначения нового значения элементу с этим индексом передайте в `set()` индекс и его новое значение.

## Класс `HashSet`

Класс `HashSet` расширяет `AbstractSet` и реализует интерфейс `Set`. Он создает коллекцию, которая использует хэш-таблицу для хранения коллекций. Как

большинство читателей, вероятно, знает, хэш-таблица хранит информацию, используя механизм, называемый хэшированием. При *хэшировании* информационное содержание ключа используется, чтобы определить уникальное значение, называемое его хэш-кодом. Затем хэш-код применяется как индекс (номер) элемента, в котором хранятся данные, связанные с ключом. Преобразование ключа в его хэш-код выполняется автоматически — вы никогда не видите сам хэш-код. Ваш код не может также прямо индексировать хэш-таблицу. Преимущество хэширования состоит в том, что оно сохраняет постоянным время выполнения основных операций, таких как *add()*, *contains()*, *remove()* и *size()*, даже для больших наборов.

В классе определены следующие конструкторы:

```
HashSet()
HashSet(Collection c)
HashSet(int capacity)
HashSet(int capacity, float fillRatio)
```

Первая форма организует заданный по умолчанию хэш-набор. Вторая инициализирует хэш-набор, используя элементы *c*. Третья создает емкость хэш-набора величиной *capacity*. Четвертая форма инициализирует как емкость (*capacity*), так и отношение (коэффициент) заполнения (*fillRatio*), называемое также емкостью загрузки хэш-набора. Коэффициент заполнения должен быть между 0.0 и 1.0, и он определяет, насколько полным может быть хэш-набор, прежде чем он увеличивает свой размер. Когда число элементов больше, чем емкость хэш-набора, умноженная на его коэффициент заполнения, хэш-набор расширяется. Для конструкторов, которые не принимают коэффициент заполнения, используется множитель 0.75.

*HashSet* не определяет никаких дополнительных методов, кроме тех, что обеспечены его суперклассами и интерфейсами.

Важно обратить внимание, что хэш-набор не гарантирует упорядочивания его элементов, потому что процесс хэширования<sup>1</sup> не предназначен для создания сортируемых наборов. Если вам нужна сортируемая память, то лучше выбрать другую коллекцию, такую как *TreeSet*.

Пример, который демонстрирует *HashSet*, выглядит так:

```
// Демонстрирует класс HashSet.
import java.util.*;

class HashSetDemo {
    public static void main(String args[]) {
        // создать хэш-набор
        HashSet hs = new HashSet();
```

<sup>1</sup> Хэширование — это процесс *рандомизации*, т. е. случайного перемешивания элементов. — Примеч. пер.

```

// добавить элементы в хэш-набор
hs.add("B");
hs.add("A");
hs.add("D");
hs.add("E");
hs.add("C");
hs.add("F");

System.out.println(hs);
}
}

```

Вывод этой программы таков:

```
[F, E, D, C, B, A]
```

Итак, мы видим, что элементы хэш-набора не хранятся в сортированном порядке. Размещение элементов в таком наборе случайное (рандомизированное).

## Класс TreeSet

Класс TreeSet обеспечивает реализацию интерфейса Set и использует иерархическую (древовидную) структуру для хранения данных. Объекты хранятся в сортированном по возрастанию порядке. Время доступа и поиска в такой структуре не велико, что делает TreeSet превосходным выбором для хранения больших количеств сортированной информации, которая должна быть быстро найдена.

В классе определены следующие конструкторы:

```

TreeSet()
TreeSet(Collection c)
TreeSet(Comparator comp)
TreeSet(SortedSet ss)

```

Первая форма создает пустой древовидный набор, который будет сортироваться в восходящем порядке согласно естественному порядку его элементов. Вторая — формирует древовидный набор, который содержит элементы коллекции *c*. Третья форма создает пустой древовидный набор, который будет сортироваться в соответствии с компаратором *comp*. (Компараторы описаны далее в этой главе.) Четвертая — строит древовидный набор, который содержит элементы сортированного набора *ss*.

Вот пример, который демонстрирует TreeSet:

```

// Демонстрирует класс TreeSet.
import java.util.*;

```

```

class TreeSetDemo {
    public static void main(String args[]) {
        // создать древовидный набор
        TreeSet ts = new TreeSet();

        // добавить элементы в древовидный набор
        ts.add("C");
        ts.add("A");
        ts.add("B");
        ts.add("E");
        ts.add("F");
        ts.add("D");

        System.out.println(ts);
    }
}

```

Вывод этой программы:

[A, B, C, D, E, F]

Как говорилось ранее, из-за того, что TreeSet хранит свои элементы в дереве, они автоматически размещаются в отсортированном порядке, что и подтверждает этот вывод.

## Доступ к коллекции через итератор

Часто нужно циклически проходить элементы в коллекции, например, для отображения их на экране. Самый простой способ сделать это — использовать *итератор* (iterator), т. е. специальный объект, который реализует один из интерфейсов — либо Iterator, либо ListIterator. Интерфейс Iterator дает возможность циклически пройти через коллекцию, получая или удаляя ее элементы. Интерфейс ListIterator расширяет Iterator, обеспечивая двунаправленный обход списка, и модификацию элементов. В интерфейсе Iterator объявляются методы, перечисленные в табл. 15.6. Методы, объявленные в ListIterator, показаны в табл. 15.7.

**Таблица 15.6. Методы класса Iterator**

Метод	Описание
<code>boolean hasNext()</code>	Возвращает true, если в коллекции присутствует следующий элемент. Иначе возвращает false
<code>Object next()</code>	Возвращает следующий элемент. Выбрасывает исключение типа NoSuchElementException, если следующего элемента нет

**Таблица 15.6 (окончание)**

Метод	Описание
<code>void remove()</code>	Удаляет текущий элемент. Выбрасывает исключение типа <code>IllegalStateException</code> , если сделана попытка вызвать метод <code>remove()</code> , которой не предшествует вызов <code>next()</code>

**Таблица 15.7. Методы класса `ListIterator`**

Метод	Описание
<code>void add(Object obj)</code>	Вставляет <code>obj</code> в список перед элементом, который будет возвращен следующим вызовом <code>next()</code>
<code>boolean hasNext()</code>	Возвращает <code>true</code> , если существует следующий элемент. Иначе возвращает <code>false</code>
<code>boolean hasPrevious()</code>	Возвращает <code>true</code> , если существует предыдущий элемент. Иначе возвращает <code>false</code>
<code>Object next()</code>	Возвращает следующий элемент. Выбрасывается исключение типа <code>NoSuchElementException</code> , если нет следующего элемента
<code>int nextIndex()</code>	Возвращает индекс следующего элемента. Если следующего элемента нет, возвращает размер списка
<code>Object previous()</code>	Возвращает предыдущий элемент. Выбрасывается исключение типа <code>NoSuchElementException</code> , если предыдущего элемента нет
<code>int previousIndex()</code>	Возвращает индекс предыдущего элемента. Если предыдущего элемента нет, возвращает <code>-1</code>
<code>void remove()</code>	Удаляет текущий элемент из списка. Выбрасывается исключение типа <code>IllegalStateException</code> , если метод <code>remove()</code> вызывается, прежде чем вызван метод <code>next()</code> или <code>previous()</code>
<code>void set(Object obj)</code>	Назначает <code>obj</code> на текущий элемент. Это последний элемент, возвращенный вызовом метода <code>next()</code> или <code>previous()</code>

## Использование итератора

Перед обращением к коллекции через итератор вы должны получить его. В каждом из коллекционных классов определен метод `iterator()`, который возвращает итератор к началу коллекции. Используя этот итерационный

объект, вы можете обращаться к каждому элементу в коллекции поодиночке. Вообще, чтобы применять итератор для циклического прохода содержимого коллекции, выполните следующие действия:

1. Вызывая коллекционный метод `iterator()`, получите итератор, чтобы стартовать коллекцию.
2. Установите цикл, который делает обращение к методу `hasNext()`. Повторяйте итерации, пока `hasNext()` возвращает `true`.
3. Внутри цикла, получайте каждый элемент коллекции, вызывая метод `next()`.

Для коллекций, которые реализуют интерфейс `List`, можно таким же способом получать итератор, вызывая методы интерфейса `ListIterator`. Списковый итератор дает возможность обращаться к коллекции в прямом или в обратном направлении, а также позволяет изменять элемент коллекции. Иначе говоря, `ListIterator` применяется точно так же, как `Iterator`.

Ниже показан пример, который реализует описанные выше действия и демонстрирует интерфейсы `Iterator` и `ListIterator`. Он использует объект типа `ArrayList`, но общие принципы применимы к коллекции любого типа. Конечно, интерфейс `ListIterator` доступен только тем коллекциям, которые реализуют интерфейс `List`.

```
// Демонстрирует итераторы.
import java.util.*;

class IteratorDemo {
    public static void main(String args[]) {
        // создать список массива
        ArrayList al = new ArrayList();

        // добавить элементы в список массива
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");

        // использовать итератор для показа содержимого объекта al
        System.out.print("Исходное содержимое al: ");
        Iterator itr = al.iterator();
        while(itr.hasNext()) {
            Object element = itr.next();
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

```

// модифицировать итерируемые объекты
ListIterator litr = al.listIterator();
while(litr.hasNext()) {
    Object element = litr.next();
    litr.set(element + "+");
}

System.out.print("Модифицированное содержимое al: ");
itr = al.iterator();
while(itr.hasNext()) {
    Object element = itr.next();
    System.out.print(element + " ");
}
System.out.println();

// теперь показать список в обратном порядке
System.out.print("Модифицированный (обратный) список: ");
while(litr.hasPrevious()) {
    Object element = litr.previous();
    System.out.print(element + " ");
}
System.out.println();
}
}
}

```

Вывод этого примера:

Исходное содержимое al: C A E B D F

Модифицированное содержимое al: C+ A+ E+ B+ D+ F+

Модифицированный (обратный) список: F+ D+ B+ E+ A+ C+

Обратите особое внимание на то, как отображается обратный список. После изменения списка объектная переменная litr указывает на его конец. (Напомним, метод litr.hasNext() возвращает false, когда достигается конец списка.) Чтобы пройти список в обратном направлении, программа продолжает использовать litr, но на сей раз она выясняет, имеет ли он предыдущий элемент. Пока это так, такой элемент получается и отображается.

## Коллекции пользовательских классов

Ради простоты предыдущие примеры хранили в коллекции встроенные объекты типа String или Integer. Конечно, коллекций не ограничиваются хранением встроенных объектов. Совсем наоборот. Мощь коллекций состоит в том, что они могут хранить любой тип объекта, включая объекты классов, которые вы создаете сами. Рассмотрим следующий пример, использующий объект класса LinkedList для хранения почтовых адресов:

```
// Простой пример почтового списка.
import java.util.*;

class Address {
    private String name;
    private String street;
    private String city;
    private String state;
    private String code;

    Address(String n, String s, String c, String st, String cd) {
        name = n;
        street = s;
        city = c;
        state = st;
        code = cd;
    }

    public String toString() {
        return name + "\n" + street + "\n" +
               city + " " + state + " " + code;
    }
}

class MailList {
    public static void main(String args[]) {
        LinkedList ml = new LinkedList();

        // добавить элементы в связный список
        ml.add(new Address("J.W. West", "11 Oak Ave",
                           "Urbana", "IL", "61801"));
        ml.add(new Address("Ralph Baker", "1142 Maple Lane",
                           "Mahome", "IL", "61853"));
        ml.add(new Address("Tom Carlton", "867 Elm St",
                           "Champaign", "IL", "61820"));

        Iterator itr = ml.iterator();
        while(itr.hasNext()) {
            Object element = itr.next();
            System.out.println(element + "\n");
        }
        System.out.println();
    }
}
```

Вывод этой программы:

J.W. West  
11 Oak Ave  
Urbana IL 61801

Ralph Baker  
 1142 Maple Lane  
 Mahomet IL 61853

Tom Carlton  
 867 Elm St  
 Champaign IL 61820

Кроме сохранения в коллекции пользовательского класса, обратите внимание на другую важную особенность предыдущей программы — она довольно короткая. Когда вы видите, что создается связный список, который может хранить, отыскивать и обрабатывать почтовые адреса, приблизительно в пятидесяти строках кода, мощь структуры коллекций начинает становиться очевидной. Как знает большинство читателей, если бы все эти функциональные возможности нужно было кодировать вручную, то программа была бы в несколько раз длиннее. Коллекции предлагают практические решения самых разнообразных проблем программирования. Вы должны использовать их всякий раз, когда позволяет ситуация.

## Работа с картами отображений

Как объяснено в начале этой главы, в дополнение к коллекциям в Java 2 в `java.util` добавляются карты отображений. *Карта отображений*<sup>1</sup> (map) — это объект, который хранит ассоциации (связи) между ключами и значениями, или пары ключ/значение. По заданному ключу вы можете найти его значение. И ключи и значения являются объектами. Ключи должны быть уникальными, но значения могут быть дублированными. Одни карты (отображений) допускают `null`-ключи и `null`-значения, другие — нет.

### Интерфейсы карт

Поскольку характер и природу карт отображений определяют соответствующие интерфейсы, обсуждение начнем именно с них. Интерфейсы, поддерживающие карты отображений, представлены в табл. 15.8.

**Таблица 15.8. Интерфейсы карт**

Интерфейс	Описание
Мар	Отображает уникальные ключи в значения
Map.Entry	Описывает элемент (пару ключ/значение) карты отображений. Это внутренний класс Map-класса

<sup>1</sup> Или просто "карта" (определяющий термин "отображений" в остальной части текста будем чаще всего опускать). — Примеч. пер.

Таблица 15.8 (окончание)

Интерфейс	Описание
SortedMap	Расширяет класс Map так, что ключи поддерживаются в возрастающем порядке

## Интерфейс Map

Интерфейс Map отображает уникальные ключи в значения. Ключ — это объект, который вы используете для отыскания соответствующего значения в произвольный момент времени. Задав ключ, вы можете сохранить его значение в Map-объекте, а затем воспользоваться им. Методы, объявленные в Map, представлены в табл. 15.9. Когда в вызывающей карте никаких элементов не существует, некоторые методы выбрасывают исключение типа NoSuchElementException. Если объект несовместим с элементами карты, выбрасывается исключение типа ClassCastException. Исключение типа NullPointerException выбрасывается при осуществлении попытки использовать null-указатель (т. к. он недопустим в карте). Когда делается усилие изменить немодифицируемую карту, выбрасывается исключение типа UnsupportedOperationException.

Таблица 15.9. Методы класса Map

Метод	Описание
<code>void clear()</code>	Удаляет все пары ключ/значение из вызывающей карты
<code>boolean containsKey(Object k)</code>	Возвращает true, если вызывающая карта содержит объект <i>k</i> в качестве ключа. Иначе — false
<code>boolean containsValue(Object v)</code>	Возвращает true, если карта содержит объект <i>v</i> в качестве значения. Иначе — false
<code>Set entrySet()</code>	Возвращает Set-объект, который содержит входы карты. Набор содержит объекты типа Map.Entry. Этот метод обеспечивает представление вызывающей карты в виде набора
<code>boolean equals(Object obj)</code>	Возвращает true, если объект <i>obj</i> имеет тип Map и содержит те же самые входы (что и вызывающая карта). Иначе — false
<code>Object get(Object k)</code>	Возвращает значение, связанное с ключом <i>k</i>

Таблица 15.9 (окончание)

Метод	Описание
<code>int hashCode()</code>	Возвращает хэш-код вызывающей карты
<code>boolean isEmpty()</code>	Возвращает <code>true</code> , если вызывающая карта пуста. Иначе — <code>false</code>
<code>Set keySet()</code>	Возвращает <code>Set</code> -объект, который содержит ключи вызывающей карты. Метод обеспечивает представление ключа вызывающей карты в виде набора
<code>Object put(Object k, Object v)</code>	Помещает вход в вызывающую карту, переписывая любое предыдущее значение, связанное с ключом. Ключ и значение представлены параметрами <code>k</code> и <code>v</code> , соответственно. Возвращает <code>null</code> , если ключ еще не существует. Иначе — предыдущее значение, связанное с ключом
<code>void putAll(Map m)</code>	Помещает все входы карты отображений <code>m</code> в данную карту
<code>Object remove(Object k)</code>	Удаляет вход, чей ключ равен <code>k</code>
<code>int size()</code>	Возвращает размер (число пар ключ/значение) вызывающей карты
<code>Collection values()</code>	Возвращает коллекцию, содержащую значения вызывающей карты. Этот метод обеспечивает представление значений вызывающей карты в виде коллекции ( <code>collection-view</code> )

Карты отображений используют две основные операции — `get()` и `put()`. Чтобы поместить в карту значение, применяйте метод `put()`, определяя (в его аргументах) ключ и значение. Чтобы получить значение, вызовите метод `get()`, передавая ему ключ в качестве аргумента. Данный метод возвращает значение, соответствующее полученному ключу.

Как упоминалось ранее, карты — это не коллекции, но вы можете получить представление карты в виде коллекции (`collection-view`). Для этого, например, служит метод `entrySet()`. Он возвращает `Set`-объект, который содержит элементы карты. Чтобы получить представление ключей в виде коллекции, используйте метод `keySet()`. Чтобы получить представление значений в виде коллекции, используйте метод `values()`. Представления в виде коллекций — это средства, с помощью которых карты отображений интегрируются в структуру коллекций.

## Интерфейс *SortedMap*

Интерфейс *SortedMap* расширяет *Map*-интерфейс. Он гарантирует, что входы поддерживаются в восходящем порядке ключей. Методы, объявленные в *SortedMap*, показаны в табл. 15.10. Когда в вызывающей карте нет элементов, некоторые методы выбрасывают исключение типа *NoSuchElementException*. Если объект несовместим с элементами вызывающей карты, выбрасывается исключение типа *ClassCastException*. При попытке использовать null-объект (в то время как null-указатели не разрешены в карте отображений) выбрасывается исключение типа *NullPointerException*.

**Таблица 15.10. Методы класса *SortedMap***

Метод	Описание
<code>Comparator comparator()</code>	Возвращает вызывающий компаратор отсортированной карты. Если используется естественное упорядочение для вызывающей карты, возвращает null (пустой указатель)
<code>Object firstKey()</code>	Возвращает первый ключ вызывающей карты
<code>SortedMap headMap (Object end)</code>	Возвращает отсортированную карту для тех входов карты, ключи которых меньше чем <code>end</code>
<code>Object lastKey()</code>	Возвращает последний ключ вызывающей карты
<code>SortedMap subMap (Object start, Object end)</code>	Возвращает отсортированную карту, содержащую те входы, чьи ключи больше или равны <code>start</code> и меньше чем <code>end</code>
<code>SortedMap tailMap (Object start)</code>	Возвращает отсортированную карту, содержащую те входы, ключи которых больше или равны <code>start</code>

Отсортированные карты отображений допускают очень эффективные манипуляции над субкартами (т. е. с подмножествами карт). Для получения субкарты используйте методы `headMap()`, `tailMap()` или `subMap()`. Для получения первого ключа в наборе вызовите метод `firstKey()`, а для последнего — метод `lastKey()`.

## Интерфейс *Map.Entry*

Интерфейс *Map.Entry* дает возможность работать со входами карт. Напомним, что метод `entrySet()`, объявленный интерфейсом *Map*, возвращает *Set*-

объект, содержащий входы карты. Каждый из этих входов является объектом типа `Map.Entry`. Методы, объявленные в этом интерфейсе, перечислены в табл. 15.11.

**Таблица 15.11. Методы интерфейса `Map.Entry`**

Метод	Описание
<code>boolean equals(Object obj)</code>	Возвращает <code>true</code> , если объект <code>obj</code> является <code>Map.Entry</code> -объектом, чей ключ и значение равны соответствующим элементам вызывающего объекта
<code>Object getKey()</code>	Возвращает ключ текущего входа вызывающей карты
<code>Object getValue()</code>	Возвращает значение текущего входа вызывающей карты
<code>int hashCode()</code>	Возвращает хэш-код текущего входа вызывающей карты
<code>Object setValue(Object v)</code>	Устанавливает значение <code>v</code> -объекта в текущем входе вызывающей карты. Если аргумент, соответствующий периметру <code>v</code> , имеет некорректный для вызывающей карты тип, выбрасывается исключение <code>ClassCastException</code> . Если имеется проблема с <code>v</code> , выбрасывается исключение <code>IllegalArgumentException</code> . Если <code>v</code> является <code>null</code> -объектом, а вызывающая карта не разрешает <code>null</code> -ключей, выбрасывается исключение <code>NullPointerException</code> . Если карта не может быть изменена, выбрасывается исключение <code>UnsupportedOperationException</code>

## Классы карт отображений

Реализацию интерфейсов карт отображений обеспечивают классы, перечисленные в табл. 15.12.

**Таблица 15.12. Классы карт отображений**

Класс	Описание
<code>AbstractMap</code>	Реализует большую часть интерфейса <code>Map</code>
<code>HashMap</code>	Расширяет <code>AbstractMap</code> , используя хэш-таблицу
<code>TreeMap</code>	Расширяет <code>AbstractMap</code> , используя дерево

Таблица 15.12 (окончание)

Класс	Описание
<code>WeakHashMap</code>	Расширяет <code>AbstractMap</code> , используя хэш-таблицу со "слабыми" ключами

Обратите внимание, что `AbstractMap` является суперклассом для трех конкретных реализаций (классов) карт отображений. Класс `WeakHashMap` реализует карту, которая использует так называемые "слабые ключи". Для элементов такой карты разрешена сборка мусора, когда их ключи не используются. Этот класс далее не обсуждается. Два других конкретных класса описаны ниже.

## Класс `HashMap`

Класс `HashMap` для реализации `Map`-интерфейса использует хэш-таблицу. Это позволяет оставить постоянным время выполнения основных операций, таких как `get()` и `put()`, даже для больших наборов отображений.

Определены следующие конструкторы:

```
HashMap()
HashMap(Map m)
HashMap(int capacity)
HashMap(int capacity, float fillRatio)
```

Первая форма (без параметров) создает хэш-карту, используемую по умолчанию. Вторая инициализирует хэш-карту элементами карты `m`. Третья форма создает хэш-карту, размер (емкость) которой задает параметр `capacity`. Четвертая инициализирует хэш-карту с помощью двух параметров — `capacity` (емкость) и `fillRatio` (коэффициент заполнения). Это те же характеристики, что были описаны ранее для `HashSet`-коллекций.

Класс `HashMap` реализует интерфейс `Map` и расширяет класс `AbstractMap`. Он не добавляет никаких собственных методов.

Нужно заметить, что хэш-карта не гарантирует упорядоченности своих элементов. Поэтому порядок, в котором элементы добавляются к хэш-карте, не обязательно совпадает с порядком их считывания итератором.

Следующая программа иллюстрирует использование класса `HashMap`. Она отображает имена в баланс счетов. Обратите внимание, как получается и используется представление карты в виде набора (set-view).

```
import java.util.*;
class HashMapDemo {
    public static void main(String args[]) {
```

```
// создать хэш-карту
HashMap hm = new HashMap();

// поместить элементы в карту
hm.put("John Doe", new Double(3434.34));
hm.put("Tom Smith", new Double(123.22));
hm.put("Jane Baker", new Double(1378.00));
hm.put("Todd Hall", new Double(99.22));
hm.put("Ralph Smith", new Double(-19.08));

// получить набор входов
Set set = hm.entrySet();

// получить итератор
Iterator i = set.iterator();

// показать элементы
while(i.hasNext()) {
    Map.Entry me = (Map.Entry)i.next();
    System.out.print(me.getKey() + ": ");
    System.out.println(me.getValue());
}
System.out.println();

// перечислить 1000 на депозитный счет Джона До (John Doe)
double balance = ((Double)hm.get("John Doe")).doubleValue();
hm.put("John Doe", new Double(balance + 1000));
System.out.println("Новый баланс John Doe: " +
    hm.get("John Doe"));
}
}
```

Вывод этой программы:

```
Ralph Smith: -19.08
Tom Smith: 123.22
John Doe: 3434.34
Todd Hall: 99.22
Jane Baker: 1378.0

Новый баланс John Doe: 4434.34
```

Программа начинается с создания хэш-карты и затем добавляет отображения имен к балансам. Далее, содержимое карты выводится на экран (при этом используется представление карты в виде набора, которое получается с помощью вызова метода `entrySet()`). Ключи и значения отображаются с помощью вызова методов `getKey()` и `getValue()`, которые определены в интерфейсе `Map.Entry`. Обратите внимание на то, как вносится депозит в учетную запись (счет) Джона До. Метод `put()` автоматически заменяет любое

существовавшее ранее значение, которое связано с указанным ключом, новым значением. Таким образом, после модификации учетной записи Джона До хэш-карта будет все еще содержать только одну учетную запись с именем "John Doe".

## Класс `TreeMap`

Класс `TreeMap` реализует интерфейс `Map`, используя структуру дерева. Он обеспечивает эффективные средства хранения пар ключ/значение в отсортированном порядке и допускает быстрый поиск и извлечение данных. Заметим, что, в отличие от хэш-карт, древовидная карта гарантирует, что ее элементы будут сортироваться в порядке возрастания ключей.

В `TreeMap` определены следующие конструкторы:

```
TreeMap()
TreeMap(Comparator comp)
TreeMap(Map m)
TreeMap(SortedMap sm)
```

Первая форма создает пустую древовидную карту, которая будет сортироваться с естественным порядком своих ключей. Вторая создает пустую древовидную карту, которая будет сортироваться с помощью компаратора `comp`. (Компараторы обсуждаются позже в этой главе.) Третья форма инициализирует древовидную карту входами карты `m`, которые будут отсортированы по ключам (в естественном, т. е. возрастающем порядке). Четвертая инициализирует древовидную карту входами сортированной карты `sm`, которые будут отсортированы в том же порядке, что и `sm`.

Класс `TreeMap` реализует интерфейс `SortedMap` и расширяет класс `AbstractMap`. Никаких дополнительных собственных методов в нем не определяется.

Следующая программа переделывает предыдущий пример так, чтобы он использовал класс `TreeMap`:

```
import java.util.*;

class TreeMapDemo {
    public static void main(String args[]) {
        // создать древовидную карту
        TreeMap tm = new TreeMap();

        // поместить элементы в карту
        tm.put("John Doe", new Double(3434.34));
        tm.put("Tom Smith", new Double(123.22));
        tm.put("Jane Baker", new Double(1378.00));
        tm.put("Tod Hall", new Double(99.22));
        tm.put("Ralph Smith", new Double(-19.08));
```

```
// получить набор входов
Set set = tm.entrySet();

// получить итератор
Iterator i = set.iterator();

// показать элементы на экране
while(i.hasNext()) {
    Map.Entry me = (Map.Entry)i.next();
    System.out.print(me.getKey() + ": ");
    System.out.println(me.getValue());
}
System.out.println();

// перечислить 1000 на депозитный счет Джона До (John Doe)
double balance = ((Double)tm.get("John Doe")).doubleValue();
tm.put("John Doe", new Double(balance + 1000));
System.out.println("Новый баланс John Doe: " +
    tm.get("John Doe"));
}
```

Вывод у этой программы следующий:

```
Jane Baker: 1378.0
John Doe: 3434.34
Ralph Smith: -19.08
Todd Hall: 99.22
Tom Smith: 123.22

Новый баланс John Doe: 4434.34
```

Заметим, что TreeMap сортирует ключи. Однако в этом случае они упорядочиваются по именам вместо фамилий. Вы можете изменить это поведение, указав компаратор при создании карты (см. второй конструктор). В следующем разделе описывается, как это можно сделать.

## Компараторы

Как TreeSet, так и TreeMap хранят элементы в отсортированном виде. Однако точное определение порядка сортировки выполняет *компаратор*. По умолчанию, эти классы хранят свои элементы в порядке, который Java называет "естественным" (natural). Это упорядочение по возрастанию значение, которое вы обычно и ожидаете (A перед B, 1 перед 2, и т. д.). Если вы хотите упорядочить элементы другим способом, то укажите (в аргументе конструктора) объект типа Comparator, когда создаете набор или карту. Это обеспечит возможность точно управлять сортировкой элементов, хранящихся в сортированных коллекциях и картах.

В интерфейсе *Comparator* определяется два метода — *compare()* и *equals()*. Метод *compare()*, имеющий сигнатуру:

```
int compare(Object obj1, Object obj2)
```

сравнивает два объектных элемента *obj1* и *obj2* с учетом их порядка. Метод возвращает нуль, если объекты равны. Если *obj1* больше, чем *obj2*, он возвращает положительное значение, иначе — отрицательное значение. Метод может выбросить исключение *ClassCastException*, если типы объектов не совместимы при сравнении. Переопределяя метод *compare()*, вы можете изменять способ упорядочивания объектов. Например, чтобы отсортировать их в обратном порядке, можно создать компаратор, который реверсирует результат сравнения.

Метод *equals()* формата:

```
boolean equals(Object obj)
```

проверяет, равны ли объекты вызывающего компаратора. Параметр *obj* — это объект, который будет проверен на равенство (с вызывающим объектом). Метод возвращает *true*, если и *obj* и вызывающий объект являются объектами типа *Comparator* и используют одно и то же упорядочение. Иначе он возвращает *false*. В переопределении метода *equals()* нет необходимости, и большинство простых компараторов не будет этого делать.

## Использование компаратора

Следующий пример демонстрирует мощь заказного компаратора. Он реализует метод *compare()* так, что работает над своими параметрами в обратном порядке. Это заставляет сохранять древовидный набор тоже в обратном порядке.

```
// Пример использования заказного компаратора.  
import java.util.*;  
  
// Обратный компаратор для строк.  
class MyComp implements Comparator {  
    public int compare(Object a, Object b) {  
        String aStr, bStr;  
  
        aStr = (String) a;  
        bStr = (String) b;  
  
        // реверсировать сравнение  
        return bStr.compareTo(aStr);  
    }  
  
    // нет необходимости переопределять метод equals  
}
```

```

class CompDemo {
    public static void main(String args[]) {
        // создать древовидный набор
        TreeSet ts = new TreeSet(new MyComp());

        // добавить элементы в древовидный набор
        ts.add("C");
        ts.add("A");
        ts.add("B");
        ts.add("E");
        ts.add("F");
        ts.add("D");

        // получить итератор
        Iterator i = ts.iterator();

        // показать элементы на экране
        while(i.hasNext()) {
            Object element = i.next();
            System.out.print(element + " ");
        }
        System.out.println();
    }
}

```

Как показывает следующий вывод, дерево теперь хранится в обратном порядке:

F E D C B A

Рассмотрим внимательнее класс `MyComp`, который реализует интерфейс `Comparator` и переопределяет метод `compare()`. (Как было объяснено ранее, в переопределении метода `equals()` нет необходимости.) Внутри `compare()` `String`-метод `compareTo()` сравнивает две строки. Однако метод `compareTo()` вызывает объект `bStr`, а не `aStr`. Это и приводит к конверсии результата.

Очередная программа представляет собой более практичный пример и является модифицированной версией программы `TreeMap`, продемонстрированной ранее (хранение балансов счетов). В предыдущей версии, учетные записи сортировались по первому имени (first name), т. е. просто по имени. Следующая программа сортирует учетные записи по последнему имени (last name), т. е. по фамилии. Для этого она использует компаратор, который сравнивает фамилии каждой учетной записи, что приводит к сортировке карты по фамилиям.

```

// Использование компаратора для сортировки счетов по фамилиям.
import java.util.*;

// сравнить два последних полных слова в двух строках
class TComp implements Comparator {

```

```
public int compare(Object a, Object b) {
    int i, j, k;
    String aStr, bStr;

    aStr = (String) a;
    bStr = (String) b;

    // найти индекс начала фамилии
    i = aStr.lastIndexOf(' ');
    j = bStr.lastIndexOf(' ');

    k = aStr.substring(i).compareTo(bStr.substring(j));
    if(k==0)           // фамилии согласованы, проверить полное имя
        return aStr.compareTo(bStr);
    else
        return k;
}

// нет необходимости переопределять метод equals
}

class TreeMapDemo2 {
    public static void main(String args[]) {
        // создать древовидную карту
        TreeMap tm = new TreeMap(new TComp());

        // поместить элементы в карту
        tm.put("John Doe", new Double(3434.34));
        tm.put("Tom Smith", new Double(123.22));
        tm.put("Jane Baker", new Double(1378.00));
        tm.put("Tod Hall", new Double(99.22));
        tm.put("Ralph Smith", new Double(-19.08));

        // получить набор входов
        Set set = tm.entrySet();

        // получить итератор
        Iterator itr = set.iterator();

        // показать элементы на экране
        while(itr.hasNext()) {
            Map.Entry me = (Map.Entry)itr.next();
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
        }
        System.out.println();

        // перечислить 1000 на депозитный счет Джона До (John Doe)
        double balance = ((Double)tm.get("John Doe")).doubleValue();
        tm.put("John Doe", new Double(balance + 1000));
    }
}
```

```

        System.out.println("Новый баланс John Doe: " +
            tm.get("John Doe"));
    }
}

```

Ниже показан вывод этой программы. Обратите внимание, что учетные записи теперь сортируются по фамилиям:

Jane Baker: 1378.0

John Doe: 3434.34

Todd Hall: 99.22

Ralph Smith: -19.08

Tom Smith: 123.22

Новый баланс John Doe: 4434.34

Класс компаратора `TComp` сравнивает две строки, которые содержат имя и фамилию, причем первыми сравниваются фамилии. Для этого он находит индекс (номер) последнего пробела в каждой строке и затем сравнивает подстроки каждого элемента, которые начинаются в этой точке. Если фамилии эквивалентны, то тогда сравниваются имена. Данный алгоритм приводит к сортировке древовидной карты по фамилиям, а внутри фамилий — по именам. Это заметно потому, что в выводе строка `Ralph Smith` находится перед строкой `tom Smith`.

## Алгоритмы коллекций

Структура коллекций содержит несколько алгоритмов, которые могут применяться к коллекциям и картам отображений. Эти алгоритмы определены как статические методы в классе `Collections` и описаны в табл. 15.13.

**Таблица 15.13. Алгоритмы класса `Collections`**

Метод	Описание
<code>static int binarySearch (List list, Object value, Comparator c)</code>	Отыскивает объект <code>value</code> из списка <code>list</code> , упорядоченного с помощью компаратора <code>c</code> . Возвращает позицию <code>value</code> в списке <code>list</code> , или <code>-1</code> , если значение не найдено
<code>static int binarySearch (List list, Object value)</code>	Отыскивает объект <code>value</code> в списке <code>list</code> . Список должен быть сортированным. Возвращает позицию <code>value</code> в списке <code>list</code> , или <code>-1</code> , если значение не найдено

Таблица 15.13 (продолжение)

Метод	Описание
<code>static void copy(List list1,                  List list2)</code>	Копирует элементы <i>list2</i> в <i>list1</i>
<code>static Enumeration enumeration(Collection c)</code>	Возвращает перечисление коллекции <i>c</i> . (См. "Интерфейс Enumeration" далее в этой главе)
<code>static void fill(List list,                  Object obj)</code>	Присваивает объект <i>obj</i> каждому элементу списка <i>list</i>
<code>static Object max(Collection c,                  Comparator comp)</code>	Возвращает максимальный элемент коллекции <i>c</i> , использующий компаратор <i>comp</i>
<code>static Object max(Collection c)</code>	Возвращает максимальный элемент коллекции <i>c</i> , использующий естественное упорядочение. Коллекция может быть несортированной
<code>static Object min(Collection c,                  Comparator comp)</code>	Возвращает минимальный элемент коллекции <i>c</i> , использующий компаратор <i>comp</i> . Коллекция может быть несортированной
<code>static Object min(Collection c)</code>	Возвращает минимальный элемент коллекции <i>c</i> , использующий естественное упорядочение
<code>static List nCopies(int num,                  Object obj)</code>	Возвращает <i>num</i> копий объекта <i>obj</i> в форме неизменяемого списка. <i>num</i> должен быть больше или равен нулю
<code>static void reverse(List list)</code>	Реверсирует последовательность списка <i>list</i>
<code>static Comparator reverseOrder()</code>	Возвращает обратный компаратор (компаратор, который реверсирует вывод результата сравнения двух элементов)
<code>static void shuffle(List list,                  Random r)</code>	Перетасовывает (т. е. randomизирует) элементы в списке <i>list</i> , используя <i>r</i> как источник случайных чисел
<code>static void shuffle(List list)</code>	Перетасовывает (т. е. randomизирует) элементы в списке <i>list</i> (с randomизатором по умолчанию)

Таблица 15.13 (продолжение)

Метод	Описание
<code>static Set singleton(Object obj)</code>	Возвращает <i>obj</i> как неизменяемый набор. Это простой способ преобразования одиночного объекта в набор
<code>static void sort(List list, Comparator comp)</code>	Сортирует элементы списка <i>list</i> по правилам компаратора <i>comp</i>
<code>static void sort(List list)</code>	Сортирует элементы списка <i>list</i> в естественном порядке
<code>static Collection synchronizedCollection(Collection c)</code>	Возвращает синхронизированную (поточно-безопасную) коллекцию, поддерживаемую коллекцией <i>c</i>
<code>static List synchronizedList (List list)</code>	Возвращает синхронизированный (поточно-безопасный) список, поддерживаемый списком <i>list</i>
<code>static Map synchronizedMap(Map m)</code>	Возвращает синхронизированную (поточно-безопасную карту) отображений, поддерживаемую картой <i>m</i>
<code>static Set synchronizedSet(Set s)</code>	Возвращает синхронизированный (поточно-безопасный) набор, поддерживаемый набором <i>s</i>
<code>static SortedMap synchronizedSortedMap(SortedMap sm)</code>	Возвращает синхронизированную (поточно-безопасную) отсортированную карту, поддерживаемую отсортированной картой <i>sm</i>
<code>static SortedSet synchronizedSortedSet(SortedSet ss)</code>	Возвращает синхронизированный (поточно-безопасный) отсортированный набор, поддерживаемый отсортированным набором <i>ss</i>
<code>static Collection unmodifiableCollection(Collection c)</code>	Возвращает неизменяемую коллекцию, поддерживаемую коллекцией <i>c</i>
<code>static List unmodifiableList (List list)</code>	Возвращает неизменяемый список, поддерживаемый списком <i>list</i>
<code>static Map unmodifiableMap(Map m)</code>	Возвращает неизменяемую карту отображений, поддерживаемую картой <i>m</i>

Таблица 15.13 (окончание)

Метод	Описание
<code>static Set unmodifiableSet(Set s)</code>	Возвращает неизменяемый набор, поддерживаемый набором <i>s</i>
<code>static SortedMap unmodifiableSortedMap(SortedMap sm)</code>	Возвращает неизменяемую отсортированную карту отображений, поддерживаемую отсортированной картой <i>sm</i>
<code>static SortedSet unmodifiableSortedSet(SortedSet ss)</code>	Возвращает неизменяемый отсортированный набор, поддерживаемый отсортированным набором <i>ss</i>

Некоторые из методов могут выбрасывать исключения `ClassCastException`, которые происходят при попытке сравнения несовместимых типов, или исключения `UnsupportedOperationException`, возникающие во время изменения неизменяемой коллекции.

Обратите внимание, что некоторые методы, такие как `synchronizedList()` и `synchronizedSet`, используются для получения синхронизированных, *поточно-безопасных* (*thread-safe*) копий различных коллекций. Как уже объяснялось ранее, ни одна из стандартных реализаций коллекций не синхронизирована. Для обеспечения синхронизации следует применять алгоритмы синхронизации. И последнее, итераторы синхронизированных коллекций должны использоваться внутри синхронизированных блоков.

Набор методов, имена которых начинаются со строки `unmodifiable`, возвращают представления (виды) различных коллекций, которые не могут изменяться. Они будут полезны, когда вы захотите допустить, чтобы некоторый процесс читал, но не записывал коллекции.

В `Collections` определены две статические переменные: `EMPTY_SET` и `EMPTY_LIST` (обе — неизменяемые).

Следующая программа демонстрирует некоторые алгоритмы. Она создает и инициализирует связанный список. Метод `ReverseOrder()` возвращает `Comparator`, который реверсирует сравнение `Integer`-объектов. Элементы списка сортируются согласно этому компаратору и затем отображаются. Далее, список randomизируется с помощью вызова метода `shuffle()`, и затем отображаются его минимальное и максимальное значения.

```
// Демонстрирует различные алгоритмы.
import java.util.*;

class AlgorithmsDemo {
    public static void main(String args[]) {

```

```

// создать и инициализировать связанный список
LinkedList ll = new LinkedList();
ll.add(new Integer(-8));
ll.add(new Integer(20));
ll.add(new Integer(-20));
ll.add(new Integer(8));

// создать и реверсировать компаратор
Comparator r = Collections.reverseOrder();

// сортировать список, используя компаратор
Collections.sort(ll, r);

// получить итератор
Iterator li = ll.iterator();

System.out.print("Список, сортированный в обратном порядке: ");
while(li.hasNext())
    System.out.print(li.next() + " ");
System.out.println();

Collections.shuffle(ll);

// показать на экране randomизированный список
li = ll.iterator();
System.out.print("Randomизированный список: ");
while(li.hasNext())
    System.out.print(li.next() + " ");
System.out.println();

System.out.println("Минимум: " + Collections.min(ll));
System.out.println("Максимум: " + Collections.max(ll));
}
}

```

Вывод этой программы:

```

Список, сортированный в обратном порядке: 20 8 -8 -20
Randomизированный список: 20 -20 8 -8
Минимум: -20
Максимум: 20

```

Заметим, что методы `min()` и `max()` работают на списке после того, как он был перетасован (randomизирован). Ни один из них не требует наличия отсортированного списка для своих операций.

## Массивы

В Java 2 к `java.util` добавлен новый класс с именем `Arrays`. Он обеспечивает различные методы, которые полезны при работе с массивами. Хотя эти методы технически не являются частью структуры коллекций, они помогают

заполнить пробел между коллекциями и массивами. В данном разделе рассматриваются все методы `Arrays`.

Метод `asList()` возвращает `List`-объект, который поддерживается указанным (в его параметре) массивом объектов. Иначе говоря, как список, так и массив обращаются к одной и той же области объектной памяти. Метод имеет следующую сигнатуру:

```
static List<Object[]> asList(Object[] array)
```

где `array` — массив, содержащий данные.

Метод `binarySearch()` использует двоичный поиск, чтобы найти указанное значение. Данный метод должен применяться к сортированным массивам. Он имеет следующие формы:

```
static int binarySearch(byte[] array, byte value)
static int binarySearch(char[] array, char value)
static int binarySearch(double[] array, double value)
static int binarySearch(float[] array, float value)
static int binarySearch(int[] array, int value)
static int binarySearch(long[] array, long value)
static int binarySearch(short[] array, short value)
static int binarySearch(Object[] array, Object value)
static int binarySearch(Object[] array, Object value, Comparator c)
```

Здесь `array` — массив, в котором ведется поиск; `value` — значение, положение которого будет определено. Две последние формы выбрасывают исключение типа `ClassCastException`, если массив содержит элементы, которые не могут быть сравнены (например, `Double` и `StringBuffer`), или если значение не совместимо с типами массива `array`. В последней форме параметр `Comparator c` используется для того, чтобы определить порядок элементов в массиве `array`. Во всех случаях, если значение `value` существует в массиве, то возвращается индекс (номер) элемента. Иначе возвращается отрицательное значение.

Метод `equals()` возвращает `true`, если два массива эквивалентны. Иначе он возвращает `false`. Метод `equals()` имеет следующие формы:

```
static boolean equals(boolean array1[], boolean array2[])
static boolean equals(byte array1[], byte array2[])
static boolean equals(char array1[], char array2[])
static boolean equals(double array1[], double array2[])
static boolean equals(float array1[], float array2[])
static boolean equals(int array1[], int array2[])
static boolean equals(long array1[], long array2[])
static boolean equals(short array1[], short array2[])
static boolean equals(Object array1[], Object array2[])
```

Здесь `array1` и `array2` — два массива, которые сравниваются на равенство.

Метод `fill()` присваивает значение всем элементам массива. Другими словами, он заполняет массив указанным значением. Метод `fill()` имеет две версии. Первая версия заполняет весь массив. Она имеет следующие формы:

```
static void fill(boolean array[ ], boolean value)
static void fill(byte array[ ], byte value)
static void fill(char array[ ], char value)
static void fill(double array[ ], double value)
static void fill(float array[ ], float value)
static void fill(int array[ ], int value)
static void fill(long array[ ], long value)
static void fill(short array[ ], short value)
static void fill(Object array[ ], Object value)
```

Здесь `value` — значение, назначаемое всем элементам массива.

Вторая версия метода `fill()` присваивает значения подмножеству массива. Она имеет такие формы:

```
static void fill(boolean array[ ], int start, int end, boolean value)
static void fill(byte array[ ], int start, int end, byte value)
static void fill(char array[ ], int start, int end, char value)
static void fill(double array[ ], int start, int end, double value)
static void fill(float array[ ], int start, int end, float value)
static void fill(int array[ ], int start, int end, int value)
static void fill(long array[ ], int start, int end, long value)
static void fill(short array[ ], int start, int end, short value)
static void fill(Object array[ ], int start, int end, Object value)
```

Здесь `value` — значение, которым заполняются элементы массива, расположенные от позиции `start` до позиции `end - 1`. Все перечисленные методы могут выбрасывать исключение типа `IllegalArgumentException`, если `start` большее чем `end`, или типа `ArrayIndexOutOfBoundsException`, если `start` или `end` находятся вне границ массива.

Метод `sort()` сортирует массив так, чтобы он был размещен в восходящем порядке. Метод `sort()` имеет две версии. Первая версия сортирует весь массив:

```
static void sort(byte array[ ])
static void sort(char array[ ])
static void sort(double array[ ])
static void sort(float array[ ])
static void sort(int array[ ])
static void sort(long array[ ])
static void sort(short array[ ])
static void sort(Object array[ ])
static void sort(Object array[ ], Comparator c)
```

Здесь *array* — массив, который нужно отсортировать. В последней форме *c* — объект типа *Comparator*, который используется для упорядочивания элементов массива. Формы, которые сортируют *Object*-массивы, могут также выбрасывать исключения типа *ClassCastException*, если элементы сортируемого массива не сопоставимы.

Вторая версия *sort()* дает возможность определить диапазон в сортируемом массиве. Ее формы таковы:

```
static void sort(byte array[ ], int start, int end)
static void sort(char array[ ], int start, int end)
static void sort(double array[ ], int start, int end)
static void sort(float array[ ], int start, int end)
static void sort(int array[ ], int start, int end)
static void sort(long array[ ], int start, int end)
static void sort(short array[ ], int start, int end)
static void sort(Object array[ ], int start, int end)
static void sort(Object array[ ], int start, int end, Comparator c)
```

Здесь будет отсортирован диапазон, начинающийся в *start* и простирающийся до *end* — 1 внутри массива. В последней форме *c* — объект типа *Comparator*, который используется для упорядочивания элементов массива. Все эти методы могут выбрасывать исключение типа *IllegalArgumentException*, если *start* больше чем *end*, или типа *ArrayIndexOutOfBoundsException*, если *start* или *end* находятся вне границ массива. Две последние формы могут также выбрасывать исключение типа *ClassCastException*, если элементы сортируемого массива не сопоставимы.

Следующая программа иллюстрирует использование некоторых методов класса *Arrays*:

```
// Демонстрирует класс Arrays.
import java.util.*;

class ArraysDemo {
    public static void main(String args[]) {
        // распределить в памяти и инициализировать массив
        int array[] = new int[10];
        for(int i = 0; i < 10; i++)
            array[i] = -3 * i;

        // показать, сортировать, показать
        System.out.print("Исходное содержимое: ");
        display(array);
        Arrays.sort(array);
        System.out.print("Отсортированное содержимое: ");
        display(array);
    }
}
```

```

// заполнить и показать
Arrays.fill(array, 2, 6, -1);
System.out.print("После fill(): ");
display(array);

// сортировать и показать
Arrays.sort(array);
System.out.print("Опять после сортировки: ");
display(array);

// двоичный поиск значения -9
System.out.print("Значение -9 находится в позиции");
int index =
    Arrays.binarySearch(array, -9);
System.out.println(index);
}

static void display(int array[]) {
    for(int i = 0; i < array.length; i++)
        System.out.print(array[i] + " ");
    System.out.println("");
}
}
}

```

Вывод этой программы:

```

Исходное содержимое: 0 -3 -6 -9 -12 -15 -18 -21 -24 -27
Отсортированное содержимое: -27 -24 -21 -18 -15 -12 -9 -6 -3 0
После fill(): -27 -24 -1 -1 -1 -9 -6 -3 0
Опять после сортировки: -27 -24 -9 -6 -3 -1 -1 -1 -1 0
Значение -9 находится в позиции 2

```

## Наследованные классы и интерфейсы

Как объяснялось в начале этой главы, первоначальная версия `java.util` не включала структуру коллекций. Вместо этого в ней определялось несколько классов и интерфейсов, которые обеспечивали специальный (*ad hoc*<sup>1</sup>) метод хранения объектов. С добавлением коллекций Java 2, некоторые из первоначальных классов были перепроектированы для того, чтобы поддержать интерфейсы коллекций. Таким образом, они полностью совместимы со структурой коллекций. Хотя никакие классы фактически не были исключены, один из них был представлен как устаревший. Конечно, когда коллекция дублирует функциональные возможности наследованного (из Java предыдущих версий) класса, для нового кода следует использовать именно коллек-

<sup>1</sup> Ad hoc (лат.) — подходящий только для данного случая. — Примеч. пер.

цию. Вообще, наследованные классы поддерживаются потому, что существует большая база кодов, которая все еще их использует (включая коды, применяемые в Java 2 API).

Ни один из классов коллекций не синхронизирован, тогда как все наследованные классы — синхронизированы. Это различие может быть важным в некоторых ситуациях. Конечно, коллекции тоже можно легко синхронизировать, используя один из алгоритмов, определенных в классе `Collections`.

Ниже перечислены наследованные классы, определенные в пакете `java.util`:

- `Dictionary`
- `Stack`
- `Hashtable`
- `Vector`
- `Properties`

Существует также один наследованный интерфейс `Enumeration`. В следующих разделах рассматривается интерфейс `Enumeration` и каждый из наследованных классов.

## Интерфейс *Enumeration*

Интерфейс `Enumeration` определяет методы, с помощью которых вы можете *перечислить* (получить по одному) элементы коллекции объектов. Этот наследованный интерфейс намного уступает `Iterator`-интерфейсу коллекций. `Enumeration` рассматривается устаревшим для нового кода, хотя и не исключен из пакета. Однако он используется некоторыми методами наследованных классов (таких как `Vector` и `Properties`), другими API-классами и в настоящее время широко применяется в прикладных кодах.

В `Enumeration` определено два следующих метода:

```
boolean hasMoreElements()  
Object nextElement()
```

После реализации метод `hasMoreElements()` должен возвратить `true`, если в перечислении еще присутствует некоторое количество элементов для извлечения, и `false`, когда все элементы уже были перечислены. Метод `nextElement()` возвращает следующий объект перечисления в виде ссылки на родовой объект (типа `Object`). То есть каждое обращение к `nextElement()` получает следующий объект перечисления. Вызывающая подпрограмма должна привести тип этого объекта к типу объекта, содержащегося в перечислении.

## Класс *Vector*

Класс `Vector` реализует динамический массив. Он подобен классу `ArrayList`, но с двумя отличиями: `Vector` синхронизирован и содержит много наследо-

ванных методов, которые не являются частью структуры коллекций. С выпуском Java 2 класс `Vector` был перепроектирован так, чтобы расширить класс `AbstractList` и реализовать интерфейс `List`. Поэтому он теперь полностью совместим с коллекциями.

Конструкторы класса `Vector` такие:

```
Vector()
Vector(int size)
Vector(int size, int incr)
Vector(Collection c)
```

Первая форма создает заданный умалчивающий вектор, который имеет начальный размер 10. Вторая форма образует вектор, чья начальная емкость (размер) определяется параметром `size`. Третья форма — вектор, чья начальная емкость определяется в `size`, а приращение емкости указывается в параметре `incr`. Приращение задает число элементов, выделяемых вектору каждый раз, когда он увеличивается в размерах. Четвертая форма создает вектор, который содержит элементы коллекции `c`. Этот конструктор был добавлен в Java 2.

Все векторы начинают свое существование с некоторой начальной емкости. После того как эта начальная емкость исчерпана (в тот момент, когда вы пытаетесь сохранить в векторе очередной объектный компонент), вектору автоматически распределяется пространство для этого компонента плюс добавочный участок памяти для дополнительных компонентов. Распределение большей памяти, чем требуется для размещения очередного компонента, приводит к уменьшению числа дополнительных распределений. Это важно потому, что, т. к. на распределения затрачивается много дополнительного времени, сокращение числа таких распределений приводит к уменьшению времени выполнения (т. е. к повышению эффективности) программы. Количество дополнительного пространства, распределяемого в течение каждого перераспределения, задается приращением, которое вы указываете во втором аргументе конструктора, когда создаете вектор (см. параметр `incr` конструктора). Если вы не определяете приращения, размер вектора удваивается на каждом цикле распределения.

В классе `Vector` определяются следующие защищенные (`protected`) компоненты данных:

```
int capacityIncrement;
int elementCount;
Object elementData[];
```

Значение приращения (памяти при распределении) сохраняется в переменной `capacityIncrement`. Текущее число элементов вектора хранится в `elementCount`. Массив объектных компонентов вектора хранится в `elementData[]`.

В дополнение к коллекционным методам из интерфейса *List* в классе *Vector* определено несколько наследованных методов, которые перечислены в табл. 15.14.

Таблица 15.14. Методы класса *Vector*

Метод	Описание
<code>final void addElement(Object element)</code>	Объект, указанный параметром <i>element</i> , добавляется к вектору
<code>final int capacity()</code>	Возвращает емкость (число элементов) вектора
<code>Object clone()</code>	Возвращает клон (дубликат) вектора вызова
<code>final boolean contains(Object element)</code>	Возвращает <code>true</code> , если <i>element</i> содержится в векторе, и <code>false</code> , если нет
<code>final void copyInto(Object array[ ])</code>	Элементы, содержащиеся в вызывающем векторе, копируются в массив <i>array</i>
<code>final Object elementAt(int index)</code>	Возвращает элемент (объект), расположенный в позиции <i>index</i>
<code>final Enumeration elements()</code>	Возвращает перечисление элементов в векторе
<code>final void ensureCapacity(int size)</code>	Устанавливает минимальную емкость (размер) вектора, как указывает <i>size</i>
<code>final Object firstElement()</code>	Возвращает первый элемент вектора
<code>final int indexOf(Object element)</code>	Возвращает индекс первого размещения объекта <i>element</i> в вызывающем векторе. Если объект не расположен в векторе, возвращает <code>-1</code>
<code>final int indexOf(Object element, int start)</code>	Возвращает индекс первого размещения объекта <i>element</i> , начиная с позиции <i>start</i> или после нее. Если объект не находится в этой части вектора, возвращает <code>-1</code>
<code>final void insertElementAt(Object element, int index)</code>	Добавляет <i>element</i> к вызывающему вектору в положение, указанное в <i>index</i>

Таблица 15.14 (продолжение)

Метод	Описание
<code>final boolean isEmpty()</code>	Возвращает <code>true</code> , если вектор пуст, и <code>false</code> , если он содержит один или большее количество элементов
<code>final Object lastElement()</code>	Возвращает последний элемент вектора
<code>final int lastIndexOf(Object element)</code>	Возвращает индекс последнего размещения объекта <code>element</code> . Если объект не находится в векторе, возвращает <code>-1</code>
<code>final int lastIndexOf(Object element, int start)</code>	Возвращает индекс последнего размещения объекта <code>element</code> перед индексом <code>start</code> . Если объект не в этой части вектора, возвращает <code>-1</code>
<code>final void removeAllElements()</code>	Удаляет все элементы вектора. После того, как этот метод выполняется, размер вектора становится нулевым
<code>final boolean removeElement (Object element)</code>	Удаляет <code>element</code> из вектора. Если в векторе существуют несколько экземпляров указанного объекта, то удаляется первый. Возвращает <code>true</code> , если удаление успешно, и <code>false</code> , если объект не найден
<code>final void removeElementAt(int index)</code>	Удаляет элемент с номером, указанным в <code>index</code>
<code>final void setElementAt (Object element, int index)</code>	Объект <code>element</code> устанавливается в положение, указываемое параметром <code>index</code>
<code>final void setSize(int size)</code>	Устанавливает число элементов в векторе, как указывает параметр <code>size</code> . Если новый размер меньше чем старый, элементы теряются. Если новый размер больше чем старый, добавляются <code>null</code> -элементы
<code>final int size()</code>	Возвращает текущее число элементов вектора
<code>String toString()</code>	Возвращает строчный эквивалент вектора

Таблица 15.14 (окончание)

Метод	Описание
<code>final void trimToSize()</code>	Устанавливает емкость (размер) вектора равным числу элементов, которые в настоящее время в нем содержатся

Поскольку класс `Vector` реализует интерфейс `List`, вы можете использовать вектор точно так же, как вы используете экземпляр типа `ArrayList`. Вы можете манипулировать вектором, вызывая наследованные методы этого класса. Например, после создания экземпляра класса `Vector`, вы можете добавить в него элементы, обратясь к методу `addElement()`. Чтобы получить элемент с определенным номером (индексом), вызывайте метод `elementAt()`. Первый элемент вектора вы получите, вызвав метод `firstElement()`. Чтобы получить последний элемент, вызовите метод `lastElement()`. Индекс элемента можно получить, используя методы `indexOf()` и `lastIndexOf()`. Для удаления элемента вызовите метод `removeElement()` или `removeElementAt()`.

Следующая программа использует вектор, чтобы хранить различные типы числовых объектов. Она демонстрирует некоторые наследованные методы класса `Vector`, а также интерфейс `Enumeration`.

```
// Демонстрирует различные операции класса Vector.
import java.util.*;

class VectorDemo {
    public static void main(String args[]) {
        // исходный размер 3, инкремент равен 2
        Vector v = new Vector(3, 2);

        System.out.println("Исходный размер: " + v.size());
        System.out.println("Исходная емкость: " + v.capacity());

        v.addElement(new Integer(1));
        v.addElement(new Integer(2));
        v.addElement(new Integer(3));
        v.addElement(new Integer(4));

        System.out.println("Емкость после четырех добавлений: " +
                           v.capacity());
        v.addElement(new Double(5.45));
        System.out.println("Текущая емкость: " + v.capacity());
        v.addElement(new Double(6.08));
        v.addElement(new Integer(7));
```

```

System.out.println("Текущая емкость: " + v.capacity());
v.addElement(new Float(9.4));
v.addElement(new Integer(10));

System.out.println("Текущая емкость: " + v.capacity());
v.addElement(new Integer(11));
v.addElement(new Integer(12));

System.out.println("Первый элемент: " + (Integer)v.firstElement());
System.out.println("Последний элемент: " +(Integer)v.lastElement());

if(v.contains(new Integer(3)))
    System.out.println("Вектор содержит 3.");

// перечислить элементы вектора
Enumeration vEnum = v.elements();

System.out.println("\nЭлементы вектора:");
while(vEnum.hasMoreElements())
    System.out.print(vEnum.nextElement() + " ");
System.out.println();
}
}

```

Вывод этой программы:

```

Исходный размер: 0
Исходная емкость: 3
Емкость после четырех добавлений: 5
Текущая емкость: 5
Текущая емкость: 7
Текущая емкость: 9
Первый элемент: 1
Последний элемент: 12
Вектор содержит 3.

```

```

Элементы вектора:
1 2 3 4 5.45 6.08 7 9.4 10 11 12

```

С выпуском Java 2 в класс `Vector` добавлена поддержка итераторов. Вместо того, чтобы полагаться на перечисление для циклического прохода объектов (как это делает предыдущая программа), вы теперь можете применять итератор. Например, в программу можно подставить следующий код, использующий итератор:

```

// Использование итератора для показа содержимого.
Iterator vItr = v.iterator();

System.out.println("\nElements in vector:");
while(vItr.hasNext())

```

```
System.out.print(vItr.next() + " ");
System.out.println();
```

Поскольку перечисления не рекомендуются для нового кода, для перечисления содержимого вектора следует применять итератор. Конечно, существует много наследованного кода, который использует перечисления. К счастью, перечисления и итераторы работают примерно в одной манере.

## Класс *Stack*

Класс *Stack* является подклассом *Vector*-класса, который реализует стандартный стек LIFO<sup>1</sup>. В *Stack* определен только один (задаваемый по умолчанию) конструктор, который создает пустой стек. *Stack* включает все методы, определенные в классе *Vector*, и добавляет несколько собственных, показанных в табл. 15.15.

**Таблица 15.15. Методы класса *Stack***

Метод	Описание
<code>boolean empty()</code>	Возвращает <code>true</code> , если стек пуст, и <code>false</code> , если стек содержит элементы
<code>Object peek()</code>	Возвращает элемент на вершине стека, но не удаляет его
<code>Object pop()</code>	Возвращает элемент на вершине стека, удаляя его
<code>Object push(Object element)</code>	Помещает <code>element</code> в стек и возвращает его
<code>int search(Object element)</code>	Ищет <code>element</code> в стеке. Если находит, то возвращает его смещение от вершины стека. Иначе возвращает <code>-1</code>

Чтобы поместить объект на вершину стека, вызовите метод `push()`. Для удаления и возвращения верхнего элемента вызовите метод `pop()`. Если вы обращаетесь к `pop()`, когда вызывающий стек пуст, выбрасывается исключение типа `EmptyStackException`. Метод `peek()` можно использовать для того, чтобы возвратить, но не удалять, верхний объект. Метод `empty()` возвращает `true`, если стек пуст. Метод `search()` определяет, существует ли объект в стеке, и возвращает количество выталкиваний, которые требуются, чтобы вынести его на вершину стека. Ниже приведен пример, который создает стек, помещает в него несколько объектов типа `Integer` и затем снова выталкивает их:

<sup>1</sup> LIFO (Last-In, First-Out) — "последним вошел, первым вышел". — Примеч. пер.

```
// Демонстрирует класс Stack.
import java.util.*;

class StackDemo {
    static void showpush(Stack st, int a) {
        st.push(new Integer(a));
        System.out.println("push(" + a + ")");
        System.out.println("stack: " + st);
    }

    static void showpop(Stack st) {
        System.out.print("pop -> ");
        Integer a = (Integer) st.pop();
        System.out.println(a);
        System.out.println("stack: " + st);
    }

    public static void main(String args[]) {
        Stack st = new Stack();

        System.out.println("stack: " + st);
        showpush(st, 42);
        showpush(st, 66);
        showpush(st, 99);
        showpop(st);
        showpop(st);
        showpop(st);
        try {
            showpop(st);
        } catch (EmptyStackException e) {
            System.out.println("пустой стек");
        }
    }
}
```

Далее следует вывод этой программы. Обратите внимание, что обработчик исключения `EmptyStackException` включается так, чтобы вы могли изящно обработать исчезновение значений в стеке:

```
stack: []
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop ->> 99
stack: [42, 66]
pop ->> 66
```

```
stack: [42]
pop -> 42
stack: []
pop -> пустой стек
```

## Класс *Dictionary*

*Dictionary* — абстрактный класс, который представляет архив для хранения данных типа ключ/значение и работает во многом аналогично классу *Map*. Задавая ключи, вы можете сохранять значения в объекте класса *Dictionary*. Как только значение сохранено, вы можете отыскать и извлечь его, используя ключ. Таким образом, подобно карте отображений, словарь можно представлять себе как список пар ключ/значение. Хотя класс *Dictionary* фактически не исключен из Java 2, он классифицирован как устаревший, потому что сильно уступает по своим свойствам классу *Map*. Однако в настоящее время *Dictionary* применяется довольно широко, и поэтому мы обсудим его подробнее.

Абстрактные методы, определенные в *Dictionary*, перечислены в табл. 15.16.

**Таблица 15.16. Абстрактные методы класса *Dictionary***

Метод	Цель
<code>Enumeration elements()</code>	Возвращает перечисление значений, содержащихся в словаре
<code>Object get(Object key)</code>	Возвращает объект, который содержит значение, связанное с ключом <code>key</code> . Если <code>key</code> -ключа нет в словаре, возвращает <code>null</code>
<code>boolean isEmpty()</code>	Возвращает <code>true</code> , если словарь пуст, и <code>false</code> , если он содержит по крайней мере один ключ
<code>Enumeration keys()</code>	Возвращает перечисление ключей, содержащихся в словаре
<code>Object put(Object key, Object value)</code>	Вставляет ключ и его значение в словарь. Возвращает <code>null</code> , если <code>key</code> -ключа еще нет в словаре; возвращает предыдущее значение, связанное с <code>key</code> , если этот ключ в словаре уже присутствует
<code>Object remove(Object key)</code>	Удаляет ключ и его значение. Возвращает значение, связанное с ключом <code>key</code> . Если ключа <code>key</code> нет в словаре, возвращает <code>null</code>
<code>int size()</code>	Возвращает число входов в словарь

Для добавления в словарь ключа и значения используйте метод `put()`. Вызывайте метод `get()`, чтобы отыскать и извлечь значение данного ключа. Перечисления ключей и значений могут возвратить методы `keys()` и `elements()`, соответственно. Метод `size()` служит для определения числа пар ключ/значение, хранящихся в словаре, а метод `isEmpty()` возвращает `true`, когда словарь пуст. Чтобы удалить пару ключ/значение, вы можете использовать метод `remove()`.

### Замечание

Класс `Dictionary` считается устаревшим. Вместо него для использования функциональных возможностей хранения данных типа ключ/значение следует реализовывать `Мап-интерфейс`.

## Класс `Hashtable`

Класс `Hashtable` был частью исходного пакета `java.util` и является конкретной реализацией `Dictionary`. Однако в Java 2 `Hashtable` перепроектирован так, что он реализует еще и интерфейс `Map`. Таким образом, `Hashtable` теперь интегрирован в структуру коллекций. Он подобен классу `HashMap`, но синхронизирован.

Как и `HashMap`, `Hashtable` хранит пары ключ/значение в хэш-таблице. При применении `hashtable` вы указываете объект, который используется как ключ, и значение, которое хотите связать с этим ключом. Затем ключ хешируется (рандомизируется), и результирующий хэш-код используется как индекс (номер элемента), с которым значение хранится внутри таблицы.

Хэш-таблица может хранить только объекты, которые переопределяют методы `hashCode()` и `equals()` класса `Object`. Метод `hashCode()` должен вычислить и возвратить хэш-код объекта. Метод `equals()` сравнивает два объекта. К счастью, многие из встроенных классов Java уже реализуют метод `hashCode()`. Например, наиболее общий тип `Hashtable` использует `String`-объект как ключ. Класс `String` реализует как метод `hashCode()`, так и метод `equals()`.

Конструкторы `Hashtable` таковы:

```
Hashtable()
Hashtable(int size)
Hashtable(int size, float fillRatio)
Hashtable(Map m)
```

Первая версия — это конструктор, используемый по умолчанию. Вторая создает хэш-таблицу, которая имеет начальный размер, указанный параметром `size`. Третья версия организует хэш-таблицу, которая имеет начальный размер, указанный в `size`, и коэффициент заполнения, указанный в `fillRatio`.

Данный коэффициент должен иметь значение между 0.0 и 1.0. Он определяет, насколько заполненной может быть хэш-таблица, прежде чем она увеличит свой размер. Когда число элементов становится больше, чем емкость хэш-таблицы умноженная на ее коэффициент заполнения, хэш-таблица расширяется. Если вы не указываете параметр коэффициента заполнения, то используется коэффициент 0.75. Наконец, четвертая версия создает хэш-таблицу, которая инициализируется элементами карты `m`. Емкость хэш-таблицы устанавливается как удвоенное число элементов `m` и используется умалчивающий коэффициент заполнения (загрузки) 0.75. Четвертый конструктор был добавлен в Java 2.

В дополнение к методам интерфейса `Map`, которые теперь реализует `Hashtable`, в `Hashtable` определены наследованные методы, перечисленные в табл. 15.17.

**Таблица 15.17. Наследованные методы класса `Hashtable`**

Метод	Описание
<code>void clear()</code>	Сбрасывает и освобождает хэш-таблицу
<code>Object clone()</code>	Возвращает клон (дубликат) вызывающего объекта
<code>boolean contains (Object value)</code>	Возвращает <code>true</code> , если в хэш-таблице существует некоторое значение, равное <code>value</code> . Возвращает <code>false</code> , если такое значение не найдено
<code>boolean containsKey (Object key)</code>	Возвращает <code>true</code> , если в хэш-таблице существует некоторый ключ, равный <code>key</code> . Возвращает <code>false</code> , если такой ключ не найден
<code>boolean containsValue (Object value)</code>	Возвращает <code>true</code> , если в хэш-таблице существует некоторое значение, равное <code>value</code> . Возвращает <code>false</code> , если такое значение не найдено. (Этот не <code>Map</code> -метод добавлен в Java 2 для согласованности)
<code>Enumeration elements()</code>	Возвращает перечисление значений, содержащихся в хэш-таблице
<code>Object get(Object key)</code>	Возвращает объект, который содержит значение, связанное с <code>key</code> . Если ключ не находится в хэш-таблице, то возвращает <code>null</code> -объект
<code>boolean isEmpty()</code>	Возвращает <code>true</code> , если хэш-таблица пуста, и <code>false</code> , если она содержит, по крайней мере, один ключ
<code>Enumeration keys()</code>	Возвращает перечисление ключей, содержащихся в хэш-таблице

Таблица 15.17 (окончание)

Метод	Описание
<code>Object put(Object key,               Object value)</code>	Вставляет ключ и значение в хэш-таблицу. Возвращает null, если ключ еще не в хэш-таблице. Возвращает предыдущее значение, связанное с ключом <code>key</code> , если ключ уже в хэш-таблице
<code>void rehash()</code>	Увеличивает размер хэш-таблицы и перемешивает все ее ключи
<code>Object remove(Object key)</code>	Удаляет ключ <code>key</code> и его значение. Возвращает значение, связанное с ключом <code>key</code> . Если ключ не находится в хэш-таблице, возвращает null-объект
<code>int size()</code>	Возвращает число входов в хэш-таблицу
<code>String toString()</code>	Возвращает строчный эквивалент хэш-таблицы

Следующий пример переделывает показанную ранее программу банковских счетов так, чтобы она использовала класс `Hashtable` для хранения имен банковских депозиторов и их текущих балансов:

```
// Демонстрирует класс Hashtable.
import java.util.*;

class HTDemo {
    public static void main(String args[]) {
        Hashtable balance = new Hashtable();
        Enumeration names;
        String str;
        double bal;

        balance.put("John Doe", new Double(3434.34));
        balance.put("Tom Smith", new Double(123.22));
        balance.put("Jane Baker", new Double(1378.00));
        balance.put("Tod Hall", new Double(99.22));
        balance.put("Ralph Smith", new Double(-19.08));

        // показать все балансы хэш-таблицы
        names = balance.keys();
        while(names.hasMoreElements()) {
            str = (String) names.nextElement();
            System.out.println(str + ": " + balance.get(str));
        }

        System.out.println();

        // перечислить 1000 на депозитный счет Джона До (John Doe)
        bal = ((Double)balance.get("John Doe")).doubleValue();
    }
}
```

```

balance.put("John Doe", new Double(bal+1000));
System.out.println("Новый баланс John Doe: " +
                    balance.get("John Doe"));
}
}

```

Вывод этой программы:

```

Ralph Smith: -19.08
Tom Smith: 123.22
John Doe: 3434.34
Todd Hall: 99.22
Jane Baker: 1378.0

Новый баланс John Doe: 4434.34

```

И последнее, подобно классам карт отображений, `Hashtable` непосредственно не поддерживает итераторы. Таким образом, предшествующая программа использует перечисление, чтобы отобразить содержимое баланса. Однако вы можете получить представление хэш-таблицы в виде набора, которое разрешает использование итераторов. Для этого просто используйте один из методов коллекционного представления, определенных в `Map`-интерфейсе Java 2, например `entrySet()` или `keySet()`. К тому же, вы можете получить представление в виде набора (`set-view`) ключей и выполнять итерации через них. Имеется переделанная версия программы, которая показывает эту технику:

```

// Использование итераторов с классом Hashtable.
import java.util.*;

class HTDemo2 {
    public static void main(String args[]) {
        Hashtable balance = new Hashtable();
        String str;
        double bal;

        balance.put("John Doe", new Double(3434.34));
        balance.put("Tom Smith", new Double(123.22));
        balance.put("Jane Baker", new Double(1378.00));
        balance.put("Tod Hall", new Double(99.22));
        balance.put("Ralph Smith", new Double(-19.08));

        // показать все балансы в хэш-таблице
        Set set = balance.keySet();      // получить ключи в виде набора

        // получить итератор
        Iterator itr = set.iterator();
        while(itr.hasNext()) {
            str = (String) itr.next();

```

```

        System.out.println(str + ": " + balance.get(str));
    }

    System.out.println();

    // перечислить 1000 на депозитный счет Джона До (John Doe)
    bal = ((Double)balance.get("John Doe")).doubleValue();
    balance.put("John Doe", new Double(bal+1000));
    System.out.println("John Doe's new balance: " +
        balance.get("John Doe"));
}

}

```

## Класс *Properties*

*Properties* — это подкласс класса *Hashtable*. Он используется для поддержки списков значений, в которых ключ и значение являются *String*-объектами. Класс *Properties* используется многими другими классами Java. Например, это как раз тот тип объектов, который возвращает метод *System.getProperties()* при получении значений переменных среды окружения.

В *Properties* определена следующая экземплярная переменная:

```
Properties defaults;
```

Она содержит заданный по умолчанию список свойств, связанный с объектом *Properties*. В *Properties* определены следующие конструкторы:

```
Properties()
Properties(Properties propDefault)
```

Первая версия создает объект класса (типа) *Properties*, который не имеет никаких умалчиваемых значений. Вторая — создает объект, который использует *propDefault* в качестве своих значений по умолчанию. В обоих случаях список свойств пуст.

В дополнение к методам, которые *Properties* наследует от *Hashtable*, в нем определены методы, перечисленные в табл. 15.18. Класс *Properties* предыдущих версий пакета *java.util* содержал также один исключенный в Java 2 метод — *save()*. В Java 2 этот метод заменен на метод *store()*, потому что *save()* не обрабатывал правильно ошибки.

**Таблица 15.18. Наследованные методы класса *Properties***

Метод	Описание
<b>String getProperty(String key)</b>	Возвращает значение, связанное с ключом <i>key</i> . Возвращает null-объект, если <i>key</i> не обнаружен ни в вызывающем, ни в умалчивающем списке свойств

Таблица 15.18 (окончание)

Метод	Описание
<code>String getProperty (String key, String defaultProperty)</code>	Возвращает значение, связанное с ключом <code>key</code> . Возвращает <code>defaultProperty</code> , если <code>key</code> не обнаружен ни в вызывающем, ни в умалчивающем списке свойств
<code>void list(PrintStream streamOut)</code>	Посыпает список свойств в выходной поток, связанный с <code>PrintStream streamOut</code>
<code>void list(PrintWriter streamOut)</code>	Посыпает список свойств в выходной поток, связанный с <code>PrintWriter streamOut</code>
<code>void load(InputStream streamIn) throws IOException</code>	Вводит список свойств из входного потока, связанного с <code>InputStream streamIn</code>
<code>Enumeration propertyNames()</code>	Возвращает перечисление ключей, а также содержит ключи, найденные в умалчивающем списке свойств
<code>Object setProperty(String key, String value)</code>	Связывает <code>key</code> с <code>value</code> . Возвращает предыдущее значение, связанное с ключом, или <code>null</code> , если такой связи не существует. (Добавлен в Java 2)
<code>void store(OutputStream streamOut, String description)</code>	После записи строки, указанной в <code>description</code> , список свойств записывается в выходной поток, связанный с <code>streamOut</code> . (Добавлен в Java 2)

Одна из полезных возможностей класса `Properties` заключается в том, что вы можете определить умалчивающее свойство, которое будет возвращено, если никакое значение не связано с некоторым ключом. Например, умалчивающее значение может быть определено наряду с ключом в методе `getProperty()` так:

```
getProperties("имя", "умалчивающее значение")
```

Если значение "имя" не найдено, то будет возвращено "умалчивающее значение". Если вы создаете объект типа `Properties`, то ему можно передать другой экземпляр `Properties`-объекта, который будет использоваться в качестве умалчивающих свойств нового объекта. В этом случае, если вы вызовите метод `getProperty("foo")` заданного `Properties`-объекта, а "foo" среди его свойств не существует, то Java произведет поиск "foo" в умалчивающем `Properties`-объекте. Допустимо также произвольное вложение уровней умалчивающих свойств.

Следующий пример демонстрирует класс Properties. Он создает список свойств, в котором ключи являются названиями штатов, а значения — названиями их столиц. Обратите внимание, что попытка найти столицу штата Florida включает умалчиваемое значение.

```
// Демонстрирует список свойств (класс Properties).
import java.util.*;

class PropDemo {
    public static void main(String args[]) {
        Properties capitals = new Properties();
        Set states;
        String str;

        capitals.put("Illinois", "Springfield");
        capitals.put("Missouri", "Jefferson City");
        capitals.put("Washington", "Olympia");
        capitals.put("California", "Sacramento");
        capitals.put("Indiana", "Indianapolis");

        // показать все штаты и столицы в хэш-таблице.
        states = capitals.keySet();      // получить ключи в виде набора
        Iterator itr = states.iterator();

        while(itr.hasNext()) {
            str = (String) itr.next();
            System.out.println("Столица штата " + str + " — " +
                capitals.getProperty(str) + ".");
        }

        System.out.println();

        // искомого штата нет в списке — использовать умолчание
        str = capitals.getProperty("Florida", "не найдена");
        System.out.println("Столица штата Florida — " + str + ".");
    }
}
```

**Вывод этой программы:**

```
Столица штата California — Sacramento.
Столица штата Washington — Olympia.
Столица штата Missouri — Jefferson City.
Столица штата Indiana — Indianapolis.
Столица штата Illinois — Springfield.
Столица штата Florida — не найдена.
```

Так как значения Florida нет в текущем списке, используется значение по умолчанию.

Хотя допустимо использование умалчивающего значения при вызове метода `getProperty()`, как показывает предыдущий пример, для большинства приложений списков свойств имеется лучший способ обработки умалчивающих значений. Для большей гибкости укажите (в конструкторе) умалчивающий список свойств при построении `Properties`-объекта. Умалчивающий список будет найден, если желательный ключ не найден в основном списке. Ниже следует слегка переделанная версия предыдущей программы, создающая умалчивающий список штатов (вместе с их столицами). Теперь, когда разыскивается штат Florida, она (и ее столица) будут найдены в списке, заданном по умолчанию:

```
// Использование умалчивающего списка свойств.
import java.util.*;

class PropDemoDef {
    public static void main(String args[]) {
        // создать умалчивающий список свойств
        Properties defList = new Properties();
        defList.put("Florida", "Tallahassee");
        defList.put("Wisconsin", "Madison");

        Properties capitals = new Properties(defList);
        Set states;
        String str;

        capitals.put("Illinois", "Springfield");
        capitals.put("Missouri", "Jefferson City");
        capitals.put("Washington", "Olympia");
        capitals.put("California", "Sacramento");
        capitals.put("Indiana", "Indianapolis");

        // показать все штаты и столицы в хэш-таблице
        states = capitals.keySet();      // получить ключи в виде набора
        Iterator itr = states.iterator();

        while(itr.hasNext()) {
            str = (String) itr.next();
            System.out.println("Столица штата " + str + " - " +
                               capitals.getProperty(str) + ".");
        }

        System.out.println();

        // Florida теперь будет найдена в умалчивающем списке
        str = capitals.getProperty("Florida");
        System.out.println("Столица штата Florida - " + str + ".");
    }
}
```

Вывод этой программы:

Столица штата California – Sacramento.

Столица штата Washington – Olympia.

Столица штата Missouri – Jefferson City.

Столица штата Indiana – Indianapolis.

Столица штата Illinois – Springfield.

Столица штата Florida – Tallahassee.

Обратите внимание, как изменилась последняя строка вывода.

## **Использование методов *store()* и *load()***

Одним из наиболее полезных свойств класса Properties является возможность сохранения (на диске) и загрузки (с диска) информации, содержащейся в Properties-объекте, с помощью методов *store()* и *load()*. Вы в любое время можете записать Properties-объект в поток или считать его обратно. Это делает списки свойств особенно удобными для реализации простых баз данных. Например, следующая программа использует список свойств для создания простой компьютеризированной телефонной книги, которая хранит имена и телефонные номера. Чтобы найти телефонный номер абонента, вы просто вводите его имя. Для сохранения и загрузки списка программа вызывает методы *store()* и *load()*. Во время выполнения программы сначала пробует загрузить список из файла с именем phonebook.dat. Если этот файл существует, список загружается. Затем вы добавляете элементы к списку. После чего новый список сохраняется, когда программа завершает свою работу. Обратите внимание, как мал объем кода, реализующего небольшую компьютеризированную телефонную книгу.

```
/* Простая база телефонных номеров,
   которая использует список свойств. */
import java.io.*;
import java.util.*;

class Phonebook {
    public static void main(String args[])
        throws IOException
    {
        Properties ht = new Properties();
        BufferedReader br =
            new BufferedReader(new InputStreamReader(System.in));
        String name, number;
        FileInputStream fin = null;
        boolean changed = false;

        // попытка открыть файл phonebook.dat
        try {
```

```
fin = new FileInputStream("phonebook.dat");
} catch(FileNotFoundException e) {
    // игнорировать отсутствие файла
}

/* Если phonebook-файл уже существует,
загрузить существующие телефонные номера. */
try {
    if(fin != null) {
        ht.load(fin);
        fin.close();
    }
} catch(IOException e) {
    System.out.println("Error reading file.");
}

// ввод пользователем новых имен и телефонных номеров
do {
    System.out.println("Введите новое имя" +
                       " ('quit' для остановки): ");
    name = br.readLine();
    if(name.equals("quit")) continue;

    System.out.println("Введите номер: ");
    number = br.readLine();

    ht.put(name, number);
    changed = true;
} while(!name.equals("quit"));

// если данные телефонной книги были изменены, сохранить их
if(changed) {
    FileOutputStream fout = new FileOutputStream("phonebook.dat");

    ht.store(fout, "Telephone Book");
    fout.close();
}

// найти номер по заданному имени
do {
    System.out.println("Введите имя" + " ('quit' для завершения): ");
    name = br.readLine();
    if(name.equals("quit")) continue;

    number = (String) ht.get(name);
    System.out.println(number);
} while(!name.equals("quit"));
}
```

## Резюме

Структура коллекций дает программистам мощный набор хорошо спроектированных решений некоторых наиболее общих задач программирования. Используйте коллекции всякий раз, когда необходимо сохранять и извлекать информацию. Помните, коллекции не нужно использовать для только "больших задач" типа ведомственных баз данных, списков почтовых адресов или инвентарных систем. Они также эффективны, когда применяются к небольшим задачам. Например, класс TreeMap сделал бы превосходную коллекцию, содержащую структуру каталога набора файлов. TreeSet мог бы быть весьма полезным для хранения информации управления проектом. Типы проблем, которые смогут извлечь выгоду из решения, основанного на коллекциях, ограничиваются только вашим воображением.

# ГЛАВА 16



## Пакет *java.util*: сервисные классы

В этой главе продолжается обсуждение пакета *java.util* и исследуются те классы и интерфейсы, которые не являются частью структуры коллекций. Они включают классы, которые анализируют строки, работают с датами, вычисляют случайные числа и наблюдают за событиями. В конце главы кратко упомянуты также пакеты *java.util.zip* и *java.util.jar*.

### Класс *StringTokenizer*

Обработка текста часто состоит из синтаксического анализа отформатированной входной строки. *Синтаксический анализ* (*parsing*) — это деление текста на множество дискретных частей или *лексем* (*tokens*), которые в некоторой последовательности могут передавать семантическое значение. Класс  *StringTokenizer*, часто называемый *лексическим анализатором* или *сканером*, обеспечивает первый шаг в процессе такого анализа. Он реализует интерфейс *Enumeration*. Поэтому при заданной входной строке вы можете перечислять индивидуальные лексемы, содержащиеся в ней, используя класс  *StringTokenizer*.

Для применения  *StringTokenizer* вы указываете входную строку и строку, которая содержит разделители. *Разделители* — это символы, отделяющие лексемы друг от друга. Каждый символ в *строке разделителей* рассматривается как допустимый разделитель, например, ", ; :" устанавливает в качестве разделителей запятую, точку с запятой и двоеточие. Заданный по умолчанию набор разделителей состоит из пробельных (*whitespace*) символов: пробела, символа табуляции, символа новой строки (*newline*, *LF*) и перевода каретки (*CR*).

Класс  *StringTokenizer* обладает следующими конструкторами:

```
 StringTokenizer(String str)
 StringTokenizer(String str, String delimiters)
 StringTokenizer(String str, String delimiters, boolean delimsAsTokens)
```

Во всех версиях параметр `str` — анализируемая строка. В первой версии используются разделители, заданные по умолчанию. Во второй и третьей версиях они указываются в параметре `delimiters`. Кроме того, в третьей версии если `delimAsToken` — `true`, то разделители также возвращаются как лексемы при анализе строки. Иначе, разделители не возвращаются. Две первые формы не возвращают разделителей в качестве лексем.

После создания StringTokenizer-объекта для последовательного извлечения лексем используется метод `nextToken()`. Метод `hasMoreTokens()` возвращает `true`, если еще существуют лексемы для работы анализатора. Так как класс StringTokenizer реализует интерфейс Enumeration, то методы `hasMoreElements()` и `nextElement()` также реализованы и действуют подобно методам `hasMoreTokens()` и `nextToken()`, соответственно. Методы класса StringTokenizer представлены в табл. 16.1.

**Таблица 16.1. Методы класса StringTokenizer**

Метод	Описание
<code>int countTokens()</code>	Используя текущий набор разделителей, метод определяет число оставленных для анализа лексем и возвращает результат
<code>boolean hasMoreElements()</code>	Возвращает <code>true</code> , если одна или несколько лексем остаются в строке, и <code>false</code> , если нет ни одной
<code>boolean hasMoreTokens()</code>	Возвращает <code>true</code> , если одна или несколько лексем остаются в строке, и <code>false</code> , если нет ни одной
<code>Object nextElement()</code>	Возвращает следующую лексему как объект класса <code>Object</code>
<code>String nextToken()</code>	Возвращает следующую лексему как <code>String</code> -объект
<code>String nextToken(String delimiters)</code>	Возвращает следующую лексему как <code>String</code> -объект и устанавливает строку разделителей, указанную в <code>delimiters</code>

Ниже приведен пример, который создает объект типа StringTokenizer для анализа пар "ключ = значение". Последовательные наборы пар "ключ = значение" разделяются точкой с запятой.

```
// Демонстрирует строковый класс StringTokenizer.
import java.util.StringTokenizer;
```

```

class STDemo {
    static String in = "название=Java: The Complete Reference;" +
        "автор=Naughton и Schildt;" +"издатель=Osborne/McGraw-Hill;" +
        "copyright=1999";

    public static void main(String args[]) {
        StringTokenizer st = new StringTokenizer(in, "=");

        while(st.hasMoreTokens()) {
            String key = st.nextToken();
            String val = st.nextToken();
            System.out.println(key + "\t" + val);
        }
    }
}

```

Вывод из этой программы:

```

название Java: The Complete Reference
автор Naughton и Schildt
издатель Osborne/McGraw-Hill
copyright 1999

```

## Класс *BitSet*

Класс *BitSet* создает специальный тип массива, который содержит битовые значения. Массив может увеличиваться в размере, насколько нужно. И это делает его похожим на битовый вектор. Конструкторы класса *BitSet*:

```

BitSet()
BitSet(int size)

```

Первая версия создает объект по умолчанию. Вторая — позволяет указывать его начальный размер (т. е. число битов, которые он может содержать). Все биты инициализируются нулем.

*BitSet* реализует интерфейс *Cloneable* и определяет методы, перечисленные в табл. 16.2.

**Таблица 16.2. Методы класса *BitSet***

Метод	Описание
<code>void and(BitSet bitSet)</code>	Логическая операция AND содержимого вызывающего <i>BitSet</i> -объекта с объектом <i>bitSet</i> . Результат помещается в объект вызова

Таблица 16.2 (окончание)

Метод	Описание
<code>void andNot(BitSet bitSet)</code>	Для каждого единичного бита в <code>bitSet</code> , соответствующий бит в вызывающем BitSet-объекте очищается (сбрасывается в 0). (Добавлен в Java 2)
<code>void clear(int index)</code>	Обнуляет бит с номером <code>index</code>
<code>Object clone()</code>	Дублирует вызывающий BitSet-объект
<code>boolean equals(Object bitSet)</code>	Возвращает <code>true</code> , если множество битов вызова эквивалентно указанному в <code>bitSet</code> . Иначе, метод возвращает <code>false</code>
<code>boolean get(int bitIndex)</code>	Возвращает текущее состояние бита с номером <code>bitIndex</code>
<code>int hashCode()</code>	Возвращает код мусора для вызывающего объекта
<code>int length()</code>	Возвращает число битов, требуемых для размещения вызывающего BitSet-объекта. Это значение определяется положением последнего единичного бита. (Добавлен в Java 2)
<code>void or(BitSet bitSet)</code>	Логическая операция OR содержимого вызывающего BitSet-объекта с указанным в <code>bitSet</code> . Результат помещается в объект вызова
<code>void set(int index)</code>	Устанавливает (в 1) бит с номером <code>index</code>
<code>int size()</code>	Возвращает число битов в вызывающем BitSet-объекте
<code>String toString()</code>	Возвращает строчный эквивалент вызывающего BitSet-объекта
<code>void xor(BitSet bitSet)</code>	Логическая операция XOR содержимого вызывающего BitSet-объекта с указанным в <code>bitSet</code> . Результат помещается в объект вызова

Ниже представлен пример, который демонстрирует BitSet:

```
// Демонстрация класса BitSet.
import java.util.BitSet;

class BitSetDemo {
    public static void main(String args[]) {

```

```
BitSet bits1 = new BitSet(16);
BitSet bits2 = new BitSet(16);

// установить некоторые биты
for(int i=0; i<16; i++) {
    if((i%2) == 0) bits1.set(i);
    if((i%5) != 0) bits2.set(i);
}

System.out.println("Исходный образ bits1: ");
System.out.println(bits1);
System.out.println("\nИсходный образ bits2: ");
System.out.println(bits2);

// операция AND с битами
bits2.and(bits1);
System.out.println("\nbits2 AND bits1: ");
System.out.println(bits2);

// операция OR с битами
bits2.or(bits1);
System.out.println("\nbits2 OR bits1: ");
System.out.println(bits2);

// операция XOR с битами
bits2.xor(bits1);
System.out.println("\nbits2 XOR bits1: ");
System.out.println(bits2);
}
```

Когда `toString()` преобразует `BitSet`-объект в его строчный эквивалент, каждый бит набора представляется его разрядной позицией. Сброшенные биты не показываются.

Вывод этой программы:

Исходный образ bits1:  
{0, 2, 4, 6, 8, 10, 12, 14}

Исходный образ bits2:  
{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

bits2 AND bits1:  
{2, 4, 6, 8, 12, 14}

bits2 OR bits1:  
{0, 2, 4, 6, 8, 10, 12, 14}

bits2 XOR bits1:  
{}

## Класс Date

Класс Date инкапсулирует текущую дату и время. Перед началом нашего рассмотрения класса Date, важно указать, что он существенно изменился по сравнению со своей первоначальной версией, определенной в Java 1.0. Когда была выпущена версия Java 1.1, многие из функций, выполнявшихся исходным классом Date, были перемещены в классы Calendar и DateFormat, и в результате многие из исходных методов класса Date были исключены. В Java 2 к классам даты и времени прибавлено несколько новых методов, но реализуются они в форме версии 1.1. Так как исключенные из версии 1.0 методы не должны использоваться для новых кодов, они здесь не описываются.

Класс Date поддерживает следующие конструкторы:

```
Date()
Date(long millisec)
```

Первый конструктор инициализирует объект с текущей датой и временем. Второй принимает один аргумент, который равняется числу миллисекунд, прошедших с полуночи 1 января 1970 г. Неисключенные методы класса Date показаны в табл. 16.3. С появлением Java 2, Date реализует также интерфейс Comparable.

**Таблица 16.3. Неисключенные методы класса Date**

Метод	Описание
<code>boolean after(Date date)</code>	Возвращает true, если вызывающий Date-объект содержит более позднюю дату, чем та, что указана в <code>date</code> . Иначе возвращает false
<code>boolean before(Date date)</code>	Возвращает true, если вызывающий Date-объект содержит более раннюю дату, чем та, что указана в <code>date</code> . Иначе возвращает false
<code>Object clone()</code>	Дублирует вызывающий Date-объект
<code>int compareTo(Date date)</code>	Сравнивает значение вызывающего объекта с объектом параметра <code>date</code> . Возвращает 0, если значения равны. Возвращает отрицательное значение, если вызывающий объект имеет более раннюю дату, чем <code>date</code> . Возвращает положительное значение, если вызывающий объект имеет более позднюю дату, чем <code>date</code> . (Добавлен в Java 2)
<code>int compareTo(Object obj)</code>	Работает тождественно с методом <code>compareTo(Date)</code> , если <code>obj</code> имеет тип Date. Иначе выбрасывает исключение типа <code>ClassCastException</code> . (Добавлен в Java 2)

Таблица 16.3 (окончание)

Метод	Описание
<code>boolean equals(Object date)</code>	Возвращает <code>true</code> , если вызывающий <code>Date</code> -объект содержит то же самое время и дату, что указаны в <code>date</code> . Иначе, возвращает <code>false</code> .
<code>long getTime()</code>	Возвращает число миллисекунд, прошедших с 1 января 1970 г.
<code>int hashCode()</code>	Возвращает код мусора для вызывающего объекта
<code>void setTime(long time)</code>	Устанавливает время и дату, как указано в <code>time</code> (в виде целого числа миллисекунд, прошедших с полночи 1 января 1970 г.)
<code>String toString()</code>	Конвертирует вызывающий объект класса <code>Date</code> в строку и возвращает результат

Нетрудно заметить, просматривая табл. 16.3, что свойства класса `Date` не позволяют определять индивидуальные компоненты даты или времени. Как демонстрирует следующая программа, дату и время можно получить только в терминах миллисекунд или в ее заданном по умолчанию строчном представлении, которое возвращает метод `toString()`. Для выяснения более детальной информации о дате и времени нужно использовать класс `Calendar`.

```
// Показывает дату и время, используя только Date-методы.
import java.util.Date;

class DateDemo {
    public static void main(String args[]) {
        // Создать объект типа Date.
        Date date = new Date();

        // Показать дату и время с помощью toString().
        System.out.println(date);

        // Показать число миллисекунд с полуночи 1 января 1970 г. GMT.
        long msec = date.getTime();
        System.out.println("Milliseconds since Jan. 1, 1970 GMT = " + msec);
    }
}
```

Вывод этого примера:

```
Mon Jan 25 15:06:40 CST 1999
Milliseconds since Jan. 1, 1970 GMT = 917298400228
```

## Сравнение дат

Существуют три способа сравнения двух Date-объектов. Во-первых, вы можете использовать метод `getTime()` для получения числа миллисекунд, прошедших с полуночи 1 января 1970 г. (для обоих объектов), и затем сравнения этих значений. Во-вторых, вы можете использовать методы `before()`, `after()` и `equals()`. Поскольку, например, 12-й месяц наступает перед 18-м, то операция

```
new Date(99, 2, 12).before(new Date(99, 2, 18))
```

возвращает `true`. Наконец, можно использовать метод `compareTo()`, который определен интерфейсом `Comparable` и реализован классом `Date`.

## Класс `Calendar`

Абстрактный класс `Calendar` обеспечивает набор методов, который позволяет преобразовать время в миллисекундах в набор таких полезных компонентов, как год, месяц, день, час, минута и секунда. Предполагается, что подклассы `Calendar` будут обеспечивать определенные функциональные возможности для интерпретации информации времени согласно их собственным правилам. Это один из аспектов библиотеки классов Java, который дает возможность писать программы, способные работать в нескольких интернациональных средах. Пример такого подкласса — `GregorianCalendar`.

`Calendar` не содержит общих (`public`) конструкторов и определяет несколько защищенных (`protected`) экземплярных переменных:

- `boolean areFieldsSet` указывает, были ли установлены компоненты времени;
- `int fields[]` является массивом целых чисел, который содержит компоненты времени;
- `boolean isSet[]` представляет собой массив логических значений, который указывает, был ли установлен определенный временной компонент;
- `long time` содержит текущее время для этого объекта;
- `boolean isTimeSet` указывает, было ли установлено текущее время.

Некоторые, чаще всего используемые, методы класса `Calendar` показаны в табл. 16.4.

**Таблица 16.4. Некоторые методы класса `Calendar`**

Метод	Описание
<code>abstract void add(int which, int val)</code>	Прибавляет <code>val</code> к компоненту времени или даты <code>which</code> . Для вычитания укажите отрицательное значение

Таблица 16.4 (продолжение)

Метод	Описание
(прод.)	<i>which</i> должен быть одним из полей, определенных в <i>Calendar</i> , например, <i>Calendar.HOUR</i>
<b>boolean after(Object calendarObj)</b>	Возвращает <i>true</i> , если вызывающий <i>Calendar</i> -объект содержит более позднюю дату, чем указывает <i>calendarObj</i> . Иначе возвращает <i>false</i>
<b>boolean before(Object calendarObj)</b>	Возвращает <i>true</i> , если вызывающий объект <i>Calendar</i> содержит более раннюю дату, чем указывает <i>calendarObj</i> . Иначе возвращает <i>false</i>
<b>final void clear()</b>	Обнуляет все компоненты времени в вызывающем объекте
<b>final void clear(int which)</b>	Обнуляет компонент времени <i>which</i> в вызывающем объекте
<b>Object clone()</b>	Возвращает дубликат вызывающего объекта
<b>boolean equals(Object calendarObj)</b>	Возвращает <i>true</i> , если вызывающий <i>Calendar</i> -объект содержит дату, которая является равной дате, указанной в <i>calendarObj</i> . Иначе возвращает <i>false</i>
<b>final int get(int calendarField)</b>	Возвращает значение одного компонента вызывающего объекта. Компонент указывает параметр <i>calendarField</i> . Примеры компонентов, которые могут потребоваться программе: <i>Calendar.YEAR</i> , <i>Calendar.MONTH</i> , <i>Calendar.MINUTE</i> и т. д.
<b>static Locale[] getAvailableLocales()</b>	Возвращается массив <i>Locale</i> -объектов, который содержит языковые регионы, для которых доступны календари
<b>static Calendar getInstance()</b>	Возвращает <i>Calendar</i> -объект для заданного по умолчанию языкового региона и часового пояса
<b>static Calendar getInstance(TimeZone tz)</b>	Возвращает <i>Calendar</i> -объект для часового пояса, указанного в <i>tz</i> . Используется языковый регион, заданный по умолчанию

Таблица 16.4 (окончание)

Метод	Описание
<code>static Calendar getInstance(Locale locale)</code>	Возвращает Calendar-объект для языкового региона, указанного в <code>locale</code> . Используется часовой пояс, заданный по умолчанию
<code>static Calendar getInstance(TimeZone tz, Locale locale)</code>	Возвращает Calendar-объект для часового пояса, указанного в <code>tz</code> , и языкового региона, указанного в <code>locale</code>
<code>final Date getTime()</code>	Возвращается Date-объект, эквивалентный времени вызывающего объекта
<code>TimeZone getTimeZone()</code>	Возвращает часовой пояс для вызывающего объекта
<code>final boolean isSet(int which)</code>	Возвращает true, если указанный временной компонент установлен. Иначе возвращает false
<code>final void set(int which, int val)</code>	Устанавливает в компоненте даты или времени <code>which</code> вызывающего объекта значение <code>val</code> . Параметр <code>which</code> должен быть одним из полей, определенных в Calendar, например <code>Calendar.HOUR</code>
<code>final void set(int year, int month, int dayOfMonth)</code>	Устанавливает различные компоненты даты и времени вызывающего объекта
<code>final void set(int year, int month, int dayOfMonth, int hours, int minutes)</code>	Устанавливает различные компоненты даты и времени вызывающего объекта
<code>final void set(int year, int month, int dayOfMonth, int hours, int minutes, int seconds)</code>	Устанавливает различные компоненты даты и времени вызывающего объекта
<code>final void setTime(Date d)</code>	Устанавливает различные компоненты даты и времени вызывающего объекта. Эта информация получается от объекта Date <code>d</code>
<code>void setTimeZone(TimeZone tz)</code>	Устанавливает в компоненте часового пояса вызывающего объекта значение <code>tz</code>

В `Calendar` определены следующие `int`-константы (которые используются, когда вы получаете или устанавливаете компоненты календаря):

- |   |                                      |  |
|---|--------------------------------------|--|
| <input type="checkbox"/> AM                   | <input type="checkbox"/> FRIDAY      | <input type="checkbox"/> PM            |
| <input type="checkbox"/> AM_PM                | <input type="checkbox"/> HOUR        | <input type="checkbox"/> SATURDAY      |
| <input type="checkbox"/> APRIL                | <input type="checkbox"/> HOUR_OF_DAY | <input type="checkbox"/> SECOND        |
| <input type="checkbox"/> AUGUST               | <input type="checkbox"/> JANUARY     | <input type="checkbox"/> SEPTEMBER     |
| <input type="checkbox"/> DATE                 | <input type="checkbox"/> JULY        | <input type="checkbox"/> SUNDAY        |
| <input type="checkbox"/> DAY_OF_MONTH         | <input type="checkbox"/> JUNE        | <input type="checkbox"/> THURSDAY      |
| <input type="checkbox"/> DAY_OF_WEEK          | <input type="checkbox"/> MARCH       | <input type="checkbox"/> TUESDAY       |
| <input type="checkbox"/> DAY_OF_WEEK_IN_MONTH | <input type="checkbox"/> MAY         | <input type="checkbox"/> UNDECIMBER    |
| <input type="checkbox"/> DAY_OF_YEAR          | <input type="checkbox"/> MILLISECOND | <input type="checkbox"/> WEDNESDAY     |
| <input type="checkbox"/> DECEMBER             | <input type="checkbox"/> MINUTE      | <input type="checkbox"/> WEEK_OF_MONTH |
| <input type="checkbox"/> DST_OFFSET           | <input type="checkbox"/> MONDAY      | <input type="checkbox"/> WEEK_OF_YEAR  |
| <input type="checkbox"/> ERA                  | <input type="checkbox"/> MONTH       | <input type="checkbox"/> YEAR          |
| <input type="checkbox"/> FEBRUARY             | <input type="checkbox"/> NOVEMBER    | <input type="checkbox"/> ZONE_OFFSET   |
| <input type="checkbox"/> FIELD_COUNT          | <input type="checkbox"/> OCTOBER     |  |

Следующая программа демонстрирует несколько методов класса `Calendar`:

```
// Демонстрирует класс Calendar.
import java.util.Calendar;

class CalendarDemo {
    public static void main(String args[]) {
        String months[] = {
            "Январь", "Февраль", "Март", "Апрель", "Май", "Июнь", "Июль",
            "Август", "Сентябрь", "Октябрь", "Ноябрь", "Декабрь"};

        // Создать календарь, инициализированный
        // текущей датой и временем, в умалчивающей
        // языковой зоне и часовом поясе.
        Calendar calendar = Calendar.getInstance();

        // Отобразить информацию текущего времени и даты.
        System.out.print("Дата: ");
        System.out.print(months[calendar.get(Calendar.MONTH)]);
        System.out.print(" " + calendar.get(Calendar.DATE) + " ");
        System.out.println(calendar.get(Calendar.YEAR));

        System.out.print("Время: ");
        System.out.print(calendar.get(Calendar.HOUR) + ":" );
        System.out.print(calendar.get(Calendar.MINUTE) + ":" );
        System.out.println(calendar.get(Calendar.SECOND));
    }
}
```

```

// Установить дату и время и отобразить их.
calendar.set(Calendar.HOUR, 10);
calendar.set(Calendar.MINUTE, 29);
calendar.set(Calendar.SECOND, 22);

System.out.print("Обновленное время: ");
System.out.print(calendar.get(Calendar.HOUR) + ":" );
System.out.print(calendar.get(Calendar.MINUTE) + ":" );
System.out.println(calendar.get(Calendar.SECOND));
}
}
}

```

Пример вывода этой программы:

Дата: Январь 25 1999

Время: 11:24:25

Обновленное время: 10:29:22

## Класс *GregorianCalendar*

Класс *GregorianCalendar* является конкретной реализацией абстрактного класса *Calendar*, который имитирует нормальный григорианский календарь. Метод *getInstance()* класса *Calendar* возвращает объект типа *GregorianCalendar*, инициализированный текущей датой и временем с заданными по умолчанию языковом регионом и часовым поясом.

В *GregorianCalendar* определены два поля: *AD* и *BC*, представляющие две эры григорианского календаря (*Anno Domini*, лат. — наша эра и *Before Crist* — до нашей эры).

Имеется также несколько конструкторов для объектов типа *GregorianCalendar*. Конструктор по умолчанию, *GregorianCalendar()*, инициализирует объект с текущей датой и временем в заданном по умолчанию языковом регионе и часовом поясе. Ниже показаны еще три конструктора (с возрастающим уровнем специфики):

```

GregorianCalendar(int year, int month, int dayOfMonth)
GregorianCalendar(int year, int month, int dayOfMonth, int hours,
int minutes)
GregorianCalendar(int year, int month, int dayOfMonth, int hours,
int minutes, int seconds)

```

Все три версии устанавливают день, месяц, и год. Здесь *year* — число лет, прошедших с 1900 г.; *month* — номер месяца (с нулем, указывающим январь); *dayOfMonth* — номер дня месяца. Первая версия устанавливает время к полуночи. Вторая версия устанавливает также часы и минуты. Третья версия добавляет секунды.

Вы можете также создавать GregorianCalendar-объект, указывая языковой регион и/или часовой пояс. Следующие конструкторы создают объекты, инициализированные текущей датой и временем и использующие часовой пояс и/или языковой регион:

```
GregorianCalendar(Locale locale)
GregorianCalendar(TimeZone timeZone)
GregorianCalendar(TimeZone timeZone, Locale locale)
```

GregorianCalendar обеспечивает реализацию всех абстрактных методов из Calendar, а также некоторых дополнительных методов. Возможно наиболее интересен метод isLeapYear(), который проверяет, является ли год високосным. Его форма:

```
boolean isLeapYear(int year)
```

Этот метод возвращает true, если год високосный, иначе — false.

Следующая программа демонстрирует GregorianCalendar:

```
// Демонстрирует класс GregorianCalendar
import java.util.*;

class GregorianCalendarDemo {
    public static void main(String args[]) {
        String months[] = {
            "Январь", "Февраль", "Март", "Апрель", "Май", "Июнь", "Июль",
            "Август", "Сентябрь", "Октябрь", "Ноябрь", "Декабрь"};
        int year;

        // Создать календарь, инициализированный
        // текущей датой и временем, в умалчивающей
        // языковой зоне и часовом поясе.
        GregorianCalendar gcalendar = new GregorianCalendar();

        // Отобразить информацию текущего времени и даты.
        System.out.print("Дата: ");
        System.out.print(months[gcalendar.get(Calendar.MONTH)]);
        System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
        System.out.println(year = gcalendar.get(Calendar.YEAR));

        System.out.print("Время: ");
        System.out.print(gcalendar.get(Calendar.HOUR) + ":" );
        System.out.print(gcalendar.get(Calendar.MINUTE) + ":" );
        System.out.println(gcalendar.get(Calendar.SECOND));

        // Проверить текущий год, является ли он високосным.
        if(gcalendar.isLeapYear(year)) {
            System.out.println("Текущий год високосный");
        }
    }
}
```

```
        else {
            System.out.println("Текущий год не високосный");
        }
    }
}
```

Пример вывода этой программы:

Дата: Jan 25 1999

Время: 11:25:27

Текущий год не високосный

## Класс *TimeZone*

Другой связанный со временем класс — `TimeZone`. Класс `TimeZone` позволяет работать с отсчетом часового пояса от среднего времени по Гринвичу (`GMT`, Greenwich Mean Time), называемому также *координированным универсальным временем* (`UTC`, Coordinated Universal Time) и вычисляет летнее время (`daylight saving time`). `TimeZone` имеет только конструктор по умолчанию.

Некоторые методы, определенные в `TimeZone`, собраны в табл. 16.5.

**Таблица 16.5.** Некоторые методы класса *TimeZone*

Метод	Описание
<code>Object clone()</code>	Возвращает специальную <code>TimeZone</code> -версию дубликата вызывающего объект
<code>static String[ ] getAvailableIDs()</code>	Возвращает массив <code>String</code> -объектов, представляющих имена всех часовых поясов
<code>static String[ ] getAvailableIDs(int timeDelta)</code>	Возвращает массив <code>String</code> -объектов, представляющих имена всех часовых поясов, которые смещены на <code>timeDelta</code> от GMT
<code>static TimeZone getDefault()</code>	Возвращает <code>TimeZone</code> -объект, который представляет заданный по умолчанию часовой пояс, используемый на хост-компьютере
<code>StringgetID()</code>	Возвращает имя вызывающего объекта типа <code>TimeZone</code>

Таблица 16.5 (окончание)

Метод	Описание
<code>abstract int getOffset(int era,                   int year,                   int month,                   int dayOfMonth,                   int dayOfWeek,                   int millis)</code>	Возвращает смещение (число), которое должно быть добавлено к GMT (т. е. к нулю) для вычисления местного времени. Это значение допускает регулировку летнего времени. Параметры метода представляют компоненты даты и времени
<code>abstract int getRawOffset()</code>	Возвращает смещение, которое должно быть добавлено к GMT для вычисления местного времени. Это значение не допускает регулировку летнего времени
<code>static TimeZone getTimeZone(String tzName)</code>	Возвращает <code>TimeZone</code> -объект для часового пояса с именем (идентификатором) <code>tzName</code>
<code>abstract boolean inDaylightTime(Date d)</code>	Возвращает <code>true</code> , если дата, представленная в <code>d</code> , находится в режиме летнего времени (в вызывающем объекте). Иначе возвращает <code>false</code>
<code>static void setDefault(TimeZone tz)</code>	Устанавливает умалчиваемый часовой пояс, который нужно использовать на этом хост-компьютере. <code>tz</code> — ссылка на используемый <code>TimeZone</code> -объект
<code>void setID(String tzName)</code>	Устанавливает имя (точнее — идентификатор) часового пояса, указанного в <code>tzName</code>
<code>abstract void setRawOffset(int millis)</code>	Устанавливает смещение в миллисекундах от GMT
<code>abstract boolean useDaylightTime()</code>	Возвращает <code>true</code> , если вызывающий объект использует летнее время. Иначе — <code>false</code>

## Класс `SimpleTimeZone`

Класс `SimpleTimeZone` — удобный подкласс класса `TimeZone`. Он реализует абстрактные методы класса `TimeZone` и позволяет работать с часовыми поясами по григорианскому календарю. Он также вычисляет летнее время.

В классе `SimpleTimeZone` определено три конструктора. Первый имеет форму:

```
SimpleTimeZone(int timeDelta, String tzName)
```

Конструктор создает объект типа `SimpleTimeZone`. Его параметры: `timeDelta` — смещение относительно среднего времени по Гринвичу (GMT); `tzName` — имя (идентификатор) часового пояса.

Второй конструктор класса `SimpleTimeZone`:

```
SimpleTimeZone(int timeDelta, String tzId, int dstMonth0,
               int dstDayInMonth0, int dstDay0, int time0,
               int dstMonth1, int dstDayInMonth1, int dstDay1,
               int time1)
```

где `timeDelta` — смещение относительно GMT; `tzId` — идентификатор часового пояса; `dstMonth0`, `dstDayInMonth0`, `dstDay0` и `time0` — параметры начала летнего времени; `dstMonth1`, `dstDayInMonth1`, `dstDay1` и `time1` — параметры конца летнего времени.

Третий конструктор класса `SimpleTimeZone`:

```
SimpleTimeZone(int timeDelta, String tzId, int dstMonth0,
               int dstDayInMonth0, int dstDay0, int time0,
               int dstMonth1, int dstDayInMonth1, int dstDay1,
               int time1, int dstDelta)
```

где `dstDelta` — число миллисекунд, сохраняемых в течение летнего времени.

## Класс *Locale*

Класс `Locale` создает объекты, каждый из которых описывает географический или культурный регион. Это один из нескольких классов, обеспечивающих возможность написания программ, которые могут выполняться в различных международных средах. Например, форматы отображения даты, времени и чисел в разных регионах отличаются.

Интернационализация — большая тема, которую невозможно охватить в данной книге. Однако большинству программ нужно иметь дело только с ее основами, которые включают установку текущего языкового региона.

Класс `Locale` определяет следующие константы, которые полезны для работы с наиболее общими языковыми регионами:

- |  |                                  |                                   |
|--|----------------------------------|-----------------------------------|
| <input type="checkbox"/> CANADA        | <input type="checkbox"/> FRANCE  | <input type="checkbox"/> ITALY    |
| <input type="checkbox"/> CANADA_FRENCH | <input type="checkbox"/> FRENCH  | <input type="checkbox"/> JAPAN    |
| <input type="checkbox"/> CHINA         | <input type="checkbox"/> GERMAN  | <input type="checkbox"/> JAPANESE |
| <input type="checkbox"/> CHINESE       | <input type="checkbox"/> GERMANY | <input type="checkbox"/> KOREA    |
| <input type="checkbox"/> ENGLISH       | <input type="checkbox"/> ITALIAN | <input type="checkbox"/> KOREAN   |

- |   |  |                             |
|---|--|-----------------------------|
| <input type="checkbox"/> PRC                | <input type="checkbox"/> TAIWAN              | <input type="checkbox"/> UK |
| <input type="checkbox"/> SIMPLIFIED_CHINESE | <input type="checkbox"/> TRADITIONAL_CHINESE | <input type="checkbox"/> US |

Например, выражение `Locale.CANADA` представляет объект типа `Locale` для Канады.

Конструкторы `Locale` имеют следующие формы:

```
Locale(String language, String country)
Locale(String language, String country, String data)
```

Эти конструкторы строят `Locale`-объект для представления определенного языка (параметр `language`) и страны (параметр `country`). Значения параметров должны содержать стандарт ISO<sup>1</sup> языка и код страны. В `data` задается вспомогательная информация о браузере и поставщиках.

В классе `Locale` определяется несколько методов. Один из наиболее важных — `setDefault()`:

```
static void setDefault(Locale localeObj)
```

Он устанавливает `localeObj` в качестве региона по умолчанию.

Некоторые другие интересные методы:

```
final String getDisplayCountry()
final String getDisplayLanguage()
final String getDisplayName()
```

Они возвращают удобочитаемые строки, которые можно использовать для отображения названия страны, названия языка и полного описания языкового региона.

Заданный по умолчанию языковой регион можно получить с помощью метода `getDefault()`:

```
static Locale getDefault()
```

`Calendar` и `GregorianCalendar` являются примерами классов, которые работают в чувствительной к локализации<sup>2</sup> манере. Классы `DateFormat` и `SimpleDateFormat` также зависят от локализации.

## Класс `Random`

Класс `Random` является генератором псевдослучайных чисел. *Псевдослучайные числа* формируют равномерно распределенные последовательности. В `Random` определены следующие конструкторы:

---

<sup>1</sup> ISO (International Standardization Organization) — Международная организация по стандартизации. — Примеч. пер.

<sup>2</sup> Локализация — приведение программного продукта (в данном случае — языка программирования) к местному (региональному) языковому стандарту. — Примеч. пер.

`Random()``Random(long seed)`

Первая версия создает числовой генератор, который использует в качестве начального числа текущее время. Вторая форма позволяет определять значение начального числа вручную.

Когда вы инициализируете `Random`-объект начальным значением, вы определяете отправную точку для случайной последовательности. Если вы применяете то же начальное число для инициализации другого `Random`-объекта, то извлечете ту же случайную последовательность. Если же вы хотите генерировать различные последовательности, указывайте несовпадающие значения начального числа. Для этого проще всего использовать текущее время (что позволит создавать `Random`-объекты, содержащие разные случайные последовательности). Данный подход уменьшает возможность получения повторяющихся последовательностей.

Общие (public) методы класса `Random` показаны в табл. 16.6.

**Таблица 16.6. Методы класса `Random`**

Метод	Описание
<code>boolean nextBoolean()</code>	Возвращает следующее случайное число типа <code>boolean</code> . (Добавлен в Java 2)
<code>void nextBytes(byte vals[ ])</code>	Заполняет массив <code>vals</code> случайно сгенерированными значениями
<code>double nextDouble()</code>	Возвращает следующее случайное число типа <code>double</code>
<code>float nextFloat()</code>	Возвращает следующее случайное число <code>float</code>
<code>double nextGaussian()</code>	Возвращает следующее гауссово случайное число (стандартное гауссово распределение, математическое ожидание $m=0$ , среднеквадратическое отклонение $\sigma=1$ )
<code>int nextInt()</code>	Возвращает следующее случайное число типа <code>int</code>
<code>int nextInt(int n)</code>	Возвращает следующее случайное число типа <code>int</code> в диапазоне от 0 до <code>n</code> . (Добавлен в Java 2)
<code>long nextLong()</code>	Возвращает следующее случайное число типа <code>long</code>
<code>void setSeed(long newSeed)</code>	Устанавливает новое значение начального числа генератора. Это значение указывает параметр <code>newSeed</code>

Как вы видите, существует шесть типов случайных чисел, которые можно извлекать из Random-объекта. Значения типа boolean доступны через метод nextBoolean(). Случайные байты можно получить, вызывая метод nextBytes(). Целые числа можно извлечь через метод nextInt(). Длинные целые числа, равномерно распределенные по их диапазону, можно генерировать с помощью метода nextLong(). Методы nextFloat() и nextDouble() возвращают float- и double-значения, равномерно распределенные в диапазоне между 0.0 и 1.0. Наконец, метод nextGaussian() возвращает double-значение, центрированное в 0.0 со среднеквадратичным отклонением 1.0 (стандартное гауссово или нормальное распределение). Кривую плотности распределения таких значений называют "колокольчиком" (bell curve).

Ниже приводится пример, который демонстрирует генерацию последовательностей с помощью метода nextGaussian():

```
// Демонстрация случайных гауссовых значений.
import java.util.Random;

class RandDemo {
    public static void main(String args[]) {
        Random r = new Random();
        double val;
        double sum = 0;
        int bell[] = new int[10];

        for(int i=0; i<100; i++) {
            val = r.nextGaussian();
            sum += val;
            double t = -2;
            for(int x=0; x<10; x++, t += 0.5)
                if(val < t) {
                    bell[x]++;
                    break;
                }
        }
        System.out.println("Среднее значение: " + (sum/100));

        // отобразить "лежащий на боку" колокольчик распределения
        for(int i=0; i<10; i++) {
            for(int x=bell[i]; x>0; x--)
                System.out.print("*");
            System.out.println();
        }
    }
}
```

Здесь генерируются 100 случайных гауссовых значений и вычисляется их среднее. Программа также вычисляет количество значений, которые попа-

дают в интервалы равномерного разбиения диапазона  $[-\sigma, +\sigma]$  (где  $\sigma$  – среднеквадратическое отклонение стандартного гауссовского распределения,  $\sigma = 1$ ,  $m = 0$ ) с шириной интервала 0.5. Результатирующее распределение отображается на экран в виде столбиковой диаграммы (гистограммы), лежащей "на боку", как показывает вывод этой программы:

Среднее значение: 0.0702235271133344

## Класс *Observable*

Класс `Observable` используется для создания подклассов, за которыми могут наблюдать другие части вашей программы. Когда объект такого подкласса подвергается изменению, наблюдающие классы уведомляются об этом. Наблюдающие классы должны реализовывать интерфейс `Observer`, который определяет метод `update()`. Когда наблюдатель уведомляется об изменении в наблюдаемом объекте, вызывается метод `Update()`.

В Observable определены методы, показанные в табл. 16.7. Объект, за которым ведется наблюдение, должен следовать двум простым правилам. Во-первых, если он изменился, то должен вызывать метод `setChanged()`. Во-вторых, когда он готов уведомить наблюдателей относительно этого изменения, он должен вызывать метод `notifyObservers()`. Это приводит к вызову метода `update()` в наблюдающем объекте (или объектах). Будьте осторожны, — если объект вызывает `notifyObservers()` без предварительного вызова `setChanged()`, никакие действия выполняться не будут. Наблюдаемый объект должен вызвать как и `setChanged()`, так и `notifyObservers()` прежде, чем будет вызван `update()`.

**Таблица 16.7.** Методы класса Observable

Метод	Описание
<code>void addObserver(Observer obj)</code>	Добавляет <code>obj</code> к списку объектов, которые наблюдают вызывающий объект

Таблица 16.7 (окончание)

Метод	Описание
<code>protected void clearChanged()</code>	Вызов этого метода возвращает состояние вызывающего объекта на "неизменяемый"
<code>int countObservers()</code>	Возвращает число объектов, которые наблюдают вызывающий объект
<code>void deleteObserver (Observer obj)</code>	Удаляет <code>obj</code> из списка объектов, которые наблюдают вызывающий объект
<code>void deleteObservers()</code>	Удаляет всех наблюдателей вызывающего объекта
<code>boolean hasChanged()</code>	Возвращает <code>true</code> , если вызывающий объект был изменен, и <code>false</code> , если не был
<code>void notifyObservers()</code>	Уведомляет всех наблюдателей вызывающего объекта, что он был изменен с помощью вызова метода <code>update()</code> . В качестве второго аргумента методу <code>update()</code> передается <code>null</code> (пустой указатель)
<code>void notifyObservers (Object obj)</code>	Уведомляет всех наблюдателей вызывающего объекта, что он был изменен вызовом <code>update()</code> . Параметр <code>obj</code> передается методу <code>update()</code> в качестве второго аргумента
<code>protected void setChanged()</code>	Вызывается, когда объект, инициализирующий обращение, изменился

Обратите внимание, что `notifyObservers()` имеет две формы: одна — с параметрами, другая — без параметров. Если вы вызываете `notifyObservers()` с аргументом, то соответствующий ему объект передается методу наблюдателя `update()` в качестве его второго параметра. В противном случае в `update()` передается `null`-ссылка. Вы можете использовать второй параметр `update()` для пересылки ему объекта любого типа, который является подходящим для вашего приложения.

## Интерфейс `Observer`

Для организации наблюдения соответствующим объектом вы должны реализовать интерфейс `Observer`. Он определяет только один метод:

```
void update (Observable observable, Object arg)
```

где `observObj` — наблюдаемый объект; `arg` — значение, которое пересыпается методом `notifyObservers()`. Метод `update()` вызывается тогда, когда имеет место изменение в наблюдаемом объекте.

## Пример наблюдателя

Ниже показан пример, который демонстрирует наблюдаемый объект. В нем создается класс наблюдателя с именем `Watcher`, который реализует интерфейс `Observer`. Проверяемый класс называется `BeingWatched`. Он расширяет интерфейс `Observable`. Внутри `BeingWatched` есть метод `counter()`, который просто считает в обратном направлении от указанного значения. Он использует метод `sleep()` для приостановки на десятую долю секунды в каждом шаге счета. Очередной раз, когда счет изменяется, вызывается метод `notifyObservers()` с текущим значением счетчика в качестве аргумента. Это приводит к вызову метода `update()` внутри `Watcher`-объекта, который отображает текущий счет. Внутри метода `main()` создаются объекты `Watcher`, `observing` и `BeingWatched` `observed`. Затем объектная переменная `observing` добавляется к списку наблюдателей объекта `observed`. Это означает, что `observing.update()` будет вызываться каждый раз, когда метод `counter()` вызывает метод `notifyObservers()`.

```
// Демонстрирует класс Observable и интерфейс Observer.  
import java.util.*;  
  
// это наблюдающий класс  
class Watcher implements Observer {  
    public void update(Observable obj, Object arg) {  
        System.out.println("Вызван update(), count равно " +  
                           ((Integer)arg).intValue());  
    }  
}  
  
// это наблюдаемый класс  
class BeingWatched extends Observable {  
    void counter(int period) {  
        for(; period >=0; period--) {  
            setChanged();  
            notifyObservers(new Integer(period));  
            try {  
                Thread.sleep(100);  
            } catch(InterruptedException e) {  
                System.out.println("Sleep interrupted");  
            }  
        }  
    }  
}
```

```

class ObserverDemo {
    public static void main(String args[]) {
        BeingWatched observed = new BeingWatched();
        Watcher observing = new Watcher();

        // добавить наблюдающего в список наблюдателей observed object
        observed.addObserver(observing);

        observed.counter(10);
    }
}

```

Выход этой программы:

```

Вызван update(), count равно 10
Вызван update(), count равно 9
Вызван update(), count равно 8
Вызван update(), count равно 7
Вызван update(), count равно 6
Вызван update(), count равно 5
Вызван update(), count равно 4
Вызван update(), count равно 3
Вызван update(), count равно 2
Вызван update(), count равно 1
Вызван update(), count равно 0

```

Наблюдателями могут быть несколько объектов. Например, следующая программа реализует два наблюдающих класса и добавляет объект каждого класса в список BeingWatched наблюдателя. Второй наблюдатель ждет, пока счет не достигает нуля и затем звонит звонок.

```

// Объект может наблюдаться двумя и более наблюдателями.
import java.util.*;

// это первый наблюдающий класс
class Watcher1 implements Observer {
    public void update(Observable obj, Object arg) {
        System.out.println("update() called, count is " +
                           ((Integer)arg).intValue());
    }
}

// это второй наблюдающий класс
class Watcher2 implements Observer {
    public void update(Observable obj, Object arg) {
        // Ring bell when done
        if(((Integer)arg).intValue() == 0)
            System.out.println("Done" + '\7');
    }
}

```

```
// это наблюдаемый класс
class BeingWatched extends Observable {
    void counter(int period) {
        for(; period >=0; period--) {
            setChanged();
            notifyObservers(new Integer(period));
            try {
                Thread.sleep(100);
            } catch(InterruptedException e) {
                System.out.println("Sleep interrupted");
            }
        }
    }
}

class TwoObservers {
    public static void main(String args[]) {
        BeingWatched observed = new BeingWatched();
        Watcher1 observing1 = new Watcher1();
        Watcher2 observing2 = new Watcher2();

        // добавить оба наблюдателя
        observed.addObserver(observing1);
        observed.addObserver(observing2);

        observed.counter(10);
    }
}
}
```

Класс `Observable` и интерфейс `Observer` позволяют реализовать сложные архитектуры программ, основанные на методологии документ/вид. Они также полезны в многопоточных ситуациях.

## **Пакет `java.util.zip`**

Пакет `java.util.zip` обеспечивает способность читать и записывать файлы в популярном ZIP и GZIP файловых форматах. Доступны как ZIP, так и GZIP потоки ввода и вывода. Другие классы реализуют ZLIB-алгоритмы для сжатия и декомпрессии.

## **Пакет `java.util.jar`**

Пакет `java.util.jar` обеспечивает возможность читать и записывать файлы JAR (Java Archive). Вы увидите в главе 25, что эти файлы используются для хранения программ компонентов, известных как Java Beans, и любых связанных с ними файлов.



## ГЛАВА 17

# Ввод/вывод: обзор пакета *java.io*

В этой главе исследуется пакет *java.io*, который обеспечивает поддержку операций ввода/вывода (I/O, Input/Output). Система ввода/вывода языка Java была представлена в главе 12. Здесь мы рассмотрим ее более подробно.

Любому программисту с самого начала известно, что большинство программ не может достичь выполнения определенных задач без обращения к внешним данным. Данные извлекаются из источника *ввода*. Результаты выполнения программы пересылаются потребителю *вывода*. В Java эти источники или потребители определены очень широко. Например, I/O-классы Java могут манипулировать вводом/выводом таких устройств, как сетевое соединение, буфер памяти или файл на диске. Хотя они физически различны, но все используют одну и ту же абстракцию — *поток*. Поток, как объяснено в главе 12, является логической сущностью, которая или производит, или потребляет информацию. Поток связывается с физическим устройством с помощью системы ввода/вывода Java (Java I/O system). Все потоки ведут себя одинаковым образом, даже если фактические физические устройства, с которыми они связаны, различны.

### Замечание

Краткий обзор поточного ввода/вывода Java см. в гл. 12.

## Классы и интерфейсы ввода/вывода Java

I/O классы, определенные в пакете *java.io*, перечислены в следующем списке:

- BufferedInputStream*
- BufferedOutputStream*
- BufferedReader*
- BufferedWriter*
- ByteArrayInputStream*
- ByteArrayOutputStream*

- CharArrayReader
- CharArrayWriter
- DataInputStream
- DataOutputStream
- File
- FileDescriptor
- FileInputStream
- FileOutputStream
- FilePermission
- FileReader
- FileWriter
- FilterInputStream
- FilterOutputStream
- FilterReader
- FilterWriter
- InputStream
- InputStreamReader
- LineNumberReader
- ObjectInputStream
- ObjectInputStream.GetField
- ObjectOutputStream
- ObjectOutputStream.PutField
- ObjectStreamClass
- ObjectStreamField
- OutputStream
- OutputStreamWriter
- PipedInputStream
- PipedOutputStream
- PipedReader
- PipedWriter
- PrintStream
- PrintWriter
- PushbackInputStream
- PushbackReader
- RandomAccessFile
- Reader
- SequenceInputStream
- SerializablePermission
- StreamTokenizer
- StringReader
- StringWriter
- Writer

Внутренние классы ObjectInputStream.GetField и ObjectOutputStream.PutField были добавлены в Java 2. Пакет java.io также содержит два класса, которые были исключены в Java 2 и не показаны в предыдущем списке — LineNumberInputStream и StringBufferInputStream. Эти классы не должны использоваться в новых программах.

В java.io определены следующие интерфейсы:

- |   |  |  |
|---|--|--|
| <input type="checkbox"/> DataInput      | <input type="checkbox"/> FilenameFilter        | <input type="checkbox"/> ObjectStreamConstants |
| <input type="checkbox"/> DataOutput     | <input type="checkbox"/> ObjectInput           | <input type="checkbox"/> Serializable          |
| <input type="checkbox"/> Externalizable | <input type="checkbox"/> ObjectInputValidation |  |
| <input type="checkbox"/> FileFilter     | <input type="checkbox"/> ObjectOutputStream    |  |

Интерфейс FileFilter был добавлен в Java 2.

Как вы видите, в пакете java.io имеются много классов и интерфейсов. Они обслуживают байтовые и символьные потоки и сериализацию<sup>1</sup> объектов

---

<sup>1</sup> Сериализация — преобразование в последовательную форму. — Примеч. пер.

(с целью хранения и получения объектов из внешних устройств). Эта глава рассматривает некоторые из обычно используемых компонентов ввода/вывода, начиная с одного из наиболее уникальных — файлового объекта (класса `File`).

## Класс `File`

Хотя большинство классов, определенных в `java.io`, работает с потоками, класс `File` имеет дело непосредственно с файлами и файловой системой. В классе `File` не указывается, как нужно извлекать или сохранять информацию в файлах; в нем описываются свойства самого файла. Для получения или управления информацией, связанной с дисковым файлом, используется *объект* типа (класса) `File`. Речь здесь идет о таких свойствах файла, как права доступа, время и дата создания и изменения, путь в иерархии подкаталов и т. д.

Файлы являются первичным источником и потребителем данных для многих программ. Хотя существуют строгие ограничения на использование файлов в апплетах (из соображений безопасности), они все еще остаются центральным ресурсом для хранения постоянной и совместно используемой информацией. Каталог в Java трактуется просто как `File`-объект с одним дополнительным свойством — списком имен файлов, который можно просматривать с помощью метода `list()`.

Для создания `File`-объектов можно использовать следующие конструкторы:

```
File(String directoryPath)
File(String directoryPath, String filename)
File(File dirObj, String filename)
```

Здесь `directoryPath` — путь к файлу (в дереве каталогов); `filename` — имя файла; и `dirObj` — `File`-объект, который указывает каталог.

Следующий пример создает три файла: `f1`, `f2` и `f3`. Первый `File`-объект организуется конструктором с единственным аргументом — путем к файлу (`directoryPath`). Конструктор второго использует два аргумента — путь и имя файла. Конструктор третьего в первом аргументе использует путь, назначенный файловой ссылке `f1`, так что `f3` обращается к тому же файлу, что и `f2`.

### Замечание

Java правильно воспринимает разделители подкаталогов, используемые в соглашениях по именованию файлов в операционных системах UNIX и Windows/DOS. Если вы используете прямой слэш-символ (`/`) в Windows-версии Java, путь будет восприниматься правильно. Помните, если вы используете соглашения Windows/DOS с обратным слэш-символом (`\`), то в строке пути нужно использовать escape-последовательность этого символа (`\\\`). В Java применяется UNIX- и URL-стиль разделителей подкаталогов (прямой слэш).

В классе `File` определено много методов для получения стандартных свойств `File`-объекта. Например, `getName()` возвращает название имя, `getParent()` — имя каталога предыдущего уровня, а `exists()` возвращает `true`, если файл существует, и `false`, если — нет. Класс `File`, однако, не симметричен. Под этим мы подразумеваем, что есть много методов, которые позволяют *просмотреть* свойства простого файлового объекта, но нет соответствующих функций для их *изменения*. Следующий пример демонстрирует некоторые `File`-методы:

```
// Демонстрирует класс File.
import java.io.File;

class FileDemo {
    static void p(String s) {
        System.out.println(s);
    }

    public static void main(String args[]) {
        File f1 = new File("/java/COPYRIGHT");
        p("Имя файла: " + f1.getName());
        p("Путь: " + f1.getPath());
        p("Абсолютный путь: " + f1.getAbsolutePath());
        p("Надкаталог: " + f1.getParent());
        p(f1.exists() ? "существует" : "не существует");
        p(f1.canWrite() ? "для записи" : "не для записи");
        p(f1.canRead() ? "для чтения" : "не для чтения");
        p("is " + (f1.isDirectory() ? "" : "не " + "каталог"));
        p(f1.isFile() ? "нормальный файл" : "возможно именованный канал");
        p(f1.isAbsolute() ? "абсолютный" : "не абсолютный");
        p("Последняя модификация файла: " + f1.lastModified());
        p("Размер файла: " + f1.length() + " байтов");
    }
}
```

Когда вы выполните эту программу, то увидите нечто, похожее на следующий листинг:

```
Имя файла: COPYRIGHT
Путь: /Java/COPYRIGHT
Абсолютный путь: /Java/COPYRIGHT
Надкаталог: /Java
существует
для записи
для чтения
не каталог
нормальный файл
абсолютный
Последняя модификация файла: 812465204000
Размер файла: 695 байтов
```

Большинство File-методов самоочевидны<sup>1</sup>. Исключение в этом отношении составляют методы `isFile()` и `isAbsolute()`. Метод `isFile()` возвращает `true`, если он обратился к *файлу*, и `false`, если — к *каталогу*. Кроме того, `isFile()` возвращает `false` для некоторых специальных файлов, типа драйверов устройств и именованных каналов, так что этот метод можно использовать, чтобы удостовериться, что файл будет вести себя как файл. Метод `isAbsolute()` возвращает `true`, если файл имеет абсолютный путь, и `false`, если его путь — относительный.

Класс `File` также включает два полезных сервисных метода. Первый — `renameTo()` — имеет такую сигнатуру:

```
boolean renameTo(File newName)
```

Здесь имя файла, указываемое в `newName`, становится новым именем вызывающего `File`-объекта. Метод возвратит `true` после успешного переименования и `false`, если файл не может быть переименован (например, если вы пытаетесь переименовывать файл так, чтобы он переместился из одного каталога в другой, или используете существующее имя файла).

Второй сервисный метод — `delete()`, удаляющий дисковый файл, путь которого указан в вызывающем `File`-объекте. Вот его сигнатура:

```
boolean delete()
```

Метод `delete()` можно также использовать для удаления каталога, если он пуст. `delete()` возвращает `true`, если он удаляет файл, и `false`, если файл не может быть удален.

В Java 2 в класс `File` добавлено несколько новых методов, которые могут оказаться полезными в некоторых ситуациях. В табл. 17.1 описаны наиболее интересные из них.

**Таблица 17.1. Некоторые методы класса `File`**

Метод	Описание
<code>void deleteOnExit()</code>	Удаляет файл, связанный с вызывающим объектом, когда виртуальная машина Java завершает свою работу
<code>boolean isHidden()</code>	Возвращает <code>true</code> , если файл скрыт, иначе — <code>false</code>
<code>boolean (long millisec)</code>	Устанавливает временную метку на вызывающий файл (значение метки указывает параметр <code>millisec</code> в виде количества миллисекунд с 1 января 1970 г., UTC <sup>2</sup> )

<sup>1</sup> Имеется в виду, что имена методов отражают ту функцию, которую они выполняют (правда, по-английски). — Примеч. пер.

<sup>2</sup> Так называемое Скоординированное Универсальное Время (UTC, Coordinated Universal Time). — Примеч. пер.

Таблица 17.1 (окончание)

Метод	Описание
<code>boolean setReadOnly()</code>	Устанавливает свойство "только для чтения" (read only) для вызывающего файла

Кроме того, поддерживается метод `compareTo()` (поскольку класс `File` теперь поддерживает интерфейс `Comparable`).

## Каталоги

Каталог представляет собой `File`-объект, который содержит список других файлов и каталогов. Когда вы создаете объект типа `File`, и он является каталогом, вызов метода `isDirectory()` возвращает `true`. В этом случае вы можете применить метод `list()` к указанному объекту, чтобы извлечь из него список других файлов и каталогов. Метод `list()` имеет две формы. Вот первая:

```
String[] list()
```

Список файлов возвращается через массив `String`-объектов.

Приведенная ниже программа показывает, как можно использовать `list()` для просмотра содержимого каталога:

```
// Использование каталогов.
import java.io.File;

class DirList {
    public static void main(String args[]) {
        String dirname = "/java";
        File f1 = new File(dirname);

        if (f1.isDirectory()) {
            System.out.println("Каталог " + dirname);
            String s[] = f1.list();

            for (int i=0; i < s.length; i++) {
                File f = new File(dirname + "/" + s[i]);
                if (f.isDirectory())
                    System.out.println(s[i] + " - это каталог");
                else
                    System.out.println(s[i] + " - это файл");
            }
        } else {
    }
} else {
```

```

        System.out.println(dirname + " - это не каталог");
    }
}
}

```

Ниже показан образец вывода этой программы. (Конечно, ваш вывод будет другим. Он отобразит то, что находится в *вашем* каталоге.)

```

Каталог /Java
bin - это каталог
lib - это каталог
demo - это каталог
COPYRIGHT - это файл
README - это файл
index.html - это файл
include - это каталог
src.zip - это файл
.hot Java - это каталог
src - это каталог

```

## Использование интерфейса *FilenameFilter*

Часто требуется ограничить количество файлов, возвращаемых методом `list()`, чтобы включать в список только те из них, имена которых соответствуют некоторому образцу или фильтру. Для этого нужно использовать вторую форму метода `list()`:

```
String[] list(FilenameFilter FFObj)
```

Здесь `FFObj` — объект класса, который реализует интерфейс типа `FilenameFilter`.

В интерфейсе `FilenameFilter` определен только один метод `accept()`, который вызывается однажды для каждого файла в списке. Его общая форма:

```
boolean accept(File directory, String filename)
```

Метод `accept()` возвращает `true` для файлов каталога `directory`, которые должны быть включены в отфильтрованный список (с именами `filename`), и возвращает `false` для тех файлов, которые должны быть исключены из списка (имена которых не совпадают с `filename`).

Класс `OnlyExt`, показанный ниже, является реализацией интерфейса `FilenameFilter`. Он послужит для такого изменения предыдущей программы, что список имен файлов, возвращаемых методом `list()`, будет ограничиваться именами файлов с определенным расширением (которое вы указываете при создании объекта в аргументе соответствующего конструктора).

```

import java.io.*;
public class OnlyExt implements FilenameFilter {

```

```

String ext;
public OnlyExt(String ext) {
    this.ext = "." + ext;
}
public boolean accept(File dir, String name) {
    return name.endsWith(ext);
}
}
}

```

Следующая программа демонстрирует измененный (отфильтрованный) листинг каталога. Теперь будут выводиться только файлы с расширением .html.

```

// Каталог HTML-файлов.
import java.io.*;

class DirListOnly {
    public static void main(String args[]) {
        String dirname = "/java";
        File f1 = new File(dirname);
        FilenameFilter only = new OnlyExt("html");
        String s[] = f1.list(only);

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

```

## Альтернативный метод *listFiles()*

В Java 2 добавлен вариант метода `list()` с именем `listFiles()`, который тоже может оказаться полезным. Сигнатуры трех (перегруженных) вариантов `listFiles()`:

```

File[ ] listFiles()
File[ ] listFiles(FilenameFilter FFObj)
File[ ] listFiles(FileFilter FObj)

```

Эти методы возвращают список файлов в виде *массива объектов* типа `File` (не просто в виде строк). Первый метод возвращает все файлы, а второй — только те, что удовлетворяют фильтру имен `FFObj`. За исключением возврата массива `File`-объектов, эти две версии `listFiles()` работают аналогично их эквивалентному методу `list()`.

Третья версия `listFiles()` фильтрует файлы не просто по именам, а по их *полным* именам (с путями в иерархии каталогов), и возвращает файлы с теми именами пути, которые удовлетворяют файловому фильтру `FObj`. Интерфейс

`FileFilter` определяет только один метод, `accept()`, который вызывается однажды для каждого файла в списке. Его общая форма:

```
boolean accept(File path)
```

Метод возвращает `true` для файлов, которые должны быть включены в список (т. е. которые указаны в `path`), и `false` — для тех файлов, которые должны быть исключены (не указаны `path`).

## Создание каталогов

Два других полезных сервисных метода класса `File` — `mkdir()` и `mkdirs()`. Метод `mkdir()` создает каталог, возвращая `true` при успехе и `false` — при неудаче. Неудача указывает, что путь, указанный в `File`-объекте, уже существует, или что каталог не может быть создан, из-за отсутствия полного пути. Для каталога с отсутствующим путем используйте метод `mkdirs()`. Он создает как собственно каталог, так и всех его родителей.

## Поточные классы

Поточный ввод/вывод Java формируется на четырех абстрактных классах: `InputStream`, `OutputStream`, `Reader` и `Writer`. Эти классы были кратко обсуждены в главе 12. Они используются для создания нескольких конкретных поточных подклассов. Хотя ваши программы исполняют их I/O-операции через конкретные подклассы, основные функциональные возможности определяют классы верхнего уровня, общие для всех поточных классов.

Классы `InputStream` и `OutputStream` предназначены для байтовых потоков, а `Reader` и `Writer` — для символьных. Байтовые и символьные поточные классы формируют отдельные иерархии. Вообще, нужно использовать классы символьных потоков при работе с символами или строками, а классы байтовых потоков — при работе с байтами или другими двоичными объектами.

В остальной части этой главы рассматриваются как байтовые, так и символьные потоки.

## Байтовые потоки

Классы байтовых потоков обеспечивают богатую среду для обработки байт-ориентированного ввода/вывода. Байтовый поток можно использовать с любым типом объектов, включая данные в двоичном коде. Эта универсальность делает байтовые потоки важными для многих типов программ. Так как классы байтовых потоков возглавляют классы `InputStream` и `OutputStream`, с них мы и начнем обсуждение.

## Класс *InputStream*

*InputStream* — абстрактный класс, который определяет Java-модель поточного ввода байтов. Все методы данного класса в аварийных ситуациях выбрасывают исключения типа *IOException*. Методы класса *InputStream* показаны в табл. 17.2.

**Таблица 17.2. Методы, определенные в классе *InputStream***

Метод	Описание
<code>int available()</code>	Возвращает число байтов ввода, доступных в настоящее время для чтения
<code>void close()</code>	Закрывает входной источник. Дальнейшие попытки чтения генерируют исключение <i>IOException</i>
<code>void mark(int numBytes)</code>	Размещает специальный маркер в текущую точку входного потока, который остается действующим, пока не считается <i>numBytes</i> байт
<code>boolean markSupported()</code>	Возвращает <code>true</code> , если вызывающий поток поддерживает методы <i>mark()</i> / <i>reset()</i>
<code>int read()</code>	Возвращает целочисленное представление следующего доступного байта ввода. Когда встречается конец файла, возвращает <code>-1</code>
<code>int read(byte buffer[ ])</code>	Пытается читать до <i>buffer.length</i> байтов в <i>buffer</i> и возвращает фактическое число байтов, которые были успешно считаны. Когда встречается конец файла, возвращается <code>-1</code>
<code>int read(byte buffer[ ], int offset, int numBytes)</code>	Пытается читать до <i>numBytes</i> байтов в <i>buffer</i> , начиная с позиции <i>offset</i> , возвращая число успешно считанных байтов. Когда встречается конец файла, возвращается <code>-1</code>
<code>void reset()</code>	Переустанавливает входной указатель на предварительно установленный маркер
<code>long skip(long numBytes)</code>	Игнорирует (т. е. пропускает) <i>numBytes</i> байтов ввода, возвращая число фактически проигнорированных байтов

## Класс *OutputStream*

*OutputStream* — абстрактный класс, который определяет байтовый выходной поток. Все методы в этом классе возвращают значение `void` и выбрасывают в случае ошибок исключение типа *IOException*. Методы *OutputStream* класса показаны в табл. 17.3.

Таблица 17.3. Методы, определенные в классе OutputStream

Метод	Описание
<code>void close()</code>	Закрывает выходной поток. Дальнейшие попытки писать в него генерируют исключение типа <code>IOException</code>
<code>void flush()</code>	Сбрасывает состояние вывода так, что любые буферы очищаются. То есть он сбрасывает буферы вывода
<code>void write(int b)</code>	Записывает один байт в выходной поток. Обратите внимание, что параметр имеет тип <code>int</code> , который позволяет вызывать <code>write()</code> с выражениями (в аргументе) без необходимости приводить их типы обратно к типу <code>byte</code>
<code>void write(byte buffer[ ])</code>	Записывает заполненный массив байтов в выходной поток
<code>void write(byte buffer[ ], int offset, int numBytes)</code>	Записывает в выходной поток <code>numBytes</code> байтов из массива <code>buffer</code> , начиная с позиции <code>offset</code>

### Замечание

Большинство методов, описанных в табл. 17.2 и 17.3, реализовано подклассами `InputStream` и `OutputStream`. Методы `mark()` и `reset()` — исключения; обратите внимание на их использование или испытайте их нехватку в каждом из обсуждаемых ниже подклассов.

## Класс FileInputStream

Класс `FileInputStream` создает `InputStream`-объект, который можно использовать для чтения байтов из файла. Ниже показаны два его наиболее общих конструктора:

```
FileInputStream(String filepath)
FileInputStream(File fileObj)
```

Каждый может выбрасывать исключения типа `FileNotFoundException`. Параметры: `filepath` — полное имя (с путем) файла; `fileObj` — объект типа (класса) `File`, который описывает файл.

Следующий пример создает два `FileInputStreams`-объекта, которые используют один и тот же дисковый файл и каждый из этих двух конструкторов:

```
FileInputStream f0 = new FileInputStream("/autoexec.bat")
File f = new File("/autoexec.bat");
FileInputStream f1 = new FileInputStream(f);
```

Хотя первый конструктор, вероятно, используется чаще, второй позволяет подробнее рассмотреть файл, указывая методы класса `File`, прежде, чем мы прикрепляем его к входному потоку. Когда `FileInputStream`-объект создается, он также открывается для чтения. Класс `FileInputStream` переопределяет шесть методов абстрактного класса `InputStream`. Методы `mark()` и `reset()` не переопределяются, и любая попытка использовать `reset()` на `FileInputStream` генерирует исключение типа `IOException`.

Следующий пример показывает, как можно читать один байт, массив байтов и поддиапазон массива байтов. Он также иллюстрирует, как можно использовать метод `available()` для определения числа оставшихся байтов и метод `skip()` для пропуска нежелательных байтов. Программа читает свой собственный исходный файл, который должен размещаться в текущем каталоге.

```
// Демонстрирует класс FileInputStream.  
import java.io.*;  
  
class FileInputStreamDemo {  
    public static void main(String args[]) throws Exception {  
        int size;  
        InputStream f =  
            new FileInputStream("FileInputStreamDemo.java");  
  
        System.out.println("Всего доступно байтов: " +  
            (size = f.available()));  
        int n = size/40;  
        System.out.println("Первые " + n +  
            " байтов файла один read() считывает сразу");  
        for (int i=0; i < n; i++) {  
            System.out.print((char) f.read());  
        }  
        System.out.println("\nЕще доступно: " + f.available());  
        System.out.println("Чтение следующих " + n +  
            " с помощью одного read(b[]));  
        byte b[] = new byte[n];  
        if (f.read(b) != n) {  
            System.err.println("couldn't read " + n + " bytes.");  
        }  
        System.out.println(new String(b, 0, n));  
        System.out.println("\nЕще доступно: " + (size = f.available()));  
        System.out.println("Пропуск половины оставшихся байтов skip());  
        f.skip(size/2);  
        System.out.println("Еще доступно: " + f.available());  
        System.out.println("Чтение " + n/2 + " байт в конце массива");  
        if (f.read(b, n/2, n/2) != n/2) {  
            System.err.println("не может быть считано " + n/2 + " байтов.");  
        }  
    }  
}
```

```
System.out.println(new String(b, 0, b.length));
System.out.println("\nЕще доступно: " + f.available());
f.close();
}
}
```

Вывод, полученный этой программой:

```
Всего доступно байтов: 1433
Первые 35 байтов файла один read() считывает сразу
// Демонстрирует класс FileInputStream.
im
Еще доступно: 1398
Чтение следующих 35 с помощью одного read(b[])
port java.io.*;

class FileInputs

Еще доступно: 1363
Пропуск половины оставшихся байтов skip()
Еще доступно: 682
Чтение 17 байт в конце массива
port Чтение java.io.*;
read(b) != n) {
S
Еще доступно: 665
```

Этот несколько надуманный пример демонстрирует, как можно читать трёх способами, пропускать ввод и просматривать количество доступных данных в потоке.

## Класс *FileOutputStream*

Класс *FileOutputStream* создает объект *OutputStream*, который можно применять для записи байтов в файл. Обычно используются следующие конструкторы этого класса:

```
FileOutputStream(String filePath)
FileOutputStream(File fileObj)
FileOutputStream(String filePath, boolean append)
```

Они могут выбрасывать исключения типа *IOException* или *SecurityException*. Параметры: *filePath* — полное имя пути файла; *fileObj* — объект типа *File*, который описывает файл. Если *append* — *true*, файл открывается в режиме добавления.

Создание *FileOutputStream*-объекта не зависит от того, существует ли уже файл. *FileOutputStream*-конструктор создаст файл перед его открытием для

вывода, когда вы создаете объект. Если вы попытаетесь открыть файл только для чтения, будет выброшено исключение типа `IOException`.

Следующий пример создает буфер байтов, сначала создавая `String`-объект и затем используя метод `getBytes()`, чтобы заполнить эквивалентный байт-массив. Затем он создает три файла. Первый, `file1.txt`, будет содержать каждый нечетный байт исходной строки. Второй, `file2.txt`, будет содержать полный набор байтов. Третий и последний, `file3.txt`, будет содержать только последнюю четверть строки. В отличие от методов класса `FileInputStream`, все методы `OutputStream` имеют возвращаемый тип `void`. В случае возникновения ошибки, эти методы выбрасывают исключение типа `IOException`.

```
// Демонстрирует класс FileOutputStream.
import java.io.*;

class FileOutputStreamDemo {
    public static void main(String args[]) throws Exception {
        String source = "Теперь настало время всем порядочным людям\n"
            + " стране прийти на помощь своей\n"
            + " и платить причитающиеся налоги.";
        byte buf[] = source.getBytes();
        OutputStream f0 = new FileOutputStream("file1.txt");
        for (int i=0; i < buf.length; i += 2) {
            f0.write(buf[i]);
        }
        f0.close();

        OutputStream f1 = new FileOutputStream("file2.txt");
        f1.write(buf);
        f1.close();

        OutputStream f2 = new FileOutputStream("file3.txt");
        f2.write(buf,buf.length-buf.length/4,buf.length/4);
        f2.close();
    }
}
```

Ниже показано содержимое каждого файла после выполнения этой программы. Сначала `file1.txt`:

Тпр атл рм смпрдчы юя  
пит алмш ве тae  
ипаильпииащяянлг.

Затем `file2.txt`:

Теперь настало время всем порядочным людям  
прийти на помощь своей стране  
и платить причитающиеся налоги.

Наконец, file3.txt:

тить причитающиеся налоги.

## Класс *ByteArrayInputStream*

*ByteArrayInputStream* — реализация входного потока, которая использует байтовый массив как источник. Этот класс имеет два конструктора, каждый из которых использует байтовый массив в качестве источника данных:

```
ByteArrayInputStream(byte array[])
ByteArrayInputStream(byte array[], int start, int numBytes)
```

Здесь *array* — источник ввода. Второй конструктор создает *InputStream*-объект, состоящий из байтового подмассива, который начинается с позиции *start* и имеет длину *numBytes* байтов.

В следующем примере создается пара объектов типа *ByteArrayInputStreams*, которые инициализируются байтовым представлением алфавита:

```
// Демонстрирует класс ByteArrayInputStream.
```

```
import java.io.*;
```

```
class ByteArrayInputStreamDemo {
    public static void main(String args[]) throws IOException {
        String tmp = "abcdefghijklmnopqrstuvwxyz";
        byte b[] = tmp.getBytes();
        ByteArrayInputStream input1 = new ByteArrayInputStream(b);
        ByteArrayInputStream input2 = new ByteArrayInputStream(b, 0, 3);
    }
}
```

Объект *input1* содержит полный алфавит (в нижнем регистре), в то время как *input2* хранит только три его первых символа.

В классе *ByteArrayInputStream* реализуется как *mark()*, так и *reset()*. Однако в этой программе, если *mark()* не вызвался, то *reset()* устанавливает поточный указатель на начало потока, который в этом случае является началом byte-массива, передаваемого конструктору. Следующий пример показывает, как можно использовать метод *reset()* для чтения одного и того же ввода дважды. В этом случае мы читаем и печатаем буквы "abc" один раз в нижнем регистре, и затем еще раз — в верхнем регистре.

```
import java.io.*;
class ByteArrayInputStreamReset {
    public static void main(String args[]) throws IOException {
        String tmp = "abc";
        byte b[] = tmp.getBytes();
        ByteArrayInputStream in = new ByteArrayInputStream(b);
```

```
for (int i=0; i<2; i++) {  
    int c;  
    while ((c = in.read()) != -1) {  
        if (i == 0) {  
            System.out.print((char) c);  
        } else {  
            System.out.print(Character.toUpperCase((char) c));  
        }  
    }  
    System.out.println();  
    in.reset();  
}  
}
```

Данная программа сначала читает каждый символ из потока и печатает результат в нижнем регистре. Затем она сбрасывает поток и начинает читать снова, на сей раз преобразуя каждый символ в верхний регистр (перед печатью). Вывод этого примера:

abc  
ABC

## Класс *ByteArrayOutputStream*

Класс `ByteArrayOutputStream` реализует выходной поток, который использует `byte`-массив в качестве пункта назначения. `ByteArrayOutputStream` имеет два конструктора в следующей форме:

```
ByteArrayOutputStream()  
ByteArrayOutputStream(int numBytes)
```

В первой форме создается 32-байтный буфер. Во второй — буфер с размером, равным указанному в параметре `numBytes`. Буфер содержится в защищенным поле `buf` класса `ByteArrayOutputStream`. При необходимости размер буфера увеличивается автоматически. Число байтов буфера содержится в защищенным поле `count` класса `ByteArrayOutputStream`.

Следующий пример демонстрирует `ByteArrayOutputStream`:

```
// Демонстрирует класс ByteArrayOutputStream.  
import java.io.*;  
  
class ByteArrayOutputStreamDemo {  
    public static void main(String args[]) throws IOException {  
        ByteArrayOutputStream f = new ByteArrayOutputStream();  
        String s = "This should end up in the array";  
        byte buf[] = s.getBytes();
```

```
f.write(buf);
System.out.println("Буфер в форме строки содержит:");
System.out.println(f.toString());
System.out.println("Массив содержит:");
byte b[] = f.toByteArray();
for (int i=0; i<b.length; i++) {
    System.out.print((char) b[i]);
}
System.out.println("\nВывод буфера в OutputStream");
OutputStream f2 = new FileOutputStream("test.txt");

f.writeTo(f2);
f2.close();
System.out.println("Использование reset()");
f.reset();
for (int i=0; i<3; i++)
    f.write('X');
System.out.println(f.toString());
}
}
```

После выполнения программы вы получите следующий вывод (обратите внимание, как после обращения к `reset()` выводятся три символа X — каждый следующий вывод этого символа выполняется как бы с начала буфера вывода):

```
Буфер в форме строки содержит:
This should end up in the array
Массив содержит:
This should end up in the array
Вывод буфера в OutputStream
Использование reset()
XXX
```

Представленный пример использует метод удобства `writeTo()` для записи содержимого объектной переменной `f` в файл `test.txt`. Просмотр содержимого файла `test.txt`, созданного в предшествующем примере, показывает результат, который мы ожидали:

```
This should end up in the array
```

## Фильтрованные байтовые потоки

Фильтрованные потоки — просто оболочки (wrappers) вокруг основных потоков ввода или вывода, которые прозрачно обеспечивают некоторый расширенный уровень функциональных возможностей. Доступ к этим потокам обычно выполняется с помощью методов, использующих порождающий поток, связанный с суперклассом фильтрованных потоков. Типичные приме-

нения фильтрованных потоков — буферизация, трансляция символов и необработанных данных. Фильтрованные байтовые потоки — `FilterInputStream` и `FilterOutputStream`. Их конструкторы имеют форму:

```
FilterOutputStream(OutputStream os)
FilterInputStream(InputStream is)
```

Методы этих классов идентичны методам классов `InputStream` и `OutputStream`.

## Буферизированные байтовые потоки

Байтовый буферизированный поток расширяет класс фильтрованного потока, присоединяя буфер памяти к потокам ввода/вывода. Такой буфер позволяет выполнять I/O-операции не с одним, а с несколькими байтами одновременно, и, следовательно, увеличивает эффективность работы программы. При наличии буфера становится возможным такие операции, как пропуск байтов, маркировка и переустановка потока. Буферизированные поточные классы — это `BufferedInputStream` и `BufferedOutputStream`. Кроме того, буферизированный поток реализует класс `PushbackInputStream`.

### Класс `BufferedInputStream`

Буферизация ввода/вывода — очень общий способ оптимизации эффективности. Класс `BufferedInputStream` позволяет "обвернуть" любой `InputStream`-объект в буферизированный поток и достичь тем самым улучшения эффективности.

`BufferedInputStream` содержит два конструктора:

```
BufferedInputStream(InputStream inputStream)
BufferedInputStream(InputStream inputStream, int bufSize)
```

Первая форма создает буферизированный поток, используя размер буфера, заданный по умолчанию. Во второй форме размер буфера передается в параметре `bufSize`. Использование размеров, кратных странице памяти, дисковому блоку и т. д. может иметь существенное положительное воздействие на эффективность. Это, однако, зависит от реализации. Оптимальный размер буфера в общем случае зависит от операционной системы, объема имеющейся памяти и конфигурации хост-машины. Лучше всего размер буфера иметь кратным 8192 байтам, но и присоединение даже довольно маленького буфера к потоку ввода/вывода — всегда хорошая идея. Используя такой буфер, система нижнего уровня может читать в него блоки данных с диска или из сети и хранить в нем свои результаты. Таким образом, даже если вы читаете данные по одному байту, но в условиях буферизированного потока, то примерно 99.9% времени вы будете работать с быстродействующей памятью.

Кроме того, буферизация входного потока поддерживает возможность обратного перемещения в потоке имеющегося буфера. В дополнение к

методам `read()` и `skip()`, реализованным в любом `InputStream`-классе, `BufferedInputStream` поддерживает также методы `mark()` и `reset()`. Данный факт можно проверить с помощью метода `markSupported()`, возвращающего `true`, когда указанная поддержка обеспечивается.

В следующем примере изображается ситуация использования метода `mark()`, чтобы вспомнить, где мы находимся во входном потоке, и позднее применить метод `reset()`, чтобы вернуться туда. Этот пример анализирует поток, содержащий HTML-ссылку на символ авторского права. Формат подобной ссылки начинается с амперсанда (`&`), заканчивается точкой с запятой (`;`) и не имеет пробелов. Ввод программы имеет две таких ссылки (см. инициализатор переменной `s`), чтобы показать случаи, где `reset()` выполняется и где — нет.

```
// Использование буферизированного ввода.
import java.io.*;

class BufferedInputStreamDemo {
    public static void main(String args[]) throws IOException {
        String s = "Это: &copy; есть символ авторского права, " +
                   "а это: &copy - нет.\n";
        byte buf[] = s.getBytes();
        ByteArrayInputStream in = new ByteArrayInputStream(buf);
        BufferedInputStream f = new BufferedInputStream(in);
        int c;
        boolean marked = false;

        while ((c = f.read()) != -1) {
            switch(c) {
                case '&':
                    if (!marked) {
                        f.mark(32);
                        marked = true;
                    } else {
                        marked = false;
                    }
                    break;
                case ';':
                    if (marked) {
                        marked = false;
                        System.out.print("(c)");
                    } else
                        System.out.print((char) c);
                    break;
                case ' ':
                    if (marked) {
                        marked = false;
                        f.reset();
                        System.out.print("&");
                    }
            }
        }
    }
}
```

```
        } else
            System.out.print((char) c);
        break;
    default:
        if (!marked)
            System.out.print((char) c);
        break;
    }
}
}
```

Обратите внимание, что этот пример использует вызов `mark(32)`, который сохраняет маркер для чтения следующих 32 байтов (что достаточно для всех объектных ссылок HTML). Вывод этой программы:

Это: © есть символ авторского права, а это: &copy – нет.\n

## **Замечание**

Действие метода `mark()` ограничивается областью буфера. Это означает, что аргумент данного метода должен быть меньше, чем размер буфера.

### **Класс *BufferedOutputStream***

Класс `BufferedOutputStream` подобен любому классу `OutputStream`-иерархии за исключением того, что добавлен метод `flush()`. Его использование гарантирует, что буфера данных физически записываются на актуальное выходное устройство. Так как задача `BufferedOutputStream` — повысить эффективность, уменьшая количество фактических записей данных системой, вам может потребоваться вызов метода `flush()`, чтобы организовать запись любых данных, которые находятся в буфере, на устройство вывода.

В отличие от буферизированного ввода, буферизация вывода не обеспечивает дополнительных функциональных возможностей. Эффективность увеличивается только за счет буферов вывода. Существует два конструктора:

```
BufferedOutputStream(OutputStream outputStream)
```

```
BufferedOutputStream(OutputStream outputStream, int bufSize)
```

Первая форма создает буферизированный поток, используя буфер 512 байт. Во второй форме размер буфера задается в параметре *bufSize*.

### **Класс *PushbackInputStream***

Одно из новых применений буферизации — выполнение возврата считанного байта обратно во входной поток (операция `pushback`). Класс `PushbackInputStream` осуществляет эту идею. Он обеспечивает механизм, позволяющий "быстро

"взглянуть" на то, что происходит во входном потоке без его прерывания. PushbackInputStream имеет следующие конструкторы:

```
PushbackInputStream(InputStream inputStream)
PushbackInputStream(InputStream inputStream, int numBytes)
```

Первая форма создает поточный объект, который позволяет возвратить один байт во входной поток. Вторая форма создает поток, который имеет специальный pushback-буфер длиной *numBytes* байтов. Он дает возможность выполнять множественный возврат байтов во входной поток.

Кроме знакомых методов из *InputStream*, в *PushbackInputStream* определен метод *unread()* в следующих формах:

```
void unread(int ch)
void unread(byte buffer[ ])
void unread(byte buffer[], int offset, int numChars)
```

Первая форма возвращает младший байт параметра *ch*. Возвращенный байт будет прочитан следующим (за *unread()*) вызовом *read()*. Вторая форма возвращает группу байтов — весь буферный массив *buffer*. Третья форма обеспечивает получение части массива *buffer* (*numChars* байтов, начиная с номера (индекса) *offset* в буферном массиве *buffer*). Если осуществляется попытка возвратить байт при заполненном буфере возврата, будет выброшено исключение типа *IOException*.

Java 2 внес мелкие изменения в класс *PushbackInputStream*: он теперь содержит метод *skip()*.

Далее приводится пример, который демонстрирует, как синтаксический анализатор языка программирования может использовать класс *PushbackInputStream* и метод *unread()*, чтобы подчеркнуть различие между операцией сравнения (*==*) и операцией присваивания (*=*):

```
// Демонстрирует метод unread().
import java.io.*;

class PushbackInputStreamDemo {
    public static void main(String args[]) throws IOException {
        String s = "if (a == 4) a = 0;\n";
        byte buf[] = s.getBytes();
        ByteArrayInputStream in = new ByteArrayInputStream(buf);
        PushbackInputStream f = new PushbackInputStream(in);
        int c;

        while ((c = f.read()) != -1) {
            switch(c) {
                case '=':
                    if ((c = f.read()) == '=') System.out.print(".eq.");
                    else System.out.print("=");
            }
        }
    }
}
```

```
        else {
            System.out.print("<-");
            f.unread(c);
        }
        break;
    default:
        System.out.print((char) c);
        break;
    }
}
```

Вывод этого примера ( обратите внимание, что операция (==) была заменена на ".eq.", операция (=) — на "<- "):

```
if (a .eq. 4) a <- 0;
```

## **Замечание**

`PushbackInputStream` производит побочный эффект: он сделал несостоительными методы `mark()` и `reset()` класса `InputStream`. Для проверки возможности использования `mark()/reset()` на потоке, с которым вы собираетесь работать, используйте метод `markSupported()`.

## Класс *SequenceInputStream*

Класс `SequenceInputStream` позволяет склеивать множество `InputStream`-объектов. Конструкция `SequenceInputStream` отличается от любого другого класса `InputStream`-иерархии. Конструктор `SequenceInputStream` использует в качестве параметров или пару `InputStream`-объектов или `Enumeration`-объект с `InputStream`-компонентами:

```
SequenceInputStream(InputStream first, InputStream second)  
SequenceInputStream(Enumeration streamEnum)
```

Класс выполняет запросы чтения первого объекта типа `InputStream` (параметр `first`), пока он не закончится, и затем переключается на второй (параметр `second`). В случае `Enumeration`-параметра `streamEnum` это будет продолжаться через все его `InputStream`-компоненты, пока не будет прочитан конец последнего поточного компонента в перечислении.

Простой пример, использующий `SequenceInputStream` для вывода содержимого двух файлов, представлен ниже:

```
// Демонстрирует последовательный ввод.  
import java.io.*;  
import java.util.*;
```

```
class InputStreamEnumerator implements Enumeration {
    private Enumeration files;
    public InputStreamEnumerator(Vector files) {
        this.files = files.elements();
    }
    public boolean hasMoreElements() {
        return files.hasMoreElements();
    }
    public Object nextElement() {
        try {
            return new FileInputStream(files.nextElement().toString());
        } catch (Exception e) {
            return null;
        }
    }
}
class SequenceInputStreamDemo {
    public static void main(String args[]) throws Exception {
        int c;
        Vector files = new Vector();
        files.addElement("/autoexec.bat");
        files.addElement("/config.sys");
        InputStreamEnumerator e = new InputStreamEnumerator(files);
        InputStream input = new SequenceInputStream(e);
        while ((c = input.read()) != -1) {
            System.out.print((char) c);
        }
        input.close();
    }
}
```

Здесь создается *Vector*-объект (переменная *files*), и добавляются в него два файловых имени. Затем вектор имен передается конструктору класса *InputStreamEnumerator*, который обеспечивает оболочку данного вектора, где возвращаемые элементы — не имена файлов, а скорее открытые *FileInputStream*-объекты на этих именах. Объект *input*, созданный с помощью *SequenceInputStream*-конструктора, открывает каждый файл по очереди, а завершающий короткий цикл печатает на экране содержимое двух файлов.

## Класс *PrintStream*

Класс *PrintStream* обеспечивает все возможности форматирования, которыми мы пользовались (через дескриптор стандартного файла вывода — *System.out*) с самого начала этой книги. *PrintStream* имеет два конструктора:

```
PrintStream(OutputStream outputStream)
PrintStream(OutputStream outputStream, boolean flushOnNewline)
```

где `flushOnNewline` управляет сбрасыванием выходного потока каждый раз, когда встречается newline-последовательность (\n). Если `flushOnNewline` — true, сбрасывание выполняется автоматически, если — false, сбрасывание не автоматическое. Первый конструктор выполняет неавтоматический сброс.

Конструкторы `PrintStream` были исключены в Java 1.1, потому что `PrintStream` не работает с символами Unicode и, таким образом, неудобен для интернационализации. (Для нового кода вы должны использовать `PrintWriter`, который описан далее в этой главе.) Однако методы, определенные в `PrintStream`, не исключены. Это означает, что допустимо использовать `PrintStream`-объекты, но не создавать их! Сначала это кажется абсурдным, однако это не так. Резон в том, что `System.out`-объект на самом деле является `PrintStream`-объектом, который широко используется. Так как `PrintStream`-методы не исключены, то использование `System.out` все еще допустимо. Однако, для новых программ, лучше ограничить использование `System.out` простыми утилитами, отладкой и примерами программ. Любая реальная программа, которая отображает консольный вывод, должна делать это через `PrintWriter`, чтобы ее можно было использовать в глобальной среде.

`PrintStream`-объекты Java поддерживают методы `print()` и `println()` для всех типов, включая `Object`. Если аргумент — не простой тип, `PrintStream`-методы вызовут объектный метод `toString()` и затем напечатают результат.

## Класс `RandomAccessFile`

Класс `RandomAccessFile` инкапсулирует файл прямого доступа. Он не является производным от классов `InputStream` или `OutputStream`. Вместо этого, реализует интерфейсы `DataInput` и `DataOutput`, которые определяют базовые методы ввода/вывода. Он также поддерживает запросы позиционирования — т. е. вы можете устанавливать внутри файла `указатель` (файла). Класс обладает двумя конструкторами:

```
RandomAccessFile(File fileObj, String access) throws IOException
RandomAccessFile(String filename, String access) throws IOException
```

В первой форме параметр `fileObj` определяет имя файла, открываемого как `File`-объект. Во второй форме имя файла передается через параметр `filename`. В обоих случаях `access` определяет, какой тип доступа разрешается к файлу. Если (в вызове конструктора) указано значение "r", то файл можно читать, но не записывать, если — "rw", то файл открыт в режиме чтения-записи.

Метод `seek()`, представленный ниже, служит для установки текущей позиции указателя файла:

```
void seek(long newPos) throws IOException
```

где *newpos* определяет новую позицию (номер байта) указателя файла с начала файла. После обращения к *seek()* следующая операция чтения или записи будет выполнена в указанной здесь позиции файла.

*RandomAccessFile* реализует стандартные методы ввода и вывода, которые можно использовать для чтения и записи файлов произвольного доступа. Существует, однако, один новый метод, добавленный в Java 2 — *setLength()*. Он имеет следующую сигнатуру:

```
void setLength(long len) throws IOException
```

Этот метод приводит длину файла к значению, указанному в *len*. Метод может использоваться для удлинения или сокращения файла. Если файл удлиняется, то добавленная часть имеет неопределенные значения.

## Символьные потоки

Хотя классы байтовых потоков обеспечивают достаточные функциональные возможности для обработки любого типа операций ввода/вывода, они не могут работать непосредственно с Unicode-символами. Так как одной из главных целей Java была поддержка философии "писать однажды, выполнять везде", возникла необходимость включить прямую поддержку ввода/вывода для символов. В этом разделе обсуждается несколько классов символьного ввода/вывода. Как объяснялось ранее, наверху иерархии символьных потоков находятся абстрактные классы *Reader* и *Writer*. С них мы и начнем обсуждение.

### Замечание

Как обсуждалось в главе 12, классы символьного ввода/вывода были добавлены в версии Java 1.1. Поэтому вы все еще можете встретить старые конструкции, которые используют байтовые потоки там, где должны быть символьные. Такие коды лучше обновлять.

## Класс *Reader*

*Reader* — абстрактный класс, который определяет Java-модель поточного символьного ввода. Все методы в этом классе в аварийных ситуациях выбрасывают исключение типа *IOException*. Краткое описание методов класса *Reader* приводится в табл. 17.4.

**Таблица 17.4. Методы, определенные в классе *Reader***

Метод	Описание
<b>abstract void close()</b>	Закрывает источник ввода. Дальнейшие попытки чтения генерируют исключение типа <i>IOException</i>

Таблица 17.4 (окончание)

Метод	Описание
<code>void mark(int numChars)</code>	Размещает маркер в текущей точке входного потока, который будет оставаться актуальным, пока не считаются <code>numChars</code> символов
<code>boolean markSupported()</code>	Возвращает <code>true</code> , если данный поток поддерживает методы <code>mark()</code> / <code>reset()</code>
<code>int read()</code>	Возвращает целочисленное представление следующего доступного символа из вызывающего входного потока. Когда встречается признак конца файла (EOF), возвращается <code>-1</code>
<code>int read(char buffer[ ])</code>	Пытается прочитать до <code>buffer.length</code> символов в <code>buffer</code> -массиве и возвращает фактическое число символов, которые успешно прочитались. Когда встречается признак конца файла (EOF), возвращается <code>-1</code>
<code>abstract int read(char buffer[], int offset, int numChars)</code>	Пытается читать до <code>numChars</code> символов в <code>buffer</code> , начиная с позиции <code>offset</code> и возвращая число успешно считанных символов. Когда встречается признак конца файла (EOF), возвращается <code>-1</code>
<code>boolean ready()</code>	Возвращает <code>true</code> , если следующий запрос ввода не будет ждать. Иначе, возвращает <code>false</code>
<code>void reset()</code>	Сбрасывает указатель ввода на предварительно установленный маркер
<code>long skip(long numChars)</code>	Пропускает <code>numChars</code> символов ввода, возвращая число фактически пропущенных символов

## Класс Writer

Writer — абстрактный класс, который определяет Java-модель поточного символьного вывода. Все методы этого класса возвращают значение `void` и выбрасывают исключение типа `IOException` в случае ошибок. В табл. 17.5 приводится краткое описание методов класса `Writer`.

Таблица 17.5. Методы, определенные в классе Writer

Метод	Описание
<code>abstract void close()</code>	Закрывает выходной поток. Дальнейшие попытки записи генерируют исключение типа <code>IOException</code>
<code>abstract void flush()</code>	Завершает состояние вывода так, что любые буферы очищаются. То есть он сбрасывает буферы вывода
<code>void write(int ch)</code>	Записывает один символ в выходной поток вызывающего объекта. Обратите внимание, что параметр имеет тип <code>int</code> , который позволяет вам вызывать <code>write()</code> с выражениями (в аргументе) без необходимости приводить их обратно к типу <code>char</code>
<code>void write(char buffer[ ])</code>	Записывает заполненный массив символов в вызывающий выходной поток
<code>abstract void write (char buffer[ ], int offset,int numChars)</code>	Записывает <code>numChars</code> символов из массива <code>buffer</code> , начиная в позиции <code>offset</code> в вызывающий выходной поток
<code>void write(String str)</code>	Записывает строку <code>str</code> в вызывающий выходной поток
<code>void write(String str, int offset, int numChars)</code>	Записывает подстроку из <code>numChars</code> символов массива <code>str</code> , начиная с позиции <code>offset</code>

## Класс FileReader

Класс `FileReader` создает `Reader`-объект, который можно использовать для чтения содержимого файла. Ниже показаны два его обычно используемых конструктора:

```
FileReader(String filePath)
FileReader(File fileObj)
```

Любой может выбрасывать исключение типа `FileNotFoundException`. Здесь `filePath` — полное имя файла; `fileObj` — объект класса `File`, который описывает файл.

В следующем примере показывается, как можно читать строки из файла и печатать их в поток стандартного вывода. В нем читается свой собственный исходный файл, который должен быть в текущем каталоге.

```
// Демонстрирует FileReader.
import java.io.*;
```

```

class FileReaderDemo {
    public static void main(String args[]) throws Exception {
        FileReader fr = new FileReader("FileReaderDemo.java");
        BufferedReader br = new BufferedReader(fr);
        String s;

        while((s = br.readLine()) != null) {
            System.out.println(s);
        }

        fr.close();
    }
}

```

## Класс *FileWriter*

*FileWriter* создает *Writer*-объект, который можно использовать для записи файла. Его обычно используемые конструкторы имеют следующие сигнатуры:

```

FileWriter(String filePath)
FileWriter(String filePath, boolean append)
FileWriter(File fileObj)

```

Они могут выбрасывать исключения типа *IOException* или *SecurityException*. Здесь *filePath* — полное имя файла; *fileObj* — *File*-объект, который описывает файл. Если *append* — true, то вывод добавляется в конец файла.

Создание *FileWriter*-объекта не зависит от того, существует ли уже файл. *FileWriter* создаст файл перед его открытием для вывода, когда вы создаете объект. Если вы пытаетесь открывать файл только для чтения, будет выброшено исключение типа *IOException*.

Следующий пример является символьно-поточной версией примера, показанного ранее (см. раздел "Класс *FileOutputStream*" выше в этой главе). Данная версия строит буфер символов, сначала создавая *String*-объект и затем используя метод *getChars()*, чтобы извлечь эквивалент символьного массива. Затем он создает три файла. Первый, *file1.txt*, будет содержать каждый нечетный символ исходной строки. Второй, *file2.txt*, будет содержать полный набор символов. Наконец, третий, *file3.txt*, будет содержать только последнюю четверть исходной строки.

```

// Демонстрирует класс FileWriter.
import java.io.*;

class FileWriterDemo {
    public static void main(String args[]) throws Exception {
        String source = "Теперь настало время всем порядочным людям\n"
                      + " стране прийти на помощь своей\n"
                      + " и платить причитающиеся налоги.";
    }
}

```

```
char buffer[] = new char[source.length()];
source.getChars(0, source.length(), buffer, 0);

FileWriter f0 = new FileWriter("file1.txt");
for (int i=0; i < buffer.length; i += 2) {
    f0.write(buffer[i]);
}
f0.close();

FileWriter f1 = new FileWriter("file2.txt");
f1.write(buffer);
f1.close();

FileWriter f2 = new FileWriter("file3.txt");

f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);
f2.close();
}
}
```

## Класс CharArrayReader

CharArrayReader является реализацией входного потока, которая использует символьный массив как источник. Данный класс имеет два конструктора, каждый из которых получает символьный массив в качестве источника данных:

```
CharArrayReader(char array[])
CharArrayReader(char [ ], int start, int numChars)
```

где `array[ ]` — источник ввода. Второй конструктор создает Reader-объект из подмножества исходного символьного массива, который начинается с индексной позиции `start` (в исходном массиве) и имеет длину `numChars` символов.

Следующий пример использует пару CharArrayReader-объектов:

```
// Демонстрирует класс CharArrayReader.
import java.io.*;

public class CharArrayReaderDemo {
    public static void main(String args[]) throws IOException {
        String tmp = "abcdefghijklmnopqrstuvwxyz";
        int length = tmp.length();
        char c[] = new char[length];

        tmp.getChars(0, length, c, 0);
        CharArrayReader input1 = new CharArrayReader(c);
        CharArrayReader input2 = new CharArrayReader(c, 0, 5);
```

```

int i;
System.out.println("input1 содержит:");
while((i = input1.read()) != -1) {
    System.out.print((char)i);
}
System.out.println();

System.out.println("input2 содержит:");
while((i = input2.read()) != -1) {
    System.out.print((char)i);
}
System.out.println();
}
}

```

Объект `input1` создан с использованием полного алфавита (английского) в нижнем регистре, в то время как `input2` содержит только первые пять его символов. Вывод этой программы:

```

input1 содержит:
abcdefghijklmnopqrstuvwxyz
input2 содержит:
abcde

```

## Класс `CharArrayWriter`

`CharArrayWriter` — реализация выходного потока, которая записывает данные в символьный массив. `CharArrayWriter` имеет два конструктора:

```

CharArrayWriter()
CharArrayWriter(int numChars)

```

В первой форме создается буфер с размером, заданным по умолчанию. Во второй — буфер с размером, указанным в параметре `numChars`. Буфер содержится в поле `buf` класса `CharArrayWriter`. Если необходимо, размер буфера будет увеличен автоматически. Число символов, хранящихся в буфере, содержится в поле `count` класса `CharArrayWriter`. Как `buf`, так и `count` — защищенные поля.

Следующий пример демонстрирует класс `CharArrayWriter`, переделывая пример программы, представленной ранее при описании класса `ByteArrayOutputStream`. Вывод данной программы тот же самый, что и у предыдущей версии.

```

// Демонстрирует класс CharArrayWriter.
import java.io.*;

class CharArrayWriterDemo {
    public static void main(String args[]) throws IOException {

```

```
CharArrayWriter f = new CharArrayWriter();
String s = "This should end up in the array";
char buf[] = new char[s.length()];
buf[0] = s.charAt(0);
s.getChars(0, s.length(), buf, 0);
f.write(buf);
System.out.println("Буфер в форме строки содержит:");
System.out.println(f.toString());
System.out.println("Массив содержит:");

char c[] = f.toCharArray();
for (int i=0; i<c.length; i++) {
    System.out.print(c[i]);
}

System.out.println("\n Вывод буфера в FileWriter");
FileWriter f2 = new FileWriter("test.txt");
f.writeTo(f2);
f2.close();
System.out.println("Использование reset()");
f.reset();
for (int i=0; i<3; i++)
    f.write('X');
System.out.println(f.toString());
}
```

## Класс *BufferedReader*

BufferedReader улучшает эффективность, буферизуя ввод. Он содержит два конструктора:

```
BufferedReader(Reader inputStream)
BufferedReader(Reader inputStream, int bufferSize)
```

Первая форма создает буферизированный символьный поток, используя размер буфера, заданный по умолчанию. Во второй форме размер буфера передается конструктору через параметр *bufSize*.

Как и в случае с байтовыми потоками, буферизация входного символьного потока также обеспечивает поддержку перемещения символов обратного в поток (в пределах имеющегося буфера). Для поддержки этого в BufferedReader реализованы методы *mark()* и *reset()*, а метод *markSupported()* возвращает *true* (что означает поддержку этих возможностей в Java).

Следующий пример переделывает пример с классом *BufferedInputStream*, показанный ранее, так, чтобы он использовал символьный поток *BufferedReader*, а не буферизированный байтовый поток. Как и прежде, он

обращается к методам `mark()` и `reset()` для анализа потока, содержащего HTML-ссылку на символ авторского права. Такая ссылка начинается с амперсанда (&), заканчивается точкой с запятой (;) и не имеет пробелов. Ввод примера имеет два амперсанда, чтобы показать, где `reset()` (т. е. возврат указателя) происходит, а где — нет. Вывод — такой же, как в исходном варианте программы.

```
// Использование буферизированного ввода.
import java.io.*;

class BufferedReaderDemo {
    public static void main(String args[]) throws IOException {
        String s = "This is a &copy; copyright symbol " +
            "but this is &copy not.\n";
        char buf[] = new char[s.length()];
        s.getChars(0, s.length(), buf, 0);
        CharArrayReader in = new CharArrayReader(buf);
        BufferedReader f = new BufferedReader(in);
        int c;
        boolean marked = false;

        while ((c = f.read()) != -1) {
            switch(c) {
                case '&':
                    if (!marked) {
                        f.mark(32);
                        marked = true;
                    } else {
                        marked = false;
                    }
                    break;
                case ';':
                    if (marked) {
                        marked = false;
                        System.out.print("(c)");
                    } else
                        System.out.print((char) c);
                    break;
                case ' ':
                    if (marked) {
                        marked = false;
                        f.reset();
                        System.out.print("&");
                    } else
                        System.out.print((char) c);
                    break;
                default:
```

```
        if (!marked)
            System.out.print((char) c);
        break;
    }
}
}
```

## Класс *BufferedWriter*

`BufferedWriter` — это класс `Writer`, к которому добавлен метод `flush()`, гарантирующий, что буферы данных физически записываются в актуальный выходной поток. Использование `BufferedWriter` может увеличить эффективность, уменьшая количество фактических записей данных в выходной поток. `BufferedWriter` имеет два конструктора:

```
BufferedWriter(Writer outputStream)
BufferedWriter(Writer outputStream, int bufSize)
```

Первая форма создает буферизированный поток, используя буфер с размером, заданным по умолчанию. Во второй форме размер буфера передается через параметр *bufSize*.

## Класс *PushbackReader*

Класс `PushbackReader` позволяет возвращать во входной поток один или несколько символов. Это дает возможность "заглянуть вперед" во входной поток. Класс содержит два конструктора:

```
PushbackReader(Reader inputStream)  
PushbackReader(Reader inputStream, int bufSize)
```

Первая форма создает буферизированный поток, позволяющий вернуть обратно один символ. Во второй форме размер возвращаемой порции данных указывается параметром *bufSize*.

В `PushbackReader` определен метод `unread()`, который возвращает один или большее количество символов в вызывающий входной поток. Он имеет три формы:

```
void unread(int ch)
void unread(char buffer[ ])
void unread(char buffer[ ], int offset, int numChars)
```

Первая форма возвращает один символ, указанный в *ch*. Данный символ будет считан следующим вызовом метода `read()`. Вторая форма возвращает символы из *buffer*. Третья форма возвращает *numChars* символов, начиная

с индекса *offset* в буфере *buffer*. Если осуществляется попытка вернуть символ, когда буфер возврата заполнен, будет выброшено исключение типа IOException.

Следующая программа переделывает более ранний пример с классом PushBackInputStream, заменив PushBackInputStream на PushbackReader. Как и прежде, в ней показано, как синтаксический анализатор языка программирования может использовать поток с возвратом, чтобы подчеркнуть различие между операцией сравнения (==) и операцией присваивания (=).

```
// Демонстрирует метод unread().
import java.io.*;

class PushbackReaderDemo {
    public static void main(String args[]) throws IOException {
        String s = "if (a == 4) a = 0;\n";
        char buf[] = new char[s.length()];
        s.getChars(0, s.length(), buf, 0);
        CharArrayReader in = new CharArrayReader(buf);
        PushbackReader f = new PushbackReader(in);
        int c;

        while ((c = f.read()) != -1) {
            switch(c) {
                case '=':
                    if ((c = f.read()) == '=')
                        System.out.print(".eq.");
                    else {
                        System.out.print("<-");
                        f.unread(c);
                    }
                    break;
                default:
                    System.out.print((char) c);
                    break;
            }
        }
    }
}
```

## Класс PrintWriter

*PrintWriter* — по существу, символьная версия класса *PrintStream*. Он определяет методы форматного вывода *print()* и *println()*. *PrintWriter* имеет четыре конструктора:

*PrintWriter(OutputStream outputStream)*

*PrintWriter(OutputStream outputStream, boolean flushOnNewline)*

```
PrintWriter(Writer outputStream)
```

```
PrintWriter(Writer outputStream, boolean flushOnNewline)
```

где `flushOnNewline` управляет сбрасыванием выходного потока каждый раз, когда выводится символ новой строки `\n`. Если `flushOnNewline` — `true`, сбрасывание выполняется автоматически, если — `false`, сбрасывание не автоматическое. Первый и третий конструкторы организуют неавтоматический сброс.

Объекты `PrintWriter` поддерживают методы `print()` и `println()` для всех типов, включая `Object`. Если аргумент — не простой тип, `PrintWriter`-методы будут вызывать объектный метод `toString()` и затем распечатывать результат.

## Использование поточного ввода/вывода

Следующий пример демонстрирует некоторые классы и методы поточного ввода/вывода символов. Программа реализует стандартную команду "счетчик слов" (или `wc`<sup>1</sup>-команду). Программа имеет два режима: если ее запуск осуществляется без параметров, то она работает на стандартном входном потоке. Если же в качестве параметров запуска определены один или несколько имен файлов, то программа работает на каждом из них.

```
// Утилита "счетчик слов".
import java.io.*;

class WordCount {
    public static int words = 0;
    public static int lines = 0;
    public static int chars = 0;

    public static void wc(InputStreamReader isr)
        throws IOException {
        int c = 0;
        boolean lastWhite = true;
        String whiteSpace = "\t\n\r";

        while ((c = isr.read()) != -1) {
            // считать символы
            chars++;
            // Count lines
            if (c == '\n') {
                lines++;
            }
        }
    }
}
```

---

<sup>1</sup> WC (от англ. word count) — счетчик слов. — *Примеч. пер.*

```
// считать слова, определяя начало слова
int index = whiteSpace.indexOf(c);
if(index == -1) {
    if(lastWhite == true) {
        ++words;
    }
    lastWhite = false;
}
else {
    lastWhite = true;
}
}

if(chars != 0) {
    ++lines;
}

}

public static void main(String args[]) {
    FileReader fr;
    try {
        if(args.length == 0) { // мы работаем с stdin (стандартный ввод)
            wc(new InputStreamReader(System.in));
        }
        else { // мы работаем со списком файлов
            for(int i = 0; i < args.length; i++) {
                fr = new FileReader(args[i]);
                wc(fr);
            }
        }
    }
    catch(IOException e) {
        return;
    }
    System.out.println(lines + " " + words + " " + chars);
}
}
```

Метод `wc()` работает на любом входном потоке и считает число символов, строк и слов. Он прослеживает четность слов и пробелы в переменной `lastNotWhite`.

При выполнении без аргументов в классе `WordCount` создается объект типа `InputStreamReader`, использующий в качестве источника поточный объект стандартного ввода (`System.in`). Указанный поток передается методу `wc()`, который делает фактический подсчет. При выполнении `wc()` с одним или несколькими аргументами `WordCount` предполагает, что они являются именами файлов, и создает `FileReader`-объект для каждого из них, пересылая

результатирующими `FileReader`-объекты в метод `wc()`. В любом случае перед выходом из программы печатаются результаты.

## Улучшение метода `wc()` с помощью класса `StreamTokenizer`

Существует более совершенный способ поиска образцов во входном потоке. Он использует еще один класс ввода/вывода Java — `StreamTokenizer`. Подобно классу `StringTokenizer` из главы 16, `StreamTokenizer` разбивает входной поток (`InputStream`) на лексемы (`tokens`), которые разграничены наборами символов. Он содержит конструктор:

```
StreamTokenizer(Reader inStream)
```

где параметр `inStream` должен быть некоторой формой класса `Reader`.

`StreamTokenizer` определяет несколько методов. В этом примере, мы будем использовать только некоторые из них. Для переустановки умалчивающего набора разделителей мы будем применять метод `resetSyntax()`. Заданный по умолчанию набор разделителей точно настроен для разбиения на лексемы программ Java и является, таким образом, слишком специализированным для рассматриваемого примера. Мы объявляем, что наши лексемы или "слова" являются последовательными строками видимых символов, ограниченных с обеих сторон пробелами.

Мы используем метод `eolIsSignificant()`, гарантирующий, что `newline`-символы будут представляться как лексемы, так что мы можем считать не только слова, но и строки. Он имеет следующую общую форму:

```
void eolIsSignificant(boolean eolFlag)
```

Если `eolFlag` — `true`, символы конца строки возвращаются как лексемы. Если `eolFlag` — `false`, символы конца строки игнорируются.

Метод `WordChars()` используется для определения диапазона символов, которые можно использовать в словах. Его общая форма имеет вид:

```
void wordChars(int start, int end)
```

Здесь `start` и `end` определяют диапазон допустимых символов. В данной программе допустимы символы в диапазоне от 33 до 255.

Пробельные (`whitespace`) символы определяются с помощью метода `whiteSpaceChars()`. Он имеет следующую общую форму:

```
void whiteSpaceChars(int start, int end)
```

Здесь `start` и `end` указывают диапазон допустимых пробельных символов.

Следующая лексема получается из входного потока с помощью метода `nextToken()`. Он возвращает тип лексемы.

В классе StreamTokenizer определено четыре int-константы: TT\_EOF, TT\_EOL, TT\_NUMBER и TT\_WORD. Кроме того, определены три экземплярные переменные — nval, sval и ttype. nval — это общая (public) переменная типа double, которая используется для хранения распознанных численных значений. sval — это public String-переменная применяется для хранения значений распознанных слов. ttype — это общая int-переменная, указывающая тип лексемы, которая была только что считана методом nextToken(). Если лексема является словом, ttype равняется TT\_WORD. Если лексема — число, ttype равняется TT\_NUMBER. Если лексема — одиночный символ, ttype содержит его значение. Если встречается символ конца строки, ttype равняется TT\_EOL (при этом предполагается, что метод eolIsSignificant() был вызван с параметром true). Если встретился конец потока, ttype равняется TT\_EOF.

Далее следует модифицированная программа счетчика слов, использующая StreamTokenizer:

```
// Улучшенная программа счетчика слов, использующая StreamTokenizer.
import java.io.*;

class WordCount {
    public static int words=0;
    public static int lines=0;
    public static int chars=0;

    public static void wc(Reader r) throws IOException {
        StreamTokenizer tok = new StreamTokenizer(r);

        tok.resetSyntax();
        tok.wordChars(33, 255);
        tok.whitespaceChars(0, ' ');
        tok.eolIsSignificant(true);

        while (tok.nextToken() != tok.TT_EOF) {
            switch (tok.ttype) {
                case tok.TT_EOL:
                    lines++;
                    chars++;
                    break;
                case tok.TT_WORD:
                    words++;
                    break;
                default:
                    chars += tok.sval.length();
                    break;
            }
        }
    }

    public static void main(String args[]) {
        if (args.length == 0) { // мы работаем с stdin (стандартный ввод)

```

```
try {
    wc(new InputStreamReader(System.in));
    System.out.println(lines + " " + words + " " + chars);
} catch (IOException e) {}
} else { // We're working with a list of files
    int twords = 0, tchars = 0, tlines = 0;
    for (int i=0; i<args.length; i++) {
        try {
            words = chars = lines = 0;
            wc(new FileReader(args[i]));
            twords += words;
            tchars += chars;
            tlines += lines;
            System.out.println(args[i] + ": " +
                lines + " " + words + " " + chars);
        } catch (IOException e) {
            System.out.println(args[i] + ": error.");
        }
    }
    System.out.println("total: " +
        tlines + " " + twords + " " + tchars);
}
}
```

## Сериализация

*Сериализация* (*serialization*) — это процесс записи состояния объекта в форме байтового потока. Это полезно, когда вы хотите сохранить состояние своей программы в постоянной области памяти, например, в файле. Позднее вы можете восстановить эти объекты, используя процесс *десериализации* (*deserialization*).

Сериализация необходима также для реализации механизма вызова удаленных методов (RMI, Remote Method Invocation). RMI позволяет Java-объекту на одной машине вызывать метод Java-объекта, находящегося на удаленной машине. Удаленный объект можно указывать как аргумент удаленного метода. Посылающая машина сериализует объект и передает его. Принимающая машина десериализует принятый объект. (Более подробная информация относительно RMI находится в главе 24.)

Предположим, что объект, который нужно сериализовать, имеет ссылки на другие объекты, которые, в свою очередь, имеют ссылки на еще большее количество объектов. Этот набор объектов и отношений между ними формирует направленный граф. В объектном графе могут также содержаться циклические ссылки. То есть объект X может содержать ссылку к объекту Y,

а объект Y может содержать обратную ссылку к объекту X. Объекты могут также иметь ссылки на себя. Средства сериализации и десериализации объектов были разработаны для правильной работы в этих сценариях. Если вы пытаетесь сериализовать объект, находящийся наверху объектного графа, все другие связанные с ним всевозможными ссылками объекты выстраиваются в рекурсивную последовательность и сериализуются. Точно так же в процессе десериализации все эти объекты и их ссылки восстанавливаются в правильную иерархию.

Далее следует краткий обзор интерфейсов и классов, которые поддерживают сериализацию.

## Интерфейс *Serializable*

Средства сериализации могут работать только с объектами, которые реализуют интерфейс *Serializable*. Интерфейс *Serializable* не определяет никаких членов. Он просто указывает на то, что класс может быть сериализован (т. е. преобразован в последовательную форму). Если класс — сериализован (т. е. реализует интерфейс *Serializable*), то все его подклассы также сериализованы.

Переменные, которые объявлены как *transient* или *static*, не сохраняются средствами сериализации.

## Интерфейс *Externalizable*

Средства сериализации и десериализации Java были разработаны так, чтобы большая часть работы по сохранению и восстановлению состояния объекта выполнялась автоматически. Однако есть ситуации, в которых программисту нужно иметь контроль над этими процессами. Например, если желательно использовать технику сжатия или криптографического кодирования. Именно для таких ситуаций и разработан интерфейс *Externalizable*. В нем определяются два метода:

```
void readExternal(ObjectInput inStream)
    throws IOException, ClassNotFoundException
void writeExternal(ObjectOutput outStream)
    throws IOException
```

Параметры этих методов: *inStream* — байтовый поток, из которого объект должен быть считан; *outStream* — байтовый поток, в который объект должен быть записан.

## Интерфейс *ObjectOutput*

Интерфейс *ObjectOutput* расширяет интерфейс *DataOutput* и поддерживает сериализацию объектов. В нем определены методы, показанные в табл. 17.6.

Обратите особое внимание на метод `writeObject()`. Он вызывается, чтобы сериализовать объект. Все перечисляемые методы в ошибочных ситуациях выбрасывают исключение типа `IOException`.

**Таблица 17.6. Методы, определенные в `ObjectOutput`**

Method	Описание
<code>void close()</code>	Закрывает вызывающий поток. Дальнейшие попытки записи генерируют исключение типа <code>IOException</code>
<code>void flush()</code>	Завершает состояние вывода так, чтобы любые буферы были очищены. То есть сбрасывает (в выходной поток) буферы вывода
<code>void write(byte buffer[ ])</code>	Записывает (в вызывающий поток) массив байтов
<code>void write(byte buffer[ ], int offset, int numBytes)</code>	Записывает <code>numBytes</code> байтов массива <code>buffer</code> , начиная с позиции <code>offset</code>
<code>void write(int b)</code>	Записывает один байт (причем записывается только младший байт)
<code>void writeObject(Object obj)</code>	Записывает объект <code>obj</code> (в вызывающий поток)

## Класс `ObjectOutputStream`

Класс `ObjectOutputStream` расширяет интерфейс `OutputStream` и реализует интерфейс `ObjectOutput`. Он отвечает за запись *объектов* в поток. Конструктор этого класса:

```
ObjectOutputStream(OutputStream outStream) throws IOException
```

где параметр `outStream` — выходной поток, в который будут записываться сериализованные объекты.

Чаще всего в указанном классе используются методы, представленные в табл. 17.7. В аварийных ситуациях они выбрасывают исключение типа `IOException`. Java 2 добавляет внутренний класс к `ObjectOutputStream` с именем `PutField`. Он облегчает запись постоянных полей, но описание его использования — вне рамок этой книги.

**Таблица 17.7. Обычно используемые методы класса `ObjectOutputStream`**

Метод	Описание
<code>void close()</code>	Закрывает вызывающий поток. Дальнейшие попытки записи будут генерировать исключение типа <code>IOException</code>

Таблица 17.7 (окончание)

Метод	Описание
<code>void flush()</code>	Завершает состояние вывода так, чтобы любые буферы были очищены. То есть сбрасывают буферы вывода
<code>void write(byte buffer[ ])</code>	Записывает массив байтов в поток вывода
<code>void write(byte buffer[ ], int offset, int numBytes)</code>	Записывает <code>numBytes</code> байтов массива <code>buffer</code> , начиная с позиции <code>offset</code>
<code>void write(int b)</code>	Записывает один байт в поток вывода (причем записывается только младший байт)
<code>void writeBoolean(boolean b)</code>	Записывает boolean-значение в поток вывода
<code>void writeByte(int b)</code>	Записывает byte-значение в поток вывода (причем записывается только младший байт)
<code>void writeBytes(String str)</code>	Записывает несколько байтов (строки <code>str</code> ) в поток вывода
<code>void writeChar(int c)</code>	Записывает char-значение в поток вывода
<code>void writeChars(String str)</code>	Записывает символы строчного параметра <code>str</code> в поток вывода
<code>void writeDouble(double d)</code>	Записывает double-значение в поток вывода
<code>void writeFloat(float f)</code>	Записывает float-значение в поток вывода
<code>void writeInt(int i)</code>	Записывает int-значение в поток вывода
<code>void writeLong(long l)</code>	Записывает long-значение в поток вывода
<code>final void writeObject (Object obj)</code>	Записывает <code>obj</code> (т. е. объект типа Object) в поток вывода
<code>void writeShort(int i)</code>	Записывает short-значение в поток вывода

## Интерфейс `ObjectInput`

Интерфейс `ObjectInput` расширяет интерфейс `DataInput` и определяет методы, показанные в табл. 17.8. Он поддерживает сериализацию объектов. Обратите особое внимание на метод `readObject()`. Он вызывается для десериализации объектов. Все предлагаемые методы в случае аварийных ситуаций выбрасывают `IOException`-исключение.

Таблица 17.8. Методы, определенные в ObjectInput

Метод	Описание
<code>int available()</code>	Возвращает число байтов, которые теперь доступны в буфере ввода
<code>void close()</code>	Закрывает поток ввода. Дальнейшие попытки чтения генерируют исключение типа IOException
<code>int read()</code>	Возвращает целое представление следующего доступного байта ввода. Когда считывается символ конца файла, возвращается <code>-1</code>
<code>int read(byte buffer[ ])</code>	Пытается читать до <code>buffer.length</code> байтов в <code>buffer</code> , возвращая число байтов, которые были успешно считаны. Когда считывается символ конца файла, возвращается <code>-1</code>
<code>int read(byte buffer[ ], int offset, int numBytes)</code>	Пытается читать до <code>numBytes</code> байтов в массив <code>buffer</code> , начиная с позиции <code>offset</code> , возвращая число последовательно считанных байтов. Когда считывается символ конца файла, возвращается <code>-1</code>
<code>Object readObject()</code>	Читает объект из входного потока
<code>long skip(long numBytes)</code>	Игнорирует (т. е. пропускает) <code>numBytes</code> байтов в потоке ввода, возвращая число фактически пропущенных байтов

## Класс ObjectInputStream

Класс `ObjectInputStream` расширяет класс `InputStream` и реализует интерфейс `ObjectInput`. `ObjectInputStream` отвечает за чтение объектов из входного потока. Конструктор этого класса:

```
ObjectInputStream(InputStream inStream)
    throws IOException, StreamCorruptedIOException
```

Параметр `inStream` — входной поток, из которого должны считываться сериализованные объекты.

Обычно используемые методы этого класса показаны в табл. 17.9. В аварийных ситуациях они выбрасывают исключение типа `IOException`. Java 2 добавляет к `ObjectInputStream` внутренний класс с именем `GetField`. Он облегчает чтение постоянных полей, но его использование — вне рамок этой книги. Кроме того, из Java 2 был исключен и больше не должен использоваться метод `readLine()`.

Таблица 17.9. Обычно используемые методы класса *ObjectInputStream*

Метод	Описание
<code>int available()</code>	Возвращает число байтов, которые теперь доступны в буфере ввода
<code>void close()</code>	Закрывает поток ввода. Дальнейшие попытки чтения генерируют исключение типа <code>IOException</code>
<code>int read()</code>	Возвращает целое представление следующего доступного байта ввода. Когда считывается символ конца файла, возвращается <code>-1</code>
<code>int read(byte buffer[ ], int offset, int numBytes)</code>	Пытается читать до <code>numBytes</code> байтов в массив <code>buffer</code> , начиная с позиции <code>offset</code> , возвращая число последовательно считанных байтов. Когда считывается символ конца файла, возвращается <code>-1</code>
<code>boolean readBoolean()</code>	Читает и возвращает <code>boolean</code> -значение из входного потока
<code>byte readByte()</code>	Читает и возвращает <code>byte</code> -значение из входного потока
<code>char readChar()</code>	Читает и возвращает <code>char</code> -значение из входного потока
<code>double readDouble()</code>	Читает и возвращает <code>double</code> -значение из входного потока
<code>float readFloat()</code>	Читает и возвращает <code>float</code> -значение из входного потока
<code>void readFully(byte buffer[ ])</code>	Читает <code>buffer.length</code> байт в <code>buffer</code> . Возвращает управление только тогда, когда все байты были считаны
<code>void readFully(byte buffer[ ], int offset, int numBytes)</code>	Читает <code>numBytes</code> байтов в массив <code>buffer</code> , начиная с позиции <code>offset</code> . Возвращает управление только тогда, когда считались все <code>numBytes</code> байтов
<code>int readInt()</code>	Читает и возвращает <code>int</code> -значение из входного потока
<code>long readLong()</code>	Читает и возвращает <code>long</code> -значение из входного потока
<code>final Object readObject()</code>	Читает и возвращает объект из входного потока

Таблица 17.9 (окончание)

Метод	Описание
<code>short readShort()</code>	Читает и возвращает short-значение из входного потока
<code>int readUnsignedByte()</code>	Читает и возвращает byte без знака из потока вызова
<code>int readUnsignedShort()</code>	Читает и возвращает short-значение без знака из входного потока

## Пример с сериализацией

В следующей программе используются сериализация и десериализация объектов. Она начинается с создания объекта класса `MyClass` (`object1`). Этот объект имеет три экземплярные переменные типа `String`, `int` и `double`. Эту информацию мы хотим сохранить и восстановить.

Создается `FileOutputStream`-объект (`fos`), который ссылается на файл, названный "serial", и `ObjectOutputStream`-объект (`oos`) для этого файлового потока. Затем для сериализации объекта `object1` вызывается метод `WriteObject()` класса `ObjectOutputStream`. Объектный выходной поток сбрасывается и закрывается.

Затем создается `FileInputStream`-объект (`fis`), который ссылается на файл, так же названный "serial", и `ObjectInputStream`-объект (`ois`) для этого файлового потока. Затем для десериализации нашего объекта используется метод `ReadObject()` класса `ObjectInputStream` и объектный входной поток закрывается.

Обратите внимание, что `MyClass` определен, как расширение интерфейса `Serializable`. Если этого не сделать, то будет выброшено исключение типа `NotSerializableException`. Попробуйте поэкспериментировать с этой программой, объявляя некоторые из экземплярных переменных класса `MyClass`, со спецификатором `transient`. Такие данные не сохраняются во время инициализации.

```
import java.io.*;

public class SerializationDemo {
    public static void main(String args[]) {
        // сериализация объекта
        try {
            MyClass object1 = new MyClass("Hello", -7, 2.7e10);
            ...
        }
    }
}
```

```
System.out.println("object1: " + object1);
FileOutputStream fos = new FileOutputStream("serial");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(object1);
oos.flush();
oos.close();
}
catch(Exception e) {
    System.out.println("Исключение во время сериализации: " + e);
    System.exit(0);
}

// десериализация объекта
try {
    MyClass object2;
    FileInputStream fis = new FileInputStream("serial");
    ObjectInputStream ois = new ObjectInputStream(fis);
    object2 = (MyClass)ois.readObject();
    ois.close();
    System.out.println("object2: " + object2);
}
catch(Exception e) {
    System.out.println("Исключение во время десериализации: " + e);
    System.exit(0);
}
}

class MyClass implements Serializable {
String s;
int i;
double d;
public MyClass(String s, int i, double d) {
    this.s = s;
    this.i = i;
    this.d = d;
}
public String toString() {
    return "s=" + s + "; i=" + i + "; d=" + d;
}
}
```

Эта программа демонстрирует, что экземплярные переменные объектов `object1` и `object2` идентичны. Вывод этой программы:

```
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10
```

## Преимущества потоков

Поточный интерфейс ввода/вывода в Java обеспечивает четкую абстракцию для сложных и часто громоздких задач. Композиция фильтрованных поточных классов позволяет динамически формировать заказной поточный интерфейс, удовлетворяющий ваши требования к передаче данных. Программы, правильно использующие абстрактные высокогорневые классы `InputStream`, `OutputStream`, `Reader` и `Writer`, будут функционировать должным образом и в будущем, даже когда будут изобретены новые и улучшенные конкретные поточные классы. Как вы увидите в следующей главе, эта модель очень хорошо работает при переключении с набора потоков, основанных на файловой системе, к сетевым потокам, основанным на сокетах. Наконец ожидается, что в будущем все более и более важную роль в Java-программировании будет играть сериализация объектов. Java-классы сериализованного ввода/вывода обеспечивают мобильное (переносимое) решение этой довольно сложной задачи.



## ГЛАВА 18

# Работа в сети

Данная глава использует описание пакета `java.net`, который обеспечивает поддержку работы в сети. Создатели назвали его пакетом "для программирования в Internet". В самом языке программирования Java очень мало возможностей для записи программ, работающих в сети (так же, как, скажем, и в языках C++ или ФОРТРАН). Что действительно делает Java хорошим языком для работы в сети, так это классы, определенные в пакете `java.net`.

Эти *сетевые* классы инкапсулируют парадигму *сокета*, разработанную в Berkeley Software Distribution (BSD) из Университета Калифорнии в Berkeley. Никакое обсуждение работающих в сети Internet библиотек не было бы полным без упоминания истории UNIX и BSD-сокетов.

## Основы работы в сети

Кен Томпсон (Ken Thompson) и Дэннис Ритчи (Dennis Ritchie) разработали операционную систему UNIX совместно с языком C в Bell Telephone Laboratories, Murray Hill, New Jersey, в 1969 г. В течение многих лет развитие UNIX выполнялось в Лабораториях Бэлла, нескольких университетах и научно-исследовательских учреждениях, имевших PDP-машины фирмы DEC, для работы на которых UNIX и была разработана. В 1978 г. Билл Джой (Bill Joy) вел проект в калифорнийском Университете с целью добавления ряда новых свойств к UNIX, таких как виртуальная память и полноэкранные возможности дисплея. К началу 1984 г., как раз, когда Билл покинул университет и основал фирму Sun Microsystems, он выпустил систему 4.2BSD, известную как Berkeley UNIX.

Система 4.2BSD была выпущена с быстрой файловой системой, надежными сигналами, межпроцессорной связью и, что наиболее важно, работающую в сети. Поддержка работы в сети, впервые найденная в версии 4.2, в конечном счете, стала фактическим стандартом для Internet. Berkeley-реализация протоколов TCP/IP остается исходным эталоном для связи в Internet. Парадиг-

ма сокета для межпроцессорной и сетевой связи также была широко принята вне Berkeley. Даже Windows и Macintosh в конце 80-х начали говорить "Berkeley-сокеты".

## Обзор сокетов

*Сетевой сокет* (network socket) очень похож на электрический соединитель (разъем). Различные сетевые соединители обеспечивают стандартные пути поставки полезной нагрузки. Все, что понимает стандартный протокол, может "подключаться" к сокету и вступать в связь. Для электрических соединителей не имеет значения, подключаете ли вы лампочку или тостер. До тех пор пока они обеспечивают подачу электричества 60 Гц, 115 В, устройства будут работать. Подумайте, как создается ваш счет за электричество. Существует измеритель где-то между вашим домом и остальной частью сети. За каждый киловатт мощности, которая проходит через этот измеритель, вам приходит счет. Счет поступает в ваш "адрес". Хотя электричество "течет" свободно по силовой сети, все разъемы в вашем доме имеют специфический адрес.

Та же самая идея применяется и к сетевым разъемам — сокетам, за исключением того, что мы говорим о протоколах TCP/IP и IP-адресах, а не об электронах и уличных адресах. *IP* (Internet Protocol, Протокол Интернета) — это протокол маршрутизации нижнего уровня, который разделяет данные на небольшие пакеты и посыпает их по различным адресам через сеть, не гарантирующую доставку отправленных пакетов пункту назначения. *TCP* (Transmission Control Protocol, Протокол управления передачей) — это протокол более высокого уровня, который умеет прочно соединять вместе пакеты, сортируя и ретранслируя их по мере необходимости для надежной передачи данных. Третий протокол — *UDP* (User Datagram Protocol, Дейтаграммный протокол пользователя) — следует за TCP и может применяться непосредственно для поддержки быстрой, без установки соединения, но, правда, ненадежной транспортировки пакетов.

## Клиент-сервер

Вы часто слышите термин *клиент-сервер*, упоминаемый в контексте работы в сети. Он кажется сложным, когда вы читаете о нем в маркетинговых инструкциях корпораций, но в действительности он весьма прост. *Сервер* — это все, что имеет некоторый разделяемый (коллективно используемый) ресурс. Существуют *вычислительные серверы*, которые обеспечивают вычислительную мощность; *серверы печати*, которые управляют совокупностью принтеров; *дисковые серверы*, которые предоставляют работающее в сети дисковое пространство, и *Web-серверы*, которые хранят Web-страницы. *Клиент* — просто любой другой объект, который хочет получить доступ к специальному серверу. Взаимодействие между клиентом и сервером точно такое же, как взаимодействие между лампочкой и электрическим разъемом. Силовая

сеть дома — сервер, а лампочка — клиент этой сети. Сервер — это постоянно доступный ресурс, в то время как клиент может "отключиться" после того, как он был обслужен.

В Berkeley-сокетах понятие сокета позволяет отдельному компьютеру обслуживать много различных клиентов сразу, так же, как и обслуживать множество различных типов информации. Для управления таким обслуживанием вводится понятие *порта*, который является пронумерованным сокетом на конкретной машине. Говорят, что процесс сервера "слушает" порт, пока клиент не соединится с ним. Серверу позволяют принять много клиентов, присоединенных к одному и тому же номеру порта, хотя каждый сеанс уникален. Чтобы управлять множественными подключениями клиента, процесс сервера должен быть многопоточным или иметь некоторые другие средства мультиплексирования одновременного ввода/вывода.

## Зарезервированные сокеты

После соединения протокол высокого уровня гарантирует жесткую зависимость соединения от того порта, который вы используете. TCP/IP резервирует 1024 нижних порта для определенных протоколов. Порт с номером 21 — для протокола FTP, 23 — для протокола Telnet, 25 — для протокола электронной почты (e-mail), 79 — для протокола Finger; 80 — для протокола HTTP, 119 — для протокола телеконференций и т. д. В задачу каждого протокола входит поддержка определенного способа взаимодействия с портом.

Например, HTTP (HyperText Transfer Protocol, Протокол передачи гипертекста) — это протокол, который Web-браузеры и серверы используют для пересылки страниц гипертекста и изображений. Это совсем простой протокол для основного Web-сервера, просматривающего страницы. Вот, как он работает. Когда клиент запрашивает файл с HTTP-сервера (действие, известное как "стук" (hit)<sup>1</sup>), он просто печатает имя файла в специальном формате к предопределенному порту и считывает содержимое файла. Сервер отвечает также кодовым числом состояния, чтобы сообщить клиенту, может ли запрос быть выполнен и почему. Пример взаимосвязанных процедур клиента, требующего отдельный файл (с именем index.html), и сервера, отвечающего, что он успешно нашел файл и посыпает его клиенту:

Сервер	Клиент
Слушает порт 80	Соединяется с портом 80
Принимает соединение	Записывает "GET /index.html HTTP/1.0\r\n\r"

<sup>1</sup> Аналог русского "стук в дверь", например. — Примеч. пер.

(окончание)

Сервер	Клиент
Читает вплоть до второго символа конца строки (\n)	
Видит, что GET — известная команда и что HTTP/1.0 является правильной версией протокола	
Читает локальный файл с именем /index.html	
Записывает "HTTP/1.0 200 OK\n\n"	"200" означает "Прибывает файл"
Копирует содержание файла в сокет	Читает и отображает содержание файла
Отбой	Отбой

Очевидно, протокол HTTP намного сложнее, чем приведенный пример, но это действительная транзакция<sup>1</sup>, которую вы могли бы иметь с любым соседним Web-сервером.

## Proxy-серверы

*Proxy-сервер*<sup>2</sup> объясняет клиентскую сторону протокола другому серверу. Это требуется, когда клиенты имеют некоторые ограничения на то, с какими серверами они могут соединяться. Таким образом, клиент соединяется с proxy-сервером, который не имеет таких ограничений, а тот, в свою очередь, осуществляет связь для клиента. Proxy сервер имеет дополнительные возможности фильтровать некоторые запросы или кэшировать результаты этих запросов для будущего использования. Кэширующий proxy HTTP-сервер может помочь уменьшить требования к полосе пропускания на соединении локальной сети с Internet. Когда к популярному Web-сайту обращаются сотни пользователей, proxy-сервер может один раз получить содержимое популярных страниц Web-сервера, экономя дорогие межсетевые передачи и обеспечивая клиентам более быстрый доступ к этим страницам.

Далее в настоящей главе, мы будем фактически строить полный кэширующий proxy-сервер HTTP. Интересная сторона этой программы заключается в том, что она является как клиентом, так и сервером. Чтобы обслуживать не-

<sup>1</sup> В данном контексте термин "транзакция" означает *групповое действие*. — Примеч. пер.

<sup>2</sup> *Proxy-сервер* — это специальная служебная сетевая программа, обычно работающая на компьютере провайдера, предназначенная для нескольких целей: ускорения работы клиента (например, считывания HTML-страниц), защиты клиента от вирусов, ограничения доступа клиента к определенным протоколам, серверам и т. п. — Примеч. пер.

которые страницы, она должна действовать как клиент других серверов, стремящийся получить копию требуемого содержимого.

## Адресация Internet

Любой компьютер в Internet имеет *адрес*. Адрес Internet — это число, которое уникально идентифицирует каждый компьютер в сети. В IP-адресах 32 бита, и мы используем их как последовательность из четырех чисел между 0 и 255, разделенных точками. Это делает их проще для запоминания, потому что они назначены не случайно, а иерархически. Первые несколько бит адреса представляют класс сети (классы обозначаются буквами A, B, C, D или E). Большинство пользователей Internet находится в сетях класса C. Класс C содержит более двух миллионов сетей. Первая часть адреса сети класса C находится между 192 и 224, а последняя — идентифицирует индивидуальный компьютер. (В одиночной сети класса C допустимо до 256 компьютеров.) Эта схема позволяет существовать на сетях класса C примерно полумилиарду устройств.

## Служба доменных имен (DNS)

Сеть Internet не была бы очень удобным местом для навигации, если бы каждый должен был использовать *числовую* адресацию. Трудно, например, вообразить рекламный адрес в виде "http://192.9.9.1/". К счастью, для параллельной иерархии имен существует иной — *символьный* способ адресации. В сетевой адресации широко используется служба *доменных имен* (DNS, Domain Naming Service). Подобно тому, как четыре числа IP-адреса описывают сетевую иерархию слева направо, символьный Internet-адрес, называемый *доменным адресом*, описывает размещение машины в пространстве имен справа налево. Например, адрес www.starwave.com находится в домене com<sup>1</sup> (зарезервированном для американских коммерческих сайтов), сайт называется starwave (по имени компании), а www — имя специфичного компьютера, который является Web-сервером starwave. www соответствует самому правому номеру в эквивалентном IP-адресе.

## Java и сеть

Рассмотрим, как Java относится ко всем этим сетевым концепциям. Java поддерживает протоколы TCP/IP как путем расширения уже установленного интерфейса поточного ввода/вывода, представленного в главе 17, так и добавляя свойства, необходимые для построения сетевых объектов ввода/вывода. Java поддерживает два семейства протоколов — TCP и UDP. TCP-про-

<sup>1</sup> Крайнее правое слово в доменном адресе (в нашем случае — com) называется именем корневого домена. — Примеч. ред.

токолы используются для надежного поточного ввода/вывода через сеть. UDP-протоколы поддерживают более простую и, следовательно, более быстрой двухточечную модель, ориентированную на дейтаграммы<sup>1</sup>.

## Сетевые классы и интерфейсы

Классы, содержащиеся в пакете `java.net`:

- Authenticator (Java 2)
- ContentHandler
- DatagramPacket
- DatagramSocket
- DatagramSocketImpl
- HttpURLConnection
- InetAddress
- JarURLConnection (Java 2)
- MulticastSocket
- NetPermission
- PasswordAuthentication (Java 2)
- Server-Socket
- Socket
- SocketImpl
- SocketPermission
- URL
- URLClassLoader (Java 2)
- URLConnection
- URLDecoder (Java 2)
- URLEncoder
- URLStreamHandler

Интерфейсы пакета `java.net`:

- ContentHandlerFactory
- FileNameMap
- SocketImplFactory
- SocketOptions
- URLStreamHandlerFactory

В следующих разделах мы рассмотрим главные сетевые классы и покажем несколько примеров, в которых они применяются.

## Класс `InetAddress`

Звоните ли вы по телефону, посыпаете ли почту или устанавливаете соединение через Internet, во всех этих процессах фундаментальное значение имеют адреса. Класс `InetAddress` используется, чтобы инкапсулировать как числовой IP-адрес, который мы обсуждали ранее, так и доменный адрес. Вы взаимодействуете с этим классом, используя доменный адрес хост<sup>2</sup>-компьютера, который более удобен и понятен, чем его числовой эквивалент. Этот числовой адрес скрыт внутри класса `InetAddress`.

<sup>1</sup> Дейтаграмма — пакет в сети передачи данных, передаваемый через сеть независимо от других пакетов без установки логического соединения и квитирования. — Примеч. ред.

<sup>2</sup> Хост (от англ. host) — любое устройство, подключенное к сети и использующее протоколы TCP/IP. — Примеч. пер.

## Производственные методы

Класс `InetAddress` не имеет видимых конструкторов. Для создания `InetAddress` объекта нужно использовать один из доступных *производственных* методов. *Производственные методы* (*factory methods*) — просто соглашение, с помощью которого статические методы в классе возвращают экземпляр данного класса. Это сделано вместо перегрузки конструктора различными списками параметров, когда наличие уникальных имен методов приводит к более ясным результатам. Для создания экземпляров типа `InetAddress` можно использовать три метода — `getLocalHost()`, `getByName()` и `getAllByName()` с форматами:

```
static InetAddress getLocalHost()
    throws UnknownHostException

static InetAddress getByName(String hostName)
    throws UnknownHostException

static InetAddress[] getAllByName(String hostName)
    throws UnknownHostException
```

Метод `getLocalHost()` просто возвращает объект `InetAddress`, который представляет локальный хост-компьютер. Метод `getByName()` возвращает объект типа (класса) `InetAddress` для компьютера, имя которого передается ему в параметре `hostName`. Если эти методы не способны распознать имя компьютера, они выбрасывают исключение типа `UnknownHostException`.

Для Internet является обычным использование одиночного имени для представления нескольких машин. В мире Web-серверов это один из способов обеспечить некоторую степень масштабирования сети. Производственный метод `getAllByName()` возвращает массив `InetAddress` объектов, представляющий все адреса, связанные с указанным именем. Он также выбрасывает исключение `UnknownHostException`, если не может построить, по крайней мере, один адресный объект для указанного имени компьютера.

Следующий пример печатает адреса и имена локальной машины двух хорошо известных Web-сайтов:

```
// Демонстрирует InetAddress.
import java.net.*;

class InetAddressTest
{
    public static void main(String args[]) throws UnknownHostException {
        InetAddress Address = InetAddress.getLocalHost();
        System.out.println(Address);
        Address = InetAddress.getByName("starwave.com");
        System.out.println(Address);
        InetAddress SW[] = InetAddress.getAllByName("www.nba.com");
        for (int i=0; i<SW.length; i++)
```

```

        System.out.println(SW[i]);
    }
}
}

```

Вывод, выполненный этой программой (ваш вывод будет, конечно, немного другим):

```

default/206.148.209.138
starwave.com/204.202.129.90
www.nba.com/204.202.130.223

```

## Методы экземпляра

Класс InetAddress имеет также несколько нестатических методов, которые можно применять к объектам, возвращаемым только что обсужденными методами (табл. 18.1).

**Таблица 18.1. Нестатические методы класса InetAddress**

Метод	Описание
<code>boolean equals(Object other)</code>	Возвращает true, если возвращаемый объект имеет тот же Internet-адрес, что и объект, полученный через параметр <code>other</code>
<code>byte[ ] getAddress()</code>	Возвращает четырехэлементный массив байтов, который представляет IP-адрес обрабатываемого объекта
<code>String getHostAddress()</code>	Возвращает строку, которая представляет адрес хост-компьютера, связанного с объектом класса InetAddress
<code>String getHostName()</code>	Возвращает строку, которая представляет имя хост-компьютера, связанного с InetAddress-объектом
<code>int hashCode()</code>	Возвращает хэш-код вызывающего объекта
<code>boolean isMulticastAddress()</code>	Возвращает true, если IP-адрес объекта этого класса — групповой (multicast). Иначе, возвращает false
<code>String toString()</code>	Возвращает строку, которая перечисляет доменный и IP-адрес хост-компьютера (например, "starwave.com/192.147.170.6")

IP-адреса отыскиваются в ряде иерархически кэшированных серверов. Это означает, что ваш локальный компьютер может знать как свой собственный

адрес, так и адреса близлежащих серверов в автоматически отображаемом смешанном формате "доменный адрес/IP-адрес". Для других имен (т. е. имен, не входящих в указанную группу) информацию об IP-адресах ваш компьютер может запрашивать у локального сервера DNS. Если данный сервер не имеет специфического адреса, он может обратиться к удаленному сайту и запросить у него этот адрес. Такая последовательность запроса адреса может продолжаться по всему пути до корневого сервера, называемого InterNIC-сервером (его DNS адрес — internic.net). Этот процесс может занять много времени, так что разумнее структурировать ваш код так, чтобы хэшировать информацию IP-адресов локально, а не отыскивать ее повторно.

## Сокеты TCP/IP клиентов

*TCP/IP сокеты* используются для того, чтобы осуществить надежные, двунаправленные, постоянные, двухточечные и поточные соединения между хост-компьютерами в Internet. Такие сокеты можно использовать для подключения системы ввода/вывода Java к другим программам, которые могут постоянно находиться на локальной машине или на любой другой машине в Internet.

### Замечание

Апплеты могут устанавливать сокет-соединения только с хост-машиной, с которой апплет был загружен. Это ограничение существует потому, что было бы опасно для апплетов, загруженных через межсетевую защиту, иметь доступ к любой произвольной машине.

В Java имеются два вида TCP-сокетов: один — для серверов, а другой — для клиентов. Класс *ServerSocket* разработан так, чтобы быть "слушателем", который перед выполнением любой операции ожидает соединения с клиентами. Класс *Socket* разработан так, чтобы соединяться с серверными сокетами и инициализировать протокольные обмены.

Создание *Socket*-объекта неявно устанавливает соединение между клиентом и сервером. Нет никаких методов или конструкторов, которые явно демонстрируют подробности установки этого соединения. В табл. 18.2 описаны два конструктора, которые используются для создания клиентских сокетов.

**Таблица 18.2. Конструкторы для создания *Socket*-объектов**

Синтаксис конструктора	Описание
<code>Socket(String hostName, int port)</code>	Создает сокет, соединяющий локальную хост-машину с именованной хост-машиной и портом; может выбрасывать исключение <code>UnknownHostException</code> или <code>IOException</code>

Таблица 18.2 (окончание)

Синтаксис конструктора	Описание
<code>Socket(InetAddress ipAddress, int port)</code>	Создает сокет, аналогичный предыдущему, но используется уже существующий объект класса InetAddress и порт; может выбрасывать исключение IOException

Сокет может в любое время просматривать связанную с ним адресную и портовую информацию при помощи методов, представленных в табл. 18.3.

Таблица 18.3. Методы просмотра адресной и портовой информации

Метод	Описание
<code>InetAddress getInetAddress()</code>	Возвращает InetAddress-объект, связанный с Socket-объектом
<code>Int getPort()</code>	Возвращает удаленный порт, с которым соединен данный Socket-объект
<code>int getLocalPort()</code>	Возвращает локальный порт, с которым соединен данный Socket-объект

После создания Socket-объект можно также применять для получения доступа к связанным с ним потокам ввода/вывода. Данные потоки используются точно так же, как потоки ввода/вывода, описанные в главе 17 (для отправки и приема данных). Если сокеты становятся несостоятельными от потери соединения с сетью, то каждый из этих методов может выбрасывать исключение типа IOException (табл. 18.4).

Таблица 18.4. Исключения потоков ввода/вывода

Исключение	Описание
<code>InputStream getInputStream()</code>	Возвращает InputStream-объект (объект входного потока), связанный с активным сокетом
<code>OutputStream getOutputStream()</code>	Возвращает OutputStream-объект (объект выходного потока), связанный с активным сокетом
<code>void close()</code>	Закрывает объекты InputStream и OutputStream (т. е. соответствующие потоки)

## Пример работы с сокет-соединением (программа Whois)

В следующем очень простом примере сначала открывается соединение с портом "whois" на сервере InterNIC, далее этому серверу пересыпается аргумент командной строки и затем печатаются данные, которые он возвращает. Сервер InterNIC пытается найти аргумент в форме зарегистрированного (в Internet) доменного адреса и затем посыпает назад IP-адрес и контактную информацию своего сайта.

```
// Демонстрирует сокеты.
import java.net.*;
import java.io.*;

class Whois {
    public static void main(String args[]) throws Exception {
        int c;
        Socket s = new Socket("internic.net", 43);
        InputStream in = s.getInputStream();
        OutputStream out = s.getOutputStream();
        String str = (args.length == 0 ? "starwave-dom" : args[0]) + "\n";
        byte buf[] = str.getBytes();
        out.write(buf);
        while ((c = in.read()) != -1) {
            System.out.print((char) c);
        }
        s.close();
    }
}
```

Если бы, например, вы ввели в командную строку (запускающую эту программу на выполнение) аргумент sportszone.com, то получили бы нечто похожее на следующий листинг:

```
Registrant:
Star-wave Corporation (SPORTS ZONE-DOM)
13810 SE Eastgate Way
Bellevue/ WA 98005
US
```

Domain Name: SPORTSZONE.COM

```
Administrative Contact, Technical Contact, Zone Contact:
Domain Administrator (DA4894-ORG) dns-admin@STARWAVE.COM 425.957.2000
Fax- 425.957.2009
```

Record last updated on 19-Feb-98.

Database last updated on 28-Jan-99 04:31:51 EST.

Domain servers in listed order:

DNS1.STARWAVE.COM	204.202.132.51
DNS3.NWNET.NET	192.220.250.7
DNS4.NWNET.NET	192.220.251.7

The InterNIC Registration Services database contains ONLY non-military and non-US Government Domains and contacts.

Other associated whois servers:

American Registry for Internet Numbers	- whois.arin.net
European IP Address Allocations	- whois.ripe.net
Asia Pacific IP Address Allocations	- whois.apnic.net
US Military	- whois.nic.mil
US Government	- whois.nic.gov

## Использование URL

Последний пример был не очень понятным потому, что современный Internet не ориентирован на старые протоколы, подобные whois, finger и FTP. Он ориентирован на протоколы системы World Wide Web (WWW, Все-мирная Паутина). *Web-протоколы* — это свободная (не связанная) совокупность протоколов высокого уровня и файловых форматов, объединенная в Web-браузере (программе навигации по сети). Один из наиболее важных аспектов этой совокупности заключается в том, что Тим Бернерс-Ли (Tim Berners-Lee) изобрел масштабируемый способ размещения всех ресурсов сети. Возможность надежно именовать все что угодно и где угодно, становится очень мощной парадигмой (общим принципом). Данный принцип воплощен через универсальный локатор ресурсов (URL, Uniform Resource Locator).

URL обеспечивает разумную, понятную форму уникальной идентификации адресной информации в Internet-сетях. URL-адреса вездесущи. Каждый браузер использует их, чтобы идентифицировать Web-информацию. На самом деле Web-система — это то же самое, что старый Internet со всеми его адресованными (с помощью URL) ресурсами плюс HTML-файлы. Внутри сетевой библиотеки классов java.net имеется класс URL, который обеспечивает простой и краткий API-интерфейс для доступа к Internet-информации с использованием URL-адресов.

## Формат

Два примера URL-адресов: <http://www.starwave.com/> и <http://www.starwave.com:80/index.html>. Спецификация URL основана на четырех компонентах. Первый — используемый протокол, отделенный от остальной части локатора двоеточием (:). Общими протоколами являются http, ftp, gopher и file, хотя

на сегодня почти все делается через http (фактически, большинство браузеров продолжит правильно работать, если вы уберете "http://" из URL-спецификации). Второй компонент — имя (или IP-адрес) используемого хост-компьютера, ограничен слева двойным слэшем (//) и справа одинарным слэшем (/) или (не обязательно) двоеточием (:). Третий компонент — номер порта, является необязательным параметром, разграниченным слева (от имени хост-компьютера) двоеточием (:), а справа — слэшем (/). (По умолчанию принимается порт 80, предопределенный порт HTTP. Таким образом, запись ":80" избыточна.) Четвертый компонент — фактический путь файла. Большинство HTTP-серверов добавляет к URL-адресам, которые обращаются прямо к ресурсу каталога, файл с именем index.html. Итак, `http://www.starwave.com/` — то же самое, что `http://www.starwave.com/index.html`.

Java-класс URL имеет несколько конструкторов, каждый из которых может выбрасывать исключение `MalformedURLException`. Обычно используемая форма конструктора определяет URL, который идентичен тому, что отображается в раскрывающемся списке **Адрес** окна браузера:

`URL(String urlSpecifier)`

Следующие две формы конструктора позволяют разбивать URL на его компонентные элементы:

`URL(String protocolName, String hostName, int port, String path)`

`URL(String protocolName, String hostName, String path)`

Другой часто используемый конструктор позволяет применять существующий URL как ссылочный контекст и затем создавать новый URL из того контекста. Хотя это звучит немного непонятно, но в действительности совсем просто и полезно.

`URL(URL urlObj, String urlSpecifier)`

В следующем примере мы создадим URL к домашней странице Патрика Ноутона в хост-компьютере Starwave и затем просмотрим его свойства:

```
// Демонстрирует URL.
import java.net.*;
class patrickURL {
    public static void main(String args[]) throws MalformedURLException {
        URL hp = new URL("http://www.starwave.com/people/naughton/");
        System.out.println("Протокол: " + hp.getProtocol());
        System.out.println("Порт: " + hp.getPort());
        System.out.println("Хост: " + hp.getHost());
        System.out.println("Файл: " + hp.getFile());
        System.out.println("URL: " + hp.toExternalForm());
    }
}
```

Когда вы ее выполните, то получите следующий вывод:

```
Протокол: http
Порт: -1
Хост: www.starwave.com
Файл: /people/naughton/
URL: http://www.starwave.com/people/naughton/
```

Обратите внимание, что порт имеет номер `-1`. Это означает, что он не был явно установлен. Теперь, когда мы создали `URL`-объект, мы хотим отыскать и вернуть связанные с ним данные. Чтобы получить доступ к фактическим битам или информации о содержимом класса `URL`, нужно создать объект `URLConnection` и вызвать его метод `openConnection()`, примерно так:

```
url.openConnection()
```

Метод `openConnection()` имеет следующую общую форму:

```
URLConnection openConnection()
```

Он возвращает объект `URLConnection`, связанный с вызывающим `URL`-объектом, и может выбрасывать исключение типа `IOException`.

## Класс `URLConnection`

`URLConnection` — класс общего назначения для доступа к атрибутам удаленного ресурса. Выполнив соединение с удаленным сервером, можно использовать `URLConnection` для просмотра свойства удаленного объекта перед фактической его транспортировкой в локальную программу. Эти атрибуты определяются спецификацией HTTP-протокола и, как таковые, имеют смысл только для `URL`-объектов, которые используют данный протокол. Здесь мы рассмотрим наиболее полезные элементы `URLConnection`.

В следующем примере мы создаем класс, используя метод `openConnection()` объекта `URL`, и затем используем его для просмотра свойств и содержимого документов:

```
// Демонстрирует URLConnection.
import java.net.*;
import java.io.*;
import java.util.Date;

class JCDemo
{
    public static void main(String args[]) throws Exception {
        int c;
        URL hp = new URL("http://www.starwave.com/people/naughton/");
        URLConnection hpCon = hp.openConnection();
```

```
System.out.println("Дата: " + new Date(hpCon.getDate()));
System.out.println("Тип содержимого: " + hpCon.getContentType());
System.out.println("Срок хранения: " + hpCon.getExpiration());
System.out.println("Последнее изменение: " +
    new Date(hpCon.getLastModified()));
int len = hpCon.getContentLength();
System.out.println("Content-Length: " + len);
if (len > 0) {
    System.out.println("==== Content ===");
    InputStream input = hpCon.getInputStream();
    int i = len;
    while (((c = input.read()) != -1) && (--i > 0)) {
        System.out.print((char) c);
    }
    input.close();
} else {
    System.out.println("Нет содержимого");
}
}
```

Программа устанавливает HTTP-соединение с [www.starwave.com](http://www.starwave.com) через порт 80 и запрашивает документ `/people/naughton/`. Затем выводится список значений заголовка и извлекается содержимое. Первые несколько строк вывода:

Дата: Fri Jan 29 16:32:41 CST 1999  
Тип содержимого: text/html  
Срок хранения: 0  
Последнее изменение: Wed Jan 20 18:37:54 CST 1999  
Content-Length: 275  
= = = Content = = =  
<body onload=doRedirect()>  
<script language="JavaScript">  
<!--  
function doRedirect() {  
 location="http://homepages.go.com/~pjn"  
}

Классы `URL` и `URLConnection` удобны для простых программ, которые хотят присоединяться к HTTP-серверам, чтобы извлекать их содержимое. Для более сложных приложений лучше прекратить изучение спецификаций HTTP-протокола и реализовать собственные упаковщики.

## **Сокеты TCP/IP серверов**

Java предоставляет другой сокет-класс для создания серверных приложений. Класс `ServerSocket` используется для создания серверов, которые прослушивают

вают либо локальные, либо удаленные программы клиента, чтобы соединяться с ними на опубликованных портах. Так как Web-система очень активно используется в сетях Internet, то в этом разделе показана разработка операционного Web-сервера (HTTP-сервера).

Класс `ServerSocket` сильно отличается от нормального класса `Socket`. Когда вы создаете объект класса `ServerSocket`, он регистрирует себя в системе, как имеющий интерес к соединениям клиента. Конструкторы `ServerSocket` указывают номер порта, на который вы хотите принимать соединения, и (по желанию) какой длины нужна очередь для указанного порта. Длина очереди сообщает системе, сколько подключений клиента она может оставлять задержанными прежде, чем должна просто отвергнуть соединения. Значение по умолчанию — 50. При неблагоприятных условиях конструкторы могут выбрасывать исключение типа `IOException`. В табл. 18.5 приводятся сигнатуры<sup>1</sup> и краткие описания конструкторов.

**Таблица 18.5. Конструкторы класса `ServerSocket`**

Конструктор	Описание
<code>ServerSocket(int port)</code>	Создает сокет сервера на указанном порте с длиной очереди по умолчанию (50)
<code>ServerSocket(int port, int maxQueue)</code>	Создает сокет сервера на указанном порте с максимальной длиной очереди <code>maxQueue</code>
<code>ServerSocket(int port, int maxQueue, InetAddress localAddress)</code>	Создает сокет сервера на указанном порте с максимальной длиной очереди <code>maxQueue</code> . На групповом <sup>2</sup> хост-компьютере <code>localAddress</code> определяет IP-адрес, с которым этот сокет связан

Класс `ServerSocket` имеет дополнительный метод (`accept()`), который является блокирующим вызовом: он будет ждать клиента, чтобы инициализировать связь, и затем вернет нормальный `Socket`-объект, который будет использоваться для связи с клиентом.

## Кэширующий proxy HTTP-сервер

В конце этого раздела показана разработка простого кэширующего proxy HTTP-сервера, названного `http`, и демонстрирующего сокеты клиента и сер-

<sup>1</sup> Сигнатура — сочетание имени метода, его параметров и ключевых слов, определяющих тип возвращаемого значения и спецификаторы доступа (в формате заголовка определителя метода). — Примеч. пер.

<sup>2</sup> Групповой хост — компьютер, который имеет несколько сетевых адаптеров или сконфигурирован с несколькими IP-адресами для одиночного сетевого адаптера. — Примеч. пер.

вера. Сервер http поддерживает только операции GET и очень ограниченный диапазон жестко закодированных MIME-типов. *MIME-типы* — описатели типов мультимедийного содержимого в документах электронной почты. Proxy HTTP-сервер — это однопоточная программная система, в которой каждый запрос обрабатывается по очереди, в то время как все другие ждут обработки. В нем реализованы довольно наивные стратегии кэширования — он сохраняет все и навсегда в оперативной памяти. Действуя как proxy-сервер, http также копирует каждый принимаемый файл в локальный кэш, для которого он не имеет никакой стратегии для регенерации или сборки мусора. Если оставить все эти рассуждения в стороне, то http — это продуктивный пример сокетов клиента и сервера, представляющий определенный интерес для эксплуатации и легко расширяемый.

## Исходный код

Реализация этого HTTP-сервера представлена пятью классами и одним интерфейсом. Более полная реализация, вероятно, разделила бы на части многие методы главного класса (`httpd`), чтобы абстрагировать в них больше компонентов. Из-за ограниченного объема этой книги, большинство функциональных возможностей реализовано в одном классе, а небольшие поддерживающие классы действуют только как структуры данных. Чтобы понять, как работает этот сервер, рассмотрим каждый класс и метод подробнее (начиная с классов поддержки и заканчивая главной программой).

### MimeHeader.java

*MIME* — стандарт Internet для передачи мультимедийного содержимого по системам электронной почты. Этот стандарт был создан Натом Боренстайном (Nat Borenstein) в 1992 году. Протокол использует и расширяет нотацию MIME-заголовков, чтобы передать общие пары атрибут/значение между HTTP-клиентом и сервером.

**Конструкторы.** Класс `MimeHeader` является подклассом `Hashtable`, так что в нем удобно хранить и отыскивать пары ключ/значение, связанные с MIME-заголовком. Он имеет два конструктора. Один создает пустой `MimeHeader`-объект без ключей. Другой берет строку, отформатированную как MIME-заголовок, и анализирует ее, чтобы определить начальное содержимое объекта (см. следующий подраздел).

**Метод `parse()`.** Метод `parse()` получает (через свой параметр) строку MIME-формата, выполняет ее синтаксический анализ, извлекает пары ключ/значение и вводит их в заданный экземпляр класса `MimeHeader`. Чтобы разбить входные данные на индивидуальные строки, завершающиеся последовательностью CRLF (`\r\n`), метод использует объект типа  `StringTokenizer`. Затем он выполняет итерации через каждую строку, применяя каноническую последовательность `while... hasMoreTokens()... nextToken()`.

Метод `parse()` разбивает каждую строку MIME-заголовка на две строки, разделенные двоеточием. Метод `substring()` извлекает символы перед двоеточием, после двоеточия и следующие за ними пробельные символы и сохраняет их в двух переменных — `key` и `val`. После этого вызывается метод `put()`, чтобы передать эту ассоциацию ключа и значения объекту `Hashtable`.

**Метод `toString()`.** Метод `toString()` (использующий оператор конкатенации строк `+`) — инверсия метода `parse()`. Он берет текущие пары ключ/значение, сохраненные в `MimeHeader`, и возвращает их строчное представление в MIME-формате, где за ключами печатаются двоеточие и пробел, а затем — значение, за которым следует последовательность CRLF.

**Методы `put()`, `get()` и `fix()`.** Методы `put()` и `get()` в `Hashtable` работали бы прекрасно для этого приложения, если бы не одна довольно странная вещь. Спецификации MIME определили несколько важных ключей, таких как `Content-Type` и `Content-Length`. Некоторые ранние реализации MIME-систем, особенно Web-браузеры, позволяли себе вольности с применением заглавных букв в этих полях. Некоторые используют `Content-type`, другие — `content-type`. Чтобы избежать неудач, наш HTTP-сервер пробует конвертировать все входящие и исходящие ключи `MimeHeader` в каноническую форму `Content-Type`. Мы освобождаем методы `put()` и `get()` от подобных преобразований, используя в качестве аргументов их вызовов метод `fix()`.

**Код.** Исходный код для `MimeHeader`:

```
import java.util.*;  
  
class MimeHeader extends Hashtable {  
    void parse(String data) {  
        StringTokenizer st = new StringTokenizer(data, "\r\n");  
  
        while (st.hasMoreTokens()) {  
            String s = st.nextToken();  
            int colon = s.indexOf(':');  
            String key = s.substring(0, colon);  
            String val = s.substring(colon + 2); // пропуск ":"  
            put(key, val);  
        }  
    }  
  
    MimeHeader() {}  
  
    MimeHeader(String d) {  
        parse(d);  
    }  
  
    public String toString() {  
        String ret = "";  
        Enumeration e = keys();
```

```

while(e.hasMoreElements()) {
    String key = (String) e.nextElement();
    String val = (String) get(key);
    ret += key + ": " + val + "\r\n";
}
return ret;
}

// Эта простая функция конвертирует MIME-строку из
// любого варианта капитализации в каноническую форму.
// Например: CONTENT-TYPE или content-type к Content-Type,
// или Content-length или Content-LENGTH к Content-Length.
private String fix(String ms) {
    char chars[] = ms.toLowerCase().toCharArray();
    boolean upcaseNext = true;

    for (int i = 0; i < chars.length - 1; i++) {
        char ch = chars[i];
        if (upcaseNext && 'a' <= ch && ch <= 'z') {
            chars[i] = (char) (ch - ('a' - 'A'));
        }
        upcaseNext = ch == '-';
    }
    return new String(chars);
}

public String get(String key) {
    return (String) super.get(fix(key));
}

public void put(String key, String val) {
    super.put(fix(key), val);
}
}
}

```

## HttpResponse.java

Класс `HttpResponse` — упаковщик всего, связанного с ответом от HTTP-сервера. Он используется прою частью класса `httpd`. Когда вы посыаете запрос на HTTP-сервер, тот отвечает целочисленным кодом состояния, который мы храним в переменной `statusCode`, и текстовым эквивалентом, который мы храним в переменной `reasonPhrase`. (Эти имена переменных взяты из формулировок в официальной спецификации HTTP.) За односторонним ответом следует MIME-заголовок, который содержит остальную информацию ответа. Для анализа данной строки мы используем рассмотренный ранее объект `MimeHeader`, хранящийся внутри класса `HttpResponse` в переменной `mh`. Все перечисленные переменные не объявлены частными (`private`) так, что класс `httpd` может использовать их напрямую.

**Конструкторы.** При создании объекта класса `HttpServletResponse` конструктор со строчным параметром получает необработанный ответ HTTP-сервера и передает его в описанный далее метод `parse()` (для инициализации объекта). Альтернативный конструктор может передавать методу `parse()` вычисленный заранее код состояния, причинную фразу и MIME-заголовок.

**Метод `parse()`.** Метод `parse()` берет необработанные данные, которые были считаны из HTTP-сервера, извлекает из первой строки этих данных `statusCode` (код состояния) и `reasonPhrase` (причинную фразу), а из остальных строк полученных данных конструирует `MimeHeader`-объект (в переменной `mh`).

**Метод `toString()`.** Метод `toString()` — инверсия `parse()`. Он берет текущие значения объекта `HttpServletResponse` и возвращает строку, которую HTTP-клиент ожидает получить в качестве отклика сервера.

**Код.** Исходный код для `HttpServletResponse`:

```
import java.io.*;  
/* HttpResponse  
 * анализирует возвращаемое сообщение и MIME-заголовок от сервера.  
 * HTTP/1.0 302 Found = повторный вызов, проверить размещение.  
 * HTTP/1.0 200 OK = после MIME-заголовка идут данные файла. */  
  
class HttpServletResponse  
{  
    int statusCode;           // код состояния (Status-Code в MIME-спец-ях)  
    String reasonPhrase;     // фраза причины (Reason-Phrase в MIME-спец-ях)  
    MimeHeader mh;  
    static String CRLF = "\r\n";  
  
    void parse(String request) {  
        int fsp = request.indexOf(' ');  
        int nsp = request.indexOf(' ', fsp+1);  
        int eol = request.indexOf('\n');  
        String protocol = request.substring(0, fsp);  
        statusCode = Integer.parseInt(request.substring(fsp+1, nsp));  
        reasonPhrase = request.substring(nsp+1, eol);  
        String raw_mime_header = request.substring(eol + 1);  
        mh = new MimeHeader(raw_mime_header);  
    }  
  
    HttpResponse(String request) {  
        parse(request);  
    }  
}
```

```

    HttpResponse(int code, String reason, MimeHeader m) {
        statusCode = code;
        reasonPhrase = reason;
        mh = m;
    }

    public String toString() {
        return "HTTP/1.0 " + statusCode + " " + reasonPhrase + CRLF +
            mh + CRLF;
    }
}

```

## UrlCacheEntry.java

Чтобы кэшировать содержимое документа на сервере, нам нужно объединить URL-адрес, который использовался для извлечения документа, и описание самого документа. Документ описывается его MIME-заголовком (т. е. классом `MimeHeader`) и необработанными данными. Например, изображение могло бы быть описано классом `MimeHeader` с ключом `Content-Type:image/gif` и необработанными данными изображения, которые являются массивом байтов. Аналогично, Web-страница будет иметь пару ключ/значение `Content-Type:text/html` в ее классе `MimeHeader`, в то время как необработанные данные являются содержимым HTML-страницы. Переменные экземпляра не отмечены как `private`, так что `httpd` может иметь к ним прямой доступ.

**Конструктор.** Конструктор объекта `UrlCacheEntry` получает (через параметры) строку URL и MIME-заголовок (в форме `String` и `MimeHeader`-объектов). Если `MimeHeader` объект имеет поле с названием `Content-Length` (в большинстве случаев такое поле имеется), то, чтобы вместить такое содержимое, предварительно выделенная область данных должна быть достаточно большой.

**Метод `append()`.** Метод `append()` используется для добавления данных в объект `UrlCacheEntry`. Он имеет дело с тремя ситуациями. В первом случае буфер данных не распределяется вообще. Во втором — буфер данных слишком мал, чтобы вместить входящие данные, так что он перераспределяется. В последнем случае входящие данные прекрасно согласованы и вставляются прямо в буфер. Текущий размер буфера данных отслеживается в экземплярной переменной `length`, так что она всегда содержит действительный размер этого буфера.

**Код.** Исходный код для `UrlCacheEntry`:

```

class UrlCacheEntry
{
    String url;
    MimeHeader mh;
}

```

```

byte data[];
int length = 0;

public UrlCacheEntry(String u, MimeHeader m) {
    url = u;
    mh = m;
    String cl = mh.get("Content-Length");
    if (cl != null) {
        data = new byte[Integer.parseInt(cl)];
    }
}

void append(byte d[], int n) {
    if (data == null) {
        data = new byte[n];
        System.arraycopy(d, 0, data, 0, n);
        length = n;
    } else if (length + n > data.length) {
        byte old[] = data;
        data = new byte[old.length + n];
        System.arraycopy(old, 0, data, 0, old.length);
        System.arraycopy(d, 0, data, old.length, n);
    } else {
        System.arraycopy(d, 0, data, length, n);
        length += n;
    }
}
}
}

```

## LogMessage.java

Простой интерфейс LogMessage объявляет единственный метод — log(), который имеет только один String-параметр. Он используется, чтобы абстрагировать (выделить) из httpd вывод сообщений. В случае приложения этот метод реализует печать в форме стандартного консольного вывода (т. е. выводит строки сообщений прямо на экран, после командной строки). В случае апплета, данные добавляются в оконный текстовый буфер.

**Код.** Исходный код LogMessage-интерфейса:

```

interface LogMessage {
    public void log(String msg);
}

```

## Httpd.java

Это действительно большой класс, выполняющий множество действий. Мы просмотрим его метод за методом.

**Конструктор.** Существует пять главных экземплярных переменных: `port`, `docRoot`, `log`, `cache` и `stopFlag`, и все они — частные (`private`). Три из них можно установить единственным конструктором:

```
httpd(int p, String dr, LogMessage lm)
```

Он инициализирует порт прослушивания, каталог для извлекаемых файлов, и интерфейс для отправки сообщений.

Четвертая экземплярная переменная (`cache`), является объектом `Hashtable`, где все файлы кэшируются в оперативной памяти и инициализируются при создании объекта. Выполнением программы управляет переменная `stopFlag`.

**Статический раздел.** В этом классе есть несколько важных статических переменных. В переменной `version` можно найти версию, сообщаемую в поле `Server` MIME-заголовка. Определены несколько констант: `mime_text_html` — MIME-тип для HTML-файлов; `CRLF` — MIME-последовательность конца строки; `indexfile` — имя HTML-файла, возвращаемого вместо необработанных запросов каталога, и `buffer_size` — размер буфера, используемого для ввода/вывода данных.

Массив `mt` определяет список расширений имен файлов и соответствующих MIME-типов для этих файлов. Переменная `Hashtable types` статически инициализирована в следующем блоке так, чтобы содержать массив `mt` из чередующихся ключей и значений. Метод `fnameToMimeType()` можно использовать для возврата подходящего типа MIME для каждого переданного в него имени файла (`filename`). Если `filename` не имеет ни одного из расширений таблицы `mt`, метод `fnameToMimeType()` возвращает `defaultExt`, или `"text/plain"`.

**Статистические счетчики.** Затем объявляются еще пять экземплярных переменных. Они определены без `private`-модификатора, чтобы внешний монитор мог просматривать эти значения с целью их графического отображения. (Мы покажем эти действия позже.) Данные переменные представляют статистику использования нашего Web-сервера. Необработанное число нажатий и обслуживаемых байтов сохраняются в `hits_served` и `bytes_served`. Число файлов и байтов, размещенных в текущий момент в кэш-памяти, сохраняются в `files_in_cache` и `bytes_in_cache`. Наконец, мы храним число нажатий, которые успешно обслуживались из кэша в `hits_to_cache`.

**Метод `toBytes()`.** Метод удобств `toBytes()` относится к специальному типу процедур, называемых "подпрограммами удобств" (`convenience routine`). Он конвертирует свой строчный параметр в массив байтов. Это необходимо, потому что `String`-объекты Java хранятся как символы Unicode, в то время как общепринятый язык Internet-протоколов, таких как HTTP, — это хороший старый 8-разрядный код ASCII.

**Метод `makeMimeType()`.** Метод `makeMimeType()` — другая "подпрограмма удобств", которая используется для создания `MimeHeader`-объекта с нескольки-

кими вставленными ключевыми значениями. `MimeHeader`-объект, возвращаемый из этого метода, содержит в поле `Date` текущие время и дату, в поле `Server` — имя и версию нашего сервера, в поле `Content-Type` — параметр `type` и в поле `Content-Length` — параметр `length`.

**Метод `error()`.** Метод `error()` используется для форматирования HTML-страницы, отправляемой Web-клиентам, приславшим запросы, которые не могут быть успешно завершены. Первый параметр, `code`, является кодом возвращаемой ошибки. Как правило, он имеет значения из диапазона 400—499. Наш сервер посыпает обратно ошибки 404 и 405. Он использует класс `HttpResponse`, чтобы инкапсулировать код возврата в соответствующий MIME-заголовок (`MimeHeader`). Метод возвращает строчное представление этого ответа, сцепленное с HTML-страницей, предназначеннной для показа пользователю. Страница включает читаемую версию кода ошибки, сообщение (`msg`) и URL запроса, который вызвал ошибку.

**Метод `getRawRequest()`.** Метод `getRawRequest()` очень прост. Он читает данные из потока, пока не получит два последовательных символа `newline`. Он игнорирует переводы каретки и ищет только `newline`. Найдя второе вхождение `newline`, он вводит массив байтов в `String`-объект и возвращает его. Он вернет `null` (пустой указатель), если входной поток не производит двух последовательных `newline` перед своим окончанием. Покажем, как отформатированы сообщения от серверов и клиентов HTTP. Они начинаются одной строкой состояния, за которой немедленно следует MIME-заголовок. Конец MIME-заголовка отделен от остальной части содержимого двумя символами `newlines`.

**Метод `logEntry()`.** Метод `logEntry()` используется для формирования стандартного отчета на каждый запрос клиента к HTTP-серверу. Формат, который этот метод воспроизводит, может показаться нечетким, но он соответствует текущему стандарту для журнальных файлов HTTP. Этот метод имеет несколько вспомогательных переменных и методов, которые используются для форматирования отметок даты на каждом регистрационном входе. Чтобы конвертировать номер месяца в его строковое представление, используется строковый массив `months[]`. Переменная `host` устанавливается главным HTTP-циклом, когда он принимает соединение от заданного хост-компьютера. Метод `fmt02d()` преобразует однозначные целые числа (0,..., 9) в двузначную форму — с ведущим нулем. Затем результирующая строка пересыпается через интерфейс `LogMessage` переменный `log`.

**Метод `writeString()`.** Это еще один метод удобств. Им пользуются, чтобы скрыть преобразование `String`-объекта в массив байтов (для того, чтобы можно было представить его в форме выходного потока).

**Метод `writeUCE()`.** Метод `writeUCE()` имеет параметры типа `OutputStream` и `UrlCacheEntiy`. Он извлекает информацию из входов кэша и затем посыпает

Web-клиенту сообщение, содержащее подходящий код ответа, MIME-заголовок и содержимое.

**Метод *serveFromCache()*.** Этот булевский метод пытается найти определенный URL в кэше. Если поиск успешен, то содержимое соответствующего входа кэша отправляется клиенту (с помощью метода `writeUCE()`), затем переменная `hits_to_cache` увеличивается на 1 и вызывающей программе возвращается `true`. В противном случае (когда поиск неудачен) он просто возвращает `false`.

**Метод *loadFile()*.** Этот метод получает (через аргументы вызова) объекты типа `InputStream` (для поточного ввода файла), `String` (для URL соответствующего файла) и `MimeHeader` (для MIME-заголовка этого URL). Сначала создается новый объект класса `UrlCacheEntry` с информацией, хранящейся в полученном `MimeHeader`-объекте. Затем входной поток читается (по кускам размерами `buffer_size` байтов) и добавляется к URL-входу кэша `UrlCacheEntry`. Заполненный таким образом `UrlCacheEntry` сохраняется в кэше. Наконец, выполняется обновление переменных `Files_in_cache` и `bytes_in_cache`, и `UrlCacheEntry`-объект возвращается в вызывающую программу.

**Метод *readFile()*.** Метод `readFile()` может показаться избыточным по сравнению с методом `loadFile()`. Это не так. Данный метод используется только для чтения файлов, размещенных вне локальной файловой системы, тогда как `loadFile()` используется для загрузки потоков любого вида. Если `File`-объект `f` существует, то для него создается объект `InputStream`. Определяется размер файла (в переменной `file_length`) и строка имени файла приводится к MIME-форме (в переменной `mime_type`). Затем обе переменные используются для создания соответствующего MIME-заголовка (в переменной `MimeHeader mh`). Затем вызывается метод `loadFile()`, чтобы выполнить фактическое чтение и кэширование.

**Метод *writeDiskCache()*.** Метод `writeDiskCache()` получает объект `UrlCacheEntry` и записывает его на локальный диск. Сначала он создает (из URL-строки) имя каталога, заменяя слэш (/) символами, зависящими от системы (хранившимися в `File`-переменной `separatorChar`). Затем он вызывает метод `mkdirs()`, чтобы удостовериться, что локальный дисковый путь существует для заданного URL. Наконец, он создает объект `FileOutputStream`, имитируя открытие потока файлового ввода, записывает в него все данные и закрывает его.

**Метод *handleProxy()*.** Метод `handleProxy()` определяет один из двух главных режимов этого сервера. Основная идея заключается в следующем: если вы настраиваете ваш браузер на использование этого сервера в качестве прокси-сервера, то запросы, посылаемые ему, будут включать полный URL-адрес, откуда нормальные GET-операции удаляют начальные части ("http://" и имя хост-машины). Наш же метод, получив (через параметр `url`) полный URL, отыскивает последовательность "://", следующий слэш (/) и необязательное

двоеточие (: ) (для серверов, использующих нестандартные номера портов). После нахождения этих символов ему становятся известны запрашивающий хост-компьютер, номер порта, а также URL-адрес некоторого документа, который требуется выбрать с сервера. Затем он пытается загрузить предварительно сохраненную версию этого документа не из своего оперативного кэша. Если данное действие заканчивается неудачно, то можно попытаться загрузить ее из файловой системы в RAM-кэш и повторить попытку загрузки из кэша. Если и это не удается, то он должен читать документ из удаленного сайта.

Чтобы реализовать предложенный алгоритм, метод открывает сокет (т. е. организует сетевое соединение) с удаленным сайтом и портом и посыпает туда GET-запрос, требуя URL, который был послан к нему. Какой бы заголовок ответа не был получен от удаленного сайта, последний пересыпает его клиенту. Если код состояния в этом заголовке был равен 200 (что сигнализирует об успешной передаче файла), то метод также читает следующий за заголовком поток данных в новый кэш-вход UrlCacheEntry и записывает его на сокет клиента. После этого метод вызывает writeDiskCache(), чтобы сохранить результаты этой передачи на локальном диске. Наконец, он регистрирует транзакцию (вызывая метод logEntry()), закрывает сокеты и выполняет возврат.

**Метод handleGet().** Параметры метода handleGet() указывают, куда ему следует записывать результаты, поисковый URL и MIME-заголовок от запрашивающего Web-браузера. Этот MIME-заголовок будет включать строку User-Agent (агент пользователя) и другие полезные атрибуты. Сначала осуществляется попытка обслуживать URL вне RAM-кэша. Потерпев неудачу, метод заглядывает в файловую систему в поисках файла по его URL-адресу. Если файл не существует или нечитабелен, он сообщает Web-клиенту об ошибке. Иначе, он просто использует метод readFile(), чтобы получить содержимое файла и поместить его в кэш. Затем вызывается метод writeUCE(), чтобы послать содержимое файла сокету клиента.

**Метод doRequest().** Метод doRequest() вызывается один раз при каждом соединении с сервером. Сначала он анализирует строку запроса и входящий MIME-заголовок. Если в строке запроса обнаруживается последовательность "://", то он вызывает метод handleProxy(), иначе вызывается handleGet(). Если в запросе указываются методы, отличные от GET, например, HEAD или POST, то эта подпрограмма возвращает клиенту ошибку с кодом 405. Обратите внимание, что HTTP-запрос игнорируется, если stopFlag имеет значение true.

**Метод run().** Метод run() вызывается тогда, когда стартует поток сервера. Он создает новое соединение (объект ServerSocket) на заданном порте, входит в бесконечный цикл, вызывающий accept() на сокете сервера, и затем передает результирующий Socket-объект на обработку в метод doRequest().

**Методы *start()* и *stop()*.** Эти методы используются для запуска и остановки серверного процесса. Они устанавливают значения переменной *stopFlag*.

**Метод *main()*.** Метод *main()* можно использовать для того, чтобы запустить данное приложение из командной строки. Он устанавливает параметр *LogMessage*, чтобы самому быть сервером, и затем обеспечивает простой консольный вывод с помощью метода *log()*.

**Код.** Исходный код для *httpd*:

```
import java.net.*;
import java.io.*;
import java.text.*;
import java.util.*;

class httpd implements Runnable, LogMessage {
    private int port;
    private String docRoot;
    private LogMessage log;
    private Hashtable cache = new Hashtable();
    private boolean stopFlag;

    private static String version = "1.0";
    private static String mime_text_html = "text/html";
    private static String CRLF = "\r\n";
    private static String indexfile = "index.html";
    private static int buffer_size = 8192;
    static String mt[] = {           // отображение расширения файла в Mime-тип
        "txt", "text/plain",
        "html", mime_text_html,
        "htm", "text/html",
        "gif", "image/gif",
        "jpg", "image/jpg",
        "jpeg", "image/jpeg",
        "class", "application/octet-stream"
    };
    static String defaultExt = "txt";
    static Hashtable types = new Hashtable();
    static {
        for (int i=0; i<mt.length;i+=2)
            types.put(mt[i], mt[i+1]);
    }

    static String fnameToMimeType(String filename) {
        if (filename.endsWith("/"))      // специально для index файлов
            return mime_text_html;
        int dot = filename.lastIndexOf('.');
        String ext = (dot > 0) ? filename.substring(dot + 1) : defaultExt;
        String ret = (String) types.get(ext);
        return ret;
    }
}
```

```
    return ret != null ? ret : (String)types.get(defaultExt);
}

int hits_served = 0;
int bytes_served = 0;
int files_in_cache = 0;
int bytes_in_cache = 0;
int hits_to_cache = 0;

private final byte toBytes(String s) {
    byte b[] = s.getBytes();
    return b;
}

private MimeHeader makeMimeHeader(String type, int length) {
    MimeHeader mh = new MimeHeader();
    Date curDate = new Date();
    TimeZone gmtTz = TimeZone.getTimeZone("GMT");
    SimpleDateFormat sdf =
        new SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
    sdf.setTimeZone(gmtTz);
    mh.put("Date", sdf.format(curDate));
    mh.put("Server", "JavaCompleteReference/" + version);
    mh.put("Content-Type", type);
    if (length >= 0)
        mh.put("Content-Length", String.valueOf(length));
    return mh;
}

private String error(int code, String msg, String url) {
    String html_page = "<body>" + CRLF +
        "<h1>" + code + " " + msg + "</h1>" + CRLF;
    if (url != null)
        html_page += "Error when fetching URL: " + url + CRLF;
    html_page += "</body>" + CRLF;
    MimeHeader mh = makeMimeHeader(mime_text_html, html_page.length());
    HttpResponse hr = new HttpResponse(code, msg, mh);

    logEntry("GET", url, code, 0);
    return hr + html_page;
}

// Читает 'in', пока не будет получено два символа \n в строке.
// Сбрасывает все символы \r.
private String getRawRequest(InputStream in)
    throws IOException {
    byte buf[] = new byte[buffer_size];
    int pos=0;
    int c;
```

```
while ((c = in.read()) != -1) {  
    switch (c) {  
        case '\r':  
            break;  
        case '\n':  
            if (buf[pos-1] == c) {  
                return new String(buf, 0, pos);  
            }  
        default:  
            buf[pos++] = (byte) c;  
    }  
}  
return null;  
}  
  
static String months[] = {  
    "Jan", "Feb", "Mar", "Apr", "May", "Jun",  
    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"  
};  
private String host;  
// fmt02d эквивалент С функции printf("%02d", i)  
private final String fmt02d(int i) {  
    if(i < 0) {  
        i = -i;  
        return ((i < 9) ? "-0" : "-") + i;  
    }  
    else {  
        return ((i < 9) ? "0" : "") + i;  
    }  
}  
}  
private void logEntry(String cmd, String url, int code, int size) {  
    Calendar calendar = Calendar.getInstance();  
    int tzmin = calendar.get(Calendar.ZONE_OFFSET)/(60*1000);  
    int tzhour = tzmin / 60;  
    tzmin -= tzhour * 60;  
    log.log(host + " -- [" +  
        fmt02d(calendar.get(Calendar.DATE)) + "/" +  
        months[calendar.get(Calendar.MONTH)] + "/" +  
        calendar.get(Calendar.YEAR) + ":" +  
        fmt02d(calendar.get(Calendar.HOUR)) + ":" +  
        fmt02d(calendar.get(Calendar.MINUTE)) + ":" +  
        fmt02d(calendar.get(Calendar.SECOND)) + " " +  
        fmt02d(tzhour) + fmt02d(tzmin) +  
    "] \\" +  
    cmd + " " +  
    url + " HTTP/1.0\\" +  
    code + " " +  
    size + "\n");  
}
```

```
    hits_served++;
    bytes_served += size;
}

private void writeString(OutputStream out, String s)
    throws IOException {
    out.write(toBytes(s));
}

private void writeUCE(OutputStream out, UrlCacheEntry uce)
    throws IOException {
    HttpResponse hr = new HttpResponse(200, "OK", uce.mh);
    writeString(out, hr.toString());
    out.write(uce.data, 0, uce.length);
    logEntry("GET", uce.url, 200, uce.length);
}

private boolean serveFromCache(OutputStream out, String url)
    throws IOException {
    UrlCacheEntry uce;
    if ((uce = (UrlCacheEntry)cache.get(url)) != null) {
        writeUCE(out, uce);
        hits_to_cache++;
        return true;
    }
    return false;
}

private UrlCacheEntry loadFile(InputStream in, String url,
                               MimeHeader mh)
    throws IOException {

    UrlCacheEntry uce;
    byte file_buf[] = new byte[buffer_size];
    uce = new UrlCacheEntry(url, mh);
    int size = 0;
    int n;
    while ((n = in.read(file_buf)) >= 0) {
        uce.append(file_buf, n);
        size += n;
    }
    in.close();
    cache.put(url, uce);
    files_in_cache++;
    bytes_in_cache += uce.length;
    return uce;
}

private UrlCacheEntry readFile(File f, String url)
    throws IOException {
```

```
if (!f.exists())
    return null;
InputStream in = new FileInputStream(f);
int file_length = in.available();
String mime_type = fnameToMimeType(url);
MimeHeader mh = makeMimeHeader(mime_type, file_length);
UrlCacheEntry uce = loadFile(in, url, mh);
return uce;
}

private void writeDiskCache(UrlCacheEntry uce)
throws IOException {

String path = docRoot + uce.url;
String dir = path.substring(0, path.lastIndexOf("/"));
dir.replace('/', File.separatorChar);
new File(dir).mkdirs();
FileOutputStream out = new FileOutputStream(path);
out.write(uce.data, 0, uce.length);
out.close();
}

// Клиент посыпает серверу запрос в следующей форме:
// http://the.internet.site/the/url
// Сервер собирается получить запрос с сайта и возвратить его...
private void handleProxy(OutputStream out, String url,
                         MimeHeader inmh) {
    try {
        int start = url.indexOf(":/") + 3;
        int path = url.indexOf('/', start);
        String site = url.substring(start, path).toLowerCase();
        int port = 80;
        String server_url = url.substring(path);
        int colon = site.indexOf(':');
        if (colon > 0) {
            port = Integer.parseInt(site.substring(colon + 1));
            site = site.substring(0, colon);
        }
        url = "/cache/" + site + ((port != 80) ? ":" + port) : "" + server_url;
        if (url.endsWith("/"))
            url += indexfile;
        if (!serveFromCache(out, url)) {
            if (readFile(new File(docRoot + url), url) != null) {
                serveFromCache(out, url);
                return;
            }
        }
    }
}
```

```
// Если эта страница уже не была кэширована, открыть
// сокет-соединение с портом сайта и послать ему GET-команду.
// Сервер изменяет "user-agent" для добавления своих...

Socket server = new Socket(site, port);
InputStream server_in = server.getInputStream();
OutputStream server_out = server.getOutputStream();
inmh.put("User-Agent", inmh.get("User-Agent") +
          " via JavaCompleteReferenceProxy/" + version);
String req = "GET " + server_url + " HTTP/1.0" + CRLF +
             inmh + CRLF;
writeString(server_out, req);
String raw_request = getRawRequest(server_in);
HttpResponse server_response =
    new HttpResponse(raw_request);
writeString(out, server_response.toString());
if (server_response.statusCode == 200) {
    UrlCacheEntry uce = loadFile(server_in, url,
                                  server_response.mh);
    out.write(uce.data, 0, uce.length);
    writeDiskCache(uce);
    logEntry("GET", site + server_url, 200, uce.length);
}
server_out.close();
server.close();
}
} catch (IOException e) {
    log.log("Exception: " + e);
}
}

private void handleGet(OutputStream out, String url,
                      MimeHeader inmh) {
    byte file_buf[] = new byte[buffer_size];
    String filename = docRoot + url +
        (url.endsWith("/") ? indexfile : "");
    try {
        if (!serveFromCache(out, url)) {
            File f = new File(filename);
            if (!f.exists()) {
                writeString(out, error(404, "Not Found", filename));
                return;
            }
            if (!f.canRead()) {
                writeString(out, error(404, "Permission Denied", filename));
                return;
            }
            UrlCacheEntry uce = readFile(f, url);
            out.write(uce.data, 0, uce.length);
            writeDiskCache(uce);
            logEntry("GET", site + url, 200, uce.length);
        }
    } catch (IOException e) {
        log.log("Exception: " + e);
    }
}
```

```
        writeUCE(out, uce);
    }
} catch (IOException e) {
    log.log("Exception: " + e);
}
}

private void doRequest(Socket s) throws IOException {
    if(stopFlag)
        return;
    InputStream in = s.getInputStream();
    OutputStream out = s.getOutputStream();

    String request = getRawRequest(in);
    int fsp = request.indexOf(' ');
    int nsp = request.indexOf(' ', fsp+1);
    int eol = request.indexOf('\n');
    String method = request.substring(0, fsp);
    String url = request.substring(fsp+1, nsp);
    String raw_mime_header = request.substring(eol + 1);

    MimeHeader inmh = new MimeHeader(raw_mime_header);
    request = request.substring(0, eol);

    if (method.equalsIgnoreCase("get")) {
        if (url.indexOf(":/") >= 0) {
            handleProxy(out, url, inmh);
        } else {
            handleGet(out, url, inmh);
        }
    } else {
        writeString(out, error(405, "Method Not Allowed", method));
    }
    in.close();
    out.close();
}

public void run() {
    try {
        ServerSocket acceptSocket;
        acceptSocket = new ServerSocket(port);
        while (true) {
            Socket s = acceptSocket.accept();
            host = s.getInetAddress().getHostName();
            doRequest(s);
            s.close();
        }
    } catch (IOException e) {
```

```
    log.log("accept loop IOException: " + e + "\n");
} catch (Exception e) {
    log.log("Exception: " + e);
}
}

private Thread t;
public synchronized void start() {
    stopFlag = false;
    if (t == null) {
        t = new Thread(this);
        t.start();
    }
}

public synchronized void stop() {
    stopFlag = true;
    log.log("Stopped at " + new Date() + "\n");
}

public httpd(int p, String dr, LogMessage lm) {
    port = p;
    docRoot = dr;
    log = lm;
}

// Это методы main и log, позволяющие запускать httpd с консоли.
public static void main(String args[]) {
    httpd h = new httpd(80, "c:\\www", null);
    h.log = h;
    h.start();
    try {
        Thread.currentThread().join();
    } catch (InterruptedException e) {};
}

public void log(String m) {
    System.out.print(m);
}
}
```

## HTTP.java

В качестве дополнительного приза рассматривается класс апплета, который придает HTTP-серверу функциональную "лицевую панель". Этот апплет имеет два параметра, которые можно использовать для конфигурирования сервера: port и docroot. Это очень простой апплет. Он создает экземпляр httpd, пересылаемый самому себе как интерфейс LogMessage. Затем он создает панель, которая имеет простую метку сверху, текстовую область (TextArea)

в середине — для отображения сообщений (LogMessages), и панель внизу, которая содержит две кнопки и еще одну метку. Методы start() и stop() апплета вызывают соответствующие методы класса httpd. Кнопки с метками **Start** и **Stop** также вызывают соответствующие методы httpd. При каждой регистрации сообщения нижний правый Label-объект обновляется так, чтобы содержать самую последнюю статистику из httpd.

```
import java.util.*;
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class HTTP extends Applet implements LogMessage, ActionListener
{
    private int m_port = 80;
    private String m_docroot = "c:\\www";
    private httpd m_httpd;
    private TextArea m_log;
    private Label status;

    private final String PARAM_port = "port";
    private final String PARAM_docroot = "docroot";
    public HTTP()
    {
    }

    public void init()
    {
        setBackground(Color.white);
        String param;

        // port: номер порта прослушивания
        param = getParameter(PARAM_port);
        if (param != null)
            m_port = Integer.parseInt(param);

        // docroot: путь Web-документа
        param = getParameter(PARAM_docroot);
        if (param != null)
            m_docroot = param;

        setLayout(new BorderLayout());
        Label lab = new Label("Java HTTPD");
        lab.setFont(new Font("SansSerif", Font.BOLD, 18));
        add("North", lab);
        m_log = new TextArea("", 24, 80);
        add("Center", m_log);
        Panel p = new Panel();
        p.setLayout(new FlowLayout(FlowLayout.CENTER, 1, 1));
        add("South", p);
    }

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == start)
            m_httpd.start();
        else if (e.getSource() == stop)
            m_httpd.stop();
        else if (e.getSource() == status)
            status.setText(m_log.getText());
    }
}
```

```
Button bstart = new Button("Start");
bstart.addActionListener(this);
p.add(bstart);
Button bstop = new Button("Stop");
bstop.addActionListener(this);
p.add(bstop);
status = new Label("raw");
status.setForeground(Color.green);
status.setFont(new Font("SansSerif", Font.BOLD, 10));
p.add(status);
m_httpd = new httpd(m_port, m_docroot, this);
}

public void destroy() {
    stop();
}

public void paint(Graphics g)  {

}

public void start()  {
    m_httpd.start();
    status.setText("Running  ");
    clear_log("Log started on " + new Date() + "\n");
}

public void stop()  {
    m_httpd.stop();
    status.setText("Stopped  ");
}

public void actionPerformed(ActionEvent ae) {
    String label = ae.getActionCommand();
    if(label.equals("Start")) {
        start();
    }
    else {
        stop();
    }
}

public void clear_log(String msg) {
    m_log.setText(msg + "\n");
}

public void log(String msg) {
    m_log.append(msg);
    status.setText(m_httpd.hits_served + " hits (" +
                  (m_httpd.bytes_served / 1024) + "K), " +
                  m_httpd.files_in_cache + " cached files (" +
```

```

        (m_httpd.bytes_in_cache / 1024) + "K), " +
        m_httpd.hits_to_cache + " cached hits");
status.setSize(status.getPreferredSize());
}
}

```

### Замечание

В файлах `httpd.java` и `HTTP.java` код построен в предположении, что документ размещается в каталоге `c:\www`. Вы можете изменить это значение для своей конфигурации. Поскольку этот апллет пишет в журнальный файл, он может работать только в том случае, если в переменной среды `CLASSPATH` указан путь к его каталогу.

## Дейтаграммы

Для большинства межсетевых потребностей вас будет удовлетворять TCP/IP-стиль работы в сети. Он обеспечивает сериализованный, предсказуемый и надежный поток пакетных данных. Не следует, однако, забывать о достаточно высокой его стоимости. Протокол TCP включает много сложных алгоритмов, имеющих дело с управлением перегрузкой на переполненных сетях и весьма пессимистическими ожиданиями потерь пакетов. Он приводит к довольно неэффективному способу транспортировки данных. Альтернативу обеспечивают дейтаграммные протоколы.

*Дейтаграммы* — это небольшие пакеты информации, независимо транспортируемые между машинами. После того как дейтаграмма направляется предназначенному ей адресату, нет никакой гарантии, что она прибудет в пункт назначения, и кто-то сможет там ее получить. Аналогично, когда дейтаграмма принимается, нет гарантии, что она не была повреждена при пересылке или что тот, кто ее послал, все еще находится в пункте передачи и готов получить ответ.

Java реализует дейтаграммы в верхней части UDP-протокола, используя два класса. Объект класса `DatagramPacket` — это контейнер данных, в то время как класс `DatagramSocket` — это механизм, используемый для посылки или приема `DatagramPacket`-объектов.

### Класс `DatagramPacket`

Объект `DatagramPacket` может быть создан одним из четырех конструкторов. Первый конструктор определяет буфер, который примет данные, и размер пакета. Он применяется для приема данных через `DatagramSocket`. Вторая форма позволяет определять смещение к буферу, в котором данные будут сохраняться. Третья форма специфицирует целевой адрес и порт, использующиеся классом `DatagramSocket`, чтобы определить, куда данные пакета

будут отправлены. Четвертая форма передает пакеты, начинающиеся с указанным смещением в наборе данных. Существует четыре конструктора:

```
DatagramPacket(byte data[ ], int size)
DatagramPacket(byte data[ ], int offset, int size)
DatagramPacket(byte data[ ], int size, InetAddress ipAddress, int port)
DatagramPacket(byte data[ ], int offset, int size, InetAddress
               ipAddress, int port)
```

Кроме того, существует несколько методов для доступа к внутреннему состоянию дейтаграммного объекта. Они предоставляют полный доступ к адресу назначения и номеру порта пакета так же, как к необработанным данным и их длине. Краткое описание методов представлено в табл. 18.6.

**Таблица 18.6. Методы для доступа к внутреннему состоянию дейтаграммного объекта**

Метод	Описание
InetAddress getAddress()	Возвращает Internet-адрес (объект типа InetAddress) пункта назначения; обычно используется для посылки гистограмм
int getPort()	Возвращает номер порта
byte[ ] getData()	Возвращает байт-массив данных, содержащихся в дейтаграмме. Главным образом используется для извлечения данных дейтаграммы после того, как она была принята
int getLength()	Возвращает длину достоверных данных, содержащихся в байт-массиве, который был возвращен методом getData(). Она обычно не равняется длине всего байт-массива

## Дейтаграммный сервер и клиент

Следующий пример реализует очень простые сетевые коммуникации клиента и сервера. Сообщения печатаются в окно на сервере и записываются через сеть на сторону клиента, где и отображаются на экране.

```
// Демонстрирует дейтаграммы.
import java.net.*;

class WriteServer {
    public static int serverPort = 666;
    public static int clientPort = 999;
    public static int buffer_size = 1024;
    public static DatagramSocket ds;
    public static byte buffer[] = new byte[buffer_size];
```

```

public static void TheServer() throws Exception {
    int pos=0;
    while (true) {
        int c = System.in.read();
        switch (c) {
            case -1:
                System.out.println("Server Quits.");
                return;
            case '\r':
                break;
            case '\n':
                ds.send(new DatagramPacket(buffer, pos,
                    InetAddress.getLocalHost(), clientPort));
                pos=0;
                break;
            default:
                buffer[pos++] = (byte) c;
        }
    }
}

public static void TheClient() throws Exception {
    while(true) {
        DatagramPacket p = new DatagramPacket(buffer, buffer.length);
        ds.receive(p);
        System.out.println(new String(p.getData(), 0, p.getLength()));
    }
}

public static void main(String args[]) throws Exception {
    if(args.length == 1) {
        ds = new DatagramSocket(serverPort);
        TheServer();
    } else {
        ds = new DatagramSocket(clientPort);
        TheClient();
    }
}
}

```

Эта выборочная программа ограничена DatagramSocket-конструктором, организующим ее выполнение между двумя портами на локальной машине. Чтобы использовать программу, запустите ее:

Java WriteServer

в одном окне (это будет клиент). Затем запустите:

Java WriteServer 1

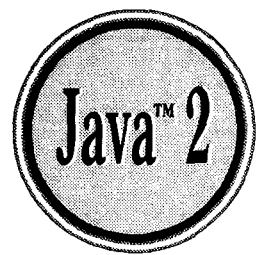
Теперь это будет сервер. Все, что напечатано в окне сервера, будет передано окну клиента после того, как будет получен символ newline.

### Замечание

Этот пример требует, чтобы ваш компьютер был подключен к Internet.

## Достоинства сети

Имея пять классов — InetAddress, Socket, ServerSocket, DatagramSocket и DatagramPacket, вы можете программировать любой существующий протокол Internet. Они также обеспечивают мощный низкоуровневый контроль над соединениями Internet. Сетевые пакеты Java очень хорошо отражают состояние Internet.



## ГЛАВА 19

# Класс *Applet*

В данной главе рассматривается класс *Applet*, обеспечивающий необходимую поддержку для аплетов. В главе 12 была представлена общая форма аплета и шаги, необходимые для его компиляции и выполнения. Здесь же мы рассмотрим аплеты подробно.

Класс *Applet* содержится в пакете `java.applet`. *Applet* включает несколько методов, которые дают детальный контроль над выполнением аплета. Дополнительно `java.applet` определяет три интерфейса: `AppletContext`, `AudioClip` и `AppletStub`.

Начнем с рассмотрения основных элементов аплета и шагов, необходимых для его создания и тестирования.

## Основы аплетов

Все аплеты являются подклассами *Applet*. Таким образом, они должны импортировать `java.applet`, а также `java.awt`. Вспомните, что AWT — сокращение *Abstract Window Toolkit* (абстрактный оконный интерфейс). Так как все аплеты выполняются в окне, необходимо включить поддержку для этого окна. Аплеты не исполняются Java-интерпретатором времени выполнения, работающим в консольном режиме. Они выполняются или Web-браузером или программой просмотра аплетов, представленные в этой главе, были созданы стандартной программой просмотра аплетов, называемой *appletviewer* и поставляемой с пакетом разработки JDK (*Java Developer Kit*, инструментарий разработчика Java). Но вы можете использовать любую программу просмотра аплетов или браузер, с которыми вы работаете.

Выполнение аплета не начинается с метода `main()`. Некоторые из них даже содержат метод `main()`, однако выполнение аплета начинается и управляется совершенно иным механизмом, который требует хотя бы краткого объяснения. Вывод в окно аплета не выполняется методом `System.out.println()`.

Скорее, он обрабатывается различными AWT-методами, такими как `drawString()`, который выводит строку в указанную точку экрана. Ввод также обрабатывается иначе, чем в приложении.

Как только апплет откомпилирован, он включается в HTML-файл, используя тег `<applet>`. Апплет будет выполняться Java-совместимым браузером, когда тот встретит в HTML-файле указанный тег. Для более удобного просмотра и проверки апплета просто включите в начало файла исходного кода Java-комментарий, который содержит тег `<applet>`. Этим способом ваш код документируется вместе с инструкциями HTML, необходимыми апплету, и вы можете проверить откомпилированный апплет, запустив программу просмотра с вашим файлом исходного кода в качестве параметра. Пример такого комментария:

```
/*
<applet code="MyApplet" width=200 height=60>
</applet>
*/
```

Этот комментарий содержит тег `<applet>`, который выполнит апплет с именем `MyApplet` в окне с размерами  $200 \times 60$  пикселов. Так как включение команды `<applet>` делает тестирование апплетов проще, все показанные в этой книге апплеты будут содержать соответствующий тег, внедренный в комментарий.

## Класс *Applet*

Класс `Applet` определяет методы, представленные в табл. 19.1. `Applet` обеспечивает всю необходимую поддержку для выполнения апплетов, такую как запуск и остановка. Он также реализует методы, которые загружают и показывают изображения, и методы, которые загружают и проигрывают аудиоклипы. `Applet` расширяет AWT-класс `Panel`. Кроме того, `Panel` расширяет `Container`, который, в свою очередь, расширяет `Component`. Эти классы обеспечивают поддержку графического интерфейса Java при работе с окнами. Таким образом, `Applet` обеспечивает всю необходимую поддержку для работы с окнами. (AWT описан подробно в следующих главах.)

**Таблица 19.1. Методы, определенные в классе *Applet***

Метод	Описание
<code>void destroy()</code>	Освобождает все ресурсы, занятые апплетом. Вызывается браузером непосредственно перед тем, как апплет завершается. Ваш апплет переопределит данный метод, если потребуется выполнить какую-нибудь дополнительную чистку перед его выполнением

Таблица 19.1 (продолжение)

Метод	Описание
<code>AppletContext getAppletContext()</code>	Возвращает контекст, связанный с апплетом
<code>String getAppletInfo()</code>	Возвращает строку, которая описывает апплет
<code>AudioClip getAudioClip(URL url)</code>	Возвращает объект AudioClip, который инкапсулирует аудиоклип, найденный по адресу, указанному в <code>url</code>
<code>AudioClip getAudioClip(URL url, String clipName)</code>	Возвращает объект AudioClip, который инкапсулирует аудиоклип, найденный по адресу, указанному в <code>url</code> , и имеющий имя, указанное в параметре <code>clipName</code>
<code>URL getCodeBase()</code>	Возвращает URL, связанный с вызывающим апплетом
<code>URL getDocumentBase()</code>	Возвращает URL HTML-документа, который вызывает апплет
<code>Image getImage(URL url)</code>	Возвращает объект Image, который инкапсулирует изображение, найденное по адресу <code>url</code>
<code>Image getImage(URL url, String imageName)</code>	Возвращает объект Image, который инкапсулирует изображение, найденное по адресу <code>url</code> , и имеющий имя, указанное в параметре <code>imageName</code>
<code>Locale getLocale()</code>	Возвращает объект Locale, который используется различными чувствительными к локализации классами и методами
<code>String getParameter(String paramName)</code>	Возвращает параметр, указанный в <code>paramName</code> . Если указанный параметр не найден, возвращается null (пустой указатель)
<code>String[ ] [ ] getParameterInfo()</code>	Возвращает таблицу строк, описывающую параметры, распознанные апплетом. Каждый вход в таблицу должен состоять из трех строк, которые содержат имя параметра, описание его типа и/или диапазона, и объяснения его цели
<code>void init()</code>	Вызывается, когда апплет начинает выполнение. Это первый метод, который вызывается для любого апплета

Таблица 19.1 (окончание)

Метод	Описание
<code>boolean isActive()</code>	Возвращает <code>true</code> , если аплет был запущен. Возвращает <code>false</code> , если аплет был остановлен
<code>static final AudioClip newAudioClip(URL url)</code>	Возвращает объект <code>AudioClip</code> , который инкапсулирует аудиоклип, найденный по адресу <code>url</code> . Этот метод подобен <code>getAudioClip()</code> за исключением того, что он статический и может быть выполнен без потребности в <code>Applet</code> -объекте. (Добавлен в Java 2.)
<code>void play(URL url)</code>	Если аудиоклип найден по адресу <code>url</code> , то он проигрывается
<code>void play(URL url, String clipName)</code>	Если аудиоклип найден по адресу <code>url</code> с именем <code>clipName</code> , то клип проигрывается
<code>void resize(Dimension dim)</code>	Изменяет размеры аплета согласно измерениям, указанным в <code>dim</code> . <code>Dimension</code> – это класс пакета <code>java.awt</code> . Он содержит два целочисленных поля: <code>width</code> и <code>height</code>
<code>void resize(int width, int height)</code>	Изменяет размеры аплета согласно размерам, указанным в <code>width</code> и <code>height</code>
<code>final void setStub(AppletStub stubObj)</code>	Делает <code>stubObj</code> -заглушку для аплета. Этот метод используется исполнительной системой Java и обычно не вызывается аплетом. Заглушка – маленькая часть кода, которая обеспечивает связь между аплетом и браузером
<code>void showStatus(String str)</code>	Отображает значение параметра <code>str</code> в окне состояния браузера или программы просмотра аплета. Если браузер не поддерживает окно состояния, то никакое действие не выполняется
<code>void start()</code>	Вызывается браузером, когда аплет должен запустить (или возобновить) выполнение. После <code>init()</code> (когда аплет впервые начинает выполнение) вызывается автоматически
<code>void stop()</code>	Вызывается браузером, чтобы приостановить выполнение аплета. После остановки аплет перезапускается, когда браузер вызывает <code>start()</code>

## Архитектура апплета

Апплет — программа, работающая с окном. Поэтому его архитектура отличается от так называемых "нормальных" программ, основанных на консольном вводе/выводе и показанных в первой части данной книги. Если вы знакомы с программированием под Windows, то будете иметь возможность писать апплеты дома. Если нет, то имеется несколько ключевых концепций, которые нужно понимать.

Во-первых, апплеты управляются событиями. Хотя мы не будем рассматривать обработку событий до следующей главы, важно общее понимание того, как управляемая событиями архитектура воздействует на проектирование апплета. Апплет походит на набор программ обработки прерываний. Процесс выполняется так: апплет ожидает возникновение некоторого события. AWT уведомляет апплет о событии вызовом обработчика события, который был обеспечен апплетом. Как только это случается, апплет должен выполнить соответствующее действие и затем быстро возвратить управление AWT. Это критический момент. По большей части ваш апплет не должен входить в режим работы, в котором он поддерживает управление длительный период. В тех ситуациях, когда апплет вынужден выполнять повторяющуюся задачу сам по себе (например, отображая прокручивающееся в его окне сообщение), вы должны запустить дополнительный поток выполнения. (Примеры увидите позже в этой главе.)

Во-вторых, пользователь инициализирует взаимодействие с апплетом. Как вы знаете, в "неоконной" программе, которая нуждается во вводе, она выдает подсказку пользователю и затем вызывает некоторый метод ввода, такой как `readLine()`. Это — не способ работы в апплете. Вместо этого, пользователь взаимодействует с апплетом, как и когда он этого захочет. Эти взаимодействия посылаются апплету как события, на которые апплет должен ответить. Например, когда пользователь щелкает мышью внутри окна апплета, генерируется событие "щелчок мышью". Если пользователь нажимает клавишу в то время, когда окно апплета имеет фокус ввода, генерируется событие "нажатие клавиши". Как вы увидите в следующих главах, апплеты могут содержать различные элементы управления, такие как кнопки и переключатели. Когда пользователь взаимодействует с одним из этих элементов управления, также генерируется событие.

В то время как архитектура апплета не столь проста для понимания, как у консольной программы, Java AWT делает ее настолько простой, насколько это возможно. Если вы писали программы для Windows, то знаете, какой пугающей может быть эта среда. К счастью, Java AWT обеспечивает более ясный подход, с которым можно справиться намного быстрее.

## Скелетная схема апплета

Все апплеты, кроме наиболее тривиальных, переопределяют набор методов, обеспечивающих основной механизм, с помощью которого браузер или программа просмотра взаимодействует с апплетом и управляет его выполнением. Четыре таких метода — `init()`, `start()`, `stop()` и `destroy()` — определены в `Applet`. Пятый, `paint()`, определен AWT-классом `Component`. Для всех этих методов обеспечены также и реализации по умолчанию. Апплетам не нужно переопределять те методы, которые они не используют. Однако только в очень простых апплетах не нужно определять все эти методы сразу. Все пять методов можно собрать в следующую скелетную схему:

```
// Скелетная схема (скелет) апплета.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/
public class AppletSkel extends Applet {
    // Вызывается первым.
    public void init() {
        // инициализация
    }

    /* Вызывается вторым, после init().
       Вызывается также для перезапуска апплета. */
    public void start() {
        // начало или продолжение выполнения
    }

    // Вызывается, когда апплет остановлен.
    public void stop() {
        // приостанавливает выполнение
    }

    /* Вызывается, когда апплет завершается.
       Это — последний выполняемый метод. */
    public void destroy() {
        // выполняет завершающие действия
    }

    // Вызывается, когда окно апплета должно быть перерисовано.
    public void paint(Graphics g) {
        // повторный показ содержимого окна
    }
}
```

Хотя этот скелет ничего не выполняет, его можно откомпилировать и запустить. После выполнения он генерирует окно, показываемое программой просмотра апплета (рис. 19.1).

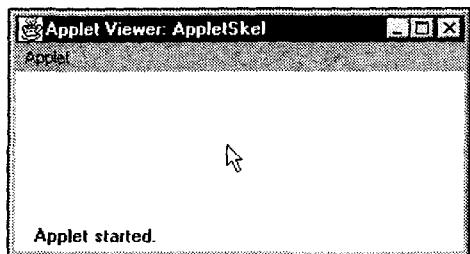


Рис. 19.1. Окно апплета AppletSkel

## Инициализация и завершение апплета

Важно понять порядок, в котором вызываются различные методы, показанные в скелетной схеме. Когда апплет начинает выполняться, AWT вызывает методы в такой последовательности:

1. `init()`
2. `start()`
3. `paint()`

При завершении апплета имеет место следующая последовательность вызовов:

1. `stop()`
2. `destroy()`

Рассмотрим подробнее эти методы.

### Метод `init()`

Метод `init()` — первый метод, который требует вызова. В нем вы должны инициализировать переменные. Вызывается он только однажды в течение времени выполнения апплета.

### Метод `start()`

Метод `start()` следует после `init()`. Он также вызывается, чтобы перезапустить апплет после его остановки. В то время как `init()` вызывается однажды (в первый момент, когда апплет загружается), `start()` запускается каждый раз, когда HTML-документ апплета отображается на экране. Так, если пользователь покидает Web-страницу и возвращается обратно, апплет возобновляет выполнение в `start()`.

## Метод *paint()*

Метод *paint()* вызывается всякий раз, когда вывод апплета должен быть перерисован. Это ситуация может возникнуть по нескольким причинам. Например, окно, в котором апплет выполняется, может быть перекрыто сверху другим окном, которое затем закрывается, или окно апплета может быть свернуто и затем восстановлено. Метод *paint()* вызывается также, когда апплет начинает выполнение. Таким образом, вне зависимости от причины *paint()* вызывается всякий раз, когда апплет должен перерисовывать свой вывод. Метод *paint()* имеет один параметр типа *Graphics*. Он должен содержать графический контекст, описывающий графическую среду, в которой выполняется апплет. Данный контекст используется всякий раз, когда требуется вывод в апплет.

## Метод *stop()*

Метод *stop()* вызывается, если Web-браузер покидает HTML-документ, содержащий апплет, при переходе к другой странице. Когда вызывается *stop()*, апплет, вероятно, продолжает выполняться. Следует использовать *stop()* для приостановки потоков, не требующих выполнения, если апплет невидим. Их можно перезапустить вызывом *start()*, когда пользователь возвращается к странице.

## Метод *destroy()*

Метод *destroy()* вызывается, когда среда решает, что апплет должен быть полностью удален из памяти. В этот момент следует освободить любые ресурсы, которые апплет может использовать. Метод *stop()* всегда вызывается перед *destroy()*.

## Переопределение метода *update()*

В некоторых ситуациях апплет может переопределить метод *update()*, определенный в AWT. Данный метод вызывается, когда требуется перерисовка части окна апплета. Заданная по умолчанию версия *update()* сначала заполняет апплет заданным по умолчанию цветом фона и затем вызывает *paint()*. Если вы заполняете фон, используя другой цвет в *paint()*, то пользователь будет видеть вспышку заданного по умолчанию фона каждый раз, когда вызывается *update()*, т. е. всякий раз, когда окно перерисовывается. Один из способов обойти указанную проблему заключается в переопределении метода *update()* так, чтобы он исполнял все необходимые действия дисплея. Тогда, вызывая *paint()*, просто запускают *update()*. Итак, для некоторых приложений скелет апплета переопределяет *paint()* и *update()* так:

```
public void update(Graphics g) {  
    // здесь повторный показ вашего окна.  
}
```

```
public void paint(Graphics g) {  
    update(g);
```

Для примеров в этой книге мы будем переопределять `update()`, только когда необходимо.

## Простые методы отображения апплетов

Как мы уже упоминали, апплеты отображаются в окне, и они используют AWT для организации ввода и вывода. Хотя мы рассмотрим методы, процедуры и технику, необходимую для полного управления оконной средой AWT, в последующих главах, некоторые из них все-таки следует описать здесь, поскольку мы будем пользоваться ими для записи простых апплетов.

Как было описано в главе 12, чтобы вывести строку в окно апплета, используют метод `drawString()`, который является членом класса `Graphics`. Как правило, он вызывается внутри или `update()`, или `paint()`. Он имеет следующую общую форму:

```
void drawString(String message, int x, int y)
```

Здесь `message` — строка, которую нужно вывести, начиная с позиции `x, y`. (В окне Java левый верхний угол имеет позицию с координатами 0,0.) Метод `drawString()` не распознает символы `newline`. Если нужно начать порцию текста с новой строки, требуется сделать это вручную, определяя точные (`x, y`) координаты, с которых вы хотите начать размещение строки. (В следующих главах показана техника, упрощающая этот процесс.)

Для установки цвета фона в окне апплета используйте метод `setBackground()`, а для цвета переднего плана (цвет, который применяется для отображения текста, например) — `setForeground()`. Оба метода определены в `Component` и имеют следующие общие формы:

```
void setBackground(Color newColor)  
void setForeground(Color newColor)
```

Здесь `newColor` — назначает новый цвет. Класс `Color` определяет константы, которые можно использовать для указания цвета:

- |  |  |
|--|--|
| <input type="checkbox"/> Color.black     | <input type="checkbox"/> Color.magenta |
| <input type="checkbox"/> Color.blue      | <input type="checkbox"/> Color.orange  |
| <input type="checkbox"/> Color.cyan      | <input type="checkbox"/> Color.pink    |
| <input type="checkbox"/> Color.darkGray  | <input type="checkbox"/> Color.red     |
| <input type="checkbox"/> Color.gray      | <input type="checkbox"/> Color.white   |
| <input type="checkbox"/> Color.green     | <input type="checkbox"/> Color.yellow  |
| <input type="checkbox"/> Color.lightGray |  |

Например, следующие вызовы устанавливают зеленым цвет фона и красным цвет текста:

```
setBackground(Color.green);
setForeground(Color.red);
```

Удачным местом указания цветов переднего плана и фона является метод `init()`. Конечно, можно изменять эти цвета так часто, как вам необходимо, во время выполнения апплета. При этом заданный по умолчанию цвет переднего плана — черный, а заданный по умолчанию цвет фона — светло-серый.

Вы можете получить текущие установки для фона и символов, вызывая методы `getBackground()` и `getForeground()`. Они определены в классе `Component` со следующими сигнатурами:

```
Color getBackground()
Color getForeground()
```

Ниже показан очень простой апплет, который устанавливает голубым цвет фона и красным цвет переднего плана (символов), а затем отображает сообщение, иллюстрирующее порядок вызова методов `init()`, `start()` и `paint()` после запуска апплета:

```
/* Простой апплет, который устанавливает цвета
символов и фона и выводит строку. */
import java.awt.*;
import java.applet.*;
/*
<applet code="Sample" width=300 height=50>
</applet>
*/
public class Sample extends Applet{
    String msg;

    // устанавливает цвета символов и фона
    public void init() {
        setBackground(Color.cyan);
        setForeground(Color.red);
        msg = "Inside init() --";
    }

    // инициализирует строку для показа
    public void start() {
        msg += " Inside start() --";
    }

    // показывает msg в окне апплета
    public void paint(Graphics g) {
```

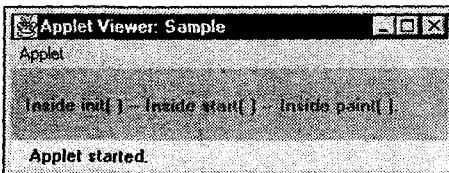
```

    msg += " Inside paint().";
    g.drawString(msg, 10, 30);
}
}

```

Этот аплет генерирует окно, представленное на рис. 19.2.

Методы `stop()` и `destroy()` не переопределяются, потому что они не нужны этому простому аплету.



**Рис. 19.2.** Окно аплета Sample

## Требование перерисовки

Существует общее правило: аплет организует вывод в свое окно только тогда, когда AWT вызывает его методы `update()` или `paint()`. Это поднимает интересный вопрос: как сам аплет может вызвать модификацию своего окна, когда его информация изменяется? Например, если аплет отображает движущийся заголовок, какой механизм используется для обновления окна каждый раз, когда этот заголовок прокручивается? Запомните одно из фундаментальных архитектурных ограничений, наложенных на аплет: он должен быстро возвратить управление в исполняющую систему AWT. Он не может создавать цикл внутри `paint()`, который, например, повторно прокручивает заголовок. Это помешало бы передаче управления обратно в AWT. При данном ограничении может показаться, что вывод в окно аплета будет, в лучшем случае, затруднен. К счастью, дело обстоит не так. Всякий раз, когда аплет должен обновить информацию, отображенную в его окне, он просто вызывает `repaint()`.

Метод `repaint()` определен в пакете AWT. Он заставляет исполняющую систему AWT вызывать метод `update()` вашего аплета, который в его реализации по умолчанию вызывает `paint()`. Таким образом, в той части аплета, где требуется отобразить нечто в окне, просто сохраните вывод и затем вызовите `repaint()`. Тогда AWT выполнит вызов `paint()`, который может отображать сохраненную информацию. Например, если часть аплета должна вывести строку, она может сохранить эту строку в `String`-переменной и затем вызвать `repaint()`. Внутри `paint()` вы будете выводить строку, используя метод `drawString()`.

Метод `repaint()` имеет четыре формы. Рассмотрим каждую по очереди. Самая простая версия `repaint()`:

```
void repaint()
```

Она вызывает перерисовку всего окна. Следующая версия определяет область, которая будет перерисована:

```
void repaint(int left, int top, int width, int height)
```

Здесь координаты верхнего левого угла области определены параметрами *left* и *top*, а ширина и высота области пересыпается в *width* и *height*. Эти измерения определены в пикселях. Указывая область для перерисовки, вы экономите время, т. к. обновление окна — длительная операция. Если нужно обновить только маленькую часть окна, более эффективно перерисовать именно эту область.

Вызов *repaint()* — это, по существу, требование, чтобы ваш *applet* был перерисован несколько позднее. Однако, если ваша система "нетороплива" или занята, *update()*, возможно, не будет вызываться немедленно. Множественные запросы перерисовки, которые происходят в пределах короткого времени, могут быть отвергнуты AWT, так что *update()* вызывается только время от времени. Это может стать проблемой во многих ситуациях, включая мультиPLICATION, где постоянно требуется время для обновления. Одно из решений этой проблемы состоит в том, чтобы использовать следующие формы *repaint()*:

```
void repaint(long maxDelay)
```

```
void repaint(long maxDelay, int x, int y, int width, int height)
```

Здесь *maxDelay* определяет максимальное число миллисекунд, на которое задерживается вызов *update()*. Однако, остерегайтесь. Если это время закончится прежде, чем *update()* может быть вызван, он вовсе не вызывается. У метода нет никакого возвращаемого значения или выброшенного исключения, так что вы должны быть внимательными.

### Замечание

Возможен вывод в окно апплета другим способом — не методами *paint()* или *update()*. Для этого следует получить графический контекст, вызывая *getGraphics()* (определенный в *Component*), и затем использовать этот контекст для вывода в окно. Однако для большинства приложений лучше и проще направлять вывод окна через *paint()* и вызывать *repaint()*, когда содержимое окна изменяется.

## Апплет с бегущим заголовком

Для демонстрации *repaint()* разработан апплет с бегущим заголовком, который прокручивает сообщение через окно апплета справа налево. Так как прокрутка сообщения — повторяющаяся задача, она выполняется отдельным потоком, создаваемым апплетом во время инициализации. Исходный код этого апплета:

```
/* Апллет с бегущим заголовком.  
Этот апллет создает поток, который прокручивает сообщение,  
содержающееся в msg, через окно аплета. */  
import java.awt.*;  
import java.applet.*;  
/*  
<applet code="SimpleBanner" width=300 height=50>  
</applet>  
*/  
  
public class SimpleBanner extends Applet implements Runnable {  
    String msg = " A Simple Moving Banner.";  
    Thread t = null;  
    int state;  
    boolean stopFlag;  
  
    // Установка цветов и инициализация потока.  
    public void init() {  
        setBackground(Color.cyan);  
        setForeground(Color.red);  
    }  
  
    // Запустить поток.  
    public void start() {  
        t = new Thread(this);  
        stopFlag = false;  
        t.start();  
    }  
  
    // Точка входа для потока, который запускает заголовок.  
    public void run() {  
        char ch;  
  
        // Показать заголовок.  
        for(;;) {  
            try {  
                repaint();  
                Thread.sleep(250);  
                ch = msg.charAt(0);  
                msg = msg.substring(1, msg.length());  
                msg += ch;  
                if(stopFlag)  
                    break;  
            } catch(InterruptedException e) {}  
        }  
    }  
  
    // Остановить заголовок.  
    public void stop() {
```

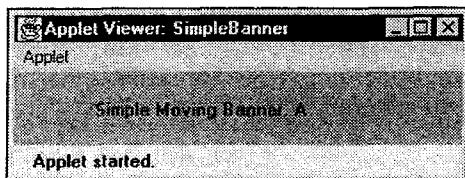
```

stopFlag = true;
t = null;
}

// Показать заголовок.
public void paint(Graphics g) {
    g.drawString(msg, 50, 30);
}
}
}

```

Пример вывода изображен на рис. 19.3.



**Рис. 19.3.** Окно апплета SimpleBanner

Рассмотрим подробнее работу этого апплета. Во-первых, заметим, что SimpleBanner расширяет класс Applet, как и ожидалось, но он также реализует интерфейс Runnable. Это необходимо, т. к. апплет будет создавать второй поток выполнения, который используется для прокручивания заголовка. Внутри init() устанавливаются цвета переднего плана и фона апплета.

После инициализации исполнительная система AWT вызывает метод start(), чтобы начать выполнение апплета. Внутри start() создается новый поток выполнения и назначается Thread-переменной t. Затем, булевой переменной stopFlag, которая управляет выполнением апплета, присваивается значение false. Далее, вызов t.start() запускает поток. Напомним, что t.start() вызывает метод, определенный классом Thread, который начинает выполнение с помощью метода run(). Он не обращается к версии start(), определенной в классе Applet. Это два отдельных метода.

Внутри run() символы в строке, содержащейся в msg, циклически смещаются влево. Между каждым смещением выполняется обращение к repaint(). В итоге вызывается метод paint() и отображается текущее содержимое msg. Между каждой итерацией run() замирает на четверть секунды. Сетевой эффект run() состоит в том, что содержимое msg прокручивается справа налево в постоянно перемещающемся представлении. Переменная stopFlag проверяется на каждой итерации. Когда она становится true, метод run() завершает свое выполнение.

Если при выполнении апплета браузер переключается на просмотр новой страницы, вызывается метод stop(), который устанавливает в stopFlag значение true и завершает работу run(). Данный механизм применяется для остановки потока, когда страница больше не находится в поле зрения браузера.

зера. Когда апплет возвращается обратно в режим просмотра, `start()` вызывается еще раз, запуская новый поток для прокрутки заголовка.

## Использование окна состояния

В дополнение к отображению информации в своем окне, апплет может также выводить сообщение в окно состояния браузера или программы просмотра апплетов (`appletviewer`), которая выполняет его. Для этого нужно вызвать метод `showStatus()`, указывая в его аргументе строку для отображения. Окно состояния обеспечивает обратную связь пользователя с работающей программой. В нем программа может показать, что происходит в апплете (например, предоставлять сведения о режимах работы или, возможно, сообщать о некоторых типах ошибок). Кроме того, окно состояния служит превосходным средством отладки, обеспечивающим простой способ вывода информации о работе апплета.

Следующий апплет демонстрирует использование `showStatus()`:

```
// Использование окна состояния.
import java.awt.*;
import java.applet.*;
/*
<applet code="StatusWindow" width=300 height=50>
</applet>
*/
public class StatusWindow extends Applet{
    public void init() {
        setBackground(Color.cyan);
    }
    // Отображает msg в окне апплета.
    public void paint(Graphics g) {
        g.drawString("This is in the applet window.", 10, 20);
        showStatus("This is shown in the status window.");
    }
}
```

Пример вывода этой программы представлен на рис. 19.4.

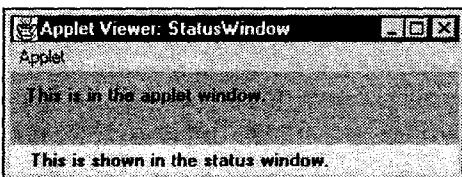


Рис. 19.4. Окно апплета StatusWindow

## Тег <applet>

Апплет можно запускать как из HTML-документа, так и из программы просмотра апплета. Для этого используется тег <applet> языка HTML. Программа просмотра апплета выполняет каждый <applet>-тег, который она находит, в отдельном окне, в то время как Web-браузеры Netscape Navigator, Internet Explorer и HotJava допускают много апплетов на одной странице. Пока мы использовали только упрощенную форму тега <applet>. Теперь пришло время взглянуть на него поближе.

Далее показан синтаксис стандартного тега <applet>. Параметры в квадратных скобках — не обязательны.

```
<applet
  [CODEBASE = codebaseURL]
  CODE = appletFile
  [ALT = altText]
  [NAME = appletInstanceName]
  WIDTH = pixels HEIGHT = pixels
  [ALIGN = alignment]
  [VSPACE = pixels] [HSPACE = pixels]
  >
  [< param NAME = AttributeName VALUE =AttributeValue >]
  [< param NAME = AttributeName2 VALUE =AttributeValue >]
  [HTML Displayed in the absence of Java]
</applet>
```

Рассмотрим каждую часть подробнее.

**CODEBASE.** CODEBASE — необязательный параметр, который определяет базовый URL-адрес кода апплета. Базовый URL — это каталог, в котором будет разыскиваться исполняемый файл апплета (имя этого файла указывается параметром CODE). Если атрибут CODEBASE не определен, то по умолчанию используется базовый URL (т. е. каталог) текущего HTML-документа. Указанный в CODEBASE URL не должен находиться на хост-компьютере, из которого был считан HTML-документ.

**CODE.** CODE — обязательный параметр, который задает имя файла, содержащего откомпилированный файл (с расширением .class) вашего апплета. Этот файл относится к базовому URL кода апплета, являющемуся каталогом, в котором находится HTML-файл, или каталогом, указанным в параметре CODEBASE (если он используется).

**ALT.** ALT — необязательный параметр, используемый для указания короткого текстового сообщения, которое должно быть отображено, если браузер понимает тег <applet>, но не может в текущий момент выполнять Java-

апплеты. (Эта ситуация отличается от того случая, когда для браузеров, не поддерживающих апллеты, вы предусматриваете альтернативный HTML-документ.)

**NAME.** NAME — необязательный параметр, используемый для определения имени экземпляра апллета. Апллеты должны быть каким-то образом названы для обеспечения поиска и связи с ними других апллетов по имени. Для того чтобы получить апллет по имени, используйте метод `getApplet()`, который определен в интерфейсе `AppletContext`.

**WIDTH и HEIGHT.** WIDTH и HEIGHT — это обязательные параметры, которые задают размер области показа апллета (в пикселях).

**ALIGN.** ALIGN — необязательный параметр, который определяет выравнивание апллета. Данный параметр трактуется так же, как HTML-тег `<img>` со следующими возможными значениями: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE и ABSBOTTOM.

**VSPACE и HSPACE.** Эти параметры необязательные. VSPACE определяет пустой интервал (в пикселях) выше и ниже области показа апллета. HSPACE задает пустой интервал (в пикселях) на каждой стороне области показа апллета. Они трактуются так же, как атрибуты VSPACE и HSPACE тега `<img>`.

**Тег `<param>` (с параметрами `NAME=` и `VALUE=`).** Вложенный тег `<param>` позволяет указывать на HTML-странице параметры, специфические для данного апллета. Апллет получает доступ к этим параметрам с помощью метода `getParameter()`.

**Работа с устаревшими браузерами.** Несколько старых браузеров не могут выполнять апллеты и не распознают тег `<applet>`. Хотя эти браузеры теперь почти вышли из употребления (и заменены Java-совместимыми браузерами), некоторое время они еще могут быть востребованы. Лучший способ проектирования HTML-страниц для таких браузеров состоит в том, чтобы включить внутрь тегов `<applet></applet>` HTML-текст и разметку. Если applet-теги не распознаются вашим браузером, то вы увидите эту альтернативную разметку. Если средства Java доступны для вашего браузера, то он использует всю разметку между тегами `<applet></applet>` и игнорирует альтернативную разметку.

Вот HTML-текст встроенного апллета для запуска Java-апплета с именем `SampleApplet` и отображения сообщения в старых браузерах:

```
<applet code="SampleApplet" width=200 height=40>
  If you were driving a Java powered Navigator/
  you'd see "A Sample Applet" here.<p>
</applet>
```

## Пересылка параметров в аплеты

Как мы только что видели, HTML тег <applet> позволяет передавать параметры включающему аплету. Для получения этих параметров следует использовать метод `getParameter()`. Он возвращает значение указанного параметра в форме `String`-объекта. Таким образом, для числовых и булевых значений нужно будет преобразовать их строчные представления во внутренние форматы. Ниже приводится пример, который демонстрирует передачу параметров:

```
// Использование параметров.
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/
public class ParamDemo extends Applet{
    String fontName;
    int fontSize;
    float leading;
    boolean active;

    // Инициализация строки для показа.
    public void start() {
        String param;

        fontName = getParameter("fontName");
        if(fontName == null)
            fontName = "Not Found";

        param = getParameter("fontSize");
        try {
            if(param != null)          // если не найден
                fontSize = Integer.parseInt(param);
            else
                fontSize = 0;
        } catch(NumberFormatException e) {
            fontSize = -1;
        }

        param = getParameter("leading");
        try {
```

```

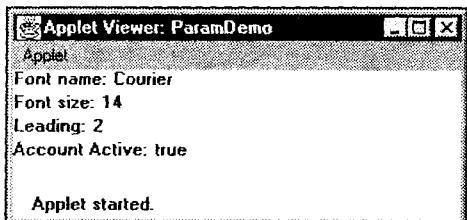
if(param != null) // если не найден
    leading = Float.valueOf(param).floatValue();
else
    leading = 0;
} catch(NumberFormatException e) {
    leading = -1;
}

param = getParameter("accountEnabled");
if(param != null)
    active = Boolean.valueOf(param).booleanValue();
}

// Показ параметров на экране.
public void paint(Graphics g) {
    g.drawString("Font name: " + fontName, 0, 10);
    g.drawString("Font size: " + fontSize, 0, 26);
    g.drawString("Leading: " + leading, 0, 42);
    g.drawString("Account Active: " + active, 0, 58);
}
}

```

Пример вывода этой программы представлен на рис. 19.5.



**Рис. 19.5.** Окно апплита ParamDemo

Как показывает программа, следует проверить значения, возвращаемые от `getParameter()`. Если параметр недоступен, `getParameter()` возвращает `null`-указатель. Кроме того, в операторе `try`, который отлавливает исключение `NumberFormatException`, должны быть выполнены преобразования к числовым типам. Непойманные исключения никогда не должны появляться в пределах апплета.

## Усовершенствованный апплет заголовка

Передачу параметра можно использовать для усовершенствования показанного ранее апплета с бегущим заголовком. В предыдущей версии бегущее сообщение было жестко закодировано в апплете. Однако с помощью методики передачи параметра можно отображать различные сообщения при каждом прогоне апплета. Ниже показана версия апплета с бегущим заголовоком,

улучшенная таким способом. Обратите внимание, что в теге <applet> в верхней части исходного файла теперь определяется параметр param с именем message (сообщение), который имеет значение бегущей строки.

```

        } catch(InterruptedException e) {}
    }
}

// Приостановить заголовок.
public void stop() {
    stopFlag = true;
    t = null;
}

// Показать заголовок на экране.
public void paint(Graphics g) {
    g.drawString(msg, 50, 30);
}
}
}

```

## Методы *getDocumentBase()* и *getCodeBase()*

Часто необходимо создавать апплеты, которые будут явно загружать среду и текст. Java позволяет загружать в апплет данные из каталога, содержащего HTML-файл, запустивший апплет (т. е. из *базы документа*), и каталога, из которого был загружен class-файл апплета (т. е. из *базы кода*). Методы *getDocumentBase()* и *getCodeBase()* возвращают указанные каталоги в виде URL-объектов, описанных в главе 18. Их можно спаять со строкой, именующей загружаемый файл. Чтобы загружать другие файлы, нужно использовать другой метод — *showDocument()*, определенный в интерфейсе *AppletContext* (он обсуждается в следующем разделе).

Рассмотрим апплет, иллюстрирующий использование этих методов:

```

// Отображение баз кода и документа.
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="Bases" width=300 height=50>
</applet>
*/

public class Bases extends Applet{
    // Отобразить базы кода и документа.
    public void paint(Graphics g) {
        String msg;

        URL url = getCodeBase();           // получить базу кода
        msg = "Code base: " + url.toString();
        g.drawString(msg, 10, 20);
    }
}

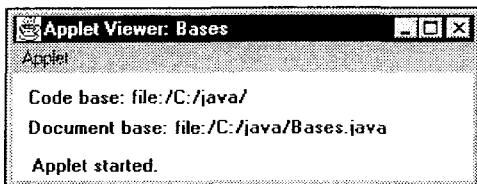
```

```

url = getDocumentBase();           // получить базу документа
msg = "Document base: " + url.toString();
g.drawString(msg, 10, 40);
}
}

```

Пример вывода этой программы представлен на рис. 19.6.



**Рис. 19.6.** Окно апплета  
Bases

## Интерфейс *AppletContext* и метод *showDocument()*

На языке Java удобно программировать *графические* средства Web-навигации — изображения и мультиплексию, которые более интересны, чем текстовые адресные гиперссылки (подчеркнутые синие слова), широко используемые для этой цели в гипертекстовых документах. Чтобы разрешить апплету передавать управление на другой URL, следует использовать метод *showDocument()*, определенный интерфейсом *AppletContext*.

Интерфейс *AppletContext* позволяет получать информацию из среды выполнения апплета. Методы, определенные в *AppletContext*, показаны в табл. 19.2. Контекст выполняющегося апплета можно получить, обращаясь к методу *getAppletContext()*, который определен в классе *Applet*.

Получив свой контекст, апплет может перейти к показу другого документа, вызывая метод *showDocument()*. Данный метод не имеет возвращаемого значения и не выбрасывает исключение, если терпит неудачу, так что используйте его с осторожностью. Существует две формы метода *showDocument()* (табл. 19.2). Метод первого формата (с одним параметром — *url*) отображает документ из каталога, указанного в аргументе его вызова. Метод второго формата (с двумя параметрами — *url* и *where*) отображает документ, определенный первым аргументом вызова, в той области окна браузера, на которую указывает второй аргумент. Параметр *where* может принимать следующие (строковые) значения: "*\_self*" (показ в текущем фрейме апплета), "*\_parent*" (показ в родительском фрейме апплета), "*\_top*" (показ в самом верхнем фрейме апплета) и "*\_blank*" (показ в новом окне браузера). Можно также указать имя, под которым документ будет отображен в новом окне браузера.

**Таблица 19.2. Абстрактные методы, определенные в интерфейсе AppletContext**

Метод	Описание
<code>Applet getApplet(String appName)</code>	Возвращает Applet-объект, имя которого специфицирует параметр <code>appName</code> (если это имя находится в текущем контексте апплета). В противном случае возвращается <code>null</code> (пустой указатель)
<code>Enumeration getApplets()</code>	Возвращает перечисление, которое содержит все апплеты из текущего контекста апплета
<code>AudioClip getAudioClip(URL url)</code>	Возвращает AudioClip-объект (который инкапсулирует аудиокlip), находящийся по адресу, указанному в аргументе вызова
<code>Image getImage(URL url)</code>	Возвращает Image-объект (который инкапсулирует изображение), находящийся по адресу, указанному в аргументе вызова
<code>void showDocument(URL url)</code>	Начинает показ документа, находящегося по адресу, указанному в аргументе вызова. Этот метод может не поддерживаться средствами просмотра апплета
<code>void showDocument(URL url, String where)</code>	Начинает показ документа, находящегося по адресу, указанному в первом аргументе вызова. Этот метод может не поддерживаться средствами просмотра апплета. Место размещения документа определяется параметром <code>where</code> , как описано в тексте раздела
<code>void showStatus(String str)</code>	Показывает строку, указанную в аргументе вызова, в окне состояния

Следующий апплет демонстрирует AppletContext и `showDocument()`. Во время выполнения, он получает текущий контекст апплета и использует его для передачи управления файлу с именем Test.html. Данный файл должен быть в том же каталоге, что и апплет. Test.html может содержать любой правильный гипертекст.

```
/* Использование контекста апплета, getCodeBase(),
и showDocument() для просмотра HTML-файла.
```

```
*/
```

```
import java.awt.*;
import java.applet.*;
import java.net.*;
```

```
/*
<applet code="ACDemo" width=300 height=50>
</applet>
*/
public class ACDemo extends Applet{
    public void start() {
        AppletContext ac = getAppletContext();
        URL url = getCodeBase();           // получить url данного апплета

        try {
            ac.showDocument(new URL(url+"Test.html"));
        } catch(MalformedURLException e) {
            showStatus("URL not found");
        }
    }
}
```

## Интерфейс *AudioClip*

Интерфейс *AudioClip* определяет следующие методы: *play()* (проигрывает клип с начала), *stop()* (останавливает проигрывание) и *loop()* (выполняет непрерывное циклическое проигрывание). Их можно использовать для воспроизведения аудиоклипа после его загрузки методом *getAudioClip()*.

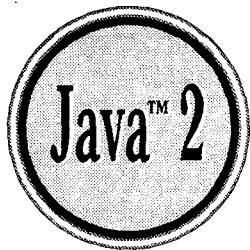
## Интерфейс *AppletStub*

Интерфейс *AppletStub* обеспечивает средства, с помощью которых апплет и браузер (или программа просмотра апплета) взаимодействуют между собой. Прикладные программисты редко реализуют этот интерфейс.

## Вывод на консоль

Хотя вывод в окно апплета должен быть организован через AWT-методы типа *drawString()*, все еще можно использовать и консольный вывод, особенно для целей отладки. Когда вызывается метод, такой как *System.out.println()*, вывод не посыпается в окно апплета. Вместо этого он появляется или в консольном сеансе, где вы запустили программу просмотра апплета, или в консоли Java, которая доступна в некоторых браузерах. Использование консольного вывода рекомендуется только для целей отладки, т. к. он нарушает основной принцип проектирования приложений — использование *графического интерфейса*.

# ГЛАВА 20



## Обработка событий

В данной главе рассматривается важный аспект языка Java, который имеет прямое отношение к апплетам и связан с *событиями*. Как объяснялось в главе 19, апплеты — это программы, управляемые событиями. Таким образом, в основе успешного апплет-программирования лежит *обработка событий*. Большинство событий, на которые должен откликаться апплет, генерируются пользователем. Эти события передаются апплету различными путями и специфическими методами, зависящими от действительного события. Существует несколько типов событий. Большинство обычным способом обрабатываемых событий генерируются мышью, клавиатурой и различными элементами управления, такими как командные кнопки. Эти события поддерживаются пакетом `java.awt.event`.

Глава начинается с краткого обзора механизма обработки событий Java. Затем рассматриваются основные классы событий и интерфейсов, и показано несколько примеров, которые демонстрируют основные принципы обработки событий. Эта глава также объясняет, как использовать классы-адаптеры, внутренние классы, и классы, упрощающие код обработки событий. Примеры, приведенные в остальной части данной книги, часто используют эту технику.

### Два механизма обработки событий

Перед началом нашего обсуждения обработки событий, сделаем важное замечание. Начиная с Java 1.1, существенно изменился способ обработки событий (по сравнению с первоначальной версией Java 1.0). Хотя метод обработки версии 1.0 все еще поддерживается, но он не рекомендуется для новых программ. Многие из методов, которые поддерживают старую модель событий версии 1.0, были исключены. Современный подход состоит в том, что события должны обрабатываться всеми новыми программами, включая

написанные для Java 2, и именно такой способ используется программами в этой книге.

## Модель делегирования событий

Современный подход к обработке событий основан на *модели делегирования событий* (delegation event model), которая определяет стандартные и непротиворечивые механизмы для генерации и обработки событий. Эта концепция весьма проста: источник генерирует событие и посыпает его одному или нескольким блокам прослушивания<sup>1</sup> (listeners) событий. В этой схеме, блок прослушивания просто ждет поступления события. Получив событие, блок прослушивания обрабатывает его и затем возвращает управление. Преимущество указанного способа состоит в том, что логика приложения, обрабатывающего события, четко отделена от логики интерфейса пользователя, генерирующего эти события. Элемент интерфейса пользователя способен "делегировать" обработку события отдельной части кода.

В модели делегирования событий блоки прослушивания должны зарегистрироваться в источнике для того, чтобы принимать уведомление (сообщение) события. Это обеспечивает важное преимущество: уведомления посыпаются только блокам прослушивания, которые хотят их принять. Это более эффективный способ обработки событий, чем метод, используемый старым подходом в Java 1.0. Прежде событие распространялось по ограниченной иерархии компонентов, пока один из них не обрабатывал это событие. Данный метод требует от компонентов принятия событий, которые они не обрабатывают, и, следовательно, тратит впустую много ценного времени. Модель делегирования событий устраниет подобные накладные расходы.

### Замечание

Java также позволяет обрабатывать события без использования модели делегирования событий, расширяя AWT-компонент. Эта методика обсуждена в конце главы 22. Однако, модель делегирования событий — привилегированный проект по только что указанным причинам.

Следующие разделы определяют события и описывают роли источников и блоков прослушивания.

## События

В модели делегирования *событие* — это объект, который описывает изменение состояния источника. Он может быть сгенерирован как последовательность взаимодействий оператора ПК с элементами в графическом интер-

<sup>1</sup> Блок прослушивания (listener) — интерфейс, создаваемый для перехвата конкретного типа события от конкретного компонента. — Примеч. пер.

файсе пользователя. Генерацию событий могут вызвать такие действия оператора, как нажатие кнопки, ввод символа через клавиатуру, выбор элемента в списке, щелчок мыши и множество других операций.

Могут происходить также события, которые непосредственно не вызываются взаимодействиями с интерфейсом пользователя. Событие, например, может сгенерировать таймер, завершающий работу, счетчик, превышающий предустановленное значение, программа в момент завершения операций или возникновения особых программных ситуаций, аппаратный отказ и т. д. Программист может сам определять события, которые будут обнаруживать и обрабатывать его приложение.

## Источники событий

*Источник* — это объект, который генерирует событие. Генерация события происходит тогда, когда каким-то образом изменяется внутреннее состояние этого объекта. Источники могут генерировать несколько типов событий.

Чтобы блоки прослушивания могли принимать уведомление об определенном типе событий, источник должен *регистрировать* эти блоки. Каждый тип событий имеет собственный метод регистрации. Общая форма таких методов:

```
public void addTypeListener(TypeListener el)
```

Здесь *type* — это имя события, а *el* — ссылка на блок прослушивания события. Например, метод, который регистрирует блок прослушивания события клавиатуры, называется *addKeyListener()*. Метод, регистрирующий блок прослушивания движения мыши, называется *addMouseMotionListener()*. Когда событие происходит, все зарегистрированные блоки прослушивания уведомляются (о произошедшем событии) и принимают копию объекта события. Это известно как *мультивещание* (*multicasting*) событий. Во всех случаях уведомления посыпаются только блокам прослушивания, которые зарегистрировались для их приема.

Некоторые источники могут позволять регистрироваться только одному блоку прослушивания. Общая форма такого метода:

```
public void addTypedListener(TypeListener el)
    throws java.util.TooManyListenersException
```

где *type* — имя события, а *el* — ссылка на блок прослушивания события. При наступлении события зарегистрированный блок прослушивания уведомляется. Это известно как *унивещание* (*unicasting*) событий.

Источник должен также обеспечить метод, который позволяет блоку прослушивания не регистрировать заинтересованность в определенном типе событий. Общая форма такого метода:

```
public void removeTypeListener(TypeListener el)
```

Здесь `type` — имя события, а `e1` — ссылка на блок прослушивания события. Например, чтобы удалить блок прослушивания клавиатуры, следует вызвать метод `removeKeyListener()`.

Методы, которые добавляют или удаляют блоки прослушивания, обеспечиваются генерирующим событие источником. Например, класс `Component` обеспечивает методы для добавления и удаления блоков прослушивания событий клавиатуры и мыши.

## Блок прослушивания событий

**Блок прослушивания** — это объект, который получает уведомление, когда происходит событие. К нему предъявляется два главных требования. Во-первых, чтобы принимать уведомления относительно определенных типов событий, он должен быть *зарегистрирован* одним или несколькими источниками. Во-вторых, он должен *реализовать* методы для приема и обработки этих уведомлений.

Методы, которые принимают и обрабатывают события, определены в наборе интерфейсов, находящихся в пакете `java.awt.event`. Например, интерфейс `MouseMotionListener` определяет два метода для приема уведомлений о событиях перетаскивания или передвижения мыши. Любой объект может принимать и обрабатывать одно или оба этих события, если он обеспечивает реализацию этого интерфейса. Далее в этой и последующих главах будут обсуждаться много других интерфейсов блоков прослушивания.

## Классы событий

В основе механизма обработки событий находятся классы, которые представляют события. Начнем наше изучение обработки событий с обзора классов событий. Они обеспечивают непротиворечивые и удобные в использовании средства инкапсуляции событий.

В корне иерархии классов событий Java находится класс `EventObject`, который размещен в пакете `java.util`. Это — суперкласс для всех событий. Один из его конструкторов:

`EventObject(Object src)`

где `src` — объект, который генерирует это событие.

`EventObject` содержит два метода: `getSource()` и `toString()`. Метод `getSource()` возвращает источник события. Его общая форма:

`Object getSource()`

Метод `toString()` возвращает строчный эквивалент события.

Класс `AWTEvent`, определенный в пакете `java.awt`, является подклассом класса `EventObject`. Это суперкласс (прямо или косвенно) всех AWT-событий, используемых моделью делегирования событий. Для определения типа события можно использовать его метод `getID()`. Сигнатура этого метода:

```
int getID()
```

Дополнительные подробности относительно класса `AWTEvent` даны в конце главы 22. Здесь же важно знать только, что все другие классы, обсуждаемые в этом разделе — подклассы `AWTEvent`.

Итак:

- `EventObject` — суперкласс всех событий.
- `AWTEvent` — суперкласс всех AWT-событий, которые обрабатываются моделью делегирования событий.

Пакет `java.awt.event` определяет несколько типов событий, которые генерируются различными элементами интерфейса пользователя. Табл. 20.1 перечисляет наиболее важные из этих классов событий и кратко описывает, когда они генерируются. Наиболее часто используемые конструкторы и методы каждого класса рассматриваются в следующих разделах.

**Таблица 20.1.** Основные классы событий `java.awt.event`

Класс событий	Описание
<code>ActionEvent</code>	Генерируется, когда нажата кнопка, дважды щелкнут элемент списка или выбран пункт меню
<code>AdjustmentEvent</code>	Генерируется при манипуляциях с полосой прокрутки
<code>ComponentEvent</code>	Генерируется, когда компонент скрыт, перемещен, изменен в размере или становится видимым
<code>ContainerEvent</code>	Генерируется, когда компонент добавляется или удаляется из контейнера
<code>FocusEvent</code>	Генерируется, когда компонент получает или теряет фокус
<code>InputEvent</code>	Абстрактный суперкласс для всех классов событий ввода компонентов
<code>ItemEvent</code>	Генерируется, когда: <ul style="list-style-type: none"> <li>• помечен флагок или элемент списка</li> <li>• сделан выбор элемента в списке выбора</li> <li>• выбран/отменен элемент меню с меткой</li> </ul>
<code>KeyEvent</code>	Генерируется, когда получен ввод от клавиатуры

Таблица 20.1 (окончание)

Класс событий	Описание
MouseEvent	Генерируется, когда объект перетасчен мышью (dragged) или перемещен (moved), произошел щелчок (clicked), нажата (pressed) или отпущена (released) кнопка мыши; также генерируется, когда указатель мыши входит или выходит в/за границы компонента
TextEvent	Генерируется, когда изменено значение текстовой области или текстового поля
WindowEvent	Генерируется, когда окно активизировано, закрыто,dezактивировано, развернуто или свернуто в значок, открыто или организован выход (exit) из него

## Класс *ActionEvent*

Событие *ActionEvent* генерируется, когда нажата кнопка, произошел двойной щелчок по элементу списка или выбран пункт меню. Класс *ActionEvent* определяет четыре целочисленные константы, которые можно использовать для идентификации любых модификаторов, связанных с событием действия: *ALT\_MASK*, *CTRL\_MASK*, *META\_MASK* и *SHIFT\_MASK*. Кроме того, существует целочисленная константа, *ACTION\_PERFORMED*, которую можно применять для идентификации *action*-события.

*ActionEvent* имеет два конструктора:

```
ActionEvent(Object src, int type, String cmd)
ActionEvent(Object src, int type, String cmd, int modifiers)
```

где *src* — ссылка на объект, который генерировал это событие; *type* — тип события; *cmd* — его командная строка; *modifiers* — указывает, какие клавиши-модификаторы (<Alt>, <Ctrl>, <META>, и/или <Shift>) были нажаты при генерации события.

Можно получить имя команды для вызова объекта *ActionEvent*, используя метод *getActionCommand()* с форматом:

```
String getActionCommand()
```

Например, когда кнопка нажата, генерируется *action*-событие, которое имеет имя команды, равное метке (надписи) на этой кнопке.

Метод *getModifiers()* возвращает значение, которое указывает, какие клавиши-модификаторы (<Alt>, <Ctrl>, <META>, и/или <Shift>) были нажаты при генерации события. Его формат:

```
int getModifiers()
```

## Класс *AdjustmentEvent*

События *AdjustmentEvent* генерируются полосой прокрутки. Существует пять типов *adjustment*-событий. Класс *AdjustmentEvent* определяет целочисленные константы, которые можно использовать для идентификации этих типов:

- **BLOCK\_DECREMENT**. Пользователь щелкнул внутри полосы прокрутки, чтобы уменьшить ее значение.
- **BLOCK\_INCREMENT**. Пользователь щелкнул внутри полосы прокрутки, чтобы увеличить ее значение.
- **TRACK**. Ползунок был перемещен.
- **UNIT\_DECREMENT**. Кнопка-стрелка в начале полосы прокрутки была нажата, чтобы уменьшить ее значение.
- **UNIT\_INCREMENT**. Кнопка-стрелка в конце полосы прокрутки была нажата, чтобы увеличить ее значение.

Кроме того, имеется целочисленная константа, **ADJUSTMENT\_VALUE\_CHANGED**, которая указывает на изменение установки полосы.

*AdjustmentEvent* имеет следующий конструктор:

```
AdjustmentEvent(Adjustable src, int id, int type, int data)
```

где *src* — ссылка на объект, который генерировал это событие; *id* — числовой идентификатор события установки, равный **ADJUSTMENT\_VALUE\_CHANGED**; *type* — определяет тип события; *data* — связанные с событием данные.

Метод *getAdjustable()* возвращает объект, который генерирован этим событием. Его формат:

```
Adjustable getAdjustable()
```

Тип события установки можно получить методом *getAdjustmentType()*. Он возвращает одну из констант, определенных в *AdjustmentEvent*. Общая форма этого метода:

```
int getAdjustmentType()
```

Значение установки полосы прокрутки можно получить методом *getValue()* с форматом:

```
int getValue()
```

Например, когда выполняются манипуляции с полосой прокрутки, этот метод возвращает значение, представляющее измененное значение полосы.

## Класс *ComponentEvent*

События *ComponentEvent* генерируются, когда размер, позиция или видимость компонента изменяются. Существует четыре типа компонентных со-

бытий. Класс `ComponentEvent` определяет четыре целочисленные константы, которые можно использовать для идентификации этих типов:

- `COMPONENT_HIDDEN`. Компонент был скрыт.
- `COMPONENT_MOVED`. Компонент был перемещен.
- `COMPONENT_RESIZED`. Размер компонента был изменен.
- `COMPONENT_SHOWN`. Компонент стал видимым.

Класс `ComponentEvent` имеет такой конструктор:

```
ComponentEvent(Component src, int type)
```

где `src` — ссылка на объект, который сгенерировал данное событие; `type` — определяет тип события.

`ComponentEvent` — это суперкласс (прямой или косвенный) классов `ContainerEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent` и `WindowEvent`.

Метод `getComponent()` возвращает компонент, который генерировал событие. Формат этого метода:

```
Component getComponent()
```

## Класс `ContainerEvent`

События класса `ContainerEvent` генерируются при добавлении или удалении компонента из контейнера. Существует два типа container-событий. Для их идентификации класс `ContainerEvent` определяет целочисленные константы: `COMPONENT_ADDED` и `COMPONENT_REMOVED`. Они указывают, что компонент был добавлен или удален из контейнера.

Класс `ContainerEvent` — это подкласс `ComponentEvent` с конструктором:

```
ContainerEvent(Component src, int type, Component comp)
```

где `src` — ссылка на контейнер, который сгенерировал это событие; `type` — тип события; `comp` — компонент, который был добавлен или удален из контейнера.

Вы можете получить ссылку на контейнер, который сгенерировал событие, используя метод `getContainer()`:

```
Container getContainer()
```

Метод `getChild()` возвращает ссылку на компонент, который был добавлен или удален из контейнера. Его общая форма:

```
Component getChild()
```

## Класс *FocusEvent*

Событие *FocusEvent* генерируется, когда компонент получает или теряет фокус ввода. Эти события идентифицируются целочисленными константами *FOCUS\_GAINED* и *FOCUS\_LOST*.

*FocusEvent* — подкласс *ComponentEvent* и имеет конструкторы:

```
FocusEvent(Component src, int type)
```

```
FocusEvent(Component src, int type, boolean temporaryFlag)
```

где *src* — ссылка на компонент, который генерировал это событие; *type* — тип события. Параметр *temporaryFlag* устанавливается как *true*, если событие фокуса временное. Иначе — устанавливается как *false*.

### Замечание

Временное событие фокуса происходит в результате другой операции интерфейса пользователя. Например, предположим, что фокус находится в текстовом поле. Если пользователь перемещает мышь, чтобы откорректировать полосу прокрутки, фокус временно теряется.

Метод *isTemporary()* указывает, является ли это изменение фокуса временным. Его форма:

```
boolean isTemporary()
```

Метод возвращает *true*, если изменение временно. Иначе, он возвращает *false*.

## Класс *InputEvent*

Абстрактный класс *InputEvent* есть подкласс *ComponentEvent* и суперкласс для событий ввода компонентов. Его подклассы — *KeyEvent* и *MouseEvent*. Класс *InputEvent* определяет следующие восемь целочисленных констант, которые можно использовать для получения информации относительно любых модификаторов, связанных с этим событием:

- |   |                                       |
|---|---------------------------------------|
| <input type="checkbox"/> ALT_MASK       | <input type="checkbox"/> BUTTON3_MASK |
| <input type="checkbox"/> ALT_GRAPH_MASK | <input type="checkbox"/> CTRL_MASK    |
| <input type="checkbox"/> BUTTON1_MASK   | <input type="checkbox"/> META_MASK    |
| <input type="checkbox"/> BUTTON2_MASK   | <input type="checkbox"/> SHIFT_MASK   |

Методы *isAltDown()*, *isAltGraphDown()*, *isControlDown()*, *isMetaDown()* и *isShiftDown()* проверяют, были ли нажаты эти модификаторы в то время, когда событие было сгенерировано. Форматы этих методов:

```
boolean isAltDown()
boolean isAltGraphDown()
boolean isControlDown()
```

```
boolean isMetaDown()
boolean isShiftDown()
```

Метод `getModifiers()` возвращает значение, которое содержит все флагги модификаторов для этого события. Его сигнатурра имеет вид:

```
int getModifiers()
```

## Класс *ItemEvent*

События `ItemEvent` генерируются при установке/сбросе флашка или щелчке по элементу списка, или когда пользователь выполняет (или отменяет) выбор пункта меню с меткой. (Флашки и списковые панели описаны дальше.) Существует два типа item-событий, которые идентифицированы целочисленными константами:

- `DESELECTED`. Пользователь отменил выбор элемента.
- `SELECTED`. Пользователь выбрал элемент.

Кроме того, `ItemEvent` определяет целочисленную константу `ITEM_STATE_CHANGED`, которая обозначает изменение состояния.

Класс `ItemEvent` имеет конструктор:

```
ItemEvent(ItemSelectable src, int type, Object entry, int state)
```

где `src` — ссылка на компонент, который генерировал это событие, например, им мог бы быть элемент типа "выбор" или "список"; `type` — определяет тип события; `entry` — передает конструктору определенный элемент, который генерировал item-событие; `state` — текущее состояние этого элемента.

Метод `getItem()` можно использовать для получения ссылки на элемент, который генерировал событие. Его сигнатурра:

```
Object getItem()
```

Метод `getItemSelectable()` можно использовать для получения ссылки на объект `ItemSelectable`, который генерировал событие. Его общая форма:

```
ItemSelectable getItemSelectable()
```

Примерами элементов интерфейса пользователя, которые реализует интерфейс `ItemSelectable`, являются (обычные) списки и списки с выбором.

Метод `getStateChange()` возвращает изменение состояния события (`SELECTED` или `DESELECTED`). Формат метода:

```
Int getStateChange()
```

## Класс *KeyEvent*

Событие `KeyEvent` генерируется, когда происходит ввод с клавиатуры. Имеются три типа key-событий, которые идентифицируются целочисленными

константами: KEY\_PRESSED, KEY\_RELEASED и KEY\_TYPED. Первые два события генерируются, когда любая клавиша нажимается или отпускается. Последнее событие происходит только при нажатии символьной клавиши. Напоминаем, что не все нажатия клавиш приводят к вводу символа. Например, нажатие клавиши <Shift> не генерирует символ.

Существуют и другие целочисленные константы, которые определены в классе KeyEvent. Например, VK\_0–VK\_9 и VK\_A–VK\_Z определяют эквиваленты ASCII-цифр и букв. Вот некоторые из них:

- |                                     |                                       |                                     |
|-------------------------------------|---------------------------------------|-------------------------------------|
| <input type="checkbox"/> VK_ALT     | <input type="checkbox"/> VK_ENTER     | <input type="checkbox"/> VK_PAGE_UP |
| <input type="checkbox"/> VK_CANCEL  | <input type="checkbox"/> VK_ESCAPE    | <input type="checkbox"/> VK_RIGHT   |
| <input type="checkbox"/> VK_CONTROL | <input type="checkbox"/> VK_LEFT      | <input type="checkbox"/> VK_SHIFT   |
| <input type="checkbox"/> VK_DOWN    | <input type="checkbox"/> VK_PAGE_DOWN | <input type="checkbox"/> VK_UP      |

Константы VK определяют *коды виртуальных клавиши* (virtual key codes) и не зависят от любых модификаторов, таких как <Ctrl>, <Shift> или <Alt>.

Класс KeyEvent является подклассом InputEvent и имеет два конструктора:

```
KeyEvent(Component src, int type, long when, int modifiers, int code)
KeyEvent(Component src, int type, long when, int modifiers, int code,
          char ch)
```

где *src* — ссылка на компонент, который генерировал это событие; *type* — тип события; *when* — параметр, передающий конструктору системное время, когда была нажата клавиша; *modifiers* — параметр, указывающий, какие модификаторы были нажаты, когда данное клавишное событие произошло; *code* — параметр, передающий конструктору код виртуальной клавиши VK\_UP, VK\_A и т. д. Параметр *ch* передает эквивалент символа (если он существует). Если никакой допустимый символ не существует, то *ch* содержит CHAR\_UNDEFINED. Для событий KEY\_TYPED параметр *code* будет содержать VK\_UNDEFINED.

Класс KeyEvent определяет несколько методов, но обычно используются getKeyChar(), возвращающий символ, который был введен, и getKeyCode(), который возвращает код клавиши. Их общие формы:

```
char getKeyChar()
int getKeyCode()
```

Если никакой допустимый символ не доступен, то getKeyChar() возвращает CHAR\_UNDEFINED. Когда происходит событие KEY\_TYPED, getKeyCode() возвращает VK\_UNDEFINED.

## Класс MouseEvent

Существует семь типов mouse-событий. Класс MouseEvent определяет целочисленные константы, которые могут использоваться для их идентификации:

- MOUSE\_CLICKED**. Пользователь щелкнул кнопкой мыши.
- MOUSE\_DRAGGED**. Пользователь перетащил мышь.
- MOUSE\_ENTERED**. Указатель мыши введен в компонент.
- MOUSE\_EXITED**. Указатель мыши выведен из компонента.
- MOUSE\_MOVED**. Мыши передвинута.
- MOUSE\_PRESSED**. Кнопка мыши нажата.
- MOUSE\_RELEASED**. Кнопка мыши освобождена.

MouseEvent — это подкласс InputEvent. Он имеет конструктор:

```
MouseEvent(Component src, int type, long when, int modifiers,  
           int x, int y, int clicks, boolean triggersPopup)
```

где **src** — ссылка на компонент, который генерировал это событие; **type** — тип события; **when** — параметр, передающий конструктору системное время, когда клавиша была нажата; **modifiers** — параметр, указывающий, какие модификаторы были нажаты, когда произошло mouse-событие. Координаты мыши передаются параметрами **x** и **y**. Счет щелчков передается в **clicks**. Флажок **triggersPopup** указывает, приводит ли это событие к появлению раскрывающегося меню на данной платформе.

Обычно используемые методы в этом классе — `getX()` и `getY()`. Они возвращают (x, y)-координаты мыши, когда событие произошло. Их форматы:

```
int getX()  
int getY()
```

Альтернативно, чтобы получить координаты мыши, можно использовать метод `getPoint()` с форматом:

```
Point getPoint()
```

Он возвращает объект `Point`, который содержит (x, y)-координаты в его целых членах **x** и **y**. Метод `translatePoint()` изменяет положение события. Его формат:

```
void translatePoint(int x, int y)
```

Здесь аргументы **x** и **y** добавляются к координатам события.

Метод `getClickCount()` получает число щелчков мыши для этого события. Его сигнатура:

```
int getClickCount()
```

Метод `isPopupTrigger()` проверяет, приводит ли событие к появлению всплывающего меню на данной платформе. Его формат:

```
boolean isPopupTrigger()
```

## Класс *TextEvent*

Экземпляры этого класса описывают события, генерируемые текстовыми полями и текстовыми областями, когда пользователь или программа вводят в них символы. В *TextEvent* определена целая константа *TEXT\_VALUE\_CHANGED*.

Один из конструкторов указанного класса:

```
TextEvent(Object src, int type)
```

где *src* — ссылка на компонент, который генерировал это событие; *type* — тип события.

Объект *TextEvent* не включает немедленно символы в текстовый компонент, который сгенерировал событие. Чтобы извлечь эту информацию, программа должна использовать другие методы. Данная операция отличается от других событийных объектов, рассмотренных в текущем разделе. По этой причине здесь не обсуждаются никакие методы класса *TextEvent*. Уведомление *text*-события нужно представлять себе как сигнал, извещающий блок прослушивания о том, что он должен сам извлечь информацию из определенного текстового компонента.

## Класс *WindowEvent*

Существует семь типов *window*-событий. Чтобы идентифицировать их, класс *WindowEvent* определяет следующие целые константы:

- WINDOW\_ACTIVATED*. Окно активизировано.
- WINDOW\_CLOSED*. Окно закрыто.
- WINDOW\_CLOSING*. Пользователь потребовал, чтобы окно было закрыто.
- WINDOW\_DEACTIVATED*. Окно деактивизировано.
- WINDOW\_DEICONIFIED*. Окно развернуто из пиктограммы.
- WINDOW\_ICONIFIED*. Окно свернуто в пиктограмму.
- WINDOW\_OPENED*. Окно открыто.

*WindowEvent* — подкласс *ComponentEvent* и имеет следующий конструктор:

```
WindowEvent(Window src, int type)
```

где *src* — ссылка на компонент, который генерировал это событие; *type* указывает тип события.

В этом классе чаще всего используется метод *getWindow()*. Он возвращает *window*-объект, который сгенерировал событие. Его общая форма:

```
Window getWindow()
```

## Элементы-источники событий

Некоторые из компонентов интерфейса пользователя, которые могут генерировать события, описанные в предыдущем разделе, перечислены в табл. 20.2. В дополнение к этим элементам графического интерфейса пользователя, события могут генерировать другие компоненты, такие как апплеты. Например, от апплета принимаются key- и mouse-события. (Можно также строить свои собственные компоненты, которые генерируют события.) В этой главе мы будем обрабатывать только события мыши и клавиатуры, но в следующих двух главах будет рассмотрена обработка событий из источников, показанных в табл. 20.2.

**Таблица 20.2. Примеры источников событий**

Источник события	Описание
Кнопка (Button)	Генерирует action-события, когда нажимается кнопка
Флажок (Checkbox)	Генерирует item-события, когда флажок устанавливается или сбрасывается
Список с выбором (Choice)	Генерирует item-события, когда изменяется выбор элемента в списке с выбором
Список (List)	Генерирует action-события, когда на элементе списка выполнен двойной щелчок (мышью). Генерирует item-события, когда элемент выделяется или выделение снимается
Пункт меню (Menu Item)	Генерирует action-события, когда пункт меню выделен; генерирует события элемента, когда пункт меню с меткой выделен или выделение отменяется
Полоса прокрутки (Scrollbar)	Генерирует adjustment-события при манипуляциях с полосой прокрутки
Текстовые компоненты	Генерирует text-события, когда пользователь вводит символ
Окно (window)	Генерирует window-события, когда окно активизируется, закрывается, деактивизируется, сворачивается в пиктограмму, разворачивается из пиктограммы, открывается или выполняется выход из него (quit)

## Интерфейсы прослушивания событий

Модель делегирования событий содержит две части: источники событий и блоки прослушивания событий. Блоки прослушивания событий создаются пу-

тем реализации одного или нескольких интерфейсов прослушивания событий, определяемых пакетом `java.awt.event`. Когда событие происходит, источник события вызывает соответствующий метод, определенный блоком прослушивания, и передает ему объект события в качестве параметра. Табл. 20.3 перечисляет используемые чаще всего интерфейсы прослушивания и приводит краткое описание методов, которые эти блоки прослушивания определяют.

**Таблица 20.3. Интерфейсы прослушивания событий**

Интерфейс	Описание
ActionListener	Определяет (один) метод для приема action-события
AdjustmentListener	Определяет (один) метод для приема adjustment-события
ComponentListener	Определяет четыре метода, распознающих события, связанные со скрытием, перемещением, изменением и показом компонента
ContainerListener	Определяет два метода, распознающих события добавления или удаления компонента из контейнера
FocusListener	Определяет два метода, распознающих события приобретения или потери компонентом фокуса клавиатуры
ItemListener	Определяет (один) метод, распознающий события изменения состояний элемента
KeyListener	Определяет три метода, распознающих события нажатия, отпускания и ввода символа клавиши
MouseListener	Определяет пять методов, распознающих события щелчка, входа в границы компонента, выхода из границ компонента, нажатия и отпускания клавиши мыши
MouseMotionListener	Определяет два метода, распознающих события перетаскивания или перемещения мыши
TextListener	Определяет (один) метод, распознающий события изменения текстового значения
WindowListener	Определяет семь методов, распознающих события активизации, деактивизации, открытия, закрытия, сворачивания/разворачивания (в значок) и выхода из окна

Следующие разделы рассматривают специфические методы, которые содержатся в каждом интерфейсе.

## Интерфейс *ActionListener*

Этот интерфейс определяет метод `actionPerformed()`, который вызывается при наступлении action-события. Его общая форма:

```
void actionPerformed(ActionEvent ae)
```

## Интерфейс *AdjustmentListener*

Этот интерфейс определяет метод `adjustmentValueChanged()`, который вызывается при наступлении adjustment-события. Его общая форма:

```
void adjustmentValueChanged(AdjustmentEvent ae)
```

## Интерфейс *ComponentListener*

Этот интерфейс определяет четыре метода, которые вызываются, когда компонент изменяется в размерах, перемещается, отображается на экране или скрывается. Их общие формы:

```
void componentResized(ComponentEvent ce)
void componentMoved(ComponentEvent ce)
void componentShown() (ComponentEvent ce)
void componentHidden(ComponentEvent ce)
```

### Замечание

AWT обрабатывает события изменения размеров и перемещения. Методы `componentResized()` и `componentMoved()` обеспечены только для целей уведомления.

## Интерфейс *ContainerListener*

Этот интерфейс содержит два метода. Когда компонент добавляется к контейнеру, вызывается метод `componentAdded()`. Когда компонент удаляется из контейнера, вызывается метод `componentRemoved()`. Их общие формы:

```
void componentAdded(ContainerEvent ce)
void componentRemoved(ContainerEvent ce)
```

## Интерфейс *FocusListener*

Этот интерфейс определяет два метода. Когда компонент получает фокус клавиатуры, вызывается метод `focusGained()`. Когда компонент теряет фокус клавиатуры, вызывается метод `focusLost()`. Их общие формы:

```
void focusGained(FocusEvent fe)
void focusLost(FocusEvent fe)
```

## Интерфейс *ItemListener*

Этот интерфейс определяет метод `itemStateChanged()`, который вызывается при изменении состояния элемента. Его общая форма:

```
void itemStateChanged(ItemEvent ie)
```

## Интерфейс *KeyListener*

Этот интерфейс определяет три метода. Методы `keyPressed()` и `keyReleased()` вызываются, когда клавиша нажата и отпущена, соответственно. Метод `keyTyped()` вызывается при вводе символа.

Например, если пользователь нажимает и отпускает буквенную клавишу `<A>`, последовательно генерируются три события — "клавиша нажата", "символ введен" и "клавиша отпущена". Если пользователь нажимает и отпускает клавишу `<Home>`, последовательно генерируются два события — "клавиша нажата" и "клавиша отпущена".

Общие формы этих методов:

```
void keyPressed(KeyEvent ke)
void keyReleased(KeyEvent ke)
void keyTyped(KeyEvent ke)
```

## Интерфейс *MouseListener*

Этот интерфейс определяет пять методов. Если кнопка мыши нажата и сразу же отпущена, вызывается метод `mouseClicked()`. Когда указатель мыши входит в границы компонента, вызывается метод `mouseEntered()`, а когда выходит — вызывается метод `mouseExited()`. Методы `mousePressed()` и `mouseReleased()` вызываются, когда кнопка мыши нажимается и отпускается, соответственно.

Общие формы этих методов:

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
```

## Интерфейс *MouseMotionListener*

Этот интерфейс определяет два метода. Метод `mouseDragged()` вызывается много раз, когда мышь перетаскивается (`dragged`). Метод `mouseMoved()` вызывается много раз, когда мышь перемещается (`moved`). Их общие формы:

```
void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)
```

## Интерфейс *TextListener*

Этот интерфейс определяет метод `textChanged()`, который вызывается при изменении содержимого текстовой области или текстового поля. Его общая форма:

```
void textChanged(TextEvent te)
```

## Интерфейс *WindowListener*

Этот интерфейс определяет семь методов. Методы `windowActivated()` и `windowDeactivated()` вызываются, когда окно активизируется или деактивируется, соответственно. Если окно сворачивается в пиктограмму, вызывается метод `windowIconified()`. Когда окно разворачивается из пиктограммы, вызывается метод `windowDeiconified()`. Когда окно открывается или закрывается, вызываются методы `windowOpened()` или `windowClosed()`, соответственно. Метод `windowClosing()` вызывается, когда обнаруживается, что окно закрыто. Общие формы этих методов:

```
void windowActivated(WindowEvent we)
void windowClosed(WindowEvent we)
void windowClosing(WindowEvent we)
void windowDeactivated(WindowEvent we)
void windowDeiconified(WindowEvent we)
void windowIconified(WindowEvent we)
void windowOpened(WindowEvent we)
```

## Использование модели делегирования событий

Теперь, когда изучена теория, которая кроется за моделью делегирования событий, и сделан краткий обзор различных ее компонентов, пришло время увидеть ее на практике. Программирование апплетов, использующее модель делегирования событий, является достаточно простым делом. Выполняйте всегда два следующих шага:

1. Реализуйте соответствующий интерфейс в блоке прослушивания, чтобы он принимал события желаемого типа.
2. Реализуйте код для регистрации (или не регистрации, если это необходимо) блока прослушивания в качестве получателя уведомлений о событии.

Помните, что источник может генерировать несколько типов событий. Каждое событие должно быть зарегистрировано отдельно. Объект тоже может регистрироваться, чтобы принимать несколько типов событий, но он должен реализовать все интерфейсы, которые требуются для приема этих событий.

Чтобы увидеть, как модель делегирования работает на практике, рассмотрим примеры, демонстрирующие обработку событий двух чаще всего используемых генераторов — мыши и клавиатуры.

## Обработка событий мыши

Для обработки событий мыши (*mouse-события*) нужно реализовать интерфейсы *MouseListener* и *MouseMotionListener*. Следующий аплет представляет этот процесс. Он отображает текущие координаты мыши в окне состояния аплета. Каждый раз, когда кнопка нажимается, в месте расположения указателя отображается слово "Down". Каждый раз, когда кнопка отпускается, показывается слово "Up". Если произведен щелчок кнопкой мыши, в левом верхнем углу области показа аплета выводится сообщение "Mouse clicked".

Когда указатель мыши входит или выходит из окна апплета, в левом верхнем углу области апплета выводится сообщение "Mouse entered". При перетаскивании мыши показывается символ \*, который отслеживает указатель мыши. Обратите внимание, что две переменные, `mouseX` и `mouseY`, хранят положение курсора мыши, когда происходят события нажатия, освобождения кнопки или перетаскивания мыши. Эти координаты затем используются методом `paint()`, чтобы отобразить вывод в той точке, где эти события происходят.

```
    addMouseMotionListener(this);      // регистрация (себя) как блока
}                                     // прослушивания MouseMotion-событий

// Обработка щелчка мыши.
public void mouseClicked(MouseEvent me) {

    mouseX = 0;
    mouseY = 10;
    msg = "Mouse clicked.";
    repaint();
}

// Обработка ввода мыши в область окна.
public void mouseEntered(MouseEvent me) {
    // сохранить координаты
    mouseX = 0;
    mouseY = 10;
    msg = "Mouse entered.";
    repaint();
}

// Обработка вывода мыши из области окна.
public void mouseExited(MouseEvent me) {
    // сохранить координаты
    mouseX = 0;
    mouseY = 10;
    msg = "Mouse exited.";
    repaint();
}

// Обработка нажатия кнопки.
public void mousePressed(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down";
    repaint();
}

// Обработка освобождения кнопки.
public void mouseReleased(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Up";
    repaint();
}

// Обработка перетаскивания мыши.
public void mouseDragged(MouseEvent me) {
```

```

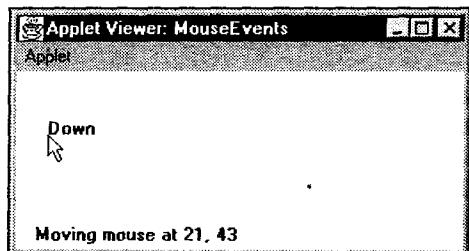
// сохранить координаты
mouseX = me.getX();
mouseY = me.getY();
msg = "*";
showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
repaint();
}

// Обработка перемещения мыши.
public void mouseMoved(MouseEvent me) {
    // отобразить состояние
    showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
}

// Вывести msg в окне апплета в положении (x, y)-координат.
public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
}
}
}

```

Пример вывода этой программы представлен на рис. 20.1.



**Рис. 20.1.** Окно апплета  
MouseEvents

Рассмотрим этот пример подробнее. Класс `MouseEvents` расширяет `Applet` и реализует как интерфейс `MouseMotionListener`, так и интерфейс `MouseListener`. Оба интерфейса содержат методы, которые принимают и обрабатывают различные типы событий мыши. Обратите внимание, что апплет является как источником, так и блоком прослушивания для этих событий (это общая ситуация для любых апплетов), т. к. `Component`, который обеспечивает методы `addMouseListener()` и `addMouseMotionListener()`, является суперклассом для `Applet`.

Внутри `init()` апплет регистрирует сам себя как блок прослушивания событий мыши. Это организуется с помощью методов `addMouseListener()` и `addMouseMotionListener()`, которые являются членами класса `Component`:

```

synchronized void addMouseListener(MouseListener ml)
synchronized void addMouseMotionListener(MouseMotionListener mml)

```

где `m1` — ссылка на объект, принимающий событие мыши; `mm1` — ссылка на объект, принимающий события движения мыши. В этой программе для обеих ссылок используется один и тот же объект.

Затем апллет реализует все методы, определенные интерфейсами `MouseMotionListener` и `MouseListener`. Они являются обработчиками для различных событий мыши. Каждый метод обрабатывает свое событие и затем возвращает управление.

## Обработка событий клавиатуры

Чтобы обрабатывать события клавиатуры, используется та же общая архитектура, что показана в примере с событиями мыши в предыдущем разделе. Различие, конечно, в том, что нужно реализовывать интерфейс `KeyListener`.

Перед примером полезно посмотреть, как генерируются key-события. Когда клавиша нажимается, генерируется событие `KEY_PRESSED`. Это приводит к запросу обработчика события `keyPressed()`. Когда клавиша отпускается, генерируется событие `KEY_RELEASED` и выполняется обработчик `keyReleased()`. Если нажатием клавиши сгенерирован символ, то посыпается уведомление о событии `KEY_TYPED` и вызывается обработчик `keyTyped()`. Таким образом, каждый раз, когда пользователь нажимает клавишу, генерируется по крайней мере два, а часто — три события. Если все, о чем вы заботитесь — фактические символы, то можно игнорировать информацию, которую посыпается нажатие клавиш. Однако если ваша программа должна обработать специальные клавиши, типа клавиш управления курсором или функциональных клавиш, то она должна наблюдать за ними через обработчик `keyPressed()`.

Предъявляется еще одно требование, которое должна выполнить программа прежде, чем она сможет обработать события клавиатуры: она должна запросить фокус ввода. Чтобы сделать это, вызовите `requestFocus()`, который определен в классе `Component`. Если вы этого не сделаете, то программа не будет принимать никаких событий клавиатуры.

Следующий пример демонстрирует ввод с клавиатуры. Он отображает в окне апллета символы клавиш и показывает состояние нажатия/освобождения каждой клавиши в окне состояния.

```
// Демонстрирует обработчики клавищных событий.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SimpleKey" width=300 height=100>
</applet>
```

```

public class SimpleKey extends Applet
    implements KeyListener {

    String msg = "";
    int X = 10, Y = 20; // координаты вывода

    public void init() {
        addKeyListener(this); // регистрация (себя) как блока
        // прослушивания key-событий
        requestFocus(); // запрос фокуса ввода
    }

    public void keyPressed(KeyEvent ke) {
        showStatus("Key Down");
    }

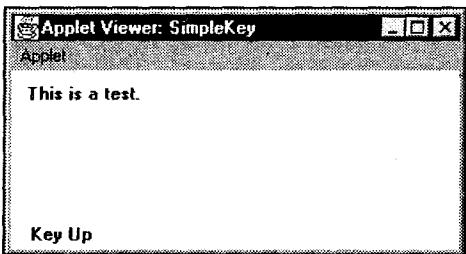
    public void keyReleased(KeyEvent ke) {
        showStatus("Key Up");
    }

    public void keyTyped(KeyEvent ke) {
        msg += ke.getKeyChar();
        repaint();
    }

    // Показывает нажатую клавишу в позиции указателя мыши.
    public void paint(Graphics g) {
        g.drawString(msg, X, Y);
    }
}

```

Вывод примера представлен на рис. 20.2.



**Рис. 20.2.** Окно апплета SimpleKey

Если нужно обработать специальные клавиши типа клавиш перемещения курсора или функциональных клавиш, следует откликнуться на них в обработчике keyPressed(). Они недоступны через keyTyped(). Чтобы идентифицировать данные клавиши, используйте код их виртуальной клавиши. Например, следующий апплет выводит имя нескольких специальных клавиши:

```
// Демонстрирует некоторые коды виртуальных клавиш.  
import java.awt.*;  
import java.awt.event.*;  
import java.applet.*;  
/*  
 <applet code="KeyEvents" width=300 height=100>  
 </applet>  
*/  
  
public class KeyEvents extends Applet  
    implements KeyListener {  
  
    String msg = "";  
    int X = 10, Y = 20; // координаты вывода  
  
    public void init() {  
        addKeyListener(this); // регистрация блока прослушивания  
        requestFocus(); // запрос фокуса ввода  
    }  
  
    public void keyPressed(KeyEvent ke) {  
        showStatus("Key Down");  
  
        int key = ke.getKeyCode();  
        switch(key) {  
            case KeyEvent.VK_F1:  
                msg += "<F1>";  
                break;  
            case KeyEvent.VK_F2:  
                msg += "<F2>";  
                break;  
            case KeyEvent.VK_F3:  
                msg += "<F3>";  
                break;  
            case KeyEvent.VK_PAGE_DOWN:  
                msg += "<PgDn>";  
                break;  
            case KeyEvent.VK_PAGE_UP:  
                msg += "<PgUp>";  
                break;  
            case KeyEvent.VK_LEFT:  
                msg += "<Left Arrow>";  
                break;  
            case KeyEvent.VK_RIGHT:  
                msg += "<Right Arrow>";  
                break;  
        }  
        repaint();  
    }  
}
```

```

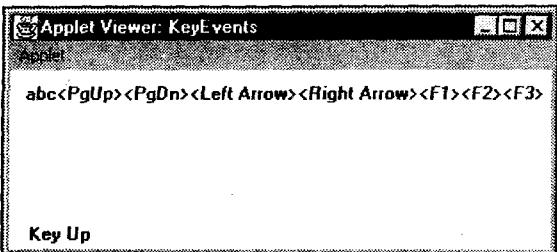
public void keyReleased(KeyEvent ke) {
    showStatus("Key Up");
}

public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
}

// Показывает нажатую клавишу в позиции указателя мыши.
public void paint(Graphics g) {
    g.drawString(msg, X, Y);
}
}

```

Вывод примера представлен на рис. 20.3.



**Рис. 20.3.** Окно апплета KeyEvents

Процедуры, показанные в предыдущих примерах с событиями клавиатуры и мыши, могут быть обобщены на любой тип обработки событий, включая события, сгенерированные элементами управления. Следующие главы продемонстрируют много примеров, обрабатывающих другие типы событий, но они будут все следовать той же основной структуре, которая описана в предыдущих программах.

## Классы-адаптеры

Java обеспечивает специальное средство, называемое *классом адаптера* (adapter class), которое может упростить в некоторых ситуациях создание обработчиков событий. Класс адаптера (или просто "адаптер") обеспечивает пустую реализацию всех методов в интерфейсе прослушивания событий. Класс полезен, когда вы хотите принимать и обрабатывать только некоторые из событий, предоставляемые конкретным интерфейсом прослушивания. Для этого нужно определить новый класс, действующий как блок прослушивания событий, расширяя один из имеющихся (в пакете `java.awt.event`) адаптеров и реализуя только те события, которые требуется обрабатывать.

Например, класс `MouseMotionAdapter` имеет два метода `mouseDragged()` и `mouseMoved()`. Сигнатуры этих пустых методов точно такие же, как в интер-

файле MouseMotionListener. Если бы вы были заинтересованы только в событиях перетаскивания мыши, то могли бы просто расширить адаптер MouseMotionAdapter и реализовать метод mouseDragged(). События же перемещения мыши обрабатывала бы пустая реализация метода mouseMoved().

Табл. 20.4 перечисляет различные классы-адаптеры, которые определены в пакете java.awt.event, и указывает те интерфейсы прослушивания, которые каждый из них реализует.

**Таблица 20.4.** Интерфейсы прослушивания событий, реализованные с помощью классов адаптеров

Класс-адаптер	Интерфейс прослушивания событий
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Следующий пример демонстрирует использование класса-адаптера. Он отображает сообщение в строке состояния программы просмотра апплета или браузера, когда выполняется перетаскивание мыши или щелчок. Однако все другие события мыши по умолчанию игнорируются. Программа содержит три класса. Класс AdapterDemo расширяет Applet. Его метод init() создает экземпляр MyMouseAdapter и регистрирует этот объект для приема уведомлений от mouse-событий. Он также создает экземпляр MyMouseMotionAdapter и регистрирует этот объект для приема уведомлений событий MouseMotion (движения мыши). В качестве аргумента оба конструктора получают ссылку на данный апплет.

Класс MyMouseAdapter реализует метод mouseClicked(). Другие события мыши "молча" игнорируются кодом, унаследованным от класса MouseAdapter.

Класс MyMouseMotionAdapter реализует метод mouseDragged(). Другое событие движения мыши игнорируется кодом, унаследованным от класса MouseMotionAdapter.

Обратите внимание, что оба класса прослушивания событий хранят ссылку на данный апплет. С помощью ссылки this эта информация передается в оба конструктора и используется позже для вызова метода showStatus().

```
// Демонстрирует адаптер.
import java.awt.*;
```

```

import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/
public class AdapterDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter(this));
        addMouseMotionListener(new MyMouseMotionAdapter(this));
    }
}
class MyMouseAdapter extends MouseAdapter {
    AdapterDemo adapterDemo;
    public MyMouseAdapter(AdapterDemo adapterDemo) {
        this.adapterDemo = adapterDemo;
    }
    // Обработка щелчка мыши.
    public void mouseClicked(MouseEvent me) {
        adapterDemo.showStatus("Mouse clicked");
    }
}
class MyMouseMotionAdapter extends MouseMotionAdapter {
    AdapterDemo adapterDemo;
    public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
        this.adapterDemo = adapterDemo;
    }
    // Обработка перетаскивания мыши.
    public void mouseDragged(MouseEvent me) {
        adapterDemo.showStatus("Mouse dragged");
    }
}

```

Глядя на программу, можно заметить, что отказ от необходимости реализовывать все методы, определенные интерфейсами `MouseMotionListener` и `MouseListener` значительно сокращает объем программы, освобождая код от загромождения пустыми методами. В качестве упражнения можно было бы попробовать переписать один из примеров ввода с клавиатуры, показанный ранее так, чтобы он использовал класс `KeyAdapter`.

## Внутренние классы

В главе 7 были объяснены основы внутренних классов. Здесь можно увидеть, почему они важны. Напомним, что *внутренний класс* — это класс, оп-

ределенный внутри другого класса, или даже внутри выражения. Этот раздел иллюстрирует, как можно использовать внутренние классы, чтобы упростить код применяющий адаптеры событий.

Чтобы понять выгоду, обеспечиваемую внутренними классами, рассмотрим апплет, показанный в следующем листинге. Он *не использует* внутренний класс. Его цель состоит в том, чтобы отобразить строку "Mouse Pressed" в строке состояния программы просмотра апплета или браузера, когда нажимается клавиша мыши. В этой программе имеются два класса верхнего уровня. Класс MousePressedDemo расширяет класс Applet, а MyMouseAdapter расширяет MouseAdapter. Метод init() класса MousePressedDemo создает экземпляр (объект) класса MyMouseAdapter и передает его (как аргумент) методу addMouseListener().

Обратите внимание, что в качестве аргумента конструктора MyMouseAdapter указана *ссылка* на данный апплет. Она сохраняется в переменной экземпляра для более позднего использования методом mousePressed(). Когда нажимается кнопка мыши, через эту ссылку вызывается метод апплета showStatus(). Другими словами, showStatus() вызывается относительно ссылки на апплет, сохраненной адаптером MyMouseAdapter.

```
// Этот апплет не использует внутренний класс.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="MousePressedDemo" width=200 height=100>
</applet>
*/
public class MousePressedDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter(this));
    }
}

class MyMouseAdapter extends MouseAdapter {
    MousePressedDemo mousePressedDemo;
    public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
        this.mousePressedDemo = mousePressedDemo;
    }
    public void mousePressed(MouseEvent me) {
        mousePressedDemo.showStatus("Mouse Pressed.");
    }
}
```

Следующий листинг демонстрирует, как предшествующая программа может быть улучшена, используя внутренний класс. Здесь InnerClassDemo — класс

верхнего уровня, который расширяет класс Applet. MyMouseAdapter — внутренний класс, который расширяет MouseAdapter. Поскольку MyMouseAdapter определен в области видимости InnerClassDemo, он имеет доступ ко всем переменным и методам в пределах этого класса. Поэтому метод mousePressed() может прямо вызвать метод showStatus(). Больше не требуется делать это через сохраненную ссылку на апплет. Таким образом, нет необходимости передавать конструктору MyMouseAdapter() ссылку на вызывающий объект.

```
// Демонстрация внутреннего класса.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="InnerClassDemo" width=200 height=100>
</applet>
*/
public class InnerClassDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter());
    }
    class MyMouseAdapter extends MouseAdapter {
        public void mousePressed(MouseEvent me) {
            showStatus("Mouse Pressed");
        }
    }
}
```

## Анонимные внутренние классы

Анонимный внутренний класс — это класс, которому не назначено имя. Этот раздел иллюстрирует, как анонимный внутренний класс может облегчать написание обработчиков событий. Рассмотрим апплет, показанный в следующем листинге. Как и прежде, его цель состоит в том, чтобы отобразить строку "Mouse Pressed" в строке состояния программы просмотра апплета или браузера, когда нажимается кнопка мыши.

```
// Демонстрация анонимного внутреннего класса.
import java.applet.*;
import java.awt.event.*;

/*
<applet code="AnonymousInnerClassDemo" width=200 height=100>
</applet>
*/
public class AnonymousInnerClassDemo extends Applet {
    public void init() {
```

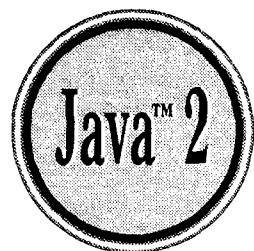
```
addMouseListener(new MouseAdapter() {  
    public void mousePressed(MouseEvent me) {  
        showStatus("Mouse Pressed");  
    }  
});  
}  
}
```

В этой программе представлен один класс верхнего уровня — `AnonymousInnerClassDemo`. Метод `init()` вызывает `addMouseListener()`. Его аргумент является выражением, которое определяет и строит экземпляр анонимного внутреннего класса. Давайте тщательно проанализируем это выражение.

Синтаксис `new MouseAdapter(){...}` указывает компилятору, что код между фигурными скобками определяет анонимный внутренний класс. Кроме того, этот класс расширяет класс `MouseAdapter`. Данный новый класс не именован, но он автоматически становится экземпляром класса, когда это выражение выполняется.

Поскольку этот анонимный внутренний класс определен в области видимости `AnonymousInnerClassDemo`, он имеет доступ ко всем переменным и методам в пределах области видимости этого класса. Поэтому он может прямо вызывать метод `showStatus()`.

Как только что было показано, как именованный, так и анонимный внутренние классы решают некоторые раздражающие проблемы простым и довольно эффективным способом. Мы видим, что они позволяют создавать более результативный код и экономят труд программиста (уменьшая объем исходного кода).



## ГЛАВА 21

# Введение в AWT: работа с окнами, графикой и текстом

Пакет AWT был представлен в главе 19, потому что он обеспечивает поддержку апплетов. В этой главе мы начинаем более глубокое его рассмотрение. AWT содержит многочисленные классы и методы, которые позволяют создавать окна и управлять ими. Полное описание AWT легко заполнило бы целую книгу. Поэтому невозможно описать подробно каждый метод, переменную экземпляра или класс, содержащийся в AWT. В этой и следующих двух главах объясняется вся техника, необходимая для эффективного использования AWT при создании собственных апплетов или автономных программ.

В данной главе вы научитесь создавать окна, управлять окнами и шрифтами, выводить текст и использовать графику. Глава 22 описывает различные элементы управления (такие как полосы прокрутки и кнопки), поддерживаемые AWT. Она также объясняет дальнейшие аспекты механизма обработки событий Java. Глава 23 рассматривает подсистемы изображений и мультиPLICATIONи AWT.

Хотя основное предназначение AWT состоит в поддержке окон апплета, ее можно также использовать для создания автономных окон, которые работают в среде GUI операционной системы Windows. Большинство примеров содержится в апплетах, так что для их выполнения нужно использовать программу просмотра апплетов или java-совместимый Web-браузер. Несколько примеров демонстрируют создание автономных оконных программ.

### Замечание

Если вы еще не читали главу 20, пожалуйста, сделайте это теперь. В ней дан обзор обработки событий, которые используются многими примерами в этой главе.

## Классы AWT

Классы AWT содержатся в пакете `java.awt`. Это один из самых больших пакетов Java. К счастью, он организован нисходящим, иерархическим способом, поэтому легок в понимании и использовании. Табл. 21.1 перечисляет некоторые из классов AWT.

**Таблица 21.1. Некоторые AWT-классы**

Класс	Описание
<code>AWTEvent</code>	Инкапсулирует AWT-события
<code>AWTEventMulticaster</code>	Рассыпает события множеству слушателей
<code>BorderLayout</code>	Менеджер граничной (Border) компоновки. Граничная компоновка использует пять компонентов: North, South, East, West и Center (Север, Юг, Восток, Запад и Центр)
<code>Button</code>	Создает элемент управления <i>командная кнопка</i>
<code>Canvas</code>	Пустое, свободное от семантики окно
<code>CardLayout</code>	Менеджер карточной (Card) компоновки. Карточная компоновка моделируют пронумерованную колоду карт. Показывается только карта, находящаяся сверху
<code>Checkbox</code>	Создает элемент управления <i>флажок</i>
<code>CheckboxGroup</code>	Создает группу элементов управления <i>флажок</i>
<code>CheckboxMenuItem</code>	Создает помеченный пункт меню
<code>Choice</code>	Создает раскрывающийся (pop-up) список
<code>Color</code>	Управляет цветами переносимым, независимым от платформы способом
<code>Component</code>	Абстрактный суперкласс для различных AWT-компонентов
<code>Container</code>	Подкласс <code>Component</code> , который может содержать другие компоненты
<code>Cursor</code>	Инкапсулирует растровый курсор
<code>Dialog</code>	Создает окно диалога
<code>Dimension</code>	Определяет измерения объекта. Ширина сохраняется в <code>width</code> , а высота — в <code>height</code>
<code>Event</code>	Инкапсулирует события
<code>EventQueue</code>	Организует очереди событий
<code>FileDialog</code>	Создает окно, из которого может быть выбран файл

Таблица 21.1 (продолжение)

Класс	Описание
FlowLayout	Менеджер поточной (Flow) компоновки. Поточная компоновка размещает компоненты слева направо, сверху вниз
Font	Инкапсулирует шрифт печати
FontMetrics	Инкапсулирует различную информацию, связанную с шрифтом. Эта информация помогает отображать текст в окне
Frame	Создает стандартное окно (фрейм), которое имеет строку заголовка, углы, изменяющие размеры и строку меню
Graphics	Инкапсулирует графический контекст. Этот контекст используется различными методами вывода для отображения вывода в окне
GraphicsDevice	Описывает графическое устройство типа экрана или принтера
GraphicsEnvironment	Описывает коллекцию доступных объектов классов Font и GraphicsDevice
GridBagConstraints	Определяет различные ограничения, касающиеся класса GridBagLayout
GridBagLayout	Менеджер ячеистой (Grid Bag) компоновки. Ячеистая компоновка отображает компоненты в ячейках, подчиненных ограничениям, указанным в GridBagConstraints
GridLayout	Менеджер сеточной (Grid) компоновки. Сеточная компоновка отображает компоненты в двумерной таблице
Image	Инкапсулирует графические изображения
Insets	Инкапсулирует границы контейнера
Label	Создает метку, которая отображает строку
List	Создает список, из которого пользователь может делать выбор. Подобен стандартному списку Windows
MediaTracker	Управляет объектами среды
Menu	Создает выпадающее (pull-down) меню
MenuBar	Создает строку меню
MenuComponent	Абстрактный класс, реализованный различными классами меню
MenuItem	Создает пункт меню
MenuShortcut	Инкапсулирует быструю клавишу (сочетание клавиш) для пункта меню

Таблица 21.1 (окончание)

Класс	Описание
Panel	Самый простой конкретный подкласс класса Container
Point	Инкапсулирует пару декартовых координат, сохраняемых в переменных x и y
Polygon	Инкапсулирует многоугольник
PopupMenu	Инкапсулирует раскрывающееся (pop-up) меню
PrintJob	Абстрактный класс, который представляет задание для печати
Rectangle	Инкапсулирует прямоугольник
Scrollbar	Создает элемент управления полоса прокрутки
ScrollPane	Контейнер, который обеспечивает горизонтальные и/или вертикальные полосы прокрутки для другого компонента
SystemColor	Содержит цвета GUI элементов управления окном, таких как окна, полосы прокрутки, текст и пр.
TextArea	Создает элемент управления с многострочным редактированием
TextComponent	Суперкласс для TextArea и TextField
TextField	Создает элемент управления с однострочным редактированием
Toolkit	Абстрактный класс, реализованный в AWT
Window	Создает окно без границы, строки меню и заголовка

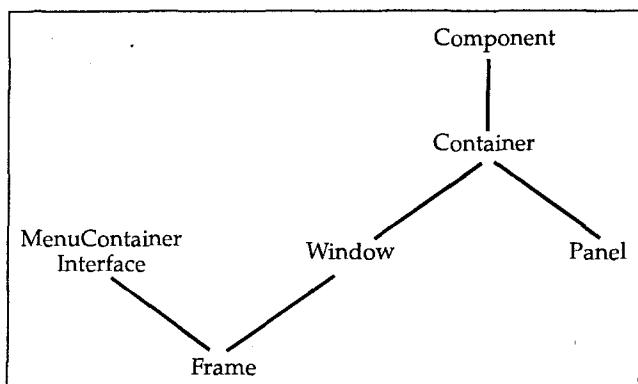
Хотя основная структура AWT остается неизменной, начиная с версии Java 1.0, в версии Java 1.1 некоторые из первоначальных методов были забракованы и заменены новыми. Для обратной совместимости, Java 2 все еще поддерживает все исходные методы версии 1.0. Однако в этой книге они не описываются, т. к. данные методы не предназначены для использования в новых Java-программах.

## Основы оконной графики

AWT определяет окна согласно иерархии классов, которая с каждым уровнем добавляет функциональные возможности и специфику. Два наиболее общих типа окон являются производными от типа Panel, который пользуется апплетами, и от типа Frame, который создает стандартное окно. Многое из функциональных возможностей этих окон получено от их родительских

классов. Описание иерархии, имеющей отношение к этим двум классам, фундаментально для их понимания. Иерархия классов, связанных с классами Panel и Frame, показана на рис. 21.1.

Рассмотрим каждый из этих классов.



**Рис. 21.1.** Иерархия классов для Panel и Frame

## Класс Component

Самый верхний в AWT-иерархии — класс Component. Это абстрактный класс, который инкапсулирует все атрибуты визуального компонента. Все элементы интерфейса пользователя, которые отображены на экране и взаимодействуют с пользователем, — это подклассы Component. В классе Component определено более сотни public-методов, которые являются ответственными за управление событиями, такими как ввод с помощью мыши и клавиатуры, позиционирование и изменение размеров окна, перерисовка и т. д. (многие из этих методов уже использовались при создании апплетов в главах 19 и 20). Объект класса Component отвечает за запоминание текущих цветов переднего плана и фона и выбранного текстового шрифта.

## Класс Container

Класс Container является подклассом Component. Он содержит дополнительные методы, которые позволяют вкладывать в него другие Component-объекты. Внутри класса Container могут храниться и его собственные объекты. Это делает Container *системой многоуровневого включения*<sup>1</sup>. Контейнер отвечает за размещение (т. е. позиционирование) любых компонентов, которые он содержит. Он делает это с помощью различных менеджеров компоновки (размещения), которые будут изучаться в главе 22.

<sup>1</sup> В СОМ-технологиях механизм *включения* (containment) позволяет одному объекту наследовать методы другого объекта с помощью простого вызова. — Примеч. пер.

## Класс *Panel*

Класс *Panel* — конкретный подкласс *Container*. Он не добавляет каких-либо новых методов. Это просто реализация класса *Container*. О *Panel* можно мыслить как о рекурсивно вкладываемом конкретном экранном компоненте. *Panel* — суперкласс для *Applet*. Когда экранный вывод направляется к апплету, он рисуется на поверхности объекта *Panel*. В сущности, объект *Panel* — это окно, которое не содержит области заголовка, строки меню и обрамления. Вот почему, когда апплет выполняется внутри браузера, вы не видите эти элементы. Если апплет выполняется с помощью утилиты просмотра апплетов (*appletviewer*), то заголовок и обрамление такого окна обеспечивается самой утилитой.

К *Panel*-объекту можно добавить другие компоненты с помощью метода *add()* (унаследованного из *Container*). Как только эти компоненты добавлены, вы можете позиционировать и изменять их размеры вручную, используя методы *setLocation()*, *setSize()* или *setBounds()*, определенные в *Component*.

## Класс *Window*

Класс *Window* создает окно верхнего уровня. *Окно верхнего уровня* не содержится в любом другом объекте. Оно находится непосредственно на Рабочем столе. Вообще, вы не будете создавать *window*-объекты непосредственно. Вместо этого, вы будете использовать подкласс *Window* с именем *Frame*.

## Класс *Frame*

Класс *Frame* инкапсулирует то, что обычно представляют как "окно"<sup>1</sup>. Это подкласс *Window* и его окно имеет строку заголовка, строку меню, обрамление и углы, изменяющие размеры окна. Если вы попытаетесь создать в апплете объект типа *Frame*, то он будет содержать сообщение вида "Warning: Applet Window", говорящее о том, что окно было создано апплете. Это сообщение предупреждает пользователей, что окно, которое они видят, было запущено апплете, а не программным обеспечением, выполняющимся на их компьютере. (Чтобы получать пароли и другую секретную информацию, неизвестную пользователям, можно было бы использовать апплет, который способен маскироваться под хост-приложение<sup>2</sup>.) Когда фрейм создается обычной программой, а не апплете, то строится нормальное окно.

<sup>1</sup> Окно этого класса далее будем называть просто *фреймом* (понимая под ним отформатированную прямоугольную область экрана в управляемой рамке с указанными компонентами). — *Примеч. пер.*

<sup>2</sup> *Хост (host-based) приложение* — это сетевое приложение, взаимодействующее с сетью, использующую TCP/IP-протоколы (в частности — с Internet). — *Примеч. пер.*

## Класс *Canvas*

Хотя *Canvas* не является частью *Container*-иерархии, существует еще один тип окна, который может оказаться полезным. Это — класс *Canvas*<sup>1</sup>. *Canvas*-объект моделирует *пустое окно*, в котором можно рисовать. Пример с *Canvas*-окном вы увидите позже в этой книге.

## Работа с фреймовыми окнами

Тип окна, который вы будете чаще всего создавать, является производным от *Frame*. Он используется для создания окон верхнего уровня и дочерних окон для апплетов и приложений. Как говорилось выше, *фрейм* — это окно со стандартным стилем (с заголовком, меню, обрамлением управляющими уголками). *Frame* поддерживает два конструктора:

```
Frame()
Frame(String заголовок)
```

Первая форма создает стандартное окно, которое не содержит заголовка. Вторая форма создает окно с заголовком, указанным в параметре *заголовок*. Обратите внимание, что вы не можете определить размеры окна, поэтому следует устанавливать размеры окна уже после того, как оно было создано.

Существует несколько методов, которые можно использовать при работе с *Frame*-окнами. Рассмотрим их.

## Установка размеров окна

Чтобы установить размеры окна, используется метод *setSize()*. Существует две формы этого метода (с разными списками параметров):

```
void setSize(int newWidth, int newHeight)
void setSize(Dimension newSize)
```

Новый размер окна специфицируется параметрами *newWidth* и *newHeight* (первая форма), или полями *width* и *height* объекта класса *Dimension*, передаваемыми параметру *newSize*. Размеры задаются в пикселях.

Метод *getSize()* используется для получения текущего размера окна. Его сигнатура:

```
Dimension getSize()
```

Данный метод возвращает текущий размер окна в полях *width* и *height* объекта класса *Dimension*.

<sup>1</sup> *Canvas* — в переводе с английского означает *полотно, холст, канва*, имея в виду пустую поверхность, на которую наносится изображение (художественными красками, вышивальными нитками и т. п.). — Примеч. пер.

## Скрытие и показ окна

После создания фрейм-окно остается невидимым до тех пор, пока вы не вызовете метод `setVisible()`. Сигнатура этого метода имеет вид:

```
void setVisible(boolean visibleFlag)
```

Компонент становится видимым, если параметр этого метода получает значение `true`, иначе он остается скрытым (невидимым).

## Установка заголовка окна

Можно изменить заголовок фрейм-окна, если вызвать метод `setTitle()`. Он имеет следующий формат:

```
void setTitle(String newTitle)
```

где `newTitle` — новый заголовок окна.

## Закрытие фрейм-окна

Когда фрейм-окно закрывается, программа должна удалить это окно с экрана, вызывая `setVisible()` с аргументом `false`:

```
...  
setVisible(false);  
...
```

Чтобы перехватить событие закрытия окна, нужно реализовать метод `windowClosing()` интерфейса `WindowListener`. Внутри `windowClosing()` необходимо удалить окно с экрана с помощью вызова `setVisible(false)`. Этую методику иллюстрирует пример, приведенный в следующем разделе.

## Создание фрейм-окна в апплете

Хотя и возможно построить фрейм-окно, просто создавая `Frame`-объект, но вы редко будете так поступать, потому что мало что сможете с ним сделать. Например, вы не будете способны принимать или обрабатывать события, которые происходят внутри него, или выводить в него информацию. Чаще всего вы будете создавать подкласс `Frame`. Это позволит переопределить методы класса `Frame` и обработку событий.

Создать новое фрейм-окно внутри апплета на самом деле очень просто. Сначала создайте подкласс `Frame`. Затем переопределите необходимые для работы с окном стандартные методы, такие как `init()`, `start()`, `stop()` и `paint()`. Наконец, реализуйте метод `windowClosing()` интерфейса `WindowListener`, вызывая `setVisible(false)`, чтобы закрыть окно.

Определив подкласс `Frame`, нужно создать объект этого класса. Однако вновь построенное фрейм-окно первоначально не будет видимым (на экране). Чтобы сделать его видимым, нужно вызвать `setVisible()` с аргументом `false`. При создании окна задают высоту и ширину по умолчанию. Чтобы установить необходимый размер окна, требуется вызвать метод `setSize()`.

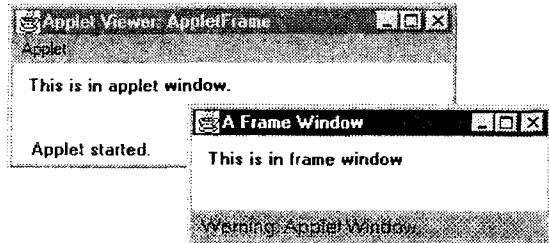
Следующий апплет создает подкласс `Frame` с именем `SampleFrame`. Оконный объект этого подкласса создается внутри метода `init()` класса `AppletFrame`. Обратите внимание, что `SampleFrame` вызывает `Frame`-конструктор. Он создает стандартное фрейм-окно с заголовком, передаваемым конструктору в аргументе `title`. Методы `start()` и `stop()` окна апплета переопределяются так, чтобы они показывали и скрывали дочернее окно. Это приводит к автоматическому удалению окна при завершении апплета, закрытии окна или при переходе к другой странице браузера. При возврате браузера к данному апплету показ дочернего окна возобновляется.

```
// Создает дочернее фрейм-окно внутри апплета.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AppletFrame" width=300 height=50>
</applet>
*/
// Создать подкласс Frame.
class SampleFrame extends Frame {
    SampleFrame(String title) {
        super(title);
        // создать объект для обработки window-событий
        MyWindowAdapter adapter = new MyWindowAdapter(this);
        // зарегистрировать его для приема этих событий
        addWindowListener(adapter);
    }
    public void paint(Graphics g) {
        g.drawString("This is in frame window", 10, 40);
    }
}

class MyWindowAdapter extends WindowAdapter {
    SampleFrame sampleFrame;
    public MyWindowAdapter(SampleFrame sampleFrame) {
        this.sampleFrame = sampleFrame;
    }
    public void windowClosing(WindowEvent we) {
        sampleFrame.setVisible(false);
    }
}
```

```
// Создать фрейм-окно.
public class AppletFrame extends Applet {
    Frame f;
    public void init() {
        f = new SampleFrame("A Frame Window");
        f.setSize(250, 250);
        f.setVisible(true);
    }
    public void start() {
        f.setVisible(true);
    }
    public void stop() {
        f.setVisible(false);
    }
    public void paint(Graphics g) {
        g.drawString("This is in applet window", 10, 20);
    }
}
```

Пример вывода этой программы представлен на рис. 21.2.



**Рис. 21.2.** Окно апплета AppletFrame и дочернее фрейм-окно

## Обработка событий фрейм-окна

Класс Frame наследует все возможности своего суперкласса (т. е. класса Component). Это означает, что вы можете управлять (и пользоваться) создаваемым фрейм-окном точно так же, как вы управляете главным окном апплета. Например, можно переопределить paint(), чтобы отобразить вывод, вызвать repaint(), когда нужно восстановить окно, а также переопределить все обработчики событий. Всякий раз, когда происходит событие, связанное с окном, будут вызываться обработчики событий, определенные для этого окна. Каждое окно обрабатывает свои собственные события. Например, следующая программа создает окно, которое откликается на события мыши. Главное окно апплета также реагирует на события мыши. Когда вы поэкспериментируете с этой программой, то увидите, что события от мыши посыпаются к тому окну, в котором они происходят.

```
// Обработка событий мыши как в дочернем окне, так и окне апплета.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="WindowEvents" width=300 height=50>
</applet>
*/
// Создать подкласс Frame.
class SampleFrame extends Frame
    implements MouseListener, MouseMotionListener {

    String msg = "";
    int mouseX=10, mouseY=40;
    int movX=0, movY=0;

    SampleFrame(String title) {
        super(title);
        // зарегистрировать этот объект, чтобы получать свои собственные
        // события мыши
        addMouseListener(this);
        addMouseMotionListener(this);
        // создать объект для обработки событий мыши
        MyWindowAdapter adapter = new MyWindowAdapter(this);
        // зарегистрировать его для получения таких событий
        addWindowListener(adapter);
    }

    // Обработать событие "Щелчок мыши".
    public void mouseClicked(MouseEvent me) {

    }

    // Обработать событие "Мышь введена".
    public void mouseEntered(MouseEvent evtObj) {
        // сохранить координаты
        mouseX = 10;
        mouseY = 54;
        msg = "Mouse just entered child.";
        repaint();
    }

    // Обработать событие "Мышь выведена".
    public void mouseExited(MouseEvent evtObj) {
        // сохранить координаты
        mouseX = 10;
        mouseY = 54;
        msg = "Mouse just left child window.";
    }
}
```

```
repaint();
}

// Обработать событие "Кнопка мыши нажата".
public void mousePressed(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down";
    repaint();
}

// Обработать событие "Кнопка мыши отпущена".
public void mouseReleased(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Up";
    repaint();
}

// Обработать событие "Мышь перетащена".
public void mouseDragged(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    movX = me.getX();
    movY = me.getY();
    msg = "*";
    repaint();
}

// Обработать событие "Мышь передвинута".
public void mouseMoved(MouseEvent me) {
    // сохранить координаты
    movX = me.getX();
    movY = me.getY();
    repaint(0, 0, 100, 60);
}

public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
    g.drawString("Mouse at " + movX + ", " + movY, 10, 40);
}
}

class MyWindowAdapter extends WindowAdapter {
    SampleFrame sampleFrame;
    public MyWindowAdapter(SampleFrame sampleFrame) {
```

```
this.sampleFrame = sampleFrame;
}
public void windowClosing(WindowEvent we) {
    sampleFrame.setVisible(false);
}
}

// Окно апплета.
public class WindowEvents extends Applet
    implements MouseListener, MouseMotionListener {

    SampleFrame f;
    String msg = "";
    int mouseX=0, mouseY=10;
    int movX=0, movY=0;

    // Создать фрейм-окно
    public void init() {
        f = new SampleFrame("Handle Mouse Events");
        f.setSize(300, 200);
        f.setVisible(true);

        // зарегистрировать этот объект, чтобы получить его собственные
        // события мыши
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    // Удалить фрейм-окно при останове апплета.
    public void stop() {
        f.setVisible(false);
    }

    // Показать фрейм-окно при старте апплета.
    public void start() {
        f.setVisible(true);
    }

    // Обработать событие "Кнопка мыши нажата".
    public void mouseClicked(MouseEvent me) {
    }

    // Обработать событие "Мышь введена".
    public void mouseEntered(MouseEvent me) {
        // сохранить координаты
        mouseX = 0;
        mouseY = 24;
        msg = "Mouse just entered applet window.";
        repaint();
    }
}
```

```
// Обработать событие "Мышь выведена".
public void mouseExited(MouseEvent me) {
    // сохранить координаты
    mouseX = 0;
    mouseY = 24;
    msg = "Mouse just left applet window.";
    repaint();
}

// Обработать событие "Кнопка мыши нажата".
public void mousePressed(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down";
    repaint();
}

// Обработать событие "Кнопка мыши отпущена".
public void mouseReleased(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Up";
    repaint();
}

// Обработать событие "Мышь перетащена".
public void mouseDragged(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    movX = me.getX();
    movY = me.getY();
    msg = "*";
    repaint();
}

// Обработать событие "Мышь передвинута".
public void mouseMoved(MouseEvent me) {
    // сохранить координаты
    movX = me.getX();
    movY = me.getY();
    repaint(0, 0, 100, 20);
}

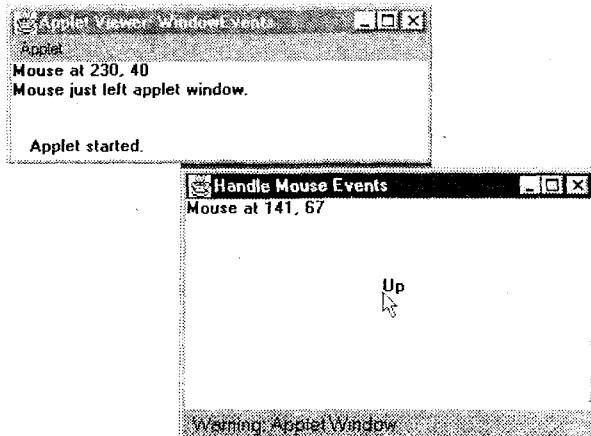
// Вывести msg в окне апплета.
public void paint(Graphics g) {
```

```

g.drawString(msg, mouseX, mouseY);
g.drawString("Mouse at " + movX + ", " + movY, 0, 10);
}
}

```

Пример вывода этой программы представлен на рис. 21.3.



**Рис. 21.3.** Окно апплета WindowEvents и дочернее фрейм-окно

## Создание оконной программы

Хотя AWT чаще всего используется для создания апплетов, с его помощью можно также проектировать и автономные приложения. Для этого нужно просто организовать один или несколько оконных объектов внутри метода main(). Следующая автономная программа создает фрейм-окно, которое отвечает на щелчки мыши и нажатия клавиш:

```

// Создать AWT-приложение.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

// Определить класс фрейм-окна.
public class AppWindow extends Frame {
    String keymsg = "";
    String mouseemsg = "";
    int mouseX=30, mouseY=30;

    public AppWindow() {
        addKeyListener(new MyKeyAdapter(this));
        addMouseListener(new MyMouseAdapter(this));
        addWindowListener(new MyWindowAdapter());
    }
}

```

```
public void paint(Graphics g) {
    g.drawString(keymsg, 10, 40);
    g.drawString(mousemsg, mouseX, mouseY);
}

// Создать фрейм-окно для приложения.
public static void main(String args[]) {
    AppWindow appwin = new AppWindow();

    appwin.setSize(new Dimension(300, 200));
    appwin.setTitle("An AWT-Based Application");
    appwin.setVisible(true);
}

}

class MyKeyAdapter extends KeyAdapter {
    AppWindow appWindow;
    public MyKeyAdapter(AppWindow appWindow) {
        this.appWindow = appWindow;
    }
    public void keyTyped(KeyEvent ke) {
        appWindow.keymsg += ke.getKeyChar();
        appWindow.repaint();
    };
}

class MyMouseAdapter extends MouseAdapter {
    AppWindow appWindow;
    public MyMouseAdapter(AppWindow appWindow) {
        this.appWindow = appWindow;
    }
    public void mousePressed(MouseEvent me) {
        appWindow.mouseX = me.getX();
        appWindow.mouseY = me.getY();
        appWindow.mousemsg = "Mouse Down at " + appWindow.mouseX +
            ", " + appWindow.mouseY;
        appWindow.repaint();
    }
}

class MyWindowAdapter extends WindowAdapter {
    public void windowClosing(WindowEvent we) {
        System.exit(0);
    }
}
```

Пример вывода этой программы представлен на рис. 21.4.

После создания фрейм-окно живет своей собственной жизнью. Обратите внимание, что main() завершается обращением к appwin.setVisible(true).

Однако, программа поддерживается в рабочем состоянии, пока вы не закроете окно. В сущности, при создании оконного приложения вы будете использовать `main()`, чтобы запустить для него окно верхнего уровня. После этого ваша программа будет функционировать как GUI-приложение, а не как консольные программы, использовавшиеся ранее.

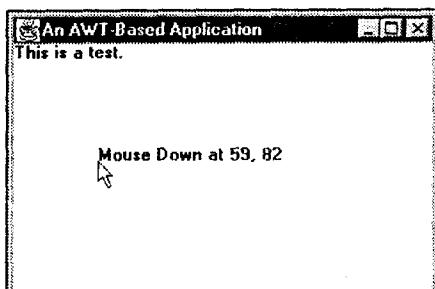


Рис. 21.4. Окно AWT-приложения

## Отображение информации в окне

В самом общем смысле окно является контейнером для разнообразной информации. Хотя в предшествующих примерах мы уже выводили небольшие порции текста в окно, мы еще не начали пользоваться способностью окна представлять высококачественный текст и графику. В действительности, большая часть средств AWT ориентирована на поддержку именно этих возможностей. По указанной причине в остальной части данной главы обсуждаются возможности обработки текста, графики и шрифтов на языке Java. Эти возможности, как вы увидите, являются достаточно мощными и гибкими.

## Работа с графикой

AWT поддерживает богатый набор графических методов. Вся графика рисуется относительно окна. Это может быть главное или дочернее окно апплета, а также окно автономного приложения. Начало координат каждого окна — в его верхнем левом углу и обозначается как  $(0, 0)$ . Координаты определяются в пикселях. Весь вывод в окно выполняется через графический контекст. *Графический контекст* инкапсулирован в классе и получается двумя способами:

- передается апплету, когда вызывается один из его многочисленных методов, таких как `paint()` или `update()`;
- возвращается методом `getGraphics()` класса `Component`.

Для остальных примеров этой главы мы будем демонстрировать графику в главном окне апплета. Однако та же техника применима к любому другому типу окна.

Класс `Graphics` определяет ряд функций рисования. Каждая форма может быть рисованной или заполненной. Объекты рисуются и заполняются выбранным в текущий момент графическим цветом, который по умолчанию является черным. Когда графический объект превышает размеры окна, вывод автоматически усекается. Рассмотрим несколько методов рисования.

## Рисование линий

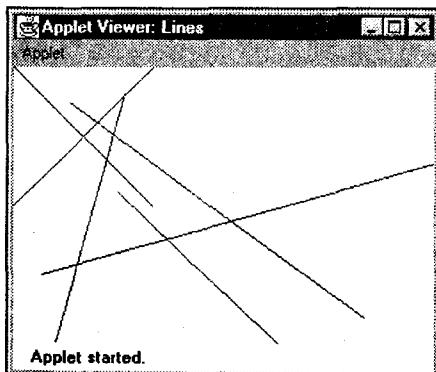
Линии рисуются методом `drawLine()` формата:

```
void drawLine(int startX, int startY, int endX, int endY)
```

`DrawLine()` отображает линию (в текущем цвете рисования), которая начинается в координатах `startX`, `startY` и заканчивается в `endX`, `endY`.

Следующий applet выводит несколько линий:

```
// Рисует линии.  
import java.awt.*;  
import java.applet.*;  
/*  
<applet code="Lines" width=300 height=200>  
</applet>  
  
*/  
public class Lines extends Applet {  
    public void paint(Graphics g) {  
        g.drawLine(0, 0, 100, 100);  
        g.drawLine(0, 100, 100, 0);  
        g.drawLine(40, 25, 250, 180);  
        g.drawLine(75, 90, 400, 400);  
        g.drawLine(20, 150, 400, 40);  
        g.drawLine(5, 290, 80, 19);  
    }  
}
```



Пример вывода этой программы представлен на рис. 21.5.

Рис. 21.5. Окно апплета Lines

## Рисование прямоугольников

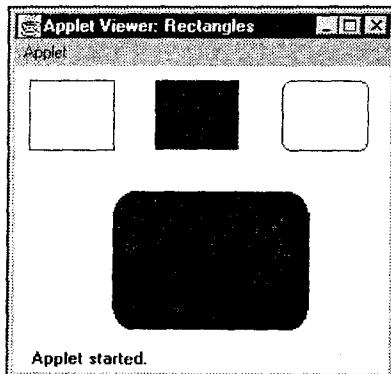
Методы `drawRect()` и `fillRect()` отображают соответственно рисованный и заполненный прямоугольник. Их формат:

```
void drawRect(int top, int left, int width, int height)
void fillRect(int top, int left, int width, int height)
```

Координаты левого верхнего угла прямоугольника — в параметрах `top` и `left`. `width` и `height` — указывают размеры прямоугольника (в пикселях).

Чтобы рисовать округленный прямоугольник, используйте `drawRoundRect()` или `fillRoundRect()` с форматами:

```
void drawRoundRect(int top, int left, int width, int height,
    int xDiam, int yDiam)
void fillRoundRect(int top, int left, int width, int height,
    int xDiam, int yDiam)
```



**Рис. 21.6.** Окно апплета Rectangles

Округленный прямоугольник имеет закругленные углы. Левый верхний угол прямоугольника задается параметрами `top`, `left`. Размеры прямоугольника определяются в `width` и `height`. Диаметр округляющейся дуги по оси X определяется в `xDiam`. Диаметр округляющейся дуги по оси Y определяется в `yDiam`. Следующий апплет рисует несколько прямоугольников:

```
// Рисует прямоугольники.
import java.awt.*;
import java.applet.*;
/*
<applet code="Rectangles" width=300 height=200>
</applet>
*/
public class Rectangles extends Applet {
    public void paint(Graphics g) {
```

```

g.drawRect(10, 10, 60, 50);
g.fillRect(100, 10, 60, 50);
g.drawRoundRect(190, 10, 60, 50, 15, 15);
g.fillRoundRect(70, 90, 140, 100, 30, 40);
}
}

```

Пример вывода этой программы представлен на рис. 21.6.

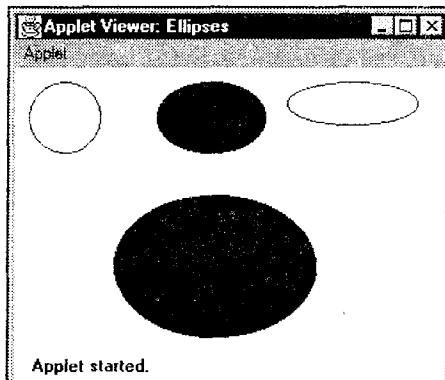
## Рисование эллипсов и кругов

Для рисования эллипса используйте `drawOval()`, а для его заполнения — `fillOval()`. Эти методы имеют формат:

```

void drawOval(int top, int left, int width, int height)
void fillOval(int top, int left, int width, int height)

```



**Рис. 21.7.** Окно апплета Ellipses

Эллипс рисуется в пределах ограничительного прямоугольника, чей левый верхний угол определяется параметрами `top` и `left`, а ширина и высота указываются в `width` и `height`. Чтобы нарисовать круг, в качестве ограничительного прямоугольника указывайте квадрат. Следующая программа рисует несколько эллипсов:

```

// Рисует эллипсы.
import java.awt.*;
import java.applet.*;
/*
<applet code="Ellipses" width=300 height=200>
</applet>
*/
public class Ellipses extends Applet {
    public void paint(Graphics g) {

```

```

g.drawOval(10, 10, 50, 50);
g.fillOval(100, 10, 75, 50);
g.drawOval(190, 10, 90, 30);
g.fillOval(70, 90, 140, 100);
}
}

```

Пример вывода этой программы представлен на рис. 21.7.

## Рисование дуг

Дуги можно рисовать методами `drawArc()` и `fillArc()`, используя форматы:

```

void drawArc(int top, int left, int width, int height, int начало,
            int конец)
void fillArc(int top, int left, int width, int height, int начало,
             int конец)

```

Дуга ограничена прямоугольником, чей левый верхний угол определяется параметрами `top`, `left`, а ширина и высота — параметрами `width` и `height`. Дуга рисуется от `начала` до углового расстояния, указанного в `конец`. Углы указываются в градусах и отсчитываются от горизонтальной оси против часовой стрелки. Дуга рисуется против часовой стрелки, если `конец` положителен, и по часовой стрелке, если `конец` отрицателен. Поэтому, чтобы нарисовать дугу от двенадцатичасового до шестичасового положений, начальный угол должен быть  $90^\circ$  и угол развертки  $180^\circ$ .

Следующий апллет рисует несколько дуг:

```

// Рисует дуги.
import java.awt.*;
import java.applet.*;
/*
<applet code="Arcs" width=300 height=200>
</applet>
*/
public class Arcs extends Applet {
    public void paint(Graphics g) {
        g.drawArc(10, 40, 70, 70, 0, 75);
        g.fillArc(100, 40, 70, 70, 0, 75);
        g.drawArc(10, 100, 70, 80, 0, 175);
        g.fillArc(100, 100, 70, 90, 0, 270);
        g.drawArc(200, 80, 80, 80, 0, 180);
    }
}

```

Пример вывода этой программы представлен на рис. 21.8.

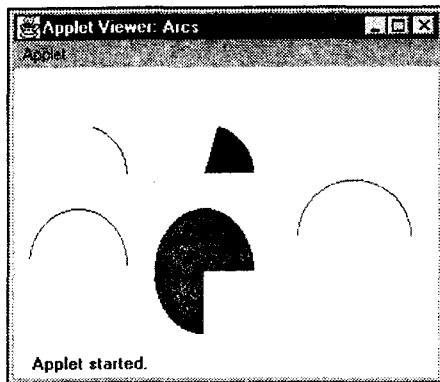


Рис. 21.8. Окно апплета Arcs

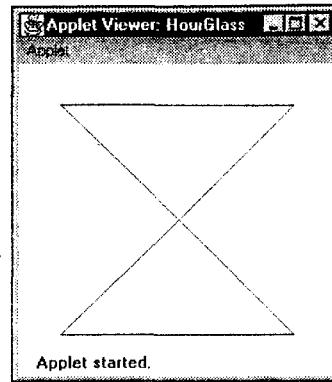


Рис. 21.9. Окно апплета HourGlass

## Рисование многоугольников

Фигуры произвольной формы можно рисовать, используя методы `drawPolygon()` и `fillPolygon()` со следующими форматами:

```
void drawPolygon(int x[ ], int y[ ], int numPoints)
void fillPolygon(int x[ ], int y[ ], int numPoints)
```

Оконечные точки многоугольника определяются координатными парами, содержащимися в массивах `x[ ]` и `y[ ]`. Число точек, определенных в этих массивах, указывается параметром `numPoints`. Имеются альтернативные формы этих методов, в которых многоугольник определяется объектом класса `Polygon`. Следующий апплет рисует форму песочных часов:

```
// Рисует многоугольник.
import java.awt.*;
import java.applet.*;
/*
<applet code="HourGlass" width=230 height=210>
</applet>
*/
public class HourGlass extends Applet {
    public void paint(Graphics g) {
        int xpoints[] = {30, 200, 30, 200, 30};
        int ypoints[] = {30, 30, 200, 200, 30};
        int num = 5;
        g.drawPolygon(xpoints, ypoints, num);
    }
}
```

Пример вывода этой программы представлен на рис. 21.9.

## Установка размеров графики

Часто нужно установить размеры графического объекта в определенной пропорции с текущими размерами окна, в котором он рисуется. Для этого сначала получают текущие размеры окна, вызывая метод `getSize()` для оконного объекта. Он возвращает размеры окна, инкапсулированные в объект класса `Dimension`. Раз вы знаете текущие размеры окна, то можете нужным образом отмасштабировать графический вывод.

Для демонстрации этой методики ниже представлен апплет, который сначала рисует квадрат размером 200×200 пикселов и затем увеличивает его по 25 пикселов в ширину и высоту с каждым щелчком мыши, пока он не станет больше чем 500×500. В этой точке следующий щелчок мыши возвратит его к размеру 200×200, и процесс начнется с начала. В окне рисуется прямоугольник вокруг внутренней границы окна; а внутри этого прямоугольника — две перекрещивающихся линии диагоналей. Апплет работает с программой `appletviewer`, но может не работать в окне браузера.

```
// Изменяет размеры вывода для установки текущих размеров окна.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code="ResizeMe" width=200 height=200>
</applet>
*/
public class ResizeMe extends Applet {
    final int inc = 25;
    int max = 500;
    int min = 200;
    Dimension d;
    public ResizeMe() {
        addMouseListener(new MouseAdapter() {
            public void mouseReleased(MouseEvent me) {
                int w = (d.width + inc) > max?min :(d.width + inc);
                int h = (d.height + inc) > max?min :(d.height + inc);
                setSize(new Dimension(w, h));
            }
        });
    }
    public void paint(Graphics g) {
        d = getSize();
        g.drawLine(0, 0, d.width-1, d.height-1);
        g.drawLine(0, d.height-1, d.width-1, 0);
        g.drawRect(0, 0, d.width-1, d.height-1);
    }
}
```

## Работа с цветом

Java обеспечивает переносимость цвета, вне зависимости от устройства вывода объекта. Цветовая система AWT позволяет указывать в программе любой желаемый цвет. Более того, AWT находит наилучшее согласование этого цвета с заданными аппаратными ограничениями дисплея, выполняющего вашу программу или апплет. Поэтому ваш код не имеет никакого отношения к различиям в способах поддержки цветов разными аппаратными устройствами. Работа с цветом поддерживается классом `Color`.

Как вы видели в главе 19, в `Color` определено несколько цветовых констант (например, `Color.black`), специфицирующих ряд обычных цветов. Вы можете также создавать собственные цвета, применяя один из цветовых конструкторов. Обычно используются следующие его формы:

```
Color(int red, int green, int blue)  
Color(int rgbValue)  
Color(float red, float green, float blue)
```

Первый конструктор получает (через указанные параметры) три целых числа, которые определяют цвет как смесь красного, зеленого и синего цвета. Эти значения должны быть между 0 и 255, как в следующем примере:

```
new Color(255, 100, 100); // светло-красный
```

Второй цветовой конструктор получает одиночное целое число, которое содержит смесь красного, зеленого и синего цветов, упакованную в целое число. Целое число организовано с красным цветом — в битах от 16 до 23, зеленым цветом — в битах от 8 до 15 и синим цветом — в битах от 0 до 7. Пример этого конструктора:

```
int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);  
Color darkRed = new Color(newRed);
```

Третий конструктор получает три `float`-значения (между 0.0 и 1.0), в которых определяется относительная смесь красного, зеленого и синего цвета.

После создания цветового объекта его можно использовать для установки цвета переднего плана и/или фона (с помощью методов `setForeground()` и `setBackground()`, описанных в главе 19). Его можно также использовать для установки текущего цвета рисования.

## Цветовые методы

Класс `Color` определяет несколько методов, которые помогают манипулировать цветом. Они рассматриваются ниже.

### Использование тона, насыщенности и яркости

Для определения специфических цветов используется две альтернативные цветовые модели: HSB (Hue-Saturation-Brightness, цветовой тон-насыщен-

нность-яркость) и RGB (Red-Green-Blue, красный-зеленый-синий). Тон (оттенок) определяется числом между 0.0 и 1.0 (для цветов радуги, расположенных в порядке возрастания: красный, оранжевый, желтый, зеленый, голубой, синий (индиго) и фиолетовый). Насыщенность — другая шкала, ранжированная от 0.0 до 1.0, представляющая изменения тона от светлого (пастельного) к интенсивному. Значения яркости также ранжированы от 0.0 до 1.0, где 1 — ярко-белый, а 0 — черный. Класс `Color` поставляет два метода, которые выполняют взаимные преобразования RGB- и HSB-моделей:

```
static int HSBtoRGB(float hue, float saturation, float brightness)
static float[ ] RGBtoHSB(int red, int green, int blue, float values[ ])
```

Метод `HSBtoRGB()` возвращает упакованное RGB-значение, совместимое с конструктором `Color(int)`. Метод `RGBtoHSB()` возвращает массив HSB-значений с плавающей точкой, соответствующих целым числам RGB. Если параметр `values` — не `null`, то этот массив содержит HSB-значения, которые возвращаются в вызывающую программу. В противном случае создается новый массив, и в нем возвращаются HSB-значения. В любом случае массив содержит тон в элементе с индексом 0, насыщенность — в элементе с индексом 1 и яркость — в элементе с индексом 2.

### **Методы `getRed()`, `getGreen()`, `getBlue()`**

Вы можете получить красные, зеленые и синие компоненты цвета, независимо используя методы `getRed()`, `getGreen()` и `getBlue()`, показанные ниже:

```
int getRed()
int getGreen()
int getBlue()
```

Каждый из этих методов возвращает цветовой компонент RGB, найденный в вызывающем `Color`-объекте в нижних восьми битах целого числа.

### **Метод `getRGB()`**

Метод `getRGB()` используется для получения упакованного RGB-представления цвета. Формат метода:

```
int getRGB()
```

Возвращаемое значение организовано так же, как описано ранее.

### **Установка текущего цвета графики**

По умолчанию, графические объекты рисуются в текущем цвете переднего плана. Можно изменить этот цвет, вызывая метод `setColor()` класса `Graphics`:

```
void setColor(Color newColor)
```

где параметр `newColor` определяет новый цвет рисунка.

Вы можете получить текущий цвет, вызывая метод `getColor()`:

```
Color getColor()
```

## Апплет с демонстрацией цветов

Следующий апплет создает несколько цветов и рисует различные объекты, используя эти цвета:

```
// Демонстрирует цвета.  
import java.awt.*;  
import java.applet.*;  
/*  
<applet code="ColorDemo" width=300 height=200>  
</applet>  
*/  
  
public class ColorDemo extends Applet {  
    // Рисовать линии  
    public void paint(Graphics g) {  
        Color c1 = new Color(255, 100, 100);  
        Color c2 = new Color(100, 255, 100);  
        Color c3 = new Color(100, 100, 255);  
  
        g.setColor(c1);  
        g.drawLine(0, 0, 100, 100);  
        g.drawLine(0, 100, 100, 0);  
  
        g.setColor(c2);  
        g.drawLine(40, 25, 250, 180);  
        g.drawLine(75, 90, 400, 400);  
  
        g.setColor(c3);  
        g.drawLine(20, 150, 400, 40);  
        g.drawLine(5, 290, 80, 19);  
  
        g.setColor(Color.red);  
        g.drawOval(10, 10, 50, 50);  
        g.fillOval(70, 90, 140, 100);  
  
        g.setColor(Color.blue);  
        g.drawOval(190, 10, 90, 30);  
        g.drawRect(10, 10, 60, 50);  
  
        g.setColor(Color.cyan);  
        g.fillRect(100, 10, 60, 50);  
        g.drawRoundRect(190, 10, 60, 50, 15, 15);  
    }  
}
```

## Установка режима рисования

Режим рисования (paint mode) определяет, как объекты рисуются в окне. По умолчанию<sup>1</sup>, новый вывод в окно записывается поверх любого предварительно существующего содержания. Однако можно выводить новые объекты с помощью метода `setXORMode()` со следующей сигнатурой:

```
void setXORMode(Color xorColor)
```

Здесь параметр `xorColor` определяет цвет, который будет использован в окне в режиме XOR<sup>2</sup>, когда объект выводится. Преимущество XOR-режима состоит в том, что новый объект всегда будет видимым, независимо от того, какого цвета объект на нем нарисован<sup>3</sup>.

Чтобы вернуться в режим перезаписи, вызовите метод `setPaintMode()` с форматом:

```
void setPaintMode()
```

Вообще, следует использовать режим перезаписи для нормального вывода, и режим XOR — для специальных целей. Например, следующая программа отображает тонкий крестик, который прослеживает указатель мыши. Такой крестик выводится в окно в режиме XOR и всегда видим, независимо от того, каков лежащий под ним цвет.

```
// Демонстрирует режим XOR.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="XOR" width=400 height=200>
</applet>
*/
public class XOR extends Applet {
    int chsX=100, chsY=100;

    public XOR() {
        addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseMoved(MouseEvent me) {
                int x = me.getX();

```

<sup>1</sup> В этом обычном режиме рисования исходный цвет точки замещается цветом накладываемого на него цвета рисунка. — Примеч. пер.

<sup>2</sup> В XOR-режиме рисования цвет результирующей точки определяется как результат операции ИСКЛЮЧАЮЩЕГО ИЛИ (XOR) над цветом рисования и исходным цветом той же точки. — Примеч. пер.

<sup>3</sup> Например, белая линия на черном фоне в итоге дает белый цвет, а белая линия на белом фоне — черный. — Примеч. пер.

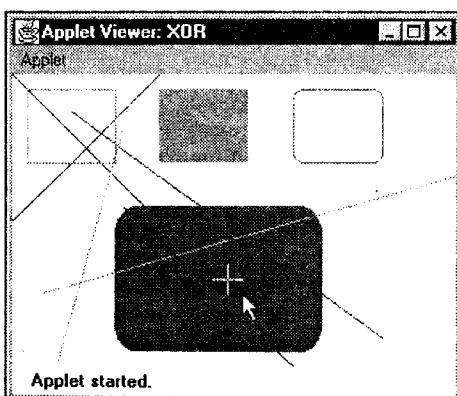
```

        int y = me.getY();
        chsX = x-10;
        chsY = y-10;
        repaint();
    }
}
}

public void paint(Graphics g) {
    g.drawLine(0, 0, 100, 100);
    g.drawLine(0, 100, 100, 0);
    g.setColor(Color.blue);
    g.drawLine(40, 25, 250, 180);
    g.drawLine(75, 90, 400, 400);
    g.setColor(Color.green);
    g.drawRect(10, 10, 60, 50);
    g.fillRect(100, 10, 60, 50);
    g.setColor(Color.red);
    g.drawRoundRect(190, 10, 60, 50, 15, 15);
    g.fillRoundRect(70, 90, 140, 100, 30, 40);
    g.setColor(Color.cyan);
    g.drawLine(20, 150, 400, 40);
    g.drawLine(5, 290, 80, 19);

    // Режим XOR для вывода перекрестья
    g.setXORMode(Color.black);
    g.drawLine(chsX-10, chsY, chsX+10, chsY);
    g.drawLine(chsX, chsY-10, chsX, chsY+10);
    g.setPaintMode();
}
}
}

```



Пример вывода этой программы представлен на рис. 21.10.

**Рис. 21.10.** Окно апплета XOR

## Работа со шрифтами

Пакет AWT поддерживает множество типов шрифтов. Шрифты появились из области традиционного набора текстов и стали важной частью компьютерных документов и дисплеев. AWT обеспечивает гибкость программирования за счет того, что берет на себя операции манипулирования шрифтами и допускает их динамический выбор.

Начиная с версии Java 2, для шрифтов различают три имени: имя семейства, логическое имя и имя гарнитуры<sup>1</sup> (face name). *Имя семейства* — общее название шрифта, например, Courier (Курьер). *Логическое имя* определяет категорию шрифта, например Monospaced (Фиксированной ширины). *Имя гарнитуры* специфицирует определенный шрифт, например, Courier Italic (Курьер курсивный).

Шрифты инкапсульированы в классе `Font`. Некоторые методы, определенные в `Font`, перечислены в табл. 21.2.

**Таблица 21.2.** Некоторые методы, определенные в `Font`

Метод	Описание
<code>static Font decode(String str)</code>	Возвращает шрифт по заданному (в параметре) имени
<code>boolean equals(Object FontObj)</code>	Возвращает <code>true</code> , если вызывающий объект содержит тот же самый шрифт, что указан в <code>FontObj</code> . Иначе возвращает <code>false</code>
<code>String getFamily()</code>	Возвращает имя семейства шрифта, которому вызывающий шрифт принадлежит
<code>static Font getFont(String property)</code>	Возвращает шрифт, связанный с системным свойством, указанным в параметре <code>property</code> . Возвращает указатель <code>null</code> , если свойство не существует
<code>static Font getFont(String property, Font defaultFont)</code>	Возвращает шрифт, связанный с системным свойством, указанным в параметре <code>property</code> . Возвращает шрифт, указанный в <code>defaultFont</code> , если свойство не существует
<code>String getFontName()</code>	Возвращает имя гарнитуры вызывающего шрифта. (Добавлен в Java 2)

<sup>1</sup> В терминах типографского набора под *гарнитурой* шрифта принято понимать *характер рисунка литер* (typeface). — Примеч. пер.

Таблица 21.2 (окончание)

Метод	Описание
<code>String getName()</code>	Возвращает логическое имя вызывающего шрифта
<code>int getSize()</code>	Возвращает размер, в пунктах, вызывающего шрифта
<code>int getStyle()</code>	Возвращает значения стиля (начертания) вызывающего шрифта
<code>int hashCode()</code>	Возвращает код мусора, связанный с вызывающим объектом
<code>boolean isBold()</code>	Возвращает <code>true</code> , если шрифт имеет Bold-начертание, иначе — <code>false</code>
<code>boolean isItalic()</code>	Возвращает <code>true</code> , если шрифт имеет Italic-начертание, иначе — <code>false</code>
<code>boolean isPlain()</code>	Возвращает <code>true</code> , если шрифт имеет Plain-начертание, иначе — <code>false</code>
<code>String toString()</code>	Возвращает строчный эквивалент вызывающего шрифта

В классе `Font` определены переменные, представленные в табл. 21.3.

Таблица 21.3. Переменные класса `Font`

Переменная	Значение
<code>String name</code>	Имя шрифта
<code>Float pointSize</code>	Размер шрифта в пунктах (дробный)
<code>Int size</code>	Размер шрифта в пунктах (целый)
<code>Int style</code>	Стиль (начертание) шрифта

## Определение доступных шрифтов

При работе со шрифтами часто необходимо знать, какой шрифт доступен на вашем компьютере. Чтобы получить эту информацию, можно использовать метод `getAvailableFontFamilyNames()`, определенный классом `GraphicsEnvironment`. Формат заголовка этого метода:

```
String[ ] getAvailableFontFamilyNames()
```

Метод возвращает массив строк, который содержит имена доступных семейств шрифтов.

Кроме того, в классе `GraphicsEnvironment` определен метод `getAllFonts()`. Его формат:

```
Font[] getAllFonts()
```

Этот метод возвращает массив `Font`-объектов для всех доступных шрифтов.

Так как перечисленные методы — члены класса `GraphicsEnvironment`, то для их вызова нужна ссылка на этот класс. Можно получить ссылку, используя статический метод `getLocalGraphicsEnvironment()`, который определен в `GraphicsEnvironment`. Его формат:

```
static GraphicsEnvironment getLocalGraphicsEnvironment()
```

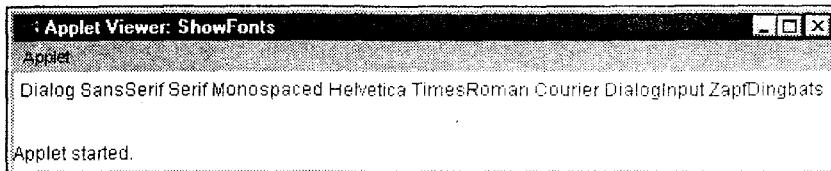
Аплет, который показывает, как получить имена доступных семейств шрифтов:

```
// Показ шрифтов.  
/*  
<applet code="ShowFonts" width=550 height=60>  
</applet>  
*/  
import java.applet.*;  
import java.awt.*;  
  
public class ShowFonts extends Applet {  
    public void paint(Graphics g) {  
        String msg = "";  
        String FontList[];  
  
        GraphicsEnvironment ge =  
            GraphicsEnvironment.getLocalGraphicsEnvironment();  
        FontList = ge.getAvailableFontFamilyNames();  
        for(int i = 0; i < FontList.length; i++)  
            msg += FontList[i] + " ";  
  
        g.drawString(msg, 4, 16);  
    }  
}
```

На рис. 21.11 показан пример вывода. Однако, когда вы выполните эту программу, то, возможно, увидите другой список шрифтов по сравнению с показанным в этой иллюстрации.

### Замечание

До Java 2 для получения списка шрифтов вы использовали бы метод `getFontList()`, определенный классом `Toolkit`. Этот метод теперь не рекомендуется и не должен использоваться в новых программах.



**Рис. 21.11.** Окно апплета ShowFonts

## Создание и выбор шрифта

Перед выбором нового шрифта нужно сначала создать объект класса `Font`, который описывает этот шрифт. Одна из форм конструктора класса `Font` имеет формат:

```
Font(String fontName, int fontStyle, int pointSize)
```

Здесь `fontName` определяет имя желательного шрифта. Имя можно указывать, используя либо логическое имя, либо имя гарнитуры. Все среды Java поддерживают следующие шрифты: `Dialog`, `DialogInput`, `Sans Serif`, `Serif`, `Monospaced` и `Symbol`. Шрифт `Dialog` используется диалоговыми окнами вашей системы. `Dialog` применяется по умолчанию, если вы явно не устанавливаете шрифт. Можно также использовать любые другие шрифты, поддерживающиеся вашей специфической средой, но будьте внимательны — они могут быть не всегда доступными.

Стиль шрифта указывается параметром `fontStyle`. Он может состоять из одной или нескольких констант: `Font.PLAIN`, `Font.BOLD` и `Font.ITALIC`. Стили можно комбинировать, объединяя эти константы операцией `&`. Например, выражение `Font.BOLD | Font.ITALIC` определяет стиль *полужирный курсив*.

Размер шрифта указывается параметром `pointSize` (в пунктах<sup>1</sup>).

Чтобы использовать шрифт, который вы создали, следует выбрать его с помощью метода `setFont()`. Он определен в классе `Component` и имеет общую форму:

```
void setFont(Font fontObj)
```

Здесь `fontObj` — объект, который содержит желательный шрифт.

Следующая программа выводит пример любого стандартного шрифта. Каждый раз, когда вы щелкаете кнопку мыши внутри ее окна, выбирается новый шрифт, и его имя отображается на экране.

```
// Показывает шрифты.
import java.applet.*;
```

<sup>1</sup> Напомним, что типографский *пункт* (point) равен 1/72 дюйма (дюйм = 2.54 см). — Примеч. *пер.*

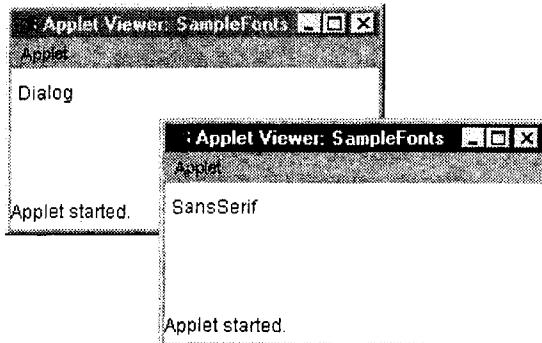
```
import java.awt.*;
import java.awt.event.*;
/*
<applet code="SampleFonts" width=200 height=100>
</applet>
*/
public class SampleFonts extends Applet {
    int next = 0;
    Font f;
    String msg;
    public void init() {
        f = new Font("Dialog", Font.PLAIN, 12);
        msg = "Dialog";
        setFont(f);
        addMouseListener(new MyMouseAdapter(this));
    }
    public void paint(Graphics g) {
        g.drawString(msg, 4, 20);
    }
}
class MyMouseAdapter extends MouseAdapter {
    SampleFonts sampleFonts;
    public MyMouseAdapter(SampleFonts sampleFonts) {
        this.sampleFonts = sampleFonts;
    }
    public void mousePressed(MouseEvent me) {
        // Переключает шрифт каждым щелчком мыши.
        sampleFonts.next++;
        switch(sampleFonts.next) {
        case 0:
            sampleFonts.f = new Font("Dialog", Font.PLAIN, 12);
            sampleFonts.msg = "Dialog";
            break;
        case 1:
            sampleFonts.f = new Font("DialogInput", Font.PLAIN, 12);
            sampleFonts.msg = "DialogInput";
            break;
        case 2:
            sampleFonts.f = new Font("SansSerif", Font.PLAIN, 12);
            sampleFonts.msg = "SansSerif";
            break;
        case 3:
            sampleFonts.f = new Font("Serif", Font.PLAIN, 12);
            sampleFonts.msg = "Serif";
            break;
        }
    }
}
```

```

    case 4:
        sampleFonts.f = new Font("Monospaced", Font.PLAIN, 12);
        sampleFonts.msg = "Monospaced";
        break;
    }
    if(sampleFonts.next == 4) sampleFonts.next = -1;
    sampleFonts.setFont(sampleFonts.f);
    sampleFonts.repaint();
}
}

```

Пример вывода этой программы представлен на рис. 21.12.



**Рис. 21.12.** Окно апплета SampleFonts

## Получение информации о шрифте

Предположим, что вы хотите получить информацию о текущем (только что выбранном) шрифте. Для этого нужно сначала получить текущий шрифт, вызывая метод `getFont()`. Он определен в классе `Graphics` как:

```
Font getFont()
```

Как только вы получили текущий шрифт, можете извлекать информацию о нем, используя различные методы, определенные в классе `Font`. Например, следующий апплет отображает имя, семейство, размер и стиль текущего шрифта:

```

// Показывает информацию о шрифте.
import java.applet.*;
import java.awt.*;
/*
<applet code="FontInfo" width=350 height=60>
</applet>
*/
public class FontInfo extends Applet {
    public void paint(Graphics g) {

```

```

Font f = g.getFont();
String fontName = f.getName();
String fontFamily = f.getFamily();
int fontSize = f.getSize();
int fontStyle = f.getStyle();

String msg = "Family: " + fontName;
msg += ", Font: " + fontFamily;
msg += ", Size: " + fontSize + ", Style: ";
if((fontStyle & Font.BOLD) == Font.BOLD)
    msg += "Bold ";
if((fontStyle & Font.ITALIC) == Font.ITALIC)
    msg += "Italic ";
if((fontStyle & Font.PLAIN) == Font.PLAIN)
    msg += "Plain ";

g.drawString(msg, 4, 16);
}
}

```

## Управление текстовым выводом с помощью класса *FontMetrics*

Для большинства шрифтов не все символы имеют одинаковую ширину (такие шрифты называют *пропорциональными*). Кроме того, высота каждого символа, длина *выносных элементов* (свисающих частей, как у символов г или р, например) и величина пробела между горизонтальными строками изменяется от шрифта к шрифту. Далее, может быть изменен размер шрифта (в пунктах). Переменный характер этих (и других) атрибутов не имел бы слишком больших последствий, если бы Java не требовал от программиста ручного управления фактически всем текстовым выводом.

Учитывая, что размеры каждого шрифта могут отличаться и что шрифты могут быть изменены во время выполнения программы, должен существовать некоторый способ для определения размеров и различных других атрибутов текущего шрифта. Например, для записи одной строки текста после другой необходимо как-то узнать, какова высота шрифта и сколько пикселов необходимо иметь между строками. Чтобы заполнить эту потребность, AWT включает класс *FontMetrics*, который инкапсулирует различную информацию о шрифте. Начнем с определения общей терминологии, используемой при описании шрифтов:

- Высота (Height)** — размер (от верха до низа) самого высокого символа в шрифте.
- Базовая линия (Baseline)** — линия, по которой выровнен низ всех символов (не считая десцендера).

- Высота надстрочного элемента, асцендер (Ascent) — расстояние от базовой линии до верха символа.
- Высота подстрочного элемента, десцендер (Descent) — расстояние от базовой линии до низа символа.
- Интерлиньяж (Leading) — расстояния между самым низом одной строки текста и самым верхом следующей строки.

Во многих из предыдущих примеров мы использовали метод `drawString()`. Он выводит строку в текущем шрифте и цвете, начиная с определенного положения в окне. Однако точка вывода находится слева на базовой линии, а не в левом верхнем углу окна, как обычно в других рисующих методах. Общая ошибка — рисовать строку в той же самой координате, в которой вы рисовали бы прямоугольник. Например, если бы вы рисовали прямоугольник в координатах 0,0 в вашем апплете, вы бы видели полный прямоугольник. Но, если бы вы рисовали строку "Typesetting" с координатами 0,0, то обнаружили бы только нижние выносные элементы (или десцендеры) символов у, р, и г. Как вы увидите, используя метрику шрифта, можно определять надлежащее размещение каждой строки, которую вы отображаете.

Класс `FontMetrics` определяет несколько методов, которые помогают управлять текстовым выводом. Наиболее используемые методы перечислены в табл. 21.4. Они помогают должным образом отобразить текст в окне.

**Таблица 21.4. Некоторые методы класса `FontMetrics`**

Метод	Описание
<code>int bytesWidth(byte b[ ], int start, int numBytes)</code>	Возвращает ширину строки, состоящей из <code>numBytes</code> -символов, содержащихся в массиве <code>b</code> . Параметр <code>start</code> указывает номер начального символа этой строки в массиве <code>b</code>
<code>int charWidth(char c[ ], int start, int numChars)</code>	Возвращает ширину строки, состоящей из <code>numChars</code> -символов, содержащихся в массиве <code>c</code> . Параметр <code>start</code> указывает номер начального символа этой строки в массиве <code>c</code>
<code>int charWidth(char c)</code>	Возвращает ширину <code>c</code>
<code>int charWidth(int c)</code>	Возвращает ширину <code>c</code>
<code>int getAscent()</code>	Возвращает асцендер шрифта
<code>int getDescent()</code>	Возвращает десцендер шрифта
<code>Font getFont()</code>	Возвращает шрифт
<code>int getHeight()</code>	Возвращает высоту строки текста. Это значение можно использовать для вывода в окно многострочного текста

Таблица 21.4 (окончание)

Метод	Описание
<code>int getLeading()</code>	Возвращает размер интерлиньяжа
<code>int getMaxAdvance()</code>	Возвращает ширину самого широкого символа. Возвращает -1, если это значение недоступно
<code>int getMaxAscent()</code>	Возвращает максимальный асцендер
<code>int getMaxDescent()</code>	Возвращает максимальный десцендер
<code>int[] getWidths()</code>	Возвращает ширины первых 256 символов
<code>int stringWidth(String str)</code>	Возвращает ширину строки, указанной в параметре <code>str</code>
<code>String toString()</code>	Возвращает строчный эквивалент вызывающего объекта

Рассмотрим несколько примеров.

## Отображение многострочного текста

Возможно, самое обычное использование `FontMetrics` — для определения интервала между строками текста. Второе — для определения длины отображаемой строки. Здесь вы увидите, как можно выполнить эти задачи.

Для отображения многострочного текста ваша программа должна вручную отслеживать текущую позицию вывода. Когда требуется вывести новую строку, координата Y должна быть смещена к началу следующей строки. Когда строка отображается, координата X должна быть установлена в точку, где заканчивается предыдущая строка. Это позволяет записывать следующую строку, начиная с конца предыдущей.

Для определения интерлиньяжа вы можете использовать значение, возвращаемое методом `getLeading()`. Чтобы определять полную высоту шрифта, прибавьте значение, возвращенное методом `getAscent()`, к значению, возвращенному методом `getDescent()`. Затем вы можете использовать эти значения, чтобы позиционировать каждую строку текста, которую вы выводите. Однако во многих случаях нет нужды использовать эти индивидуальные значения. Часто все, что нужно знать — полную высоту строки, которая является суммой межстрочного пробела, асцендера и десцендера. Самый простой способ получить эти значения — вызвать `getHeight()`. Просто увеличивайте координату Y на это значение, каждый раз, когда нужно продвинуться к следующей строке при выводе текста.

Чтобы начать вывод с конца предыдущего вывода на той же строке, вы должны знать длину (в пикселях) каждой отображаемой строки. Для получения этого значения вызовите метод `stringWidth()`. Значение можно использовать для продвижения координаты X при отображении очередной строки.

В следующем апплете показано, как можно вывести в окно множество строк текста. Кроме того, демонстрируется, как можно вывести несколько строк текста на одной строке. Обратите внимание на переменные `curX` и `curY`. Они следят за текущей позицией текстового вывода.

```
// Демонстрирует многострочный вывод.
import java.applet.*;
import java.awt.*;
/*
<applet code="MultiLine" width=300 height=100>
</applet>
*/
public class MultiLine extends Applet {
    int curX=0, curY=0;                                // текущая позиция

    public void init() {
        Font f = new Font("SansSerif", Font.PLAIN, 12);
        setFont(f);
    }
    public void paint(Graphics g) {
        FontMetrics fm = g.getFontMetrics();
        nextLine("This is on line one.", g);
        nextLine("This is on line two.", g);
        sameLine(" This is on same line.", g);
        sameLine(" This, too.", g);
        nextLine("This is on line three.", g);
    }

    // Продвинуться к следующей строке.
    void nextLine(String s, Graphics g) {
        FontMetrics fm = g.getFontMetrics();

        curY += fm.getHeight();                         // продвижение к следующей строке
        curX = 0;
        g.drawString(s, curX, curY);
        curX = fm.stringWidth(s);                      // продвижение к концу строки
    }

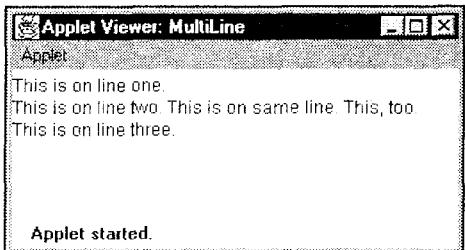
    // Показать на той же линии.
    void sameLine(String s, Graphics g) {
        FontMetrics fm = g.getFontMetrics();
```

```

    g.drawString(s, curX, curY);
    curX += fm.stringWidth(s);           // продвижение к концу строки
}
}

```

Пример вывода этой программы представлен на рис. 21.13.



**Рис. 21.13.** Окно апплета  
MultiLine

## Выравнивание текста по центру

Здесь показан пример, в котором текст выравнивается по центру в окне апплета (по горизонтали и вертикали). Программа получает асцендер, десцендер и ширину строки и с их помощью вычисляет позицию, с которой нужно отобразить в центре окна.

```

// Выравнивает текст по центру.
import java.applet.*;
import java.awt.*;
/*
<applet code="CenterText" width=200 height=100>
</applet>
*/
public class CenterText extends Applet {
    final Font f = new Font("SansSerif", Font.BOLD, 18);

    public void paint(Graphics g) {
        Dimension d = this.getSize();

        g.setColor(Color.white);
        g.fillRect(0, 0, d.width, d.height);
        g.setColor(Color.black);
        g.setFont(f);
        drawCenteredString("This is centered.", d.width, d.height, g);
        g.drawRect(0, 0, d.width-1, d.height-1);
    }

    public void drawCenteredString(String s, int w, int h,
                                   Graphics g) {
        FontMetrics fm = g.getFontMetrics();

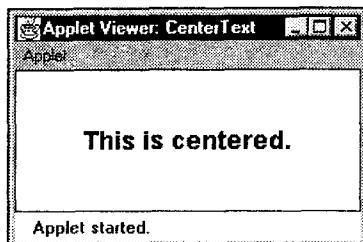
```

```

int x = (w - fm.stringWidth(s)) / 2;
int y = (fm.getAscent() + (h - (fm.getAscent()
    + fm.getDescent())))/2);
g.drawString(s, x, y);
}
}

```

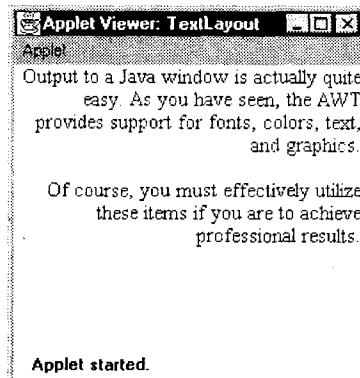
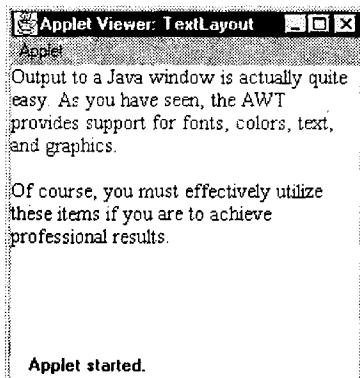
Пример вывода этой программы представлен на рис. 21.14.



**Рис. 21.14.** Окно апплита  
CenterText

## Выравнивание многострочного текста

Если вы пользовались текстовым процессором, то видели текст, выровненный так, чтобы один или оба края образовывали прямую линию. Например, большинство текстовых процессоров могут выравнивать текст влево и/или вправо, по центру. В следующей программе вы увидите, как можно выполнить эти действия.



**Рис. 21.15.** Примеры вывода апплита TextLayout

В программе происходит следующее: строка, которая будет выравниваться, разбивается на отдельные слова. Для каждого слова программа следит за его длиной в текущем шрифте и автоматически продвигается к следующей строке, если слово не будет помещаться в текущую строку. Каждая закон-

ченная строка отображается в окне в выбранном в текущий момент стиле выравнивания. Каждый раз, когда вы щелкаете мышью в окне апплета, стиль выравнивания изменяется. Пример вывода этой программы представлен на рис. 21.15.

```
// Демонстрирует выравнивание текста.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
/* <title>Text Layout</title>
<applet code="TextLayout" width=200 height=200>
<param name="text" value="Output to a Java window is actually quite easy.
As you have seen, the AWT provides support for
fonts, colors, text, and graphics. <P> Of course,
you must effectively utilize these items
if you are to achieve professional results.">
<param name="fontname" value="Serif">
<param name="fontSize" value="14">
</applet>
*/
public class TextLayout extends Applet {
    final int LEFT = 0;
    final int RIGHT = 1;
    final int CENTER = 2;
    final int LEFTRIGHT =3;
    int align;
    Dimension d;
    Font f;
    FontMetrics fm;
    int fontSize;
    int fh, bl;
    int space;
    String text;

    public void init() {
        setBackground(Color.white);
        text = getParameter("text");
        try {
            fontSize = Integer.parseInt(getParameter("fontSize"));
        catch (NumberFormatException e) {
            fontSize=14;
        }
        align = LEFT;
        addMouseListener(new MyMouseAdapter(this));
    }
}
```

```
public void paint(Graphics g) {
    update(g);
}

public void update(Graphics g) {
    d = getSize();
    g.setColor(getBackground());
    g.fillRect(0,0,d.width, d.height);
    if(f==null) f = new Font(getParameter("fontname"),
                           Font.PLAIN, fontSize);
    g.setFont(f);
    if(fm == null) {
        fm = g.getFontMetrics();
        bl = fm.getAscent();
        fh = bl + fm.getDescent();
        space = fm.stringWidth(" ");
    }

    g.setColor(Color.black);
    StringTokenizer st = new StringTokenizer(text);
    int x = 0;
    int nextx;
    int y = 0;
    String word, sp;
    int wordCount = 0;
    String line = "";
    while (st.hasMoreTokens()) {
        word = st.nextToken();
        if(word.equals("<P>")) {
            drawString(g, line, wordCount, fm.stringWidth(line), y+bl);
            line = "";
            wordCount = 0;
            x = 0;
            y = y + (fh * 2);
        }
        else {
            int w = fm.stringWidth(word);
            if(( nextx = (x+space+w)) > d.width) {
                drawString(g, line, wordCount, fm.stringWidth(line), y+bl);
                line = "";
                wordCount = 0;
                x = 0;
                y = y + fh;
            }
            if(x!=0) {sp = " ";} else {sp = "";}
            line = line + sp + word;
            x = x + space + w;
        }
    }
}
```

```
        wordCount++;
    }
}
drawString(g, line, wordCount, fm.stringWidth(line), y+bl);
}

public void drawString(Graphics g, String line,
                       int wc, int lineW, int y) {
    switch(align) {
        case LEFT: g.drawString(line, 0, y);
                     break;
        case RIGHT: g.drawString(line, d.width-lineW,y);
                     break;
        case CENTER: g.drawString(line, (d.width-lineW)/2, y);
                     break;
        case LEFTRIGHT:
            if(lineW < (int)(d.width*.75)) {
                g.drawString(line, 0, y);
            }
            else {
                int toFill = (int)((d.width - lineW)/wc);
                int nudge = d.width - lineW - (toFill*wc);
                int s = fm.stringWidth(" ");
                StringTokenizer st = new StringTokenizer(line);
                int x = 0;
                while(st.hasMoreTokens()) {
                    String word = st.nextToken();
                    g.drawString(word, x, y);
                    if(nudge>0) {
                        x = x + fm.stringWidth(word) + space + toFill + 1;
                        nudge--;
                    } else {
                        x = x + fm.stringWidth(word) + space + toFill;
                    }
                }
            }
            break;
        }
    }

class MyMouseAdapter extends MouseAdapter {
    TextLayout tl;
    public MyMouseAdapter(TextLayout tl){this.tl = tl;
        this.tl = tl;
    }
}
```

```
public void mouseClicked(MouseEvent me) {  
    tl.align = (tl.align + 1) % 4;  
    tl.repaint();  
}  
}
```

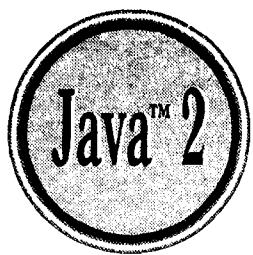
Посмотрим, как работает апплет. Сначала он создает несколько констант, которые будут использоваться для определения стиля выравнивания, и затем объявляет несколько переменных. Метод `init()` получает текст, который будет отображен. Затем апплет инициализирует размер шрифта в блоке `try-catch`, который установит размер шрифта в 14 пунктов, если в документе HTML отсутствует параметр `fontSize`. Параметр `text` является длинной строкой текста с HTML-тегом `<P>` в качестве абзаца-разделителя.

Двигателем этого примера является метод `update()`. Он устанавливает шрифт и получает базовую линию и высоту шрифта от объекта шрифтовой метрики (`FontMetrics fm`). Затем создается объект  `StringTokenizer`, который используется, чтобы извлечь следующую лексему (строку, ограниченную пробелами) из строки, определенной в переменной `text`. Если следующая лексема есть `<P>`, метод увеличивает интервал строк. Иначе, он выясняет, выходит ли длина этой лексемы за ширину колонки. Если строка заполнена текстом или нет больше лексем, строка выводится методом `drawString()`.

Три первых направления в переключателе метода `drawString()` работают идентично: каждое выравнивает строку, которая передана через параметр `line`, по левому, правому краю или по центру окна, в зависимости от стиля выравнивания. Направление `LEFTRIGHT` выравнивает как левую, так и правую стороны строки. Для этого вычисляется остаточный пробел (разность между шириной строки и шириной колонки) и распределяется между всеми словами. Последний метод в этом классе изменяет стиль выравнивания каждый раз, когда вы щелкаете мышью в окне апплета.

## Исследование текста и графики

Хотя эта глава описывает наиболее важные аспекты и общую методику, которые используются при отображении текста или графики, она не охватывает всех возможностей Java. В этой области ожидаются дальнейшие усовершенствования и улучшения, поскольку Java и вычислительная среда продолжают развиваться. Например, Java 2 добавляет к AWT новую подсистему, названную *Java 2D*. Java 2D поддерживает улучшенное управление графикой, включая трансляцию координат, вращение и масштабирование, расширенные свойства изображений. Если обработка расширенной графики представляет для вас определенный интерес, то вы, несомненно, захотите детально исследовать Java 2D.



## ГЛАВА 22

# Использование элементов управления, менеджеров компоновки и меню AWT

В этой главе продолжается исследование AWT. В ней рассматриваются стандартные элементы управления и менеджеры компоновки, а также элементы меню. В главу включено обсуждение двух высокоуровневых компонентов — диалогового окна и файлового диалогового окна. Завершает главу другой взгляд на обработку событий.

*Элементы управления* (*controls*) — это компоненты, которые предоставляют пользователю различные способы взаимодействия с приложением (например, командная кнопка (*push button*)). *Менеджер компоновки* (*layout manager*) автоматически позиционирует (размещает, располагает<sup>1</sup>) компоненты в контейнере. Вид окна, таким образом, определяется комбинацией элементов управления, содержащихся в окне, и менеджера компоновки, используемого для их размещения.

В дополнение к элементам управления, фрейм-окно может также включать *строку меню* (*menu bar*) стандартного стиля. Каждый вход в строке меню активизирует раскрывающееся меню элементов, которые пользователь может выбирать. Стока меню всегда позиционируется наверху окна. Строки меню, хотя и различаются по виду, обрабатываются похожим способом, что и другие элементы управления.

Хотя компоненты окна можно позиционировать вручную, это весьма утомительно. Менеджер компоновки предназначен для автоматизации этой задачи. В первой части данной главы, которая представляет различные элементы управления, будет использован менеджер компоновки, заданный по умолчанию. Он отображает компоненты в контейнере, размещая их по принципу "слева-направо, сверху-вниз". Затем будут рассмотрены все менеджеры ком-

<sup>1</sup> Поэтому иногда их называют менеджерами *размещения* или *расположения* компонентов. — Примеч. пер.

поновки. Там вы и увидите, как лучше управлять позиционированием элементов управления.

## Элементы управления. Основные понятия

AWT поддерживает следующие типы элементов управления:

- Текстовые метки (Labels)
- Кнопки (Push buttons)
- Флажки (Check boxes)
- Списки с выбором элементов (Choice lists)
- Списки (Lists)
- Полосы прокрутки (Scroll bars)
- Элементы редактирования текста: текстовые поля (Text fields) и текстовые области (Text areas).

Элементы управления представлены специальными классами пакета AWT, которые являются подклассами<sup>1</sup> класса Component.

## Добавление и удаление элементов управления

Для включения элемента управления в окно нужно добавить его к окну. Для этого необходимо сначала создать экземпляр желательного элемента управления и затем добавить его к окну вызовом метода add(), который определен в классе Container. Метод add() имеет несколько форм. В первой части этой главы используется следующая форма:

```
Component add(Component compObj)
```

Здесь compObj — экземпляр элемента управления, который вы хотите добавить. Метод возвращает ссылку на объект, который передается параметром compObj. Сразу после добавления элемент управления будет автоматически выводиться на экран всякий раз, когда отображается его родительское окно.

Если вы захотите удалить элемент управления из окна, когда он больше не нужен, вызывайте метод remove(), который определен в классе Container. Его общая форма:

```
void remove(Component obj)
```

Здесь obj — ссылка на элемент управления, который нужно удалить. Вызывая метод removeAll(), можно удалить все элементы управления.

<sup>1</sup> Имена этих подклассов не всегда совпадают с указанными в данном списке названиями элементов управления. — Примеч. пер.

## Реагирование на элементы управления

За исключением меток, которые являются пассивными, все элементы управления генерируют события, когда к ним обращается пользователь. Например, когда пользователь нажимает на кнопку, программе посыпается сообщение о событии, которое идентифицирует кнопку. В общем случае, ваша программа просто реализует соответствующий интерфейс и затем регистрирует блок прослушивания события для каждого элемента, которым нужно управлять. Как объяснено в главе 20, как только блок прослушивания установлен, события посыпаются к нему автоматически. В следующих разделах определен соответствующий интерфейс для каждого элемента управления.

## Текстовые метки

Самый простой для использования элемент управления — (текстовая) метка (`label`). *Текстовая метка* — это объект класса `Label`, содержащий строку, которую она отображает. Метки — пассивные элементы управления, которые не поддерживают никакого взаимодействия с пользователем. Класс `Label` определяет следующие конструкторы:

```
Label()
Label(String str)
Label(String str, int how)
```

Первая версия создает пустую метку, вторая — метку, которая содержит строку, специфицированную параметром `str`. Эта строка выровнена по левому краю. Третья версия создает метку, которая содержит строку, специфицированную параметром `str`, используя выравнивание, указанное в параметре `how`. Значением `how` должна быть одна из трех констант: `Label.LEFT`, `Label.RIGHT` или `Label.CENTER`.

Текст в метке можно устанавливать или изменять, используя метод `setText()`, а текущую метку можно получить, вызывая `getText()`. Формат этих методов:

```
void setText(String str)
String getText()
```

Параметр `str` в `setText()` определяет новую метку. Метод `getText()` возвращает текущую метку.

Вызывая метод `setAlignment()`, можно устанавливать выравнивание строки в пределах метки. Чтобы получить текущее выравнивание, вызовите метод `getAlignment()`. Формат этих методов:

```
void setAlignment(int how)
int getAlignment()
```

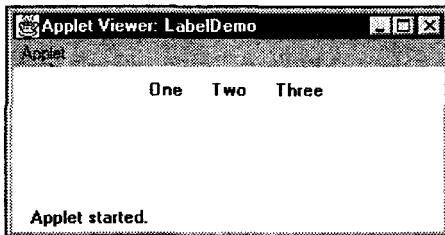
Параметр *how* должен быть одной из констант выравнивания, представленных ранее.

Следующий пример создает три метки и добавляет их к апплету:

```
// Демонстрирует метки.
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/
public class LabelDemo extends Applet {
    public void init() {
        Label one = new Label("One");
        Label two = new Label("Two");
        Label three = new Label("Three");

        // добавить метки в окно апплета
        add(one);
        add(two);
        add(three);
    }
}
```

На рис. 22.1 показано окно, созданное апплетом *LabelDemo*. Обратите внимание, что метки организованы в окне менеджером компоновки, заданным по умолчанию. Позже вы увидите, как более точно управлять размещением меток.



**Рис. 22.1.** Окно апплета  
*LabelDemo*

## Использование кнопок

Наиболее широко используемым элементом управления является кнопка<sup>1</sup> (*push button*). Кнопка — это компонент, который содержит текстовую метку

<sup>1</sup> Кнопки используются для ввода команд, поэтому их часто называют *кнопками команд* или *командными кнопками*. — Примеч. пер.

и генерирует событие, когда ее нажимают. Кнопки являются объектами класса `Button`. В классе `Button` определяются два конструктора:

```
Button()
Button(String str)
```

Первая версия создает пустую кнопку, вторая — кнопку с текстовой меткой, которая передается через параметр `str`.

После того как кнопка была создана, можно установить ее метку, вызывая метод `setLabel()`. Извлечь ее метку можно вызовом `getLabel()`. Форматы этих методов:

```
void setLabel(String str)
String getLabel()
```

Параметр `str` указывает новую метку для кнопки.

## Обработка кнопок

Каждый раз, когда кнопка нажимается, генерируется action-событие. Оно посыпается любым блокам прослушивания, которые предварительно зарегистрировали заинтересованность в приеме уведомления об action-событии от этого компонента. Каждый блок прослушивания реализует интерфейс `ActionListener`. Этот интерфейс определяет метод `actionPerformed()`, который вызывается при возникновении события. В качестве аргумента в этот метод передается объект `ActionEvent`. Он содержит как ссылку на кнопку, которая сгенерировала событие, так и ссылку на строку, которая является меткой кнопки. Для идентификации кнопки можно использовать любую из этих ссылок.

Ниже показан пример, который создает три кнопки с метками "Yes", "No" и "Undecided". Каждый раз, когда одна из них нажимается, отображается сообщение, которое докладывает, что кнопка была нажата. В этой версии метка кнопки используется для того, чтобы определить, какая из них была нажата. Метка возвращается вызовом метода `getActionCommand()` объекта `ActionEvent`, передаваемого методу `actionPerformed()`.

```
// Демонстрирует кнопки.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonDemo" width=250 height=150>
</applet>
*/
public class ButtonDemo extends Applet implements ActionListener {
    String msg = "";
    Button yes, no, maybe;
```

```

public void init() {
    yes = new Button("Yes");
    no = new Button("No");
    maybe = new Button("Undecided");

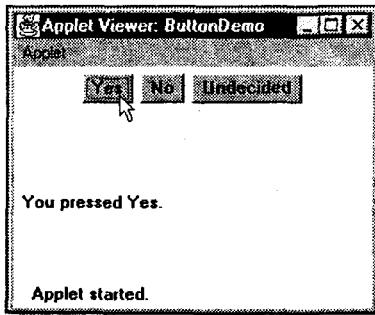
    add(yes);
    add(no);
    add(maybe);

    yes.addActionListener(this);
    no.addActionListener(this);
    maybe.addActionListener(this);
}

public void actionPerformed(ActionEvent ae) {
    String str = ae.getActionCommand();
    if(str.equals("Yes")) {
        msg = "You pressed Yes.";
    }
    else if(str.equals("No")) {
        msg = "You pressed No.";
    }
    else {
        msg = "You pressed Undecided.";
    }
    repaint();
}

public void paint(Graphics g) {
    g.drawString(msg, 6, 100);
}
}

```



Пример вывода программы ButtonDemo показан на рис. 22.2.

**Рис. 22.2.** Пример вывода апллета ButtonDemo

В дополнение к сравнению текстовых меток вы можете также определять, какая кнопка была нажата, сравнивая объект, полученный от метода getSource(), с объектами кнопок, которые вы ранее добавляли к окну. Что-

бы это сделать, нужно сохранять список объектов во время их добавления. Данный подход показывает следующий апллет:

```
// Распознавание объектов типа Button.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonList" width=250 height=150>
</applet>
*/
public class ButtonList extends Applet implements ActionListener {
    String msg = "";
    Button bList[] = new Button[3];

    public void init() {
        Button yes = new Button("Yes");
        Button no = new Button("No");
        Button maybe = new Button("Undecided");

        // сохранить ссылки на кнопки при их добавлении
        bList[0] = (Button) add(yes);
        bList[1] = (Button) add(no);
        bList[2] = (Button) add(maybe);

        // зарегистрироваться для приема событий действия
        for(int i = 0; i < 3; i++) {
            bList[i].addActionListener(this);
        }
    }

    public void actionPerformed(ActionEvent ae) {
        for(int i = 0; i < 3; i++) {
            if(ae.getSource() == bList[i]) {
                msg = "You pressed " + bList[i].getLabel();
            }
        }
        repaint();
    }

    public void paint(Graphics g) {
        g.drawString(msg, 6, 100);
    }
}
```

В этой версии программа сохраняет каждую ссылку кнопки в массиве `bList[]`, когда кнопки добавляются к окну апллета. (Напомним, что метод `add()` возвращает ссылку на добавляемую кнопку.) Затем этот массив используется внутри метода `actionPerformed()`, чтобы определить, какая кнопка была нажата.

Для простых апплетов обычно проще распознать кнопки по их меткам. Однако в ситуациях, когда требуется изменять метку кнопки во время выполнения программы, или при использовании кнопок, которые имеют одинаковые метки, может оказаться проще определить нажатую кнопку, используя ее ссылку на объект.

## Применение флажков

**Флажок** (checkbox) — это элемент управления, который используется для включения или выключения некоторой опции (режима, параметра и т. п.). Он имеет вид маленького квадрата, который может содержать (или не содержать) маркер проверки (check mark) — обычно символ "П". С каждым флажком связывается текстовая метка, которая описывает, какую опцию представляет флажок. Вы можете изменять состояние флажка щелчком мыши<sup>1</sup> по нему. Флажки можно использовать индивидуально или как часть группы. Флажки являются объектами типа Checkbox, который поддерживает следующие конструкторы:

```
Checkbox()
Checkbox(String str)
Checkbox(String str, boolean on)
Checkbox(String str, boolean on, CheckboxGroup cbGroup)
Checkbox(String str, CheckboxGroup cbGroup, boolean on)
```

Первая форма создает флажок, чья текстовая метка изначально — пробел. Состояние флажка в этом случае — "выключен" (off, unchecked). Вторая форма создает флажок, чья текстовая метка определяется параметром *str*. Состояние флажка здесь также "off". Третья форма позволяет устанавливать начальное состояние флажка. Если параметр *on* равен true, создается флажок с первоначальным состоянием "включен" ("on"); иначе (когда *on* равен false) создается "чистый" флажок — без маркера проверки (т. е. в состоянии "off"). Четвертые и пятые формы создают флажок, чья текстовая метка определена параметром *str*, а группа указана параметром *cbGroup*. Если данный флажок не является частью группы, то *cbGroup* должен иметь значение null (пустая ссылка). (Группы флажков описаны в следующем разделе.) Значение параметра *on* определяет начальное состояние флажка.

Для получения текущего состояния флажка вызовите метод getState(). Чтобы установить его состояние, вызовите setState(). Можно получить текущую строковую метку, связанную с флажком, вызывая getLabel(). Чтобы установить эту метку, вызовите setLabel(). Форматы этих методов следующие:

---

<sup>1</sup> Существует и клавиатурный способ изменения состояния флажка. — Примеч. пер.

```
boolean getState()
void setState(boolean on)
String getLabel()
void setLabel(String str)
```

Если *on* есть *true*, флагок устанавливается в состояние "on" (включен). Если *on* — *false*, флагок сбрасывается (квадратик очищается от маркера проверки, т. е. флагок устанавливается в состояние "off" (выключен)). Стока, переданная в параметре *str*, становится новой меткой, связанной с флагком.

## Обработка флагков

Каждый раз, когда флагок выбирается (или выбор отменяется), генерируется item-событие. Оно посыпается к любым блокам прослушивания, которые предварительно зарегистрировали свою заинтересованность в приеме уведомлений об item-событиях от этого компонента. Каждый блок прослушивания реализует интерфейс *ItemListener*. Этот интерфейс определяет метод *itemStateChanged()*, которому через параметр передается объект *ItemEvent*. Он содержит информацию относительно данного события (например, был ли выбор произведен или отменен).

Следующая программа создает четыре флагка. Начальное состояние первого — "on". На экране отображается информация о состоянии каждого флагка и модифицируется всякий раз, когда вы изменяете состояние какого-либо флагка (рис. 22.3).

```
// Демонстрирует флагки.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CheckboxDemo" width=250 height=200>
</applet>
*/
public class CheckboxDemo extends Applet implements ItemListener {
    String msg = "";
    Checkbox Win98, winNT, solaris, mac;

    public void init() {
        Win98 = new Checkbox("Windows 98", null, true);
        winNT = new Checkbox("Windows NT");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("MacOS");

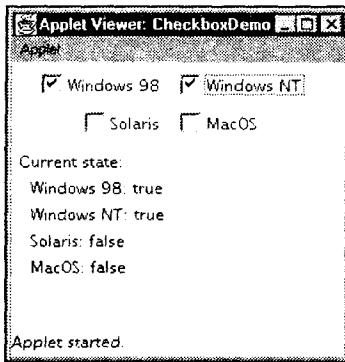
        add(Win98);
        add(winNT);
        add(solaris);
        add(mac);
    }
}
```

```

Win98.addItemListener(this);
winNT.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie) {
    repaint();
}

// отобразить текущее состояние флагков
public void paint(Graphics g) {
    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = " Windows 98: " + Win98.getState();
    g.drawString(msg, 6, 100);
    msg = " Windows NT: " + winNT.getState();
    g.drawString(msg, 6, 120);
    msg = " Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = " MacOS: " + mac.getState();
    g.drawString(msg, 6, 160);
}
}
}

```



Пример вывода программы CheckboxDemo показан на рис. 22.3.

**Рис. 22.3.** Пример вывода  
апплета CheckboxDemo

## Класс *CheckboxGroup*

Возможно создание набора (группы) взаимоисключающих флагков, в котором может быть включен один и только один из них. Такие флагки часто называются "радиокнопками" (*radio buttons*), потому что они действуют подобно селектору станций на автомобильном радиоприемнике — в любой момент может быть выбрана только одна станция. Чтобы создать набор

взаимоисключающих флажков, нужно сначала определить группу, к которой они будут принадлежать, и затем указать эту группу, когда флажки будут создаваться. Группы флажков являются объектами типа `CheckboxGroup`. Пустую группу создает только конструктор, заданный по умолчанию.

Вы можете определить, какой флажок в группе выбран в текущий момент, вызывая метод `getSelectedCheckbox()`. Установить флажок можно вызовом метода `setSelectedCheckbox()`. Форматы этих методов:

```
Checkbox getSelectedCheckbox()
void setSelectedCheckbox(Checkbox which)
```

где `which` — параметр, указывающий флажок, который нужно выбрать. При этом предварительно выбранный флажок будет выключен.

Ниже показана программа, которая использует флажки, являющиеся частью группы:

```
// Демонстрирует группу (взаимонезависимых) флажков.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CBGroup" width=250 height=200>
</applet>
*/
public class CBGroup extends Applet implements ItemListener {
    String msg = "";
    Checkbox Win98, winNT, solaris, mac;
    CheckboxGroup cbg;

    public void init() {
        cbg = new CheckboxGroup();
        Win98 = new Checkbox("Windows 98", cbg, true);
        winNT = new Checkbox("Windows NT", cbg, false);
        solaris = new Checkbox("Solaris", cbg, false);
        mac = new Checkbox("MacOS", cbg, false);

        add(Win98);
        add(winNT);
        add(solaris);
        add(mac);

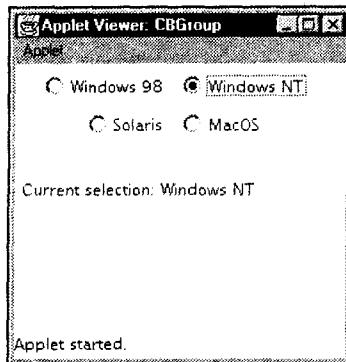
        Win98.addItemListener(this);
        winNT.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
    }
}
```

```

public void itemStateChanged(ItemEvent ie) {
    repaint();
}

// отобразить текущее состояние группы
public void paint(Graphics g) {
    msg = "Current selection: ";
    msg += cbg.getSelectedCheckbox().getLabel();
    g.drawString(msg, 6, 100);
}
}

```



Вывод, сгенерированный апплетом CBGroup, показан на рис. 22.4. Обратите внимание, что флажки теперь имеют круглую форму<sup>1</sup>.

**Рис. 22.4.** Пример вывода апплета CBGroup

## Элемент управления *Choice*

Класс Choice используется для того, чтобы создавать раскрывающийся список элементов, из которых пользователь может делать выбор. Таким образом, элемент управления "выбор" (Choice) имеет форму меню: В неактивном состоянии компонент типа Choice занимает столько места, чтобы показывать только текущий выбранный элемент. Когда пользователь щелкает по нему мышью, раскрывается полный список элементов и можно сделать новый выбор. Каждый элемент в списке — это строка, которая выровнена по левому краю и появляется в списке в том порядке, в каком она добавлялась к объекту типа Choice. Класс Choice определяет только умолчаемый конструктор, который создает пустой список. Чтобы добавить элемент выбора к списку, вызовите метод addItem() или add(). Сигнатуры этих методов:

```

void addItem(String name)
void add(String name)

```

<sup>1</sup> На самом деле, данные элементы управления не являются флажками. Так выглядят элементы типа "переключатель". — Примеч. ред.

Здесь `name` — имя добавляемого элемента. Элементы добавляются к списку в том порядке, в котором выполнялись вызовы `add()` или `addItem()`.

Для определения выбранного в настоящее время элемента можно вызвать метод `getSelectedItem()` или `getSelectedIndex()` с форматами:

```
String getSelectedItem()
int getSelectedIndex()
```

Метод `getSelectedItem()` возвращает строку, содержащую имя элемента, а метод `getSelectedIndex()` — индекс (номер) элемента. Первый элемент имеет индекс 0. По умолчанию выбирается первый элемент, добавленный к списку.

Чтобы получить количество элементов в списке, вызовите метод `getItemCount()`. Выбранный элемент можно установить текущим, вызывая метод `select()` с аргументом в виде отсчитываемого от нуля целочисленного индекса или строки, которая совпадает с одним из имен в списке. Форматы соответствующих методов:

```
int getItemCount()
void select(int index)
void select(String name)
```

Зная индекс, можно получить имя элемента с этим индексом. Для этого нужно вызвать метод `getItem()`, который имеет следующую форму:

```
String getItem(int index)
```

где `index` специфицирует индекс желательного элемента.

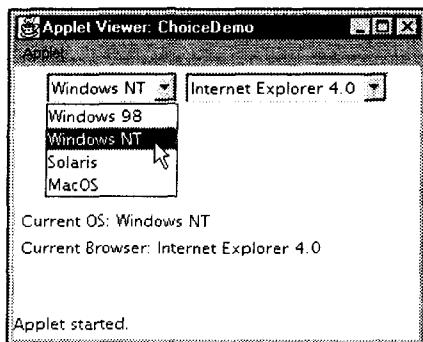
## Обработка списков типа *Choice*

Каждый раз, когда выбирается элемент Choice-списка, генерируется событие типа `item`. Оно посыпается к любым блокам прослушивания, которые предварительно зарегистрировали заинтересованность в приеме уведомлений об item-событиях от данного компонента. Каждый блок прослушивания реализует интерфейс `ItemListener`. Этот интерфейс определяет метод `itemStateChanged()`. Объект `ItemEvent` передается этому методу в качестве аргумента.

В следующем примере создаются два Choice-меню. Одно выбирает операционную систему, другое — браузер.

```
// Демонстрирует Choice-списки.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ChoiceDemo" width=300 height=180>
</applet>
*/
```

```
public class ChoiceDemo extends Applet implements ItemListener {  
    Choice os, browser;  
    String msg = "";  
  
    public void init() {  
        os = new Choice();  
        browser = new Choice();  
  
        // добавить элементы в список OS  
        os.add("Windows 98");  
        os.add("Windows NT");  
        os.add("Solaris");  
        os.add("MacOS");  
  
        // добавить элементы в список браузеров  
        browser.add("Netscape 1.1");  
        browser.add("Netscape 2.x");  
        browser.add("Netscape 3.x");  
        browser.add("Netscape 4.x");  
  
        browser.add("Internet Explorer 2.0");  
        browser.add("Internet Explorer 3.0");  
        browser.add("Internet Explorer 4.0");  
  
        browser.add("Lynx 2.4");  
        browser.select("Netscape 4.x");  
  
        // добавить choice-списки в окно  
        add(os);  
        add(browser);  
  
        // зарегистрироваться для приема item-событий  
        os.addItemListener(this);  
        browser.addItemListener(this);  
    }  
  
    public void itemStateChanged(ItemEvent ie) {  
        repaint();  
    }  
  
    // отобразить текущие выборы  
    public void paint(Graphics g) {  
        msg = "Current OS: ";  
        msg += os.getSelectedItem();  
        g.drawString(msg, 6, 120);  
        msg = "Current Browser: ";  
        msg += browser.getSelectedItem();  
        g.drawString(msg, 6, 140);  
    }  
}
```



Пример вывода программы ChoiceDemo показан на рис. 22.5.

**Рис. 22.5.** Пример вывода апллета ChoiceDemo

## Использование списков

Класс List обеспечивает компактный многоэлементный список со множественным выбором и прокруткой. В отличие от объекта типа Choice, который показывает в меню только один выбранный элемент, List-объект может быть сконструирован так, чтобы отображать любое число элементов выбора в видимом окне. Его можно создать так, чтобы разрешить множественный выбор. List содержит следующие конструкторы:

```
List()
List(int numRows)
List(int numRows, boolean multipleSelect)
```

Первая форма создает элемент управления List, который позволяет выбирать только один элемент. Во второй форме значение параметра *numRows* определяет число строк в списке, которые будут всегда видимы в панели списка (другие могут прокручиваться в панели по мере необходимости). В третьей форме, если параметр *multipleSelect* равен true, то пользователь может выбирать два или несколько элементов одновременно. Если его значение — false, то можно выбрать только один элемент.

Чтобы добавить элемент выбора к списку, вызывайте метод add(), который имеет следующие формы:

```
void add(String name)
void add(String name, int index)
```

Здесь *name* — имя элемента, добавляемого к списку. Первая форма добавляет элементы к концу списка. Вторая — добавляет элементы по индексу (номеру), указываемому параметром *index*. Индексация начинается с нуля. Чтобы добавить элемент в конец списка, нужно указать индекс, равный -1.

Для списков, которые допускают только одиночный выбор, можно определять, какой элемент выбран в текущий момент, если вызвать метод `getSelectedItem()` или `getSelectedIndex()`. Форматы этих методов:

```
String getSelectedItem()  
int getSelectedIndex()
```

Метод `getSelectedItem()` возвращает строку, содержащую имя элемента. Если выбран больше чем один элемент или если никакого выбора еще не было сделано, возвращается `null` (пустой указатель). Метод `getSelectedIndex()` возвращает индекс элемента. Первый элемент имеет индекс 0. Если выбрано больше одного элемента, или если никакого выбора еще не было сделано, возвращается `-1`.

Чтобы определить текущие выбранные элементы для списков, которые допускают множественный выбор, нужно использовать метод `getSelectedItems()` или `getSelectedIndexes()`, с форматами:

```
String[ ] getSelectedItems()  
int[ ] getSelectedIndexes()
```

`getSelectedItems()` возвращает массив, содержащий имена текущих выбранных элементов. `getSelectedIndexes()` возвращает массив, содержащий индексы текущих выбранных элементов.

Для определения количества элементов в списке вызывайте метод `getItemCount()`. Можно устанавливать текущий выбранный элемент, используя метод `select()` с отсчитываемым от нуля целым индексом. Форматы этих методов:

```
int getItemCount()  
void select(int index)
```

Зная индекс, можно получить имя, связанное с элементом с этим индексом, вызывая метод `getItem()`, который имеет следующую форму:

```
String getItem(int index)
```

Здесь `index` — указывает индекс (номер) желательного элемента.

## Обработка списков

Чтобы обрабатывать list-события (list events), нужно реализовать интерфейс `ActionListener`. Каждый раз, когда выполняется двойной щелчок на элементе типа `List`, генерируется объект `ActionEvent`. Его метод `getActionCommand()` можно использовать для извлечения имени вновь выбранного элемента. Кроме того, всякий раз, когда выбирается или отменяется выбор элемента (одиночным щелчком мыши), генерируется объект `ItemEvent`. Его метод `getStateChange()` можно использовать, чтобы определить, что породило это

событие — выбор или отмену выбора. Метод `getItemSelectable()` возвращает ссылку на объект, который породил это событие.

Ниже показан пример, который конвертирует Choice-элемент управления из предшествующего раздела в List-компоненты, один — с множественным выбором, другой — с одиночным:

```
// Демонстрирует списки.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ListDemo" width=300 height=180>
</applet>
*/
public class ListDemo extends Applet implements ActionListener {
    List os, browser;
    String msg = "";

    public void init() {
        os = new List(4, true);
        browser = new List(4, false);

        // добавить элементы в список OS
        os.add("Windows 98");
        os.add("Windows NT");
        os.add("Solaris");
        os.add("MacOS");

        // добавить элементы в список браузеров
        browser.add("Netscape 1.1");
        browser.add("Netscape 2.x");
        browser.add("Netscape 3.x");
        browser.add("Netscape 4.x");

        browser.add("Internet Explorer 2.0");
        browser.add("Internet Explorer 3.0");
        browser.add("Internet Explorer 4.0");

        browser.add("Lynx 2.4");

        browser.select(1);

        // добавить списки в окно
        add(os);
        add(browser);

        // зарегистрироваться для приема action-событий
        os.addActionListener(this);
        browser.addActionListener(this);
    }
}
```

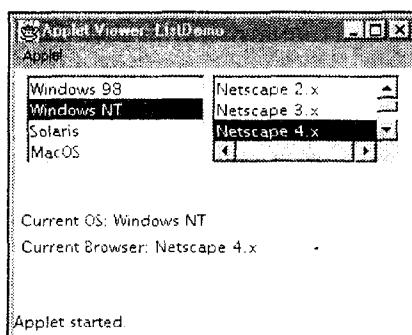
```

public void actionPerformed(ActionEvent ae) {
    repaint();
}

// отобразить текущие выборы
public void paint(Graphics g) {
    int idx[];

    msg = "Current OS: ";
    idx = os.getSelectedIndexes();
    for(int i=0; i<idx.length; i++)
        msg += os.getItem(idx[i]) + " ";
    g.drawString(msg, 6, 120);
    msg = "Current Browser: ";
    msg += browser.getSelectedItem();
    g.drawString(msg, 6, 140);
}
}

```



Пример вывода, сгенерированного аплетом ListDemo, показан на рис. 22.6. Обратите внимание, что список браузеров имеет полосу прокрутки, так как все элементы не вставили в число строк, указанное при его создании.

**Рис. 22.6.** Пример вывода аплета ListDemo

## Управление полосами прокрутки

Полосы прокрутки (scroll bars) используются для выбора непрерывных значений из некоторого интервала с конечными границами. Полосы прокрутки могут быть ориентированы горизонтально или вертикально. Полоса прокрутки фактически является композицией нескольких индивидуальных частей. На каждом конце полосы имеется кнопка-стрелка, которую можно нажимать (щелчком мыши), чтобы переместить текущее значение полосы прокрутки на одну позицию в направлении стрелки. Текущее значение полосы прокрутки относительно ее минимальных и максимальных значений обозначено ползунком<sup>1</sup> (или бегунком) полосы прокрутки. Ползунок может перетаскиваться пользователем в новую позицию. Пользователь может щел-

<sup>1</sup> Этот элемент на полосе прокрутке также называется маркером полосы прокрутки. — Примеч. ред.

кать мышью в фоновых частях полосы, находящихся с обеих сторон ползунка, чтобы заставить бегунок перескакивать в этом направлении с некоторым приращением, большим чем 1. Обычно это действие приводит к некоторой форме листания страницы вверх (page up) или вниз (page down). Полосы прокрутки инкапсулированы в классе Scrollbar. В Scrollbar определены следующие конструкторы:

```
Scrollbar()
Scrollbar(int style)
Scrollbar(int style, int initialValue, int thumbSize, int min, int max)
```

Первая форма создает вертикальную полосу прокрутки. Вторая и третья — позволяют указать ориентацию полосы прокрутки. Если параметр *style* задается как Scrollbar.VERTICAL, то создается вертикальная полоса прокрутки, если — как Scrollbar.HORIZONTAL, то — горизонтальная. В третьей форме конструктора начальное значение полосы прокрутки передается в параметре *initialValue*, а высота ползунка — в *thumbSize*. Минимальное и максимальное значения для полосы прокрутки указываются в параметрах *min* и *max*.

Если вы создаете полосу прокрутки при помощи одного из первых двух конструкторов, то перед использованием нужно установить ее параметры, вызывая метод *setValues()* следующего формата:

```
void setValues(int initialValue, int thumbSize, int min, int max)
```

Параметры имеют те же значения, как в только что описанном третьем конструкторе.

Чтобы получить текущее значение полосы прокрутки, вызовите метод *getValue()*. Он возвращает текущую установку. Чтобы установить текущее значение, вызовите *setValue()*. Форматы этих методов:

```
int getValue()
void setValue(int newValue)
```

Здесь *newValue* определяет новое значение для полосы прокрутки. Когда вы устанавливаете значение, ползунок внутри полосы прокрутки будет перемещен в позицию, отражающую новое значение.

Вы можете также отыскивать минимальное и максимальное значения через показанные ниже методы *getMinimum()* и *getMaximum()*:

```
int getMinimum()
int getMaximum()
```

Для прокрутки вверх или вниз на одну строку по умолчанию используется (строчное) приращение<sup>1</sup>, равное 1. Можно изменить это приращение, вызыв-

<sup>1</sup> Приращение — это величина, которая добавляется (к) или вычитается из текущего значения полосы прокрутки каждый раз, когда нажимаются (щелчком мыши) кнопки-стрелки на концах полосы прокрутки или области полосы, обрамляющие движок. В зависимости от места на полосе, где выполняется щелчок (кнопки-стрелки или обрамляющие области) различают строчные и страницные приращения. — Примеч. пер.

вая метод `setUnitIncrement()`. По умолчанию, страничные (page-up и page-down) приращения равны 10. Это значение можно изменять, вызывая `setBlockIncrement()`. Форматы этих методов:

```
void setUnitIncrement(int newIncr)
void setBlockIncrement(int newIncr)
```

## Обработка полос прокрутки

Для обработки событий полосы прокрутки следует реализовать интерфейс `AdjustmentListener`. Каждый раз, когда пользователь взаимодействует с полосой прокрутки, генерируется объект `AdjustmentEvent`. Чтобы определить тип настройки, можно использовать его метод `getAdjustmentType()`. Типы событий настройки следующие:

- `BLOCK_DECREMENT`. Событие page-down было сгенерировано.
- `BLOCK_INCREMENT`. Событие page-up было сгенерировано.
- `TRACK`. Абсолютное tracking-событие было сгенерировано.
- `UNIT_DECREMENT`. Кнопка "строка-вниз" (line-down) на полосе прокрутки была нажата.
- `UNIT_INCREMENT`. Кнопка "строка-вверх" (line-up) на полосе прокрутки была нажата.

Следующий пример создает как вертикальную, так и горизонтальную полосы прокрутки. На экране отображаются их текущие установки. Если вы перетаскиваете мышью элементы, находящиеся внутри окна, то координаты каждого drag-события (события перетаскивания мыши) используются для обновления полосы прокрутки. В текущей drag-позиции указателя мыши внутри окна отображается звездочка (asterisk).

```
// Демонстрирует полосы прокрутки.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SBDemo" width=300 height=200>
</applet>
*/
public class SBDemo extends Applet
    implements AdjustmentListener, MouseMotionListener {
    String msg = "";
    Scrollbar vertSB, horzSB;

    public void init() {
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));

        vertSB = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, height);
        horzSB = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, width);
```

```

add(vertSB);
add(horzSB);

// зарегистрироваться для приема adjustment-событий
vertSB.addAdjustmentListener(this);
horzSB.addAdjustmentListener(this);

addMouseMotionListener(this);
}

public void adjustmentValueChanged(AdjustmentEvent ae) {
    repaint();
}

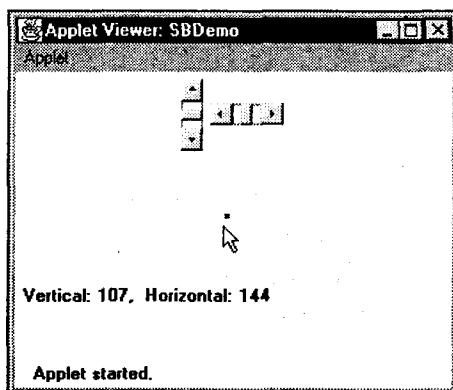
// обновить полосы прокрутки, чтобы отразить перетаскивание мыши
public void mouseDragged(MouseEvent me) {
    int x = me.getX();
    int y = me.getY();
    vertSB.setValue(y);
    horzSB.setValue(x);
    repaint();
}

// необходим для MouseMotionListener
public void mouseMoved(MouseEvent me) {
}

// отобразить текущее значение полос прокрутки
public void paint(Graphics g) {
    msg = "Vertical: " + vertSB.getValue();
    msg += ", Horizontal: " + horzSB.getValue();
    g.drawString(msg, 6, 160);

    // показать текущую drag-позицию мыши
    g.drawString("*", horzSB.getValue(), vertSB.getValue());
}
}
}

```



Пример вывода апплета SBDemo показан на рис. 22.7.

**Рис. 22.7.** Пример вывода апплета SBDemo

## Использование класса *TextField*

Класс *TextField* реализует одностороннюю область ввода текста, обычно называемую *элементами редактирования* (*edit control*). Текстовые поля дают возможность пользователю вводить строки и редактировать текст, используя клавиши-стрелки, сочетание клавиш для операций "вырезать" и "вставить", а также выборки мышью. *TextField* — подкласс *TextComponent*. *TextField* определяет следующие конструкторы:

```
TextField()
TextField(int numChars)
TextField(String str)
TextField(String str, int numChars)
```

Первая форма создает заданное текстовое поле по умолчанию. Вторая — создает текстовое поле шириной *numChars* символов. Третья форма инициализирует текстовое поле со строкой, содержащейся в *str*. Четвертая — инициализирует текстовое поле и устанавливает его ширину.

*TextField* (и его суперкласс *TextComponent*) обеспечивает несколько методов, которые позволяют использовать текстовое поле. Чтобы получить строку, содержащуюся в текущий момент в текстовом поле, вызовите метод *getText()*, а для установки текста вызовите *setText()*. Форматы этих методов следующие:

```
String getText()
void setText(String str)
```

Здесь *str* — новая строка.

Пользователь может выбирать часть текста в текстовом поле. Метод *select()* позволяет выбирать часть текста под программным управлением. Вызывая *getSelectedText()*, ваша программа может получить текущий выбранный текст. Формат этих методов:

```
String getSelectedText()
void select(int startIndex, int endIndex)
```

Метод *getSelectedText()* возвращает выбранный текст, а метод *select()* выбирает символы, начинающиеся в *startIndex* и заканчивающиеся в *endIndex* - 1.

Вызовом *setEditable()* можно управлять возможностью редактирования (изменения содержания) текстового поля пользователем. Вызовом *isEditable()* можно определить, редактируемо ли данное поле. Форматы этих методов:

```
boolean isEditable()
void setEditable(boolean canEdit)
```

`isEditable()` возвращает `true`, если текст может быть изменен, и `false` — в противном случае. В методе `setEditable()`, если `canEdit` `true`, то текст может быть изменен, а если `false` — не может.

Если нужно, чтобы пользователь мог вводить текст, который бы не отображался в секретном поле (типа пароля), то следует отключить отображение на экране вводимых символов, вызывая `setEchoChar()`. Данный метод определяет одиночный символ (эхо-символ), который будет отображаться при вводе каждого символа (таким образом, фактически вводимые символы не будут показаны в поле). С помощью метода `echoCharIsSet()` можно проверить, находится ли текстовое поле в этом режиме. Вызывая метод `getEchoChar()` можно отыскать и извлечь эхо-символ. Форматы перечисленных методов следующие:

```
void setEchoChar(char ch)
boolean echoCharIsSet()
char getEchoChar()
```

Здесь `ch` определяет эхо-символ, который будет отображаться на экране.

## Обработка *TextField*

Так как текстовые поля выполняют свои собственные функции редактирования, ваша программа вообще не будет откликаться на индивидуальные key-события, которые происходят в текстовом поле. Однако вы, может быть, захотите обработать нажатие клавиши `<Enter>`. Когда это происходит, генерируется action-событие (типа "действие").

Пример, который создает классическое окно с именем и паролем пользователя:

```
// Демонстрирует текстовое поле.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/
public class TextFieldDemo extends Applet
    implements ActionListener {

    TextField name, pass;

    public void init() {
        Label namep = new Label("Name: ", Label.RIGHT);
        Label passp = new Label("Password: ", Label.RIGHT);
        name = new TextField(12);
```

```

pass = new TextField(8);
pass.setEchoChar('*');

add(namep);
add(name);
add(passp);
add(pass);

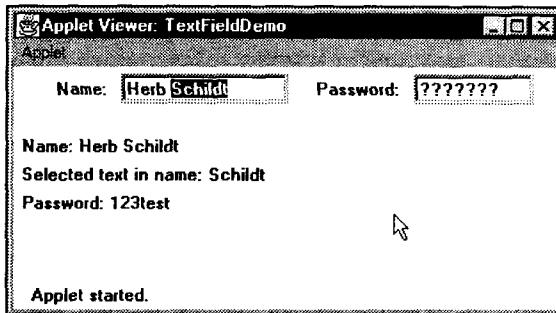
// регистрироваться для получения action-событий
name.addActionListener(this);
pass.addActionListener(this);
}

// клавиша <Enter>, нажатая пользователем
public void actionPerformed(ActionEvent ae) {
    repaint();
}

public void paint(Graphics g) {
    g.drawString("Name: " + name.getText(), 6, 60);
    g.drawString("Selected text in name: "
                + name.getSelectedText(), 6, 80);
    g.drawString("Password: " + pass.getText(), 6, 100);
}
}

```

Пример вывода апплета `TextFieldDemo` показан на рис. 22.8.



**Рис. 22.8.** Пример вывода апплета `TextFieldDemo`

## Использование `TextArea`

Иногда односторонний текстовый ввод не достаточен для данной задачи. Чтобы обрабатывать эти ситуации, AWT включает простой многострочный редактор, по имени `TextArea`. Конструкторы `TextArea`:

```

TextArea()
TextArea(int numLines, int numChars)

```

```
TextArea(String str)
TextArea(String str, int numLines, int numChars)
TextArea(String str, int numLines, int numChars, int sBars)
```

Здесь *numLines* определяет высоту текстовой области (в строках); *numChars* — ее ширину (в символах); *str* — начальный текст. В пятой форме можно определить полосы прокрутки, если вы хотите, чтобы элемент управления их имел. *sBars* должен принимать одно из следующих значений:

- SCROLLBARS\_BOTH
- SCROLLBARS\_HORIZONTAL\_ONLY
- SCROLLBARS\_NONE
- SCROLLBARS\_VERTICAL\_ONLY

TextArea — подкласс TextComponent. Поэтому он поддерживает методы *getText()*, *setText()*, *getSelectedText()*, *select()*, *isEditable()* и *setEditable()*, описанные в предыдущем разделе.

TextArea добавляет следующие методы:

```
void append(String str)
void insert(String str, int index)
void replaceRange(String str, int startIndex, int endIndex)
```

Метод *append()* добавляет строку, указанную в *str*, к концу текущего текста. *insert()* вставляет строку, передаваемую в *str*, в позицию, указанную в параметре *index*. Чтобы заменить текст, вызовите метод *replaceRange()*. Он заменяет символы от *startIndex* до *endIndex* — 1 текстом, передаваемым в *str*.

Текстовые области — почти автономный элемент управления. Ваша программа фактически не берет на себя никакого дополнительного администрирования. Текстовые области генерируют события получения и потери фокуса (*got-focus* и *lost-focus events*). Обычно такая программа просто выводит на экран текст, когда это необходимо.

Следующая программа создает элемент управления TextArea:

```
// Демонстрирует TextArea.
import java.awt.*;
import java.applet.*;
/*
<applet code="TextAreaDemo" width=300 height=250>
</applet>
*/
public class TextAreaDemo extends Applet {
    public void init() {
```

```

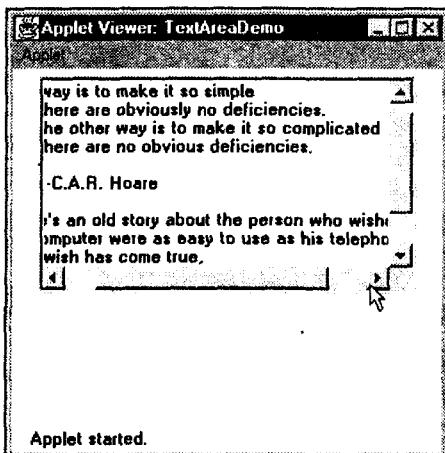
String val = "There are two ways of constructing " +
    "a software design.\n" +
    "One way is to make it so simple\n" +
    "that there are obviously no deficiencies.\n" +
    "And the other way is to make it so complicated\n" +
    "that there are no obvious deficiencies.\n\n" +
    "-C.A.R. Hoare\n\n" +
    "There's an old story about the person who wished\n" +
    "his computer were as easy to use as his telephone.\n" +
    "That wish has come true,\n" +
    "since I no longer know how to use my telephone.\n\n" +
    "-Bjarne Stroustrup, AT&T, (inventor of C++);"

```

```

TextArea text = new TextArea(val, 10, 30);
add(text);
}
}

```



Пример вывода апплета TextAreaDemo представлен на рис. 22.9.

Рис. 22.9. Окно апплета TextAreaDemo

## Понятие менеджера компоновки

Все компоненты, которые мы показывали до сих пор, были размещены (позиционированы) в контейнере менеджером компоновки, заданным по умолчанию. Как мы упомянули в начале этой главы, менеджер компоновки автоматически размещает элементы управления в пределах окна, используя некоторый тип алгоритма. Если вы программируете для других GUI-сред, таких как Windows, то привыкли к размещению элементов управления вручную. Хотя элементы управления Java тоже можно размещать вручную, в этом нет необходимости по двум причинам. Во-первых, очень утомитель-

но вручную размещать большое количество компонентов. Во-вторых, иногда, в тот момент, когда нужно размещать некоторый элемент, информация о его размерах оказывается недоступной (чаще всего потому, что необходимые для получения этой информации компоненты комплекта инструментов еще не были реализованы). Правильное же размещение компонента без определенных размеров относительно других компонентов весьма проблематично. Поэтому разработаны специальные программные компоненты (менеджеры компоновки) для автоматического управления компоновкой.

Каждый объект типа `Container` имеет связанный с ним менеджер компоновки. *Менеджер компоновки* — это экземпляр некоторого класса, который реализует интерфейс `LayoutManager`. Менеджер компоновки устанавливается методом `setLayout()`. Если вызов `setLayout()` не сделан, то используется менеджер компоновки по умолчанию (поточный). Менеджер компоновки применяется всякий раз, когда контейнер изменяется в размерах (или устанавливается впервые — с первоначальными размерами), чтобы позиционировать каждый из его внутренних компонентов.

Метод `setLayout()` имеет следующую общую форму:

```
void setLayout(LayoutManager layoutObj)
```

Здесь `layoutObj` — ссылка на необходимый менеджер компоновки. Если вы хотите отключить менеджер компоновки и позиционировать компоненты вручную, передавайте в качестве параметра `layoutObj` значение `null` (пустой указатель). Если вы это сделаете, то будете вынуждены определять форму и позицию каждого компонента вручную, указывая метод `setBounds()`, определенный в `Component`. Но, как правило, следует использовать менеджер компоновки.

Каждый менеджер компоновки хранит и отслеживает список имен компонентов. Всякий раз, когда вы добавляете компонент в контейнер, менеджер компоновки получает соответствующее уведомление. Если нужно изменять размеры контейнера, менеджер компоновки консультируется со своими методами `minimumLayoutSize()` и `preferredLayoutSize()`. Каждый компонент, который управляется менеджером компоновки, содержит методы `getPreferredSize()` и `getMinimumSize()`. Они возвращают предпочтительный и минимальный размер, требуемый для отображения каждого компонента. Если это вообще возможно, менеджер компоновки будет удовлетворять эти запросы (для поддержки целостности политики компоновки). Для элементов управления своего подкласса можно переопределить указанные методы. Иначе используются значения по умолчанию.

Java имеет несколько предопределенных классов `LayoutManager`, которые описываются ниже. Используйте тот менеджер компоновки, который наилучшим образом подходит вашему приложению.

## Менеджер *FlowLayout*

*FlowLayout* — это менеджер *поточной* компоновки. Напомним, что если метод *setLayout()* не устанавливает никакой иной компоновщик, то данный компоновщик используется по умолчанию. Этот менеджер использовали все предшествующие примеры. *FlowLayout* реализует простой стиль размещения, похожий на поток слов в текстовом редакторе. Компоненты размещаются от левого верхнего угла окна, слева направо и сверху вниз. Когда нет больше компонентов, пригодных для размещения на строке, очередной компонент размещается в начале следующей строки. Выше и ниже, справа и слева, а также между каждым компонентом оставляется маленькое пространство. Конструкторы *FlowLayout*:

```
FlowLayout()
FlowLayout(int how)
FlowLayout(int how, int horz, int vert)
```

Первая форма создает размещение по умолчанию, которое выравнивает компоненты по центру и оставляет пять пикселов пробела между каждым компонентом. Вторая форма позволяет определить, как выравнивается каждая строка. Допустимы следующие значения параметра *how*:

- `FlowLayout.LEFT`
- `FlowLayout.CENTER`
- `FlowLayout.RIGHT`

Эти значения определяют выравнивание влево, по центру и вправо, соответственно. Третья форма позволяет задавать (в параметрах *horz* и *vert*) горизонтальный и вертикальный пробел, оставляемый между компонентами (в форме целого числа пикселов).

Ниже приводится версия апплета *CheckboxDemo*, показанного ранее в этой главе, модифицированная так, чтобы использовать поточное размещение, выровненное по левой границе.

```
// Использует поточное размещение с левым выравниванием.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="FlowLayoutDemo" width=250 height=200>
</applet>
*/
public class FlowLayoutDemo extends Applet
    implements ItemListener {

    String msg = "";
    Checkbox Win98, winNT, solaris, mac;
```

```
public void init() {
    // установить поточное размещение с левым выравниванием
    setLayout(new FlowLayout(FlowLayout.LEFT));

    Win98 = new Checkbox("Windows 98", null, true);
    winNT = new Checkbox("Windows NT");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("MacOS");

    add(Win98);
    add(winNT);
    add(solaris);
    add(mac);

    // зарегистрироваться для приема item-событий
    Win98.addItemListener(this);
    winNT.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
}

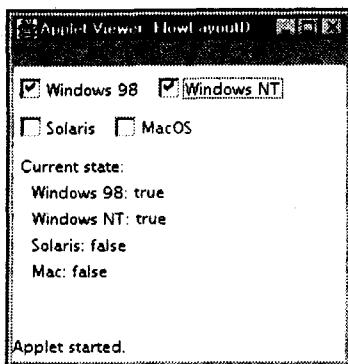
// перерисовать, когда изменяется состояние флагка
public void itemStateChanged(ItemEvent ie) {
    repaint();
}

// показать текущее состояние флагков
public void paint(Graphics g) {
    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = " Windows 98: " + Win98.getState();
    g.drawString(msg, 6, 100);
    msg = " Windows NT: " + winNT.getState();
    g.drawString(msg, 6, 120);
    msg = " Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = " Mac: " + mac.getState();
    g.drawString(msg, 6, 160);
}
}
```

Пример вывода, сгенерированного апплетом `FlowLayoutDemo`, представлен на рис. 22.10. Сравните это с выводом апплета `CheckboxDemo`, показанного ранее на рис. 22.3.

## Класс *BorderLayout*

Класс `BorderLayout` реализует *граничный* стиль компоновки, используемый для окон верхнего уровня. Он имеет четыре узких компонента фиксированной



**Рис. 22.10.** Окно апплета FlowLayoutDemo

ширины по краям и один — в виде большой области — в центре. Четыре краевых компонента называют Север (North), Юг (South), Восток (East) и Запад (West). Средняя область называется Центр (Center). Конструкторы, определенные в BorderLayout:

```
BorderLayout()
BorderLayout(int horz, int vert)
```

Первая форма создает граничное размещение, используемое по умолчанию. Вторая — позволяет указывать количество (в параметрах *horz* и *vert*, соответственно) горизонтальных и вертикальных пробелов, оставляемых между компонентами. BorderLayout определяет следующие константы, которые специфицируют области размещения:

- |  |   |
|--|---|
| <input type="checkbox"/> BorderLayout.CENTER | <input type="checkbox"/> BorderLayout.SOUTH |
| <input type="checkbox"/> BorderLayout.EAST   | <input type="checkbox"/> BorderLayout.WEST  |
| <input type="checkbox"/> BorderLayout.NORTH  |   |

При добавлении компонентов вы будете использовать эти константы со следующей формой метода add(), который определен в классе Container:

```
void add(Component compObj, Object region);
```

Здесь *compObj* — компонент, который будет добавлен, а *region* специфицирует область размещения, куда компонент будет добавлен.

Пример граничного размещения (менеджером BorderLayout) с компонентом в каждой области компоновки:

```
// Демонстрирует BorderLayout.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="BorderLayoutDemo" width=400 height=200>
</applet>
*/
```

```

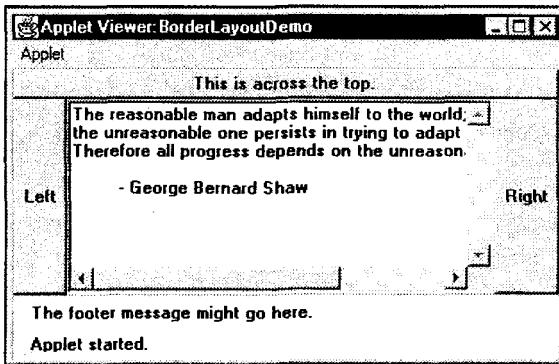
public class BorderLayoutDemo extends Applet {
    public void init() {
        setLayout(new BorderLayout());
        add(new Button("This is across the top."),
            BorderLayout.NORTH);
        add(new Label("The footer message might go here."),
            BorderLayout.SOUTH);
        add(new Button("Right"), BorderLayout.EAST);
        add(new Button("Left"), BorderLayout.WEST);

        String msg = "The reasonable man adapts " +
            "himself to the world;\n" +
            "the unreasonable one persists in " +
            "trying to adapt the world to himself.\n" +
            "Therefore all progress depends " +
            "on the unreasonable man.\n\n" +
            "           - George Bernard Shaw\n\n";
    }

    add(new TextArea(msg), BorderLayout.CENTER);
}
}

```

Пример вывода апплета BorderLayoutDemo представлен на рис. 22.11.



**Рис. 22.11.** Окно апплета BorderLayoutDemo

## Использование вставок

Иногда нужно оставить немного пустого места между контейнером, который хранит компоненты, и окном, содержащим контейнер. Для этого необходимо переопределить метод `getInsets()`, который определен в классе `Container`. Эта функция возвращает объект типа `Insets`, содержащий верхнюю, нижнюю, левую и правую вставки, которые используются во время отображения контейнера. Менеджер компоновки использует эти значения

при вставке компонентов, когда размещает окно вокруг контейнера. Конструктор Insets:

```
Insets(int top, int left, int bottom, int right)
```

Значения, пересылаемые параметрами *top*, *left*, *bottom*, *right* определяют количество пробельного пространства (в пикселях) между контейнером и включающим его окном.

Метод *getInsets()* имеет общую форму:

```
Insets getInsets()
```

При переопределении одного из этих методов нужно возвратить новый *Insets*-объект, который содержит необходимую пробельную вставку.

Ниже показан предшествующий пример с компоновщиком *BorderLayout*, измененный так, что он вставляет компоненты в десяти пикселях от каждой границы. Чтобы лучше видеть вставки, установлен голубой цвет фона.

```
// Демонстрирует BorderLayout со вставками.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="InsetsDemo" width=400 height=200>
</applet>
*/
public class InsetsDemo extends Applet {
    public void init() {
        // установить цвет фона так, чтобы вставки были легко видимы
        setBackground(Color.cyan);
        setLayout(new BorderLayout());
        add(new Button("This is across the top."), BorderLayout.NORTH);
        add(new Label("The footer message might go here."),
            BorderLayout.SOUTH);
        add(new Button("Right"), BorderLayout.EAST);
        add(new Button("Left"), BorderLayout.WEST);

        String msg = "The reasonable man adapts " +
            "himself to the world;\n" +
            "the unreasonable one persists in " +
            "trying to adapt the world to himself.\n" +
            "Therefore all progress depends " +
            "on the unreasonable man.\n\n" +
            "— George Bernard Shaw\n\n";
        add(new TextArea(msg), BorderLayout.CENTER);
    }
}
```

```
// добавить вставки
public Insets getInsets() {
    return new Insets(10, 10, 10, 10);
}
}
```

Вывод апплета InsetsDemo представлен на рис. 22.12.

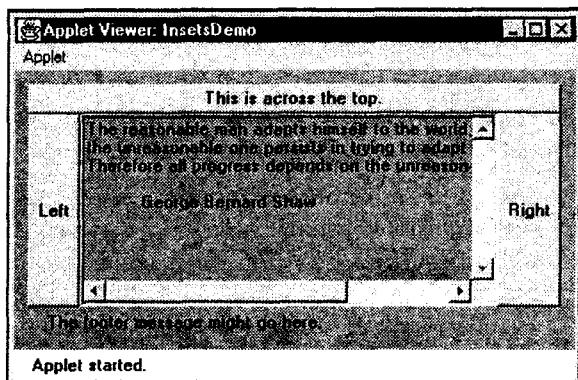


Рис. 22.12. Окно апплета InsetsDemo

## Менеджер GridLayout

Менеджер GridLayout располагает компоненты в двумерной сетке<sup>1</sup>. Число строк и столбцов сетки следует определять при создании экземпляра (объекта) GridLayout. Конструкторы, определенные в GridLayout:

```
GridLayout()
GridLayout(int numRows, int numColumns)
GridLayout(int numRows, int numColumns, int horz, int vert)
```

Первая форма создает сеточное размещение с одиночным столбцом. Вторая — сеточное размещение с указанным числом строк и столбцов. Третья форма позволяет определять горизонтальный и вертикальный пробелы, оставляемые между компонентами (в параметрах *horz* и *vert*, соответственно). Параметр *numRows* или *numColumns* может быть нулевым. Нулевая спецификация *numRows* допускает столбцы с неограниченной длиной. Нулевая спецификация *numColumns* допускает строки с неограниченной длиной.

Пример программы, которая создает сетку 4×4 и вставляет в нее 15 кнопок, каждая из которых помечена своим индексом (порядковым номером):

```
// Демонстрирует GridLayout.
import java.awt.*;
```

<sup>1</sup> Иногда такую компоновку называют *табличной*. — Примеч. пер.

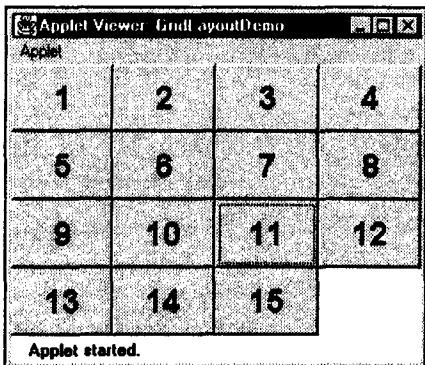
```

import java.applet.*;
/*
<applet code="GridLayoutDemo" width=300 height=200>
</applet>
*/
public class GridLayoutDemo extends Applet {
    static final int n = 4;
    public void init() {
        setLayout(new GridLayout(n, n));
        setFont(new Font("SansSerif", Font.BOLD, 24));

        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                int k = i * n + j;
                if(k > 0)
                    add(new Button(" " + k));
            }
        }
    }
}

```

Вывод, сгенерированный апплетом GridLayoutDemo, представлен на рис. 22.13.



### Совет

Вы могли бы попробовать использовать этот пример, как отправную точку для головоломки "игра в 15".

**Рис. 22.13.** Окно апплета GridLayoutDemo

## Класс CardLayout

Класс *CardLayout* уникalen среди других менеджеров компоновки тем, что хранит несколько различных размещений. Каждое размещение можно представлять отдельной пронумерованной картой в колоде, которая может быть перетасована так, чтобы в данный момент наверху находилась любая карта. Это может быть полезно для интерфейсов пользователя с необязательными компонентами, которые можно динамически включать и выключать при вводе пользователя. Вы можете подготовить другие компоновки и сделать их

скрытыми, но готовыми к активизации, когда это необходимо. CardLayout обеспечивает два конструктора:

```
CardLayout()  
CardLayout(int horz, int vert)
```

Первая форма создает карточную компоновку по умолчанию. Вторая — позволяет указывать горизонтальный и вертикальный пробел, оставляемый между компонентами (в параметрах *horz* и *vert*, соответственно).

Использование карточной компоновки требует немного большей работы, чем в других компоновках. Карты обычно содержатся в объекте типа *Panel*. Эта панель должна выбрать *CardLayout*, как свой менеджер компоновки. Карты, которые формируют колоду, также обычно являются объектами типа *Panel*. Таким образом, вы должны создать панель, которая содержит колоду и панель для каждой карты в колоде. Затем, вы добавляете к соответствующей панели компоненты, которые формируют каждую карту. Далее, вы добавляете карточные панели к панели менеджера *CardLayout*. Наконец, вы добавляете эту панель к главной панели апплета. Как только эти шаги закончены, вы должны обеспечить для пользователя некоторый способ выбора между панелями. Один из обычных подходов состоит в том, чтобы ввести по одной командной кнопке для каждой карты в колоде.

Когда карточные панели добавляются в панель колоды (т. е. компоновщица), им обычно присваиваются имена. Для добавления карт в панель колоды в большинстве случаев будет использоваться следующая форма метода *add()*:

```
void add(Component panelObj, Object name);
```

Здесь *name* — строка, которая определяет имя карты, чья панель указана в параметре *panelObj*.

После того как вы создали колоду, программа активизирует карту, вызывая один из следующих методов, определенных в *CardLayout*:

```
void first(Container deck)  
void last(Container deck)  
void next(Container deck)  
void previous(Container deck)  
void show(Container deck, String cardName)
```

Здесь *deck* — ссылка на контейнер (обычно — панель), который содержит карты, а *cardName* — имя карты. Вызов *first()* заставляет показать первую карту в колоде. Чтобы отобразить последнюю карту, вызовите *last()*. Для показа очередной карты вызывайте *next()*, а предыдущей — *previous()*. Как *next()*, так и *previous()* автоматически повторяют цикл сверху или снизу колоды, соответственно. Метод *show()* отображает карту, чье имя передается в *cardName*.

Следующий пример создает двухуровневую колоду, которая позволяет пользователю выбирать операционную систему. Операционные системы на базе Windows отображаются на одной карте, Macintosh и Solaris — на другой.

```
// Демонстрирует CardLayout.  
import java.awt.*;  
import java.awt.event.*;  
import java.applet.*;  
/*  
 <applet code="CardLayoutDemo" width=300 height=100>  
 </applet>  
*/  
public class CardLayoutDemo extends Applet  
    implements ActionListener, MouseListener {  
  
    Checkbox Win98, winNT, solaris, mac;  
    Panel osCards;  
    CardLayout cardLO;  
    Button Win, Other;  
  
    public void init() {  
        Win = new Button("Windows");  
        Other = new Button("Other");  
        add(Win);  
        add(Other);  
  
        cardLO = new CardLayout();  
        osCards = new Panel();  
        osCards.setLayout(cardLO); // установить panel-компоновку  
                                // для card-компоновки  
  
        Win98 = new Checkbox("Windows 98", null, true);  
        winNT = new Checkbox("Windows NT");  
        solaris = new Checkbox("Solaris");  
        mac = new Checkbox("MacOS");  
  
        // добавить в панель Windows флагки  
        Panel winPan = new Panel();  
        winPan.add(Win98);  
        winPan.add(winNT);  
  
        // добавить в панель другие OS-флагки  
        Panel otherPan = new Panel();  
        otherPan.add(solaris);  
        otherPan.add(mac);  
  
        // добавить панели к панели колоды карт  
        osCards.add(winPan, "Windows");  
        osCards.add(otherPan, "Other");
```

```

// добавить карты к главной панели апплета
add(osCards);

// зарегистрироваться для приема action-событий
Win.addActionListener(this);
Other.addActionListener(this);

// зарегистрировать события мыши
addMouseListener(this);

}

// цикл карт в панели
public void mousePressed(MouseEvent me) {
    cardLO.next(osCards);
}

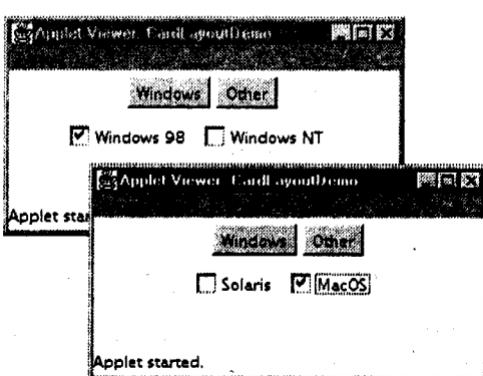
// обеспечить пустые реализации для других методов MouseListener
public void mouseClicked(MouseEvent me) {
}

public void mouseEntered(MouseEvent me) {
}
public void mouseExited(MouseEvent me) {
}
public void mouseReleased(MouseEvent me) {
}

public void actionPerformed(ActionEvent ae) {
    if(ae.getSource() == Win) {
        cardLO.show(osCards, "Windows");
    }
    else {
        cardLO.show(osCards, "Other");
    }
}
}
}

```

На рис. 22.14 представлен вывод, сгенерированный апплетом CardLayoutDemo. Каждая карта активизируется нажатием своей кнопки. Допустим также циклический просмотр карт (щелчками мыши).



**Рис. 22.14.** Окно апплета CardLayoutDemo

## Панели меню и меню

Окно верхнего уровня может содержать связанную с ним панель меню (menu bar) верхнего уровня<sup>1</sup>. В строке этого меню отображается список выбираемых элементов (выборов, choices). Каждый выбираемый элемент панели меню верхнего уровня связан с выпадающим (drop-down) меню, содержащим свои выбираемые элементы. Эти элементы меню реализованы в Java следующими классами: `MenuBar`, `Menu` и `MenuItem`. В общем случае, панель меню содержит один или несколько объектов типа `Menu`. Каждый `Menu`-объект содержит список. Любой объект типа `MenuItem` представляет нечто, что может быть выбрано пользователем. Элементом списка в `Menu`-объекте может быть не только `MenuItem`-объект, но и другой `Menu`-объект со своим списком `MenuItem`-объектов. Таким образом может быть создана *иерархия вложенных подменю*. В меню можно также включать специальные элементы — с отметками. Они являются меню-элементами типа `CheckboxMenuItem`, когда они выбираются, перед ними проставляется специальная метка (check mark<sup>2</sup>).

Чтобы создать панель меню, сначала создают экземпляр типа `MenuBar`. Этот класс определяет только конструктор по умолчанию. Затем создаются экземпляры типа `Menu`, которые определяют выбираемые элементы, расположенные на строке меню. Конструкторы класса `Menu`:

```
Menu()
Menu(String optionName)
Menu(String optionName, boolean removable)
```

Здесь `optionName` специфицирует имя выбираемого элемента меню-панели. Если параметр `removable` — `true`, выпадающее меню может быть удалено (из иерархии) и отправлено "в свободное плавание" (по окну). В противном случае это меню останется прикрепленным к панели меню. (Открепляемые меню зависят от реализации.) Первая форма создает пустое меню.

Индивидуальные пункты меню имеют тип `MenuItem`. В этом классе определены следующие конструкторы:

```
MenuItem()
MenuItem(String itemName)
MenuItem(String itemName, MenuShortcut keyAccel)
```

Где `itemName` — имя, показанное в меню; `keyAccel` — клавиши быстрого доступа для этого пункта.

<sup>1</sup> В терминологии GUI такую панель называют *главным меню*, а ее выбираемые элементы — *пунктами главного меню*. — Примеч. пер.

<sup>2</sup> По форме метка такого элемента часто совпадает с меткой элемента управления типа "флажок", хотя ее можно задавать и произвольно. — Примеч. пер.

Вы можете деактивизировать или активизировать пункт меню, используя метод `setEnabled()`. Его формат:

```
void setEnabled(boolean enabledFlag)
```

Если параметр `enabledFlag` — `true`, пункт меню активизируется, если `false` — пункт деактивизируется (блокируется).

Можно определять состояние элемента, вызывая `isEnabled()`. Формат этого метода:

```
boolean isEnabled()
```

`isEnabled()` возвращает `true`, если пункт меню активизирован. Иначе, возвращается `false`.

Можно изменять имя пункта меню, вызывая метод `setLabel()`, и извлекать имя текущего пункта, вызывая метод `getLabel()`. Форматы этих методов:

```
void setLabel(String newName)
```

```
String getLabel()
```

где `newName` устанавливает новое имя пункта меню. `getLabel()` возвращает текущее имя.

Можно создавать помечаемый пункт меню, используя подкласс `MenuItem` с именем `CheckboxMenuItem`. Он имеет следующие конструкторы:

```
CheckboxMenuItem()
```

```
CheckboxMenuItem(String itemName)
```

```
CheckboxMenuItem(String itemName, boolean on)
```

где `itemName` — имя, показываемое в меню. Отмечаемые пункты работают как флаги. Каждый раз при выборке их состояние изменяется. В первых двух формах создается пункт без отметки. В третьей форме, если параметр `on` указан как `true`, создается изначально помеченный пункт, иначе — не-помеченный.

Состояние помечаемого пункта можно получить, вызывая `getState()`. Для установки состояния такого пункта используется метод `setState()`. Форматы этих методов:

```
boolean getState()
```

```
void setState(boolean checked)
```

Если элемент отмечен, `getState()` возвращает `true`, иначе — `false`. Чтобы пометить пункт, передайте в `setState()` значение `true`. Чтобы удалить метку, передайте `false`.

Как только вы создали пункт меню, следует добавить элемент в объект типа `Menu` с помощью метода `add()`, который имеет следующую общую форму:

```
MenuItem add(MenuItem item)
```

где `item` — добавляемый пункт. Пункты добавляются к меню в порядке, в котором выполняются вызовы `add()`. Объект `item` используется и как возвращаемое значение.

Как только вы добавили все элементы к `Menu`-объекту, можно добавить этот объект в строку меню, используя следующую версию метода `add()`, определенную в `MenuBar`:

```
Menu add(Menu menu)
```

где `menu` — объект, представляющий как добавляемое меню, так и возвращаемое значение.

Меню генерируют события только тогда, когда выбираются элементы типа `MenuItem` или `CheckboxMenuItem`. Такие события не генерируются, например, когда обращаются к строке меню, чтобы открыть выпадающее меню. При каждой выборке обычного (без отметки) пункта меню генерируется объект `ActionEvent`. При очередном сбросе или установке флажка помеченного пункта меню генерируется объект `ItemEvent`. Таким образом, чтобы обработать эти меню-события, нужно реализовать интерфейсы блоков прослушивания `ActionListener` и `ItemListener`.

Метод `getItem()` класса `ItemEvent` возвращает ссылку на элемент, который генерировал это событие. Общая форма этого метода:

```
Object getItem()
```

Ниже показан пример, который добавляет ряд вложенных меню в окно раскрывающегося меню (второго уровня). Выбранный элемент выделяется в окне инверсной строкой. Отображается также состояние двух помечаемых пунктов меню (один — включен, другой — нет).

```
// Иллюстрирует меню.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MenuDemo" width=250 height=250>
</applet>
*/
// создать подкласс для Frame
class MenuFrame extends Frame {
    String msg = "";
    CheckboxMenuItem debug, test;

    MenuFrame(String title) {
        super(title);

        // создать строку главного меню и добавить ее во фрейм
        MenuBar mbar = new MenuBar();
        setMenuBar(mbar);
```

```
// создать элементы меню
Menu file = new Menu("File");
MenuItem item1, item2, item3, item4, item5;
file.add(item1 = new MenuItem("New..."));
file.add(item2 = new MenuItem("Open..."));
file.add(item3 = new MenuItem("Close"));
file.add(item4 = new MenuItem("-"));
file.add(item5 = new MenuItem("Quit..."));
mbar.add(file);

Menu edit = new Menu("Edit");
MenuItem item6, item7, item8, item9;
edit.add(item6 = new MenuItem("Cut"));
edit.add(item7 = new MenuItem("Copy"));
edit.add(item8 = new MenuItem("Paste"));
edit.add(item9 = new MenuItem("-"));
Menu sub = new Menu("Special");
MenuItem item10, item11, item12;
sub.add(item10 = new MenuItem("First"));
sub.add(item11 = new MenuItem("Second"));
sub.add(item12 = new MenuItem("Third"));
edit.add(sub);

// Это элементы меню с метками.
debug = new CheckboxMenuItem("Debug");
edit.add(debug);
test = new CheckboxMenuItem("Testing");
edit.add(test);

mbar.add(edit);

// создать объект для обработки action- и item-событий
MyMenuHandler handler = new MyMenuHandler(this);
// зарегистрировать его для приема этих событий
item1.addActionListener(handler);
item2.addActionListener(handler);
item3.addActionListener(handler);
item4.addActionListener(handler);
item5.addActionListener(handler);
item6.addActionListener(handler);
item7.addActionListener(handler);
item8.addActionListener(handler);
item9.addActionListener(handler);
item10.addActionListener(handler);
item11.addActionListener(handler);
item12.addActionListener(handler);
debug.addItemListener(handler);
test.addItemListener(handler);
```

```
// создать объект для обработки window-событий
MyWindowAdapter adapter = new MyWindowAdapter(this);
// register it to receive those events
addWindowListener(adapter);
}

public void paint(Graphics g) {
    g.drawString(msg, 10, 200);

    if(debug.getState())
        g.drawString("Debug is on.", 10, 220);
    else
        g.drawString("Debug is off.", 10, 220);

    if(test.getState())
        g.drawString("Testing is on.", 10, 240);
    else
        g.drawString("Testing is off.", 10, 240);
}
}

class MyWindowAdapter extends WindowAdapter {
    MenuFrame menuFrame;
    public MyWindowAdapter(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
    public void windowClosing(WindowEvent we) {
        menuFrame.setVisible(false);
    }
}

class MyMenuHandler implements ActionListener, ItemListener {
    MenuFrame menuFrame;
    public MyMenuHandler(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
    // Обработка action-событий.
    public void actionPerformed(ActionEvent ae) {
        String msg = "You selected ";
        String arg = (String)ae.getActionCommand();
        if(arg.equals("New..."))
            msg += "New.";
        else if(arg.equals("Open..."))
            msg += "Open.";
        else if(arg.equals("Close"))
            msg += "Close.";
        else if(arg.equals("Quit..."))
            msg += "Quit.";
    }
}
```

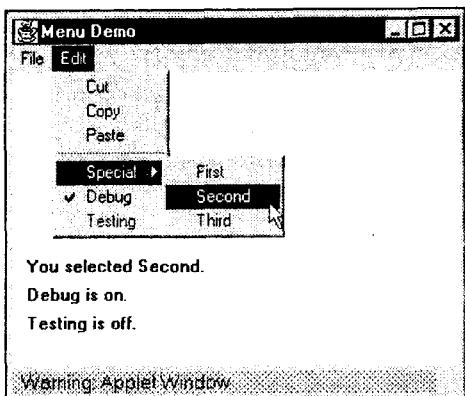
```
else if(arg.equals("Edit"))
    msg += "Edit.";
else if(arg.equals("Cut"))
    msg += "Cut.";
else if(arg.equals("Copy"))
    msg += "Copy.";
else if(arg.equals("Paste"))
    msg += "Paste.";
else if(arg.equals("First"))
    msg += "First.";
else if(arg.equals("Second"))
    msg += "Second.";
else if(arg.equals("Third"))
    msg += "Third.";
else if(arg.equals("Debug"))
    msg += "Debug.";
else if(arg.equals("Testing"))
    msg += "Testing.";
menuFrame.msg = msg;
menuFrame.repaint();
}
// обработка item-событий
public void itemStateChanged(ItemEvent ie) {
    menuFrame.repaint();
}
}

// создать frame-окно
public class MenuDemo extends Applet {
    Frame f;

    public void init() {
        f = new MenuFrame("Menu Demo");
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));

        setSize(new Dimension(width, height));

        f.setSize(width, height);
        f.setVisible(true);
    }
    public void start() {
        f.setVisible(true);
    }
    public void stop() {
        f.setVisible(false);
    }
}
```



Пример вывода апплета MenuDemo показан на рис. 22.15.

**Рис. 22.15.** Пример вывода апплета MenuDemo

Имеется еще один класс, связанный с меню, который может показаться интересным — `PopupMenu`. Он работает точно так же, как `Menu`, но создает меню, которое может быть отображено в определенном месте окна апплета. В некоторых ситуациях `PopupMenu` обеспечивает гибкую, полезную альтернативу классу `Menu`.

## Диалоговые окна

Часто необходимо использовать *диалоговое окно*, содержащее набор связанных элементов управления. Диалоговые окна первоначально использовались для получения ввода от пользователя. Они подобны фрейм-окнам, за исключением того, что диалоговые окна — всегда дочерние окна для окна верхнего уровня. Кроме того, диалоговые окна не имеют строки меню. В других отношениях они функционируют подобно фреймовым окнам. (Можно, например, добавлять к ним элементы управления тем же способом, каким добавляются элементы управления к фреймовому окну.) Диалоговые окна могут быть модальными или немодальными. Когда *модальное* диалоговое окно активно, весь ввод направляется к нему, пока оно не будет закрыто. Это означает, что вы не можете обращаться к другим частям программы до тех пор, пока не закрыли диалоговое окно. Когда *немодальное* диалоговое окно активно, фокус ввода может быть направлен другому окну вашей программы. Таким образом, другие части вашей программы остаются активными и доступными. Диалоговые окна обслуживает класс `Dialog`. Обычно используются следующие конструкторы этого класса:

```
Dialog(Frame parentWindow, boolean mode)
Dialog(Frame parentWindow, String title, boolean mode)
```

Здесь `parentWindow` — владелец диалогового окна. Если `mode` имеет значение `true`, диалоговое окно является модальным. Иначе, оно — немодальное. Заголовок диалогового окна можно передать через параметр `title`. В общем

случае, ваша программа будет подклассом класса Dialog, добавляющим функциональные возможности, необходимые вашему приложению.

Далее следует модифицированная версия предшествующей меню-программы, которая отображает немодальное диалоговое окно с выбранным пунктом New. Обратите внимание, что, когда диалоговое окно закрывается, вызывается метод dispose(). Данный метод определен в классе Window и освобождает все системные ресурсы, связанные с диалоговым окном.

```
// Демонстрирует диалоговое окно.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MenuDemo" width=250 height=250>
</applet>
*/
// создать подкласс класса Dialog
class SampleDialog extends Dialog implements ActionListener {
    SampleDialog(Frame parent, String title) {
        super(parent, title, false);
        setLayout(new FlowLayout());
        setSize(300, 200);

        add(new Label("Press this button:"));
        Button b;
        add(b = new Button("Cancel"));
        b.addActionListener(this);
    }

    public void actionPerformed(ActionEvent ae) {
        dispose();
    }

    public void paint(Graphics g) {
        g.drawString("This is in the dialog box", 10, 70);
    }
}

// создать подкласс класса Frame
class MenuFrame extends Frame {
    String msg = "";
    CheckboxMenuItem debug, test;

    MenuFrame(String title) {
        super(title);

        // создать строку меню и добавить ее к фрейму
        MenuBar mbar = new MenuBar();
        setMenuBar(mbar);
```

```
// создать пункты меню
Menu file = new Menu("File");
MenuItem item1, item2, item3, item4;
file.add(item1 = new MenuItem("New..."));
file.add(item2 = new MenuItem("Open..."));
file.add(item3 = new MenuItem("Close"));
file.add(new MenuItem("-"));
file.add(item4 = new MenuItem("Quit..."));
mbar.add(file);

Menu edit = new Menu("Edit");
MenuItem item5, item6, item7;
edit.add(item5 = new MenuItem("Cut"));
edit.add(item6 = new MenuItem("Copy"));
edit.add(item7 = new MenuItem("Paste"));
edit.add(new MenuItem("-"));

Menu sub = new Menu("Special", true);
MenuItem item8, item9, item10;
sub.add(item8 = new MenuItem("First"));
sub.add(item9 = new MenuItem("Second"));
sub.add(item10 = new MenuItem("Third"));
edit.add(sub);

// это помечаемые пункты
debug = new CheckboxMenuItem("Debug");
edit.add(debug);
test = new CheckboxMenuItem("Testing");
edit.add(test);

mbar.add(edit);

// создать объекты для обработки action- и item-событий
MyMenuHandler handler = new MyMenuHandler(this);

// зарегистрировать их для приема этих событий
item1.addActionListener(handler);
item2.addActionListener(handler);
item3.addActionListener(handler);
item4.addActionListener(handler);
item5.addActionListener(handler);
item6.addActionListener(handler);
item7.addActionListener(handler);
item8.addActionListener(handler);
item9.addActionListener(handler);
item10.addActionListener(handler);
debug.addItemListener(handler);
test.addItemListener(handler);
```

```
// создать объект для обработки window-событий
MyWindowAdapter adapter = new MyWindowAdapter(this);
// зарегистрировать его для приема этих событий
addWindowListener(adapter);
}

public void paint(Graphics g) {
    g.drawString(msg, 10, 200);

    if(debug.getState())
        g.drawString("Debug is on.", 10, 220);
    else
        g.drawString("Debug is off.", 10, 220);

    if(test.getState())
        g.drawString("Testing is on.", 10, 240);
    else
        g.drawString("Testing is off.", 10, 240);
}

}

class MyWindowAdapter extends WindowAdapter {
    MenuFrame menuFrame;
    public MyWindowAdapter(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
    public void windowClosing(WindowEvent we) {
        menuFrame.dispose();
    }
}

class MyMenuHandler implements ActionListener, ItemListener {
    MenuFrame menuFrame;
    public MyMenuHandler(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
    // обработать action-события
    public void actionPerformed(ActionEvent ae) {
        String msg = "You selected ";
        String arg = (String)ae.getActionCommand();
        // активизировать диалоговое окно, когда выбран пункт New
        if(arg.equals("New...")) {
            msg += "New.";
            SampleDialog d = new
                SampleDialog(menuFrame, "New Dialog Box");
            d.setVisible(true);
        }
        // попытка определить другие окна диалога для этих пунктов
        else if(arg.equals("Open..."))
            msg += "Open.";
    }
}
```

```
else if(arg.equals("Close"))
    msg += "Close.";
else if(arg.equals("Quit..."))
    msg += "Quit.";
else if(arg.equals("Edit"))
    msg += "Edit.";
else if(arg.equals("Cut"))
    msg += "Cut.";
else if(arg.equals("Copy"))
    msg += "Copy.";
else if(arg.equals("Paste"))
    msg += "Paste.";
else if(arg.equals("First"))
    msg += "First.";
else if(arg.equals("Second"))
    msg += "Second.";
else if(arg.equals("Third"))
    msg += "Third.";
else if(arg.equals("Debug"))
    msg += "Debug.";
else if(arg.equals("Testing"))
    msg += "Testing.";
menuFrame.msg = msg;
menuFrame.repaint();
}

public void itemStateChanged(ItemEvent ie) {
    menuFrame.repaint();
}
}

// создать frame-окно
public class DialogDemo extends Applet {
    Frame f;

    public void init() {
        f = new MenuFrame("Menu Demo");
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));

        setSize(width, height);

        f.setSize(width, height);
        f.setVisible(true);
    }

    public void start() {
        f.setVisible(true);
    }
}
```

```

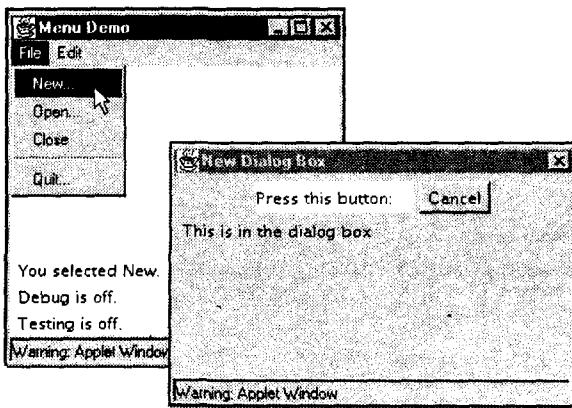
public void stop() {
    f.setVisible(false);
}
}

```

Пример вывода модифицированного апплита MenuDemo представлен на рис. 22.16.

### Совет

Пробуйте самостоятельно определить диалоговые окна для других пунктов, представленных в меню.



**Рис. 22.16.** Окно апплета MenuDemo после выбора команды **New**

## Класс *FileDialog*

Java обеспечивает встроенное диалоговое окно, которое дает возможность пользователю специфицировать файл. Для открытия этого окна программе достаточно создать конкретный экземпляр объекта типа *FileDialog*. По форме это стандартное файловое диалоговое окно, используемое операционной системой для открытия файлов (см. вывод программы). Класс *FileDialog* обеспечивает следующие конструкторы:

```

FileDialog(Frame parent, String boxName)
FileDialog(Frame parent, String boxName, int how)
FileDialog(Frame parent)

```

Здесь *parent* — владелец диалогового окна; *boxName* — имя, отображаемое в области заголовка окна. Если *boxName* опущен, заголовок диалогового окна остается пустым. Если *how* имеет значение *FileDialog.LOAD*, то окно выбирает файл для чтения, а если *how* имеет значение *FileDialog.SAVE*, окно выбирает файл для записи (с целью сохранения). Третий конструктор создает диалоговое окно с выбором файла для чтения.

FileDialog содержит методы, которые позволяют определить имя и путь файла, выбранного пользователем, например:

```
String getDirectory()  
String getFile()
```

Эти методы возвращают каталог и имя файла, соответственно. Следующая программа активизирует стандартное файловое диалоговое окно:

```
/* Демонстрирует файловое диалоговое окно.  
Это приложение, не аплет. */  
import java.awt.*;  
import java.awt.event.*;  
  
// создать подкласс класса Frame  
class SampleFrame extends Frame {  
    SampleFrame(String title) {  
        super(title);  
        // создать объект для обработки window-событий  
        MyWindowAdapter adapter = new MyWindowAdapter(this);  
        // зарегистрировать его для приема этих событий  
        addWindowListener(adapter);  
    }  
}  
  
class MyWindowAdapter extends WindowAdapter {  
    SampleFrame sampleFrame;  
    public MyWindowAdapter(SampleFrame sampleFrame) {  
        this.sampleFrame = sampleFrame;  
    }  
    public void windowClosing(WindowEvent we) {  
        sampleFrame.setVisible(false);  
    }  
}  
  
// создать фрейм-окно  
class FileDialogDemo {  
    public static void main(String args[]) {  
        Frame f = new SampleFrame("File Dialog Demo.");  
        f.setVisible(true);  
        f.setSize(100, 100);  
  
        FileDialog fd = new FileDialog(f, "File Dialog");  
        fd.setVisible(true);  
    }  
}
```

Вывод этой программы представлен на рис. 22.17.

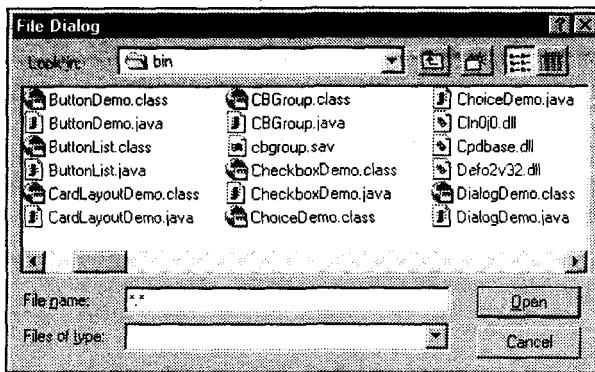


Рис. 22.17. Диалоговое окно File Dialog

## Обработка событий путем расширения AWT-компонентов

Перед завершением обзора AWT необходимо обсуждение еще одной темы: обработка событий путем расширения AWT-компонентов. Модель делегирования событий была представлена в главе 20, и все программы в этой книге пока использовали тот способ обработки событий. Но Java позволяет также обрабатывать события, создавая подклассы AWT-компонентов. Это дает возможность обрабатывать события тем же способом, что и в оригинальной 1.0 версии Java. Конечно, эта методика неудовлетворительна потому, что она имеет те же самые недостатки, что и модель событий Java 1.0 (главным из них является неэффективность). Обработка событий с помощью расширения AWT-компонентов описана в этом разделе для полноты изложения. Однако в других разделах книги данная методика не используется.

Чтобы расширить AWT-компонент, нужно вызвать метод `enableEvents()` класса. Его общая форма:

```
protected final void enableEvents(long eventMask)
```

Параметр `eventMask` — поразрядная маска, определяющая события, которые будут поставлены этому компоненту. Класс `AWTEvent` определяет следующие константы типа `int`, представляющие значения этой маски:

- `ACTION_EVENT_MASK`
- `ADJUSTMENT_EVENT_MASK`
- `COMPONENT_EVENT_MASK`
- `CONTAINER_EVENT_MASK`
- `FOCUS_EVENT_MASK`
- `INPUT_METHOD_EVENT_MASK`

- `ITEM_EVENT_MASK`
- `KEY_EVENT_MASK`
- `MOUSE_EVENT_MASK`
- `MOUSE_MOTION_EVENT_MASK`
- `TEXT_EVENT_MASK`
- `WINDOW_EVENT_MASK`

Чтобы обработать событие, следует также переопределить соответствующий метод в одном из суперклассов. Табл. 22.1 перечисляет обычно используемые методы обработки событий и классы, которые их обеспечивают.

**Таблица 22.1. Методы обработки событий**

Класс	Метод обработки
Button	processActionEvent()
Checkbox	processItemEvent()
CheckboxMenuItem	processItemEvent()
Choice	processItemEvent()
Component	processComponentEvent(), processFocusEvent(), processKeyEvent(), processMouseEvent(), processMouseMotionEvent()
List	processActionEvent(), processItemEvent()
MenuItem	processActionEvent()
Scrollbar	processAdjustmentEvent()
TextComponent	processTextEvent()

В следующих разделах приводятся простые программы, которые показывают, как можно расширить некоторые AWT-компоненты.

## Расширение класса *Button*

Следующая программа создает апплет, который отображает кнопку с надписью "Test Button". Когда кнопка нажимается, в строке состояния программы просмотра апплета или браузера отображается сообщение "action event", за которым следует счетчик числа нажатий кнопки.

Программа имеет один класс верхнего уровня с именем *ButtonDemo2*, который расширяет класс *Applet*. Определена и инициализирована нулем статическая целая переменная *i*. В нее записывается число нажатий кнопки. Метод *init()* создает экземпляр подкласса *MyButton* и добавляет его к апплету.

*MyButton* — это внутренний класс, который расширяет класс *Button*. Его конструктор использует метод *super()*, чтобы передать метку кнопки конструктору суперкласса. Затем он вызывает метод *enableEvents()*, чтобы action-события могли быть получены этим объектом. Когда генерируется action-событие, вызывается метод *processActionEvent()*. Он отображает сообщение в строке состояния и вызывает метод *processActionEvent()* для суперкласса. Поскольку *MyButton* — внутренний класс, он имеет прямой доступ к методу *showStatus()* класса *ButtonDemo2*.

```

/*
* <applet code=ButtonDemo2 width=200 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ButtonDemo2 extends Applet {
    MyButton myButton;
    static int i = 0;
    public void init() {
        myButton = new MyButton("Test Button");
        add(myButton);
    }
    class MyButton extends Button {
        public MyButton(String label) {
            super(label);
            enableEvents(AWTEvent.ACTION_EVENT_MASK);
        }
        protected void processActionEvent(ActionEvent ae) {
            showStatus("action event: " + i++);
            super.processActionEvent(ae);
        }
    }
}

```

## Расширение класса *Checkbox*

Приведенная ниже программа создает апплет, который отображает (на экран) три флашка с метками "Item 1", "Item 2" и "Item 3". Когда флашок включается или выключается, в строке состояния программы просмотра апплета или браузера отображается имя и состояние этого флашка.

Программа определяет класс верхнего уровня с именем *CheckboxDemo2*, который расширяет класс *Applet*. Метод *init()* создает три экземпляра (объекта) класса *MyCheckbox* и прибавляет их к апплету. *MyCheckbox* — внутренний класс, который расширяет класс *Checkbox*. Его конструктор использует метод *super()*, чтобы передать метку флашка конструктору суперкласса. Затем он вызывает метод *enableEvents()*, чтобы эти объекты могли получать *item*-события. Когда *item*-событие генерируется, вызывается метод *processItemEvent()*. Данный метод отображает сообщение в строке состояния и вызывает *processEvent()* для суперкласса.

```

/*
* <applet code=CheckboxDemo2 width=300 height=100>
* </applet>
*/

```

```

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class CheckboxDemo2 extends Applet {
    MyCheckbox myCheckbox1, myCheckbox2, myCheckbox3;
    public void init() {
        myCheckbox1 = new MyCheckbox("Item 1");
        add(myCheckbox1);
        myCheckbox2 = new MyCheckbox("Item 2");
        add(myCheckbox2);
        myCheckbox3 = new MyCheckbox("Item 3");
        add(myCheckbox3);
    }
    class MyCheckbox extends Checkbox {
        public MyCheckbox(String label) {
            super(label);
            enableEvents(AWTEvent.ITEM_EVENT_MASK);
        }
        protected void processItemEvent(ItemEvent ie) {
            showStatus("Checkbox name/state: " + getLabel() + "/" + getState());
            super.processItemEvent(ie);
        }
    }
}

```

## Расширение группы флажков

Следующая программа — переработка предшествующего примера для формирования такой группы флажков, в которой можно выбирать только один.

```

/*
* <applet code=CheckboxGroupDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class CheckboxGroupDemo2 extends Applet {
    CheckboxGroup cbg;
    MyCheckbox myCheckbox1, myCheckbox2, myCheckbox3;
    public void init() {
        cbg = new CheckboxGroup();
        myCheckbox1 = new MyCheckbox("Item 1", cbg, true);
        add(myCheckbox1);
        myCheckbox2 = new MyCheckbox("Item 2", cbg, false);
        add(myCheckbox2);
    }
}

```

```

myCheckbox3 = new MyCheckbox("Item 3", cbg, false);
add(myCheckbox3);
}
class MyCheckbox extends Checkbox {
    public MyCheckbox(String label, CheckboxGroup cbg, boolean flag) {
        super(label, cbg, flag);
        enableEvents(AWTEvent.ITEM_EVENT_MASK);
    }
    protected void processItemEvent(ItemEvent ie) {
        showStatus("Checkbox name/state: " + getLabel() + "/" + getState());
        super.processItemEvent(ie);
    }
}
}

```

## Расширение класса *Choice*

Представленная ниже программа создает апплет, который отображает выборочный список с элементами "Red", "Green" и "Blue". Когда элемент списка выбирается, в строке состояния программы просмотра апплета или браузера отображается имя цвета.

В программе определен один класс верхнего уровня с именем ChoiceDemo2, который расширяет класс Applet. Метод init() создает элемент выбора и добавляет его к апплету. MyChoice — внутренний класс, который расширяет Choice. Он вызывает метод enableEvents(), что позволяет объекту получать item-события. Когда генерируется item-событие, вызывается метод processItemEvent(). Он отображает сообщение в строке состояния и вызывает метод processItemEvent() для суперкласса.

```

/*
* <applet code=ChoiceDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ChoiceDemo2 extends Applet {
    MyChoice choice;
    public void init() {
        choice = new MyChoice();
        choice.add("Red");
        choice.add("Green");
        choice.add("Blue");
        add(choice);
    }
}

```

```

class MyChoice extends Choice {
    public MyChoice() {
        enableEvents(AWTEvent.ITEM_EVENT_MASK);
    }
    protected void processItemEvent(ItemEvent ie) {
        showStatus("Choice selection: " + getSelectedItem());
        super.processItemEvent(ie);
    }
}
}
}

```

## Расширение класса *List*

Приведенная далее программа изменяет предшествующий пример так, чтобы вместо выборочного меню использовался простой список. В ней определен один класс верхнего уровня с именем *ListDemo2*, который расширяет класс *Applet*. Его метод *init()* создает элемент списка и добавляет его к апплету. *MyList* — внутренний класс, который расширяет *List*. Он вызывает метод *enableEvents()*, чтобы объект мог получать как *action-*, так и *item-* события. Когда выбирается элемент списка или выбор отменяется, вызывается метод *processItemEvent()*. Когда по элементу выполняется двойной щелчок, то вызывается *processActionEvent()*. Оба метода отображают строку и затем передают управление суперклассу.

```

/*
* <applet code=ListDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ListDemo2 extends Applet {
    MyList list;
    public void init() {
        list = new MyList();
        list.add("Red");
        list.add("Green");
        list.add("Blue");
        add(list);
    }
    class MyList extends List {
        public MyList() {
            enableEvents(AWTEvent.ITEM_EVENT_MASK |
                        AWTEvent.ACTION_EVENT_MASK);
        }
    }
}

```

```
protected void processActionEvent(ActionEvent ae) {
    showStatus("Action event: " + ae.getActionCommand());
    super.processActionEvent(ae);
}

protected void processItemEvent(ItemEvent ie) {
    showStatus("Item event: " + getSelectedItem());
    super.processItemEvent(ie);
}
```

## Расширение класса *Scrollbar*

Следующая программа создает апплет, который отображает полосу прокрутки. При манипулировании с этим элементом управления в строке состояния программы просмотра апплета или браузера отображается текущее значение полосы прокрутки.

В программе определен один класс верхнего уровня с именем ScrollbarDemo2, который расширяет класс Applet. Его метод init() создает элемент полосы прокрутки и прибавляет его к апплету. MyScrollbar — внутренний класс, который расширяет Scrollbar. Он вызывает метод enableEvents(), чтобы объект мог получать adjustment-события. Когда выполняются манипуляции с полосой прокрутки, вызывается метод processAdjustmentEvent(). Когда выбирается вход прокручиваемого списка, вызывается метод processAdjustmentEvent(). Он отображает строку и затем передает управление суперклассу.

```
/*
* <applet code=ScrollbarDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ScrollbarDemo2 extends Applet {
    MyScrollbar myScrollbar;
    public void init() {
        myScrollbar = new MyScrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, 100);
        add(myScrollbar);
    }
    class MyScrollbar extends Scrollbar {
        public MyScrollbar(int style, int initial, int thumb,
                           int min, int max) {
            super(style, initial, thumb, min, max);
            enableEvents(AWTEvent.ADJUSTMENT_EVENT_MASK);
        }
    }
}
```

```
protected void processAdjustmentEvent(AdjustmentEvent ae) {  
    showStatus("Adjustment event: " + ae.getValue());  
    setValue(getValue());  
    super.processAdjustmentEvent(ae);  
}  
}  
}
```

## Исследование элементов управления, меню и менеджеров компоновки

В этой главе обсуждались классы, которые содержат элементы управления AWT, меню и менеджеры компоновки. AWT обеспечивает богатую среду программирования, продолжить исследование которой вы захотите самостоятельно. Вот некоторые предложения:

- испытайте вкладывающиеся canvas-окна (пустые окна) внутри панели апплета;
- исследуйте компонент `FileDialog`;
- экспериментируйте с ручным позиционированием компонентов, используя метод `setBounds()`;
- испытайте вкладывающиеся элементы управления внутри панелей, чтобы получить больший контроль над компоновками;
- создайте ваш собственный менеджер компоновки с помощью реализации интерфейса `LayoutManager`;
- исследуйте класс `PopupMenu`.

Чем больше вы знаете об AWT-компонентах, тем больший контроль вы будете иметь над внешним видом, чувствительностью и работоспособностью ваших апплетов и приложений.

В следующей главе мы рассмотрим еще один класс AWT — `Image`. Он используется для поддержки изображений и анимации.



## ГЛАВА 23

# Работа с изображениями

В этой главе рассматривается AWT-класс `Image` и пакет `java.awt.image`. Вместе они поддерживают работу с изображениями (отображение и манипуляции с графическими изображениями). Под *изображением* понимают прямоугольный *графический объект*. Изображения являются ключевым компонентом Web-дизайна. Включение тега `<img>` в браузер Mosaic NCSA (National Center for Supercomputer Applications, Национальный Центр Суперкомпьютерных Приложений) привело к началу взрывного роста Web в 1993 г. Этот тег был использован, чтобы встраивать изображение в поток гипертекста. Java расширяет данную базовую концепцию, допуская программное управление изображениями. Java обеспечивает интенсивную поддержку работы с изображениями.

Изображения — это объекты класса `Image`, который является частью пакета `java.awt`. Для манипулирования изображениями используются классы пакета `java.awt.image`, который содержит большое количество классов и интерфейсов изображений. Рассмотреть их всех сразу невозможно. Поэтому мы сосредоточимся только на той части пакета, которая формирует основу работы с изображениями. В настоящей главе обсуждены следующие классы `java.awt.image`:

- `CropImageFilter`
- `FilteredImageSource`
- `ImageFilter`
- `MemoryImageSource`
- `PixelGrabber`
- `RGBImageFilter`

Будут использоваться интерфейсы:

- `ImageConsumer`
- `ImageObserver`
- `ImageProducer`

Кроме того, рассматривается класс `MediaTracker`, который является частью пакета `java.awt`.

## Форматы графических файлов

Первоначально, Web-изображения могли быть только в формате GIF. Формат растровых изображений GIF (Graphics Interchange Format, формат обмена графическими данными) был создан в CompuServe Incorportion в 1987 г., для возможности просмотра встроенных изображений, что хорошо подходило для Internet. Каждое GIF-изображение может иметь не больше 256 цветов. Это ограничение заставило главных поставщиков браузеров в 1995 г. добавить поддержку изображений в формате JPEG. Формат JPEG (Joint Photographic Expert Group) был создан группой фотографических экспертов для хранения изображений с полным цветовым спектром и непрерывным тоном. Эти изображения, если они созданы должным образом, могут иметь намного более высокую точность цветовоспроизведения и более высокую степень сжатия по сравнению с GIF-кодированием. В большинстве случаев вас не будет даже интересовать, какой формат вы используете в своих программах. В языке Java все различия в кодировании изображений скрыты за ясными и удобными интерфейсами их классов.

## Создание, загрузка и просмотр изображений

Существует три общие операции, которые используются для работы с любыми изображениями: создание, загрузка и просмотр изображения на экране. Класс `Image` языка Java имеет средства для создания нового *объекта изображения* и его загрузки, и средства, с помощью которых изображение можно отобразить на экране. Отметим также, что `Image` обслуживает как изображения, находящиеся в памяти, так и изображения, которые загружаются из внешних источников.

### Создание объекта изображения

Можно было бы ожидать, что для создания изображения в памяти достаточно записать что-то вроде следующей операции:

```
Image test = new Image(200, 100); // Ошибка — не работает!
```

Но это не так. Чтобы изображения стали видимыми, их нужно *рисовать* в окне. Однако класс `Image` не имеет достаточной информации для того, чтобы создать надлежащий формат данных для экрана. Поэтому класс `Component` (из пакета `java.awt`) содержит специальный "производственный"<sup>1</sup> (*factory*) метод с именем `createImage()`, который используется для создания

<sup>1</sup> О производственных методах см. гл. 18. — Примеч. пер.

Image-объектов. (Напомним, что все AWT-компоненты являются подклассами Component, поэтому все они поддерживают данный метод.) Метод `createImage()` имеет две формы:

```
Image createImage(ImageProducer imgProd)
Image createImage(int width, int height)
```

Первая форма возвращает изображение, изготовленное параметром `imgProd`, который является объектом класса, реализующего интерфейс `ImageProducer` (производителей изображений мы рассмотрим позже). Вторая форма возвращает пустое изображение, которое имеет указанную ширину и высоту. Например:

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
```

Здесь создается экземпляр (объект) класса `Canvas` и затем вызывается производственный метод `createImage()`, чтобы фактически построить объект типа `Image`. В этом случае изображение будет пустым. Позже вы увидите, как записать в него данные.

## Загрузка изображения

Другой способ получения изображения — его загрузка. Для этого используйте метод `getImage()`, определенный классом `Applet`. Он имеет следующие формы:

```
Image getImage(URL url)
Image getImage(URL url, String imageName)
```

Первая версия возвращает `Image`-объект, который инкапсулирует изображение, найденное по (универсальному) адресу, указанному в параметре `url`. Вторая версия возвращает `Image`-объект, который инкапсулирует изображение, найденное по адресу, указанному в `url`, и имеющему имя, указанное в `imageName`.

## Просмотр изображения

Имея изображение, вы можете выводить его (на экран), используя метод `drawImage()`, который является членом класса `Graphics`. Он содержит несколько форм. Мы будем использовать метод в следующей форме:

```
boolean drawImage(Image imgObj, int left, int top, ImageObserver imgOb)
```

Он выводит изображение, переданное ему параметром `imgObj`, размещая его левый верхний угол с позиции, указанной в `left` и `top`. `imgOb` — ссылка на класс, который реализует интерфейс `ImageObserver`. Этот интерфейс реали-

зуется всеми AWT-компонентами. *Наблюдатель изображения* (*image observer*) — это объект, который может контролировать изображение, пока оно загружается. Класс *ImageObserver* описан в следующем разделе.

С помощью *getImage()* и *drawImage()* действительно очень просто загружать и просматривать изображение. Ниже показан пример апплета, который загружает и выводит одиночное изображение. Загружается файл *Seattle.jpg*, но вы можете заменить его любым файлом в формате GIF или JPG (только удостоверьтесь, что он находится в одном каталоге с HTML-файлом, который содержит апплет).

```
/*
 * <applet code="SimpleImageLoad" width=248 height=146>
 *   <param name="img" value="seattle.jpg">
 * </applet>
 */
import java.awt.*;
import java.applet.*;

public class SimpleImageLoad extends Applet
{
    Image img;

    public void init() {
        img = getImage(getDocumentBase(), getParameter("img"));
    }

    public void paint(Graphics g) {
        g.drawImage(img, 0, 0, this);
    }
}
```

В методе *init()* переменной *img* назначается изображение, возвращенное методом *getImage()*. Метод *getImage()* использует строку, возвращенную методом *getParameter("img")*, как имя файла изображения. Это изображение загружается из URL-адреса, в который метод *getDocumentBase()* возвратил URL-адрес HTML-страницы с тегом данного апплета. Имя файла, возвращенное методом *getParameter("img")*, исходит из тега *<param name="img" value="seattle.jpg">* данного апплета. Этот тег является эквивалентом, правда немного более медленным, HTML-тега **. Результат выполнения этой программы показан на рис. 23.1.

Когда этот апплет выполняется, он начинает загрузку *img* в методе *init()*. На экране можно видеть изображение по мере его загрузки из сети, потому что реализация интерфейса *ImageObserver* в классе *Applet* вызывает метод *paint()* каждый раз, когда прибывает следующая порция данных изображения.

Наблюдение загрузки изображения довольно информативно, но было бы лучше, если бы вы использовали время загрузки изображения, чтобы что-то делать параллельно. Полнотью сформированное изображение появляется на экране только в тот момент, когда оно целиком загружено. Для контроля загрузки изображения во время прорисовок экрана с другой информацией можно использовать интерфейс `ImageObserver`, описанный далее.

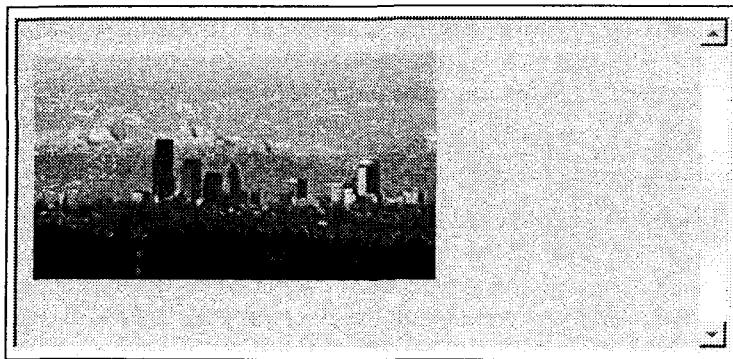


Рис. 23.1. Пример вывода апплета `SimpleImageLoad`

## Интерфейс `ImageObserver`

`ImageObserver` — это интерфейс, используемый для приема уведомлений о том, как генерируются изображения. `ImageObserver` определяет только один метод: `imageUpdate()`. Использование наблюдателя изображения позволяет выполнять (параллельно с загрузкой изображения) другие действия, такие как показ индикатора хода работы (progress-индикатора) или дополнительного экрана, которые информируют вас о ходе загрузки. Подобный вид уведомления очень полезен, когда изображение загружается по сети, где проектировщик содержимого редко принимает во внимание, что люди часто пробуют загружать апплеты через медленный модем.

Метод `ImageUpdate()` имеет следующую общую форму:

```
boolean imageUpdate(Image imgObj, int flags, int left, int top,
                    int width, int height)
```

Здесь `imgObj` — загружаемое изображение, а `flags` — целое число, которое сообщает состояние отчета обновления. Четыре целых параметра `left`, `top`, `width` и `height` представляют прямоугольник, который содержит различные значения в зависимости от передаваемых в `flags`-значений. `ImageUpdate()` должен возвратить `false`, если он завершил загрузку, и `true`, если еще имеется остаток изображения для обработки.

Параметр *flags* содержит один или несколько разрядных флагов, определенных как статические переменные внутри интерфейса *ImageObserver*. Эти флаги и информация, которую они обеспечивают, перечислены в табл. 23.1.

**Таблица 23.1. Разрядные флагги параметра *flags* метода *ImageUpdate()***

Флажок	Значение
WIDTH	Параметр <i>width</i> правilen и содержит ширину изображения
HEIGHT	Параметр <i>height</i> правilen и содержит высоту изображения
PROPERTIES	Свойства, связанные с изображением могут теперь быть получены через <i>imgObj.getProperty()</i>
SOMEBITS	Получена следующая порция пикселов, необходимых для вывода изображения. Параметры <i>left</i> , <i>top</i> , <i>width</i> , и <i>height</i> определяют прямоугольник, содержащий новые пиксели
FRAMEBITS	Получен полный фрейм, являющийся частью многофреймового изображения, которое было предварительно нарисовано. Данный фрейм может быть отображен. Параметры <i>left</i> , <i>top</i> , <i>width</i> и <i>height</i> не используются
ALLBITS	Изображение выведено целиком. Параметры <i>left</i> , <i>top</i> , <i>width</i> и <i>height</i> не используются
ERROR	Произошла ошибка с изображением, которое прослеживалось асинхронно. Изображение неполно и не может быть отображено. Никакая дальнейшая видеоинформация не будет получена. Для удобства будет также установлен флагок <i>ABORT</i> , чтобы указать, что производство изображения было прервано
ABORT	Изображение, которое прослеживалось асинхронно, было прервано прежде, чем оно было закончено. Однако, если ошибка не произошла, доступ к любой части данных изображения перезапустит производство изображения

Класс *Applet* имеет реализацию метода *imageUpdate()* интерфейса *ImageObserver*, который используется для перерисовки изображений во время их загрузки. Его можно переопределить в вашем классе, чтобы изменить поведение метода.

Простой пример метода *imageUpdate()*:

```
public boolean imageUpdate(Image img, int flags,
                           int x, int y, int w, int h) {
    if ((flags & ALLBITS) == 0) {
        System.out.println("Загрузка изображения...");
        return true;
    } else {
```

```
    System.out.println("Загрузка изображения завершена.");
    return false;
}
}
```

## Пример с *ImageObserver*

Теперь рассмотрим пример, который переопределяет `imageUpdate()` так, чтобы избавить версию апплета `SimpleImageLoad` от частого мерцания изображения. Умалчивающая реализация `imageUpdate()` в классе `Applet` имеет несколько проблем. Во-первых, она перерисовывает полное изображение каждый раз, когда прибывают какие-нибудь новые данные. Это вызывает вспышки между цветом фона и изображением. Во-вторых, он использует свойство `Applet.repaint()`, заставляющее систему перерисовывать изображение только каждую десятую долю секунды или около этого. Это вызывает чувство толчкообразного, негладкого движения изображения при его рисовании. Наконец, заданная по умолчанию реализация не знает ничего относительно изображений, которые могут быть не в состоянии загружаться должным образом. Многие начинающие программисты Java недовольны тем, что метод `getImage()` преуспевает даже тогда, когда указанное изображение не существует. Вы не узнаете об отсутствии изображения, пока не начнет работу `imageUpdate()`. Если вы используете умалчивающую реализацию `imageUpdate()`, то никогда и не узнаете, что случилось. Ваш метод `paint()` просто не будет ничего делать, когда вы вызываете `g.drawImage()`.

Следующий ниже пример фиксирует все три этих проблемы в десяти строках кода. Во-первых, он устранил мерцание с помощью двух небольших изменений. Он переопределяет метод `update()` так, чтобы тот вызвал `paint()` без первоначальной прорисовки цвета фона. Фон устанавливается через метод `setBackground()` в `init()`, так что начальный фон рисуется только однажды. Он также использует версию метода `repaint()`, которая указывает прямоугольник для рисования. Система установит область отсечения такой, чтобы ничего вне данного прямоугольника не рисовалось. Это устраняет мерцание перерисовки и улучшает выполнение.

Во-вторых, это избавит от толчкообразного, негладкого показа входящего изображения путем перерисовки каждый раз, когда `repaint()` принимает обновление. Эти обновления происходят на базе построчного сканирования, так что изображение, которое имеет 100 пикселов в высоту, будет "перерисовываться" 100 раз за время загрузки. Обратите внимание, что это не самый быстрый способ отображать изображение, а лишь самый слаженный.

Наконец, он обрабатывает ошибку, вызванную тем, что нужный файл не найден, исследуя `ABORT`-разряд параметра `flags`. Если он установлен, переменная экземпляра `error` принимает значение `true`, и затем вызывается

repaint(). Метод paint() модифицируется так, что, если переменная error равна true, то сообщение об ошибках печатается на ярко-красном фоне.

```
/*
 * <applet code="ObservedImageLoad" width=248 height=146>
 *   <param name="img" value="seattle.jpg">
 * </applet>
 */
import java.awt.*;
import java.applet.*;

public class ObservedImageLoad extends Applet {
    Image img;
    boolean error = false;
    String imgname;

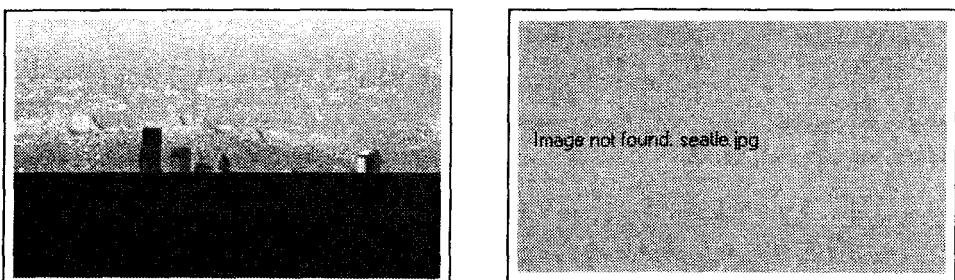
    public void init() {
        setBackground(Color.blue);
        imgname = getParameter("img");
        img = getImage(getDocumentBase(), imgname);
    }

    public void paint(Graphics g) {
        if (error) {
            Dimension d = getSize();
            g.setColor(Color.red);
            g.fillRect(0, 0, d.width, d.height);
            g.setColor(Color.black);
            g.drawString("Image not found: " + imgname, 10, d.height/2);
        } else {
            g.drawImage(img, 0, 0, this);
        }
    }

    public void update(Graphics g) {
        paint(g);
    }

    public boolean imageUpdate(Image img, int flags,
                               int x, int y, int w, int h) {
        if ((flags & SOMEBITS) != 0) {           // новые частичные данные
            repaint(x, y, w, h);                // рисует новые пиксели
        } else if ((flags & ABORT) != 0) {
            error = true;                      // файл не найден
            repaint();                         // рисовать весь апплет
        }
        return (flags & (ALLBITS|ABORT)) == 0;
    }
}
```

Рис. 23.2 показывает экраны двух отдельных прогонов этого апплета. Левый экран показывает полузагруженное изображение, а правый — отображает сообщение о том, что в теге апплета имя файла было напечатано с ошибкой.



**Рис. 23.2.** Пример вывода апплета ObservedImageLoad: полузагруженное изображение (слева) и сообщение об ошибке (справа)

Ниже показан интересный вариант `imageUpdate()`, который вы могли бы попробовать испытать. Он ждет, пока изображение не будет полностью загружено перед моментальной вставкой его в экран единственной перерисовкой.

```
public boolean imageUpdate(Image img, int flags,
                           int x, int y, int w, int h) {
    if ((flags & ALLBITS) != 0) {
        repaint();
    } else if ((flags & (ABORT|ERROR)) != 0) {
        error = true; // file not found
        repaint();
    }
    return (flags & (ALLBITS|ABORT|ERROR)) == 0;
}
```

## Двойная буферизация

Мало того, что изображения полезны для хранения картинок, но вы можете также использовать их для рисования поверхностей вне экрана. Это позволяет вам передавать любое изображение, включая текст и графику, во внешний буфер, который вы можете отображать в более позднее время. Преимущество данного механизма состоит в том, что изображение становится видимым только тогда, когда оно уже окончательно построено. Рисование сложного изображения может занимать несколько миллисекунд или больше, что может наблюдаться пользователем как мигание или мерцание. Это мигание отвлекает пользователя и заставляет его чувствовать вашу визуализацию как более медленную, чем она есть в действительности. Исполь-

зование внеэкранного изображения для ослабления мерцания называется *двойной буферизацией*, из-за того что экран рассматривается как буфер для пикселов, а внеэкранное изображение — это второй буфер, где вы можете готовить пиксели для показа на экране.

Ранее в этой главе вы видели, как можно *создать* пустой Image-объект. Теперь вы увидите, как можно *рисовать* на данном изображении, а не на экране. Для этого необходим объект типа Graphics, чтобы использовать любой из его визуализирующих методов. Удобно также, что Graphics-объект, который вы можете использовать для рисования на изображении, доступен через метод getGraphics(). Ниже представлен фрагмент кода, который создает новое изображение, получает его графический контекст, и заполняет полное изображение красными пикселями:

```
Canvas c = new Canvas();
Image test = c.createImage(200/ 100);
Graphics gc = test.getGraphics();
gc.setColor(Color.red);
gc.fillRect(0, 0, 200, 100);
```

Как только вы создали и заполнили внеэкранное изображение, оно еще не будет видимым. Для его окончательного отображения вызовите drawImage(). Чтобы продемонстрировать влияние двойной буферизации на время рисования, ниже приведен пример, рисующий изображение со значительным временем прорисовки:

```
/*
 * <applet code=DoubleBuffer width=250 height=250>
 * </applet>
 */
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class DoubleBuffer extends Applet {
    int gap = 3;
    int mx, my;
    boolean flicker = true;
    Image buffer = null;
    int w, h;

    public void init() {
        Dimension d = getSize();
        w = d.width;
        h = d.height;
        buffer = createImage(w, h);
        addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseDragged(MouseEvent me) {
                mx = me.getX();
```

```
    my = me.getY();
    flicker = false;
    repaint();
}
public void mouseMoved(MouseEvent me) {
    mx = me.getX();
    my = me.getY();
    flicker = true;
    repaint();
}
});
}
public void paint(Graphics g) {
    Graphics screengc = null;

    if (!flicker) {
        screengc = g;
        g = buffer.getGraphics();
    }

    g.setColor(Color.blue);
    g.fillRect(0, 0, w, h);

    g.setColor(Color.red);
    for (int i=0; i<w; i+=gap)
        g.drawLine(i, 0, w-i, h);
    for (int i=0; i<h; i+=gap)
        g.drawLine(0, i, w, h-i);

    g.setColor(Color.black);
    g.drawString("Press mouse button to double buffer", 10, h/2);

    g.setColor(Color.yellow);
    g.fillOval(mx - gap, my - gap, gap*2+1, gap*2+1);

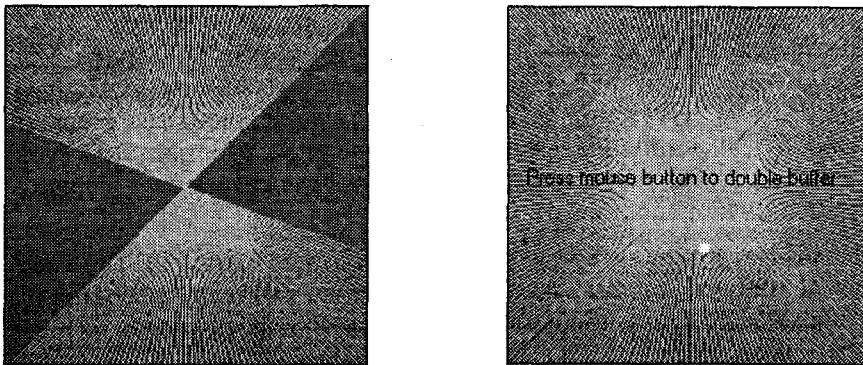
    if (!flicker) {
        screengc.drawImage(buffer, 0, 0, null);
    }
}
public void update(Graphics g) {
    paint(g);
}
}
```

Предложенный простой апплет имеет усложненный метод `paint()`. Он за-  
полняет фон синим и затем рисует сверху красный муаровый образец.  
В верхней его части выводится некоторый черный текст, и затем рисуется  
желтый круг с центром в  $(mx, my)$ -координатах. Методы `mouseMoved()` и  
`mouseDragged()` переопределены так, чтобы отслеживать позицию мыши.

Эти методы идентичны, за исключением установок булевой переменной `flicker`. `mouseMoved()` устанавливает `flicker` в `true`, а `mouseDragged()` — в `false`. Это приводит к вызову `repaint()` с установкой `flicker` в `true`, когда мышь передвигается (но никакая кнопка не нажата), и установкой ее в `false`, когда мышь перетаскивается с любой нажатой кнопкой.

Когда вызывается `paint()` с переменной `flicker`, установленной в `true`, мы видим на экране выполнение каждой операции рисования. В случае, когда кнопка мыши нажата, и `paint()` вызывается с `flicker`, установленной в `false`, мы видим совершенно иную картину. Метод `paint()` обменивает `Graphics`-ссылку `g` с графическим контекстом, который отсылает к внеэкранному полотну `buffer`, который мы создали в `init()`. Тогда все операции рисования становятся невидимыми. В конце `paint()`, мы просто вызываем `drawImage()`, чтобы показать результаты этих методов рисования все сразу.

Обратите внимание, что удобно передавать в `drawImage()` в качестве четвертого параметра `null`. Этот параметр используется для передачи объекта типа `ImageObserver`, который принимает уведомление об `image`-событиях. Так как это изображение не производится из сетевого потока, нам не нужно уведомление. Левый снимок на рис. 23.3 показывает, как выглядит апплет с не нажатыми кнопками мыши. Вы видите, что снимок был сделан, когда изображение было где-то в середине перерисовки. Правый снимок показывает, что, когда кнопка мыши нажимается, изображение всегда имеет законченный, четкий вид (благодаря двойной буферизации).



**Рис. 23.3.** Пример вывода апплета `DoubleBuffer` без двойной буферизации (слева) и с двойной буферизацией (справа)

## Класс `MediaTracker`

Многие из ранних разработчиков Java находили интерфейс `ImageObserver` слишком сложным для понимания и управления загрузкой множественных изображений. Сообщество разработчиков просило более простое решение,

которое позволило бы программистам загружать все их изображения синхронно, не беспокоясь относительно `imageUpdate()`. В ответ на это Sun Microsystems в последующем выпуске JDK добавила к `java.awt` класс с именем `MediaTracker`. `MediaTracker` создает объект, который будет параллельно проверять состояние произвольного числа изображений.

Для использования `MediaTracker` вы создаете его новый экземпляр и применяете его метод `addImage()`, чтобы прослеживать состояние загрузки изображения. Общий формат `addImage()`:

```
void addImage(Image imgObj, int imgID)
void addImage(Image imgObj, int imgID, int width, int height)
```

Здесь `imgObj` — прослеживаемое изображение. Его идентификационный номер передается в `imgID`. ID (идентификатор) номера не должны быть уникальными. Вы можете использовать тот же номер с несколькими изображениями, как средство их идентификации в качестве части группы. Во второй форме параметры `width` и `height` определяют размеры отображаемого объекта.

Как только вы зарегистрировали изображение, можете проверить, загружено ли оно, или можете ждать, когда оно полностью загрузится. Для проверки состояния изображения вызывают метод `checkID()`. Версия, используемая в этой главе, имеет формат:

```
boolean checkID(int imgID)
```

Здесь `imgID` определяет ID изображения, которое вы хотите проверить. Метод возвращает `true`, если все изображения, которые имеют указанный идентификатор, были загружены (или, когда загрузка закончилась с ошибкой или пользователь выполнил аварийное завершение работы). Иначе он возвращает `false`. Чтобы увидеть, были ли все прослеживаемые изображения загружены, можно использовать метод `checkAll()`.

`MediaTracker` следует применять при загрузке *группы* изображений. Пока все изображения не разгружены, вы можете развлечь пользователя, отобразив на экране что-нибудь еще (пока прибываю загружаемые изображения).

### Предупреждение

Если вы используете `MediaTracker`, когда уже вызвали `addImage()` для изображения, ссылка на `MediaTracker` предохранит систему от сборки мусора. Если вы хотите, чтобы система была способной к сборке мусора прослеживаемых изображений, удостоверьтесь, что она может собирать также и экземпляр `MediaTracker`.

Далее следует пример, который загружает слайд-показ с семью изображениями и отображает прогресс-полоску процесса загрузки:

```
/*
 * <applet code="TrackedImageLoad" width=300 height=400>
 * <param name="img"
 * value="vincent+leonardo+matisse+picasso+renoir+seurat+vermeer">
 * </applet>
 */
import java.util.*;
import java.applet.*;
import java.awt.*;

public class TrackedImageLoad extends Applet implements Runnable {
    MediaTracker tracker;
    int tracked;
    int frame_rate = 5;
    int current_img = 0;
    Thread motor;
    static final int MAXIMAGES = 10;
    Image img[] = new Image[MAXIMAGES];
    String name[] = new String[MAXIMAGES];
    boolean stopFlag;

    public void init() {
        tracker = new MediaTracker(this);
        StringTokenizer st = new StringTokenizer(getParameter("img"), "+");

        while(st.hasMoreTokens() && tracked <= MAXIMAGES) {
            name[tracked] = st.nextToken();
            img[tracked] = getImage(getDocumentBase(), name[tracked] + ".jpg");
            tracker.addImage(img[tracked], tracked);
            tracked++;
        }
    }

    public void paint(Graphics g) {
        String loaded = "";
        int donecount = 0;

        for(int i=0; i<tracked; i++) {
            if (tracker.checkID(i, true)) {
                donecount++;
                loaded += name[i] + " ";
            }
        }

        Dimension d = getSize();
        int w = d.width;
        int h = d.height;

        if (donecount == tracked) {
            frame_rate = 1;
        }
    }
}
```

```

Image i = img[current_img++];
int iw = i.getWidth(null);
int ih = i.getHeight(null);
g.drawImage(i, (w - iw)/2, (h - ih)/2, null);
if (current_img >= tracked)
    current_img = 0;
} else {
    int x = w * donecount           / отслежено
    g.setColor(Color.black);
    g.fillRect(0, h/3, x, 16);
    g.setColor(Color.white);
    g.fillRect(x, h/3, w-x, 16);
    g.setColor(Color.black);
    g.drawString("loaded", 10, h/2);
}
}

public void start() {
    motor = new Thread(this);
    stopFlag = false;
    motor.start();
}

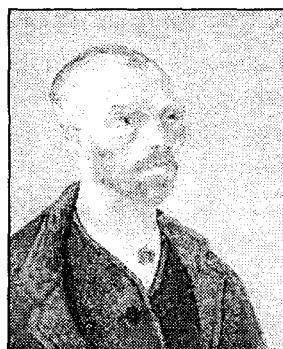
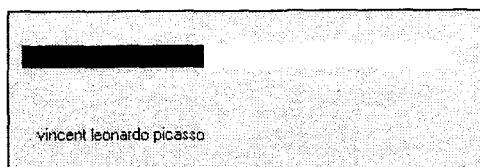
public void stop() {
    stopFlag = true;
}

public void run() {
    motor.setPriority(Thread.MIN_PRIORITY);
    while (true) {
        repaint();
        try {
            Thread.sleep(1000/frame_rate);
        } catch (InterruptedException e) { };
        if(stopFlag)
            return;
    }
}
}
}

```

Представленный пример создает новый MediaTracker в методе init() и затем добавляет каждое из именованных прослеживаемых изображений с помощью метода addImage(). В методе paint() на очередном из прослеживаемых изображений вызывается checkID(). Если все изображения загружены, они отображаются. Если нет, показывается простая прогресс-полоска числа загруженных изображений с именами полностью загруженных (отображаемых ниже этой полоски). Рис. 23.4 содержит две сцены выполнения этого апплета. Одна — это прогресс-полоска, показывающая, что уже были за-

груженены три изображения. Другая — автопортрет Ван Гога во время показа слайдов.



**Рис. 23.4.** Пример вывода апплета TrackedImageLoad: прогресс-полоска (слева) и автопортрет Ван Гога (справа)

## Интерфейс *ImageProducer*

*ImageProducer* — это интерфейс для объектов, которые хотят производить данные для изображений. Объект, реализующий интерфейс *ImageProducer*, поставляет целочисленные или байтовые массивы, которые представляют данные изображения и производят *Image*-объекты. Как вы видели ранее, одна из форм метода *createImage()* имеет объект *ImageProducer* в качестве своего параметра. Существуют два производителя изображений (*image producers*), содержащихся в *java.awt.image: MemoryImageSource* И *FilteredImageSource*. Здесь мы рассмотрим *MemoryImageSource* и создадим новый *Image* объект из данных, генерируемых внутри апплета.

## Производитель изображений *MemoryImageSource*

*MemoryImageSource* — это класс, который создает новый *Image*-объект из массива данных. Он определяет несколько конструкторов. Тот, который мы будем использовать, имеет следующую сигнатуру:

```
MemoryImageSource(int width, int height, int pixel[], int offset,
                  int scanLineWidth),
```

Объект *MemoryImageSource* создается из массива целых чисел (в формате умалчивающей цветовой модели RGB), указанного в параметре *pixel* (он-то и содержит данные для воспроизведения *Image*-объекта). В умалчивающей цветовой модели пикセル — это целое число формата 0xAARRGGBB, где A — Alpha, R — Red, G — Green, и B — Blue.

Значение Alpha представляет степень прозрачности пикселя (0 — полностью прозрачный, 255 — полностью непрозрачный). Ширина и высота результи-

рующего изображения передается в параметрах `width` и `height`. Исходную точку для начала чтения данных в массиве пикселов задает параметр `offset`. Ширина строки сканирования (которая часто совпадает с шириной изображения) задает параметр `scanLineWidth`.

Следующий короткий пример генерирует `MemoryImageSource`-объект, используя разновидность простого алгоритма (поразрядное исключающее ИЛИ  $(x,y)$ -координат каждого пикселя)<sup>1</sup>:

```
/*
 * <applet code="MemoryImageGenerator" width=256 height=256>
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class MemoryImageGenerator extends Applet {
    Image img;
    public void init() {
        Dimension d = getSize();
        int w = d.width;
        int h = d.height;
        int pixels[] = new int[w * h];
        int i = 0;

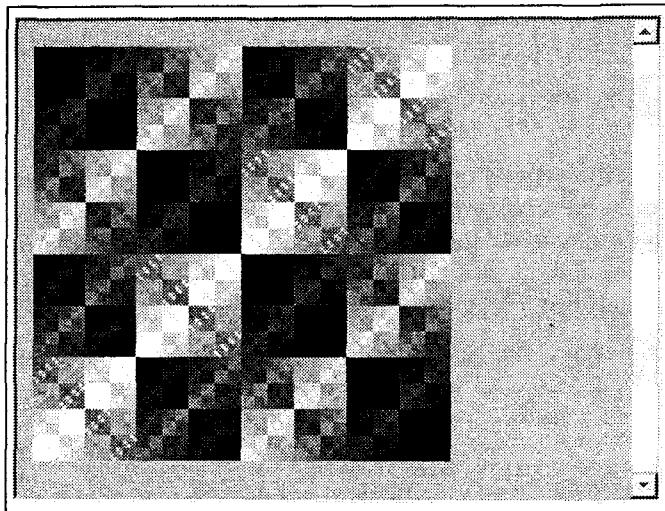
        for(int y=0; y<h; y++) {
            for(int x=0; x<w; x++) {
                int r = (x^y)&0xff;
                int g = (x*2^y*2)&0xff;
                int b = (x*4^y*4)&0xff;
                pixels[i++] = (255 << 24) | (r << 16) | (g << 8) | b;
            }
        }
        img = createImage(new MemoryImageSource(w, h, pixels, 0, w));
    }
    public void paint(Graphics g) {
        g.drawImage(img, 0, 0, this);
    }
}
```

Данные для нового `MemoryImageSource` создаются в методе `init()`. Массив целых предназначен для хранения пиксельных значений; данные генерируются во вложенных `for`-циклах, где значения `r`, `g` и `b` организуют сдвиги

---

<sup>1</sup> Пример взят из книги Gerard J. Holzmann. *Beyond Photography. The Digital Darkroom*, — Prentice Hall, 1988). — Примеч. пер.

в пикселях массива `pixels`. Наконец, вызывается метод `createImage()` с новым экземпляром `MemoryImageSource`, созданным из необработанных пиксельных данных, в качестве последнего аргумента. Рис. 23.5 показывает изображение после выполнения апплета. (В цвете оно выглядит намного лучше.)



**Рис. 23.5.** Пример вывода апплета `MemoryImageGenerator`

## Интерфейс *ImageConsumer*

`ImageConsumer` — это абстрактный интерфейс для объектов, которые хотят получать пиксельные данные изображений (скажем, от производителя) и поставлять их (скажем, на экран) уже как другой вид данных. Очевидно, что этот интерфейс является противоположностью<sup>1</sup> интерфейса `ImageProducer`. Объект, который реализует интерфейс `ImageConsumer`, собирается создавать массивы `int` или `byte`, которые представляют пиксели `Image`-объекта. Рассмотрим класс `PixelGrabber`, который является простой реализацией интерфейса `ImageConsumer`.

## Класс *PixelGrabber*

Класс `PixelGrabber` определен в `java.lang.image`. Это инверсия класса `MemoryImageSource`. Вместо построения изображения из массива пиксельных значений, он берет существующее изображение и строит из него массив пикселов. Для использования `PixelGrabber` вы сначала организуете `int`-мас-

<sup>1</sup> Если объект класса `ImageProducer` уместно называть *производителем* изображений, то объект `ImageConsumer` иногда называют *потребителем*. — Примеч. пер.

сив достаточно большой, чтобы содержать данные пикселов, и затем создаете экземпляр `PixelGrabber`, передавая ему прямоугольную область, которую вы хотите преобразовать в пикセルное представление. Наконец, вы вызываете метод `grabPixels()` этого экземпляра.

Конструктор `PixelGrabber`, который используется в этой главе, имеет следующую форму:

```
PixelGrabber(Image imgObj, int left, int top, int width, int height,
             int pixel[], int offset, int scanLineWidth)
```

Здесь `imgObj` — объект, чьи пиксели преобразуются. Значения `left` и `top` определяют левый верхний угол прямоугольника, а `width` и `height` — его размеры, из которого будут получены пиксели. Пиксели будут сохранены в массиве `pixel`, со смещением `offset`. Ширину строки сканирования (которая часто такая же, как ширина изображения) задают в `scanLineWidth`.

Метод `grabPixels()` определяется с такими сигнатурами:

```
boolean grabPixels()
    throws InterruptedException

boolean grabPixels(long milliseconds)
    throws InterruptedException
```

Оба метода возвращают `true`, если завершаются успешно, и `false` — в противном случае. Во второй форме параметр `milliseconds` определяет, как долго метод будет ожидать пиксели.

Далее показан пример, который "захватывает" пиксели изображения и затем создает гистограмму их яркости. Под *гистограммой яркости* здесь понимается распределение количества пикселов с определенной яркостью по всем значениям шкалы яркости (от 0 до 255). После того как апплет рисует изображение, он выводит (помимо этого изображения) гистограмму его яркости (рис. 23.6).

```
/*
 * <applet code=HistoGrab.class width=341 height=400>
 * <param name=img value=vermeer.jpg>
 * </applet> */
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class HistoGrab extends Applet {
    Dimension d;
    Image img;
    int iw, ih;
```

```
int pixels[];
int w, h;
int hist[] = new int[256];
int max_hist = 0;

public void init() {
    d = getSize();
    w = d.width;
    h = d.height;

    try {
        img = getImage(getDocumentBase(), getParameter("img"));
        MediaTracker t = new MediaTracker(this);
        t.addImage(img, 0);
        t.waitForID(0);
        iw = img.getWidth(null);
        ih = img.getHeight(null);
        pixels = new int[iw * ih];
        PixelGrabber pg = new PixelGrabber(img, 0, 0, iw, ih,
                                           pixels, 0, iw);
        pg.grabPixels();
    } catch (InterruptedException e) { }

    for (int i=0; i<iw*ih; i++) {
        int p = pixels[i];
        int r = 0xff & (p >> 16);
        int g = 0xff & (p >> 8);
        int b = 0xff & (p);
        int y = (int) (.33 * r + .56 * g + .11 * b);
        hist[y]++;
    }
    for (int i=0; i<256; i++) {
        if (hist[i] > max_hist)
            max_hist = hist[i];
    }
}

public void update() {}

public void paint(Graphics g) {
    g.drawImage(img, 0, 0, null);
    int x = (w - 256) / 2;
    int lasty = h - h * hist[0] / max_hist;
    for (int i=0; i<256; i++, x++) {
        int y = h - h * hist[i] / max_hist;
        g.setColor(new Color(i, i, i));
        g.fillRect(x, y, 1, h);
        g.setColor(Color.red);
```

```
    g.drawLine(x-1, lasty, x, y);
    lasty = y;
}
}
}
```

Рис. 23.6 демонстрирует изображение и гистограмму известной картины Вермера.



**Рис. 23.6.** Пример вывода апллета *HistoGrab*

## Класс *ImageFilter*

На базе интерфейсов `ImageProducer` и `ImageConsumer` (и их конкретных классов `MemoryImageSource` и `PixelGrabber`), можно создать произвольный набор преобразующих фильтров, каждый из которых берет источник пикселов, модифицирует сгенерированные ими пиксели и передает произвольному потребителю. Этот механизм аналогичен способу, с помощью которого создаются конкретные абстрактные классы ввода/вывода `InputStream`, `OutputStream`, `Reader` и `Writer` (см. гл. 17). Такая поточная модель изображений завершается введением класса `ImageFilter`. Класс `ImageFilter` пакета `java.awt.image` имеет несколько подклассов — `AreaAveragingScaleFilter`, `CropImageFilter`, `ReplicateScaleFilter` и `RGBImageFilter`. Кроме того, в пакете имеется специальная реализация интерфейса `ImageProducer` с именем `FilteredImageSource`. Объекты класса `FilteredImageSource` производят фильтрованные пиксели изображений, генерируемых объектами класса `ImageProducer`. Для фильтрации в них используются объекты одного из указанных выше подклассов `ImageFilter`. Экземпляр класса `FilteredImageSource`

передается (в качестве аргумента) в вызов метода `createImage()`, который используется в качестве производителя фильтрованных (преобразованных) изображений. В этой главе, мы рассмотрим два специальных фильтра: `CropImageFilter` и `RGBImageFilter`.

## Фильтр `CropImageFilter`

Фильтр `CropImageFilter` просто вырезает из исходного изображения небольшую прямоугольную область. Его удобно использовать, например, когда нужно работать не с целым большим изображением, а с более мелкими его частями<sup>1</sup>. Если каждое подизображение имеет один и тот же размер, то можно легко извлекать их, используя для разборки блока изображения фильтр `CropImageFilter`. Далее показан пример, в котором одиночное изображение делится фильтром на шестнадцать одинаковых прямоугольных частей и затем эти неперекрывающиеся изображения скрэмблируются (перемешиваются случайным образом) путем 32-кратного обмена пар, выбираемых случайно из шестнадцати вырезанных изображений (на рис. 23.7 — известная картина Пикассо, скрэмблированная апплетом `TileImage`).

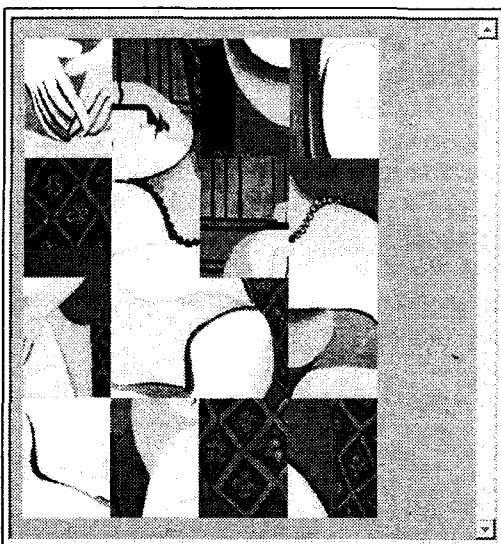


Рис. 23.7. Пример вывода  
апплета `TileImage`

```
/*
 * <applet code=TileImage.class width=288 height=399>
 * <param name=img value=picasso.jpg>
```

<sup>1</sup> Не следует, однако, забывать, что загрузка двадцати изображений с размером 2 Кбайт происходит намного дольше, чем загрузка одиночного изображения размера 40 Кбайт, которое имеет в своем составе много фреймов мультипликации. — Примеч. пер.

```
* </applet>
*/
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class TileImage extends Applet {
    Image img;
    Image cell[] = new Image[4*4];
    int iw, ih;
    int tw, th;

    public void init() {
        try {
            img = getImage(getDocumentBase(), getParameter("img"));
            MediaTracker t = new MediaTracker(this);
            t.addImage(img, 0);
            t.waitForID(0);
            iw = img.getWidth(null);
            ih = img.getHeight(null);
            tw = iw / 4;
            th = ih / 4;
            CropImageFilter f;
            FilteredImageSource fis;
            t = new MediaTracker(this);
            for (int y=0; y<4; y++) {
                for (int x=0; x<4; x++) {
                    f = new CropImageFilter(tw*x, th*y, tw, th);
                    fis = new FilteredImageSource(img.getSource(), f);
                    int i = y*4+x;
                    cell[i] = createImage(fis);
                    t.addImage(cell[i], i);
                }
            }
            t.waitForAll();
            for (int i=0; i<32; i++) {
                int si = (int)(Math.random() * 16);
                int di = (int)(Math.random() * 16);
                Image tmp = cell[si];
                cell[si] = cell[di];
                cell[di] = tmp;
            }
        } catch (InterruptedException e) { }
    }

    public void update(Graphics g) {
        paint(g);
    }
}
```

```
public void paint(Graphics g) {  
    for (int y=0; y<4; y++) {  
        for (int x=0; x<4; x++) {  
            g.drawImage(cell[y*4+x], x * tw, y * th, null);  
        }  
    }  
}
```

## Фильтр *RGBImageFilter*

Фильтр *RGBImageFilter* используется для попиксельного преобразования одного изображения в другое, трансформируя по пути цвет пикселов. Данный фильтр можно использовать для прояснения изображения, увеличения его контраста, или даже для преобразования цветного изображения к полутоновому.

Для демонстрации *RGBImageFilter* мы разработали несколько усложненный пример, который использует динамическую встроенную стратегию для обрабатывающих изображения фильтров. Мы создали интерфейс для обобщенной фильтрации изображения так, чтобы наш апллет мог просто загружать эти фильтры, основываясь на HTML-тегах *<param>*, без необходимости знать подробности обо всех фильтрах изображений. Этот пример состоит из главного апллет-класса с именем *ImageFilterDemo*, интерфейса с именем *PlugInFilter* и сервисного класса с именем *LoadedImage*, который инкапсулирует некоторые из методов класса *MediaTracker*, уже использовавшихся в текущей главе. Кроме того, в программу включены три фильтра — *Grayscale*, *Invert* и *Contrast*, которые просто манипулируют цветовым пространством исходного изображения, используя *RGBImageFilters*, и еще два класса — *Blur* и *Sharpen*, которые применяют более сложные фильтры "свертывания", изменяющие данные пикселя, основываясь на пикселях, окружающих каждый пикセル исходных данных. *Blur* и *Sharpen* — это подклассы абстрактного класса-помощника называемого *Convolver*. Рассмотрим каждую часть нашего примера.

### *ImageFilterDemo.java*

Класс *ImageFilterDemo* является каркасом апллета для других фильтров изображений. Он использует простой менеджер компоновки *BorderLayout*, с панелью в позиции *south*, содержащей кнопки, которые будут представлять каждый фильтр. Объект *Label* занимает слот *North* для информационных сообщений о ходе работы фильтра. Изображение (которое инкапсулировано в *Canvas* подклассе *LoadedImage*, описанном позже) размещается в слоте *Center*. Мы анализируем кнопки фильтров вне параметра *filters* тега *<param>* (где они отделены значками +) — с помощью класса  *StringTokenizer*.

Метод ActionPerformed() интересен тем, что он использует метку кнопки как имя класса фильтра, который он пробует загрузить с помощью метода newInstance():

```
pif = (PlugInFilter) Class.forName(a).newInstance();
```

Данный метод устойчив и выбирает адекватное действие, даже если кнопка не соответствует подходящему классу, реализующему PlugInFilter.

```
/*
 * <applet code=ImageFilterDemo width=350 height=450>
 * <param name=img value=vincent.jpg>
 * <param name=filters value="Grayscale+Invert+Contrast+Blur+Sharpen">
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class ImageFilterDemo extends Applet implements ActionListener {
    Image img;
    PlugInFilter pif;
    Image fimg;
    Image curImg;
    LoadedImage lim;
    Label lab;
    Button reset;

    public void init() {
        setLayout(new BorderLayout());
        Panel p = new Panel();
        add(p, BorderLayout.SOUTH);
        reset = new Button("Reset");
        reset.addActionListener(this);
        p.add(reset);
        StringTokenizer st = new StringTokenizer(getParameter("filters"), "+");
        while(st.hasMoreTokens()) {
            Button b = new Button(st.nextToken());
            b.addActionListener(this);
            p.add(b);
        }
        lab = new Label("");
        add(lab, BorderLayout.NORTH);

        img = getImage(getDocumentBase(), getParameter("img"));
        lim = new LoadedImage(img);
        add(lim, BorderLayout.CENTER);
    }
```

```
public void actionPerformed(ActionEvent ae) {
    String a = "";
    try {
        a = (String)ae.getActionCommand();
        if (a.equals("Reset")) {
            lim.set(img);
            lab.setText("Normal");
        }
        else {
            pif = (PlugInFilter) Class.forName(a).newInstance();
            fimg = pif.filter(this, img);
            lim.set(fimg);
            lab.setText("Filtered: " + a);
        }
        repaint();
    } catch (ClassNotFoundException e) {
        lab.setText(a + " not found");
        lim.set(img);
        repaint();
    } catch (InstantiationException e) {
        lab.setText("couldn't new " + a);
    } catch (IllegalAccessException e) {
        lab.setText("no access: " + a);
    }
}
```

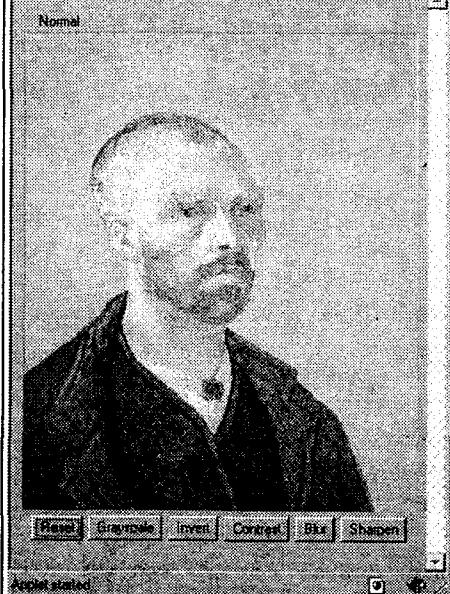


Рис. 23.8 показывает, что апплет выглядит так, как во время своей начальной загрузки, использующей тег апплита, представленный наверху этого исходного файла.

Рис. 23.8. Пример нормального вывода апплита ImageFilterDemo

## PlugInFilter.java

PlugInFilter — это простой интерфейс, используемый для абстрактной фильтрации изображения. Он имеет только один метод filter(), который берет апплет и исходное изображение и возвращает новое изображение, которое было отфильтровано некоторым способом.

```
interface PlugInFilter {  
    java.awt.Image filter(java.applet.Applet a, java.awt.Image in);  
}
```

## LoadedImage.java

LoadedImage — это удобный Canvas-подкласс, который берет изображение во время конструирования и синхронно загружает его, используя загрузчик MediaTracker. LoadedImage ведет себя должным образом под управлением менеджера компоновки LayoutManager потому, что он переопределяет методы getPreferredSize() и getMinimumSize(). Он также определяет специальный установочный метод set(). С его помощью можно добиться такого вида нового изображения (Image-объекта), чтобы оно могло отображаться в данном (пустом) окне (Canvas-объекте). Именно так фильтрованное изображение отображается после того, как подключение заканчивается.

```
import java.awt.*;  
  
public class LoadedImage extends Canvas {  
    Image img;  
  
    public LoadedImage(Image i) {  
        set(i);  
    }  
  
    void set(Image i) {  
        MediaTracker mt = new MediaTracker(this);  
        mt.addImage(i, 0);  
        try {  
            mt.waitForAll();  
        } catch (InterruptedException e) { };  
        img = i;  
        repaint();  
    }  
  
    public void paint(Graphics g) {  
        if (img == null) {  
            g.drawString("no image", 10, 30);  
        } else {  
            g.drawImage(img, 0, 0, this);  
        }  
    }  
}
```

```

public Dimension getPreferredSize() {
    return new Dimension(img.getWidth(this), img.getHeight(this));
}

public Dimension getMinimumSize() {
    return getPreferredSize();
}
}

```

## **Grayscale.java**

Фильтр `Grayscale` — это подкласс фильтра `RGBImageFilter`. Данная формулировка означает, что `Grayscale` может использоваться как `ImageFilter`-параметр для конструктора `FilteredImageSource`. Тогда все, что нужно сделать, это переопределить метод `filterRGB()` так, чтобы изменить входящие значения цветов. Он берет красные, зеленые и синие значения и вычисляет яркость пикселя, используя NTSC<sup>1</sup> фактор преобразования "цвет-яркость". Затем он просто возвращает серый пиксел, который имеет ту же яркость, что и цветной источник.

```

import java.applet.*;
import java.awt.*;
import java.awt.image.*;

class Grayscale extends RGBImageFilter implements PlugInFilter {
    public Image filter(Applet a, Image in) {
        return a.createImage(new FilteredImageSource(in.getSource(), this));
    }

    public int filterRGB(int x, int y, int rgb) {
        int r = (rgb >> 16) & 0xff;
        int g = (rgb >> 8) & 0xff;
        int b = rgb & 0xff;
        int k = (int) (.56 * g + .33 * r + .11 * b);
        return (0xff000000 | k << 16 | k << 8 | k);
    }
}

```

## **Invert.java**

Фильтр `Invert` также весьма прост. Он разбирает пикセル, отбирая красные, зеленые и синие каналы, и затем инвертирует их, вычитая их из 255. Эти инвертированные значения упаковываются обратно в значение пикселя.

---

<sup>1</sup> NTSC (National Television Standards Committee) — Национальный Телевизионный Комитет Стандартов. — Примеч. пер.

```

import java.applet.*;
import java.awt.*;
import java.awt.image.*;

class Invert extends RGBImageFilter implements PlugInFilter {
    public Image filter(Applet a, Image in) {
        return a.createImage(new FilteredImageSource(in.getSource(), this));
    }
    public int filterRGB(int x, int y, int rgb) {
        int r = 0xff - (rgb >> 16) & 0xff;
        int g = 0xff - (rgb >> 8) & 0xff;
        int b = 0xff - rgb & 0xff;
        return (0xffff000000 | r << 16 | g << 8 | b);
    }
}

```

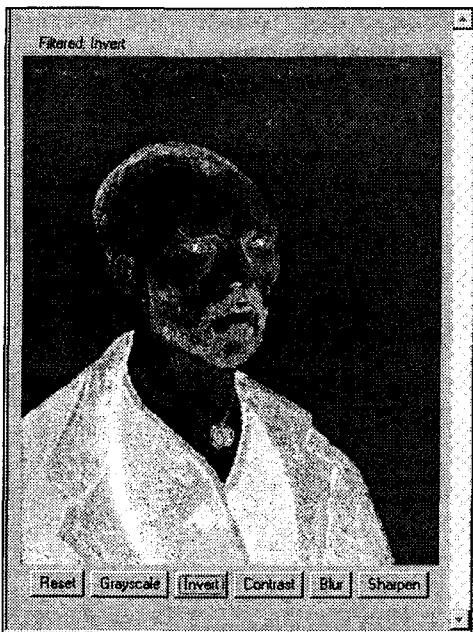


Рис. 23.9 показывает изображение после того, как оно было пропущено через фильтр Invert.

Рис. 23.9. Использование фильтра Invert с ImageFilterDemo

## *Contrast.java*

Фильтр Contrast очень похож на Grayscale, за исключением того, что его переопределение метода filterRGB() немного сложнее. Алгоритм, используемый им для улучшения контраста, берет красные, зеленые и синие значения отдельно и умножает их на 1.2, если они ярче, чем 128. Если их яркость ниже 128, то они делятся на 1.2. С помощью метода multclamp() обработанные таким способом значения, если нужно, ограничиваются величиной 255.

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class Contrast extends RGBImageFilter implements PlugInFilter {

    public Image filter(Applet a, Image in) {
        return a.createImage(new FilteredImageSource(in.getSource(), this));
    }

    private int multclamp(int in, double factor) {
        in = (int) (in * factor);
        return in > 255 ? 255 : in;
    }

    double gain = 1.2;
    private int cont(int in) {
        return (in < 128) ? (int) (in/gain) : multclamp(in, gain);
    }

    public int filterRGB(int x, int y, int rgb) {
        int r = cont((rgb >> 16) & 0xff);
        int g = cont((rgb >> 8) & 0xff);
        int b = cont(rgb & 0xff);
        return (0xffff0000 | r << 16 | g << 8 | b);
    }
}
```



Рис. 23.10 показывает изображение после сжатия контраста (с помощью кнопки **Contrast**).

Рис. 23.10. Использование фильтра  
Contrast в ImageFilterDemo

## Convolver.java

Абстрактный класс Convolver выполняет базовую обработку фильтра свертывания, реализуя интерфейс потребителя ImageConsumer для перемещения исходных пикселов в массив с именем imgpixels. Он также создает второй массив с именем newimgpixels для фильтрованных данных. Фильтры свертывания производят выборку маленького прямоугольника пикселов вокруг каждого пикселя в изображении, называемого *ядром свертывания*. Эта область ( $3 \times 3$  пикселя в данной демонстрации) используется для принятия решения, как следует изменить центральный пиксель в этой области. Два конкретных подкласса, показанные в следующем разделе, просто реализуют метод convolve(), используя imgpixels для исходных данных и newimgpixels для сохранения результата.

### Замечание

Причина того, что фильтр не может изменять массив imgpixels на месте, заключается в том, что следующий пиксель на строке сканирования пробовал бы использовать первоначальное значение для предыдущего пикселя, который был только что отфильтрован.

```

import java.applet.*;
import java.awt.*;
import java.awt.image.*;

abstract class Convolver implements ImageConsumer, PlugInFilter {
    int width, height;
    int imgpixels[], newimgpixels[];

    abstract void convolve();           // здесь идет фильтр...

    public Image filter(Applet a, Image in) {
        in.getSource().startProduction(this);
        waitForImage();
        newimgpixels = new int[width*height];

        try {
            convolve();
        } catch (Exception e) {
            System.out.println("Convolver failed: " + e);
            e.printStackTrace();
        }
        return a.createImage(
            new MemoryImageSource(width, height, newimgpixels, 0, width));
    }

    synchronized void waitForImage() {
        try { wait(); } catch (Exception e) { };
    }
}

```

```
public void setProperties(java.util.Hashtable dummy) { }
public void setColorModel(ColorModel dummy) { }
public void setHints(int dummy) { }

public synchronized void imageComplete(int dummy) {
    notifyAll();
}

public void setDimensions(int x, int y) {
    width = x;
    height = y;
    imgpixels = new int[x*y];
}

public void setPixels(int x1, int y1, int w, int h,
    ColorModel model, byte pixels[], int off, int scansize) {
    int pix, x, y, x2, y2, sx, sy;

    x2 = x1+w;
    y2 = y1+h;
    sy = off;
    for(y=y1; y<y2; y++) {
        sx = sy;
        for(x=x1; x<x2; x++) {
            pix = model.getRGB(pixels[sx++]);
            if((pix & 0xff000000) == 0)
                pix = 0x00ffffff;
            imgpixels[y*width+x] = pix;
        }
        sy += scansize;
    }
}

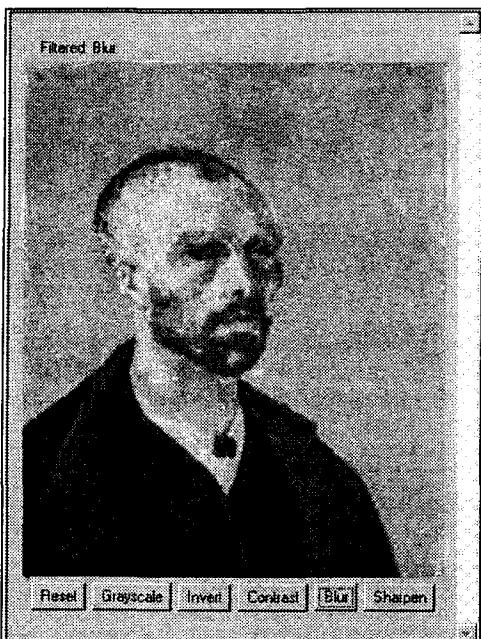
public void setPixels(int x1, int y1, int w, int h,
    ColorModel model, int pixels[], int off, int scansize) {
    int pix, x, y, x2, y2, sx, sy;

    x2 = x1+w;
    y2 = y1+h;
    sy = off;
    for(y=y1; y<y2; y++) {
        sx = sy;
        for(x=x1; x<x2; x++) {
            pix = model.getRGB(pixels[sx++]);
            if((pix & 0xff000000) == 0)
                pix = 0x00ffffff;
            imgpixels[y*width+x] = pix;
        }
    }
}
```

```
    sy += scansize;  
}  
}  
}
```

## *Blur.java*

Фильтр `Blur` — это подкласс `Convolver`. Он просто пробегает через каждый пиксель в исходном массиве изображения `imgpixels` и вычисляет среднее значение по всей окружающей его области  $3 \times 3$ . Результирующий пиксель помещается в соответствующую позицию массива `newimgpixels`. (Рис. 23.11 показывает апплет после `blur`-фильтра.)



**Рис. 23.11.** Использование фильтра Blur в ImageFilterDemo

## **Sharpen.java**

Фильтр `Sharpen` — также подкласс `Convolver` и является инверсией `Blur`. Он пробегает через каждый пиксель в исходном массиве изображения `imgpixels` и вычисляет среднее значение по окружающему  $3 \times 3$  блоку пикселов, не считая центральный. Соответствующие выводные пиксели в `newimgpixels` различаются между собой (в зависимости от окружения центрального пикселя) добавляемыми в них усреднениями. Грубо говоря, если пиксель в тридцать раз ярче, чем его окружение, подобная фильтрация делает примерно тридцать окружающих пикселов ярче. Если, однако, он в десять раз темнее, то примерно десять окружающих пикселов станут более темными. Это приводит к подчеркиванию граней изображения, оставляя гладкие области неизменными.

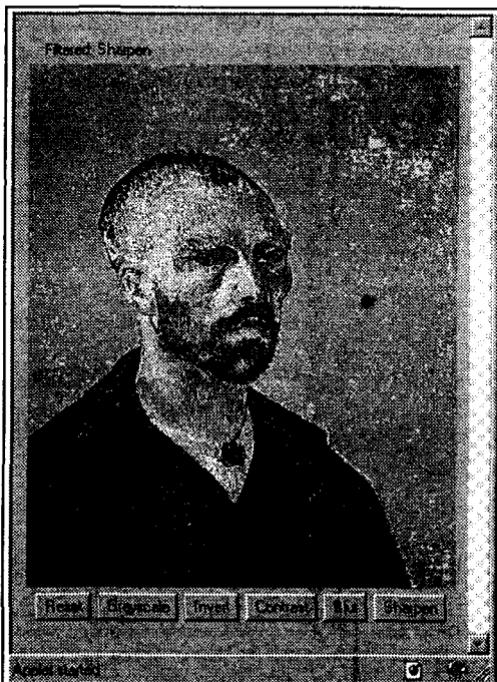


Рис. 23.12 демонстрирует апплет после Sharpen-фильтрации.

**Рис 23.12.** Использование фильтра Sharpen с ImageFilterDemo

## Анимация ячеек

Теперь, когда мы представили краткий обзор API-изображений, мы можем перейти к сборке интересного апплета, который отображает последовательность анимационных ячеек. Анимационные ячейки состоят из одиночных изображений, размещаемых в определенном порядке в ячейках сетки, специфицируемой параметрами `rows` и `cols` HTML-тега `<param>`. Каждая ячейка в изображении выделяется способом, подобным использованному ранее в примере `TileImage`. Последовательность отображения ячеек указывается параметром `sequence` тега `<param>`. Значение этого параметра — разделенный запятыми список номеров ячеек, отсчитываемых от нуля и следующих через сетку слева направо, сверху вниз.

После анализа тега `<param>` и загрузки исходного изображения апплет разрезает изображение на ряд мелких частей (подизображений). Затем запускается поток, который отображает их согласно порядку, описанному в параметре `sequence` тега `<param>`. Поток приостанавливается на достаточное время, чтобы поддерживать фреймовую скорость, указанную параметром `framerate` того же тега. Исходный код апплета `Animation`:

```
// Пример анимации изображения.
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
import java.util.*;

public class Animation extends Applet implements Runnable {
    Image cell[];
    final int MAXSEQ = 64;
    int sequence[];
    int nseq;
    int idx;
    int framerate;
    boolean stopFlag;

    private int intDef(String s, int def) {
        int n = def;
        if (s != null)
            try {
                n = Integer.parseInt(s);
            } catch (NumberFormatException e) { };
        return n;
    }

    public void init() {
        framerate = intDef(getParameter("framerate"), 5);
        int tilex = intDef(getParameter("cols"), 1);
        int tiley = intDef(getParameter("rows"), 1);
        sequence = new int[MAXSEQ];
        nseq = sequence.length;
        cell = new Image[nseq];
        for (int i = 0; i < nseq; i++)
            cell[i] = getImage("tile" + i + ".gif");
        idx = 0;
        start();
    }

    public void run() {
        while (!stopFlag) {
            repaint();
            try {
                Thread.sleep(framerate);
            } catch (InterruptedException e) { };
        }
    }

    public void stop() {
        stopFlag = true;
    }
}
```

```
int tiley = intDef(getParameter("rows"), 1);
cell = new Image[tilex*tiley];

 StringTokenizer st = new
     StringTokenizer(getParameter("sequence"), ",");
sequence = new int[MAXSEQ];
nseq = 0;
while(st.hasMoreTokens() && nseq < MAXSEQ) {
    sequence[nseq] = intDef(st.nextToken(), 0);
    nseq++;
}

try {
    Image img = getImage(getDocumentBase(), getParameter("img"));
    MediaTracker t = new MediaTracker(this);
    t.addImage(img, 0);
    t.waitForID(0);
    int iw = img.getWidth(null);
    int ih = img.getHeight(null);
    int tw = iw / tilex;
    int th = ih / tiley;
    CropImageFilter f;
    FilteredImageSource fis;
    for (int y=0; y<tiley; y++) {
        for (int x=0; x<tilex; x++) {
            f = new CropImageFilter(tw*x, th*y, tw, th);
            fis = new FilteredImageSource(img.getSource(), f);
            int i = y*tilex+x;
            cell[i] = createImage(fis);
            t.addImage(cell[i], i);
        }
    }
    t.waitForAll();
} catch (InterruptedException e) { };
}

public void update(Graphics g) { }

public void paint(Graphics g) {
    g.drawImage(cell[sequence[idx]], 0, 0, null);
}

Thread t;
public void start() {
    t = new Thread(this);
    stopFlag = false;
    t.start();
}
```

```

public void stop() {
    stopFlag = true;
}

public void run() {
    idx = 0;
    while (true) {
        paint(getGraphics());
        idx = (idx + 1) % nseq;
        try { Thread.sleep(1000/framerate); } catch (Exception e) { };
        if(stopFlag)
            return;
    }
}
}

```

Следующий тег <applet> показывает анимированное изображение известного исследования движений Эдварда Майбриджа (Eadweard Muybridge), который доказал, что лошади действительно отрывают от земли все четыре копыта сразу. В апплет, конечно, можно подставить другой графический файл.

```

<applet code=Animation width=67 height=48>
<param name=img value=horse.gif>
<param name=rows value=3>
<param name=cols value=4>
<param name=sequence value=0,1,2, 3,4,5,6,7,8,9,10>
<param name=framerate value=15>
</applet>

```

На рис. 23.13 показан результат выполнения апплета. Обратите внимание, что исходное изображение, загруженное перед анимированными, использует нормальный тег <img>.

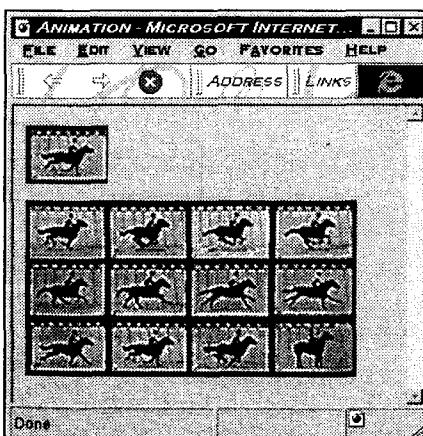
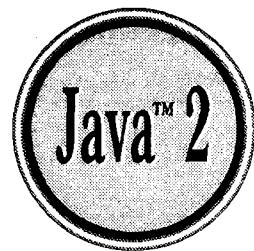


Рис. 23.13. Вывод апплета Animation

## Дополнительные классы изображений Java 2

В дополнение к классам изображений, описанным в этой главе, Java 2 обеспечивает несколько других, которые предлагают улучшенный контроль над процессом изображения и поддерживают усовершенствованную методику работы с ними. Если усовершенствованный графический вывод представляется для вас особый интерес, то в пакете `java.awt.image` вы найдете дополнительные классы для дальнейших исследований.

# ГЛАВА 24



## Дополнительные пакеты

Когда была выпущена версия Java 1.0, она включила набор из восьми пакетов, названных *ядром API* (core API). Они описаны в предшествующих главах и именно их вы будете использовать в ежедневном программировании чаще всего. Каждый последующий выпуск добавлялся к API-ядру. Сегодня API языка Java содержит большое количество пакетов. Многие из новых пакетов поддерживают специальные области, которые лежат вне рамок этой книги. Однако три пакета заслуживают краткого рассмотрения: `java.lang.reflect`, `java.rmi` и `java.text`. Они поддерживают отражение, вызов удаленных методов и форматирование текста, соответственно.

*Отражение* (reflection) — способность программного обеспечения анализировать себя. Это существенная часть технологии Java Beans, которой посвящена глава 25. Здесь приводятся примеры, представляющие эту концепцию. *Вызов удаленных методов* (RMI, Remote Method Invocation) позволяет построить приложения Java, которые распределены среди нескольких машин. В этой главе приведен простой пример клиент-сервера, который использует RMI. Возможности форматирования текста пакета `java.text` имеют много применений. Здесь рассматривается форматирование строк даты и времени.

### Пакеты ядра Java API

Все пакеты ядра API Java 2 перечислены в табл. 24.1. Там же кратко описаны их функции.

**Таблица 24.1. Пакеты API ядра Java**

Пакет	Первичная функция
<code>java.applet</code>	Поддерживает конструкцию апплета

Таблица 24.1 (продолжение)

Пакет	Первичная функция
java.awt	Обеспечивает возможности графических интерфейсов пользователя
java.awt.color	Поддерживает цветовые пространства и профили
java.awt.datatransfer	Передает данные к системному буферу обмена или от него
java.awt.dnd	Поддерживает операции перетаскивания мыши
java.awt.event	Обрабатывает события
java.awt.font	Представляет различные типы шрифтов
java.awt.geom.	Позволяет работать с геометрическими формами
java.awt.im	Разрешает ввод японских, китайских и корейских символов в компоненты редактирования текста
java.awt.image	Обрабатывает изображения
java.awt.image.renderable	Поддерживает независимые от визуализации изображения
java.awt.print	Поддерживает общие возможности печати
java.beans	Позволяет формировать программные beans-компоненты
java.beans.beancontext	Обеспечивает среду выполнения для beans-компонентов
java.io	Вводит и выводит данные
java.lang	Обеспечивает основные функциональные возможности языка
java.lang.ref	Активизирует некоторые взаимодействия со сборщиком мусора
java.lang.reflect	Анализирует код времени выполнения
java.math	Обрабатывает большие целые и десятичные числа
java.net	Поддерживает работу в сети
java.rmi	Обеспечивает удаленный вызов методов
java.rmi.activation	Активизирует постоянные объекты
java.rmi.dgc	Управляет распределенной сборкой мусора

Таблица 24.1 (продолжение)

Пакет	Первичная функция
java.rmi.registry	Отображает имена на ссылки удаленных объектов
java.rmi.server	Поддерживает вызов удаленных методов
java.security	Обрабатывает сертификаты, ключи, классификаторы, сигнатуры и другие функции защиты
java.security.acl	Управляет списками управления доступом
java.security.cert	Анализирует и управляет сертификатами
java.security.interfaces	Определяет интерфейсы для DSA-ключей (Digital Signature Algorithm, алгоритм цифровой сигнатуры)
java.security.spec	Определяет параметры ключей и алгоритма
java.sql	Общается с SQL базой данных (Structured Query Language, язык структурированных запросов)
java.text	Поддерживает форматирование, поиск и манипуляции с текстом
java.util	Содержит общие утилиты
java.util.jar	Создает и читает JAR-файлы (архивные файлы Java)
java.util.zip	Читает и записывает сжатые и несжатые ZIP-файлы
javax.swing	Содержит "облегченные" (Swing) компоненты
javax.swing.border	Рисует специализированные границы вокруг Swing-компонентов
javax.swing.colorchooser	Позволяет пользователю выбирать цвет Swing-компонента
javax.swing.event	Определяет события, генерируемые Swing-компонентами
javax.swing.filechooser	Позволяет пользователю библиотеки Swing выбирать файл (классы поддержки компонента JFileChooser)
javax.swing.plaf	Поддерживает plaf-свойства (pluggable look-and-feel) библиотеки Swing. Эти классы предназначены для разработчиков, создающих собственные модули-приложения стилей
javax.swing.plaf.basic	Реализует базовые (Basic) plaf-стили интерфейса пользователя (для создания графической среды в стиле Windows)

Таблица 24.1 (окончание)

Пакет	Первичная функция
javax.swing.plaf.metal	Реализует платформно-независимый (Metal) plaf-стиль интерфейса пользователя
javax.swing.plaf.mulfi	Сочетает вспомогательный и заданный по умолчанию plaf-стили интерфейсов пользователя (технология мультиплексирования стилей)
javax.swing.table	Обеспечивает таблицы (классы поддержки компонента JTable)
javax.swing.text	Обеспечивает текстовые компоненты (классы поддержки Swing-технологии создания документов)
javax.swing.text.html	Позволяет создавать собственные редакторы HTML-файлов (классы библиотеки HTMLEditorKit)
javax.swing.text.html.rtf	Позволяет создавать собственные редакторы RTF-файлов
javax.swing.tree	Классы поддержки для работы со Swing-компонентом JTree
javax.swing.undo	Классы, обеспечивающие реализацию функций отмены/повторения выполненных действий (в технологии Swing)
CORBA	Пакет поддержки CORBA-технологии. Содержит простой модуль ORB, написанный на языке Java
org.OMG.CORBA.DynAnyPackage	Подпакет org.OMG.CORBA
org.OMG.CORBA.ORBPackage	Подпакет org.OMG.CORBA
org.OMG.CORBA.portable	Подпакет org.OMG.CORBA
org.OMG.CORBA.TypeCodePackage	Подпакет org.OMG.CORBA
org.OMG.CORBA.CosNaming	Обеспечивает именование в языке IDL (Interface Definition Language, язык описания интерфейса)
org.OMG.CORBA.CosNaming.NamingContextPackage	Определяет исключения для именований в языке IDL (подпакет предыдущего пакета)

## Отражение

Отражение — это способность программного обеспечения к самоанализу. Его обеспечивает пакет `java.lang.reflect` и элементы класса `Class`. Отраже-

ние — важная возможность, необходимая при использовании компонентов, называемых Java Beans. Она позволяет анализировать программный компонент и описывать его возможности динамически, во время выполнения, а не во время компиляции. Например, используя отражение, можно определять, какие методы, конструкторы и поля поддерживает данный класс.

Пакет `java.lang.reflect` содержит один интерфейс, называемый `Member`, который определяет методы, позволяющие получать информацию о поле, конструкторе или методе класса. В этом пакете имеются также семь классов, перечисленные в табл. 24.2.

**Таблица 24.2. Классы, определенные в пакете `java.lang.reflect`**

Класс	Первичная функция
<code>AccessibleObject</code>	Позволяет обходить заданные по умолчанию проверки управления доступом. (Добавлен в Java 2)
<code>Array</code>	Позволяет динамически создавать и манипулировать массивами
<code>Constructor</code>	Обеспечивает информацию о конструкторе
<code>Field</code>	Обеспечивает информацию о поле
<code>Method</code>	Обеспечивает информацию о методе
<code>Modifier</code>	Обеспечивает информацию о модификаторах доступа класса и члена
<code>ReflectPermission</code>	Разрешает отражение <code>private</code> или <code>protected</code> членов класса. (Добавлен Java 2)

Следующее приложение иллюстрирует простое использование возможностей отражения Java. Оно печатает конструкторы, поля и методы класса `java.awt.Dimension`. Программа начинается использованием метода `forName()` класса `Class`, чтобы получить объект типа `Class` для `java.awt.Dimension`. Как только он получен, используются методы `getConstructors()`, `getFields()` и `getMethods()` для анализа этого объекта. Они возвращают массивы объектов `Constructor`, `Field` и `Method`, которые обеспечивают информацию об объекте. Классы `Constructor`, `Field` и `Method` определяют несколько методов, которые можно использовать для получения информации об объекте. Однако каждый поддерживает метод `toString()`. Он позволяет использовать объекты типа `Constructor`, `Field` и `Method` в качестве аргументов метода `println()`, как показано в следующей программе.

```
// Демонстрирует отражение.
import java.lang.reflect.*;
public class ReflectionDemo1 {
```

```

public static void main(String args[]) {
    try {
        Class c = Class.forName("java.awt.Dimension");
        System.out.println("Constructors:");
        Constructor constructors[] = c.getConstructors();
        for(int i = 0; i < constructors.length; i++) {
            System.out.println(" " + constructors[i]);
        }

        System.out.println("Fields:");
        Field fields[] = c.getFields();
        for(int i = 0; i < fields.length; i++) {
            System.out.println(" " + fields[i]);
        }

        System.out.println("Methods:");
        Method methods[] = c.getMethods();
        for(int i = 0; i < methods.length; i++) {
            System.out.println(" " + methods[i]);
        }
    }
    catch(Exception e) {
        System.out.println("Exception: " + e);
    }
}
}

```

Вывод этой программы:

Constructors:

```

public Java.awt.Dimension()
public Java.awt.Dimension(int, int)
public Java.awt.Dimension(Java.awt.Dimension)

```

Fields:

```

public static final long
java.io.Serializable.serialVersionUID
public int java.awt.Dimension.width
public int java.awt.Dimension.height

```

Methods:

```

public Java.lang.Object java.awt.geom.Dimension2D.clone()
public boolean java.awt.Dimension.equals(Java.lang.Object)
public final native Java.lang.Class
java.lang.Object.getClass()
public native int Java.lang.Object.hashCode()
public final native void Java.lang.Object.notify()
public final native void Java.lang.Object.notifyAll()
public Java.lang.String java.awt.Dimension.toString()
public final void Java.lang.Object.wait()
throws Java.lang.InterruptedException

```

```
public final native void Java.lang.Object.wait(long)
    throws Java.lang.InterruptedException
public final native void Java.lang.Object.wait(long, int)
    throws Java.lang.InterruptedException
public double java.awt.Dimension getHeight()
public double java.awt.Dimension getWidth()
public void java.awt.Dimension.setSize(double, double)
public void java.awt.geom.Dimension2D.setSize
    (java.awt.geom.Dimens ion2D)
public java.awt.Dimension java.awt.Dimension.getSize()
public void java.awt.Dimension.setSize(int, int)
public void java.awt.Dimension.setSize(Java.awt.Dimén sion)
```

Следующий пример применяет способности отражения Java получать public-методы класса. Программа начинается с построения экземпляра класса A. Используя его объектную ссылку, метод getClass() возвращает объект Class в качестве класса A. Метод getDeclaredMethods() возвращает массив объектов Method, который описывает только методы, объявленные этим классом. Методы, унаследованные от суперклассов, таких как Object, не включаются.

Затем обрабатывается каждый элемент массива methods. Метод getModifiers() возвращает значение типа int, содержащее флаги, которые описывают, какой модификатор доступа применяется для этого элемента. Класс Modifier обеспечивает набор методов, показанных в табл. 24.3, которые могут использоваться для просмотра этих значений. Например, статический метод isPublic() возвращает true, если его аргумент включает модификатор доступа public, иначе — false. В следующей программе, если метод поддерживает общий доступ, его имя получается методом getName() и затем распечатывается.

```
// Показывает public-методы.
import java.lang.reflect.*;
public class ReflectionDemo2 {
    public static void main(String args[]) {
        try {
            A a = new A();
            Class c = a.getClass();
            System.out.println("Public Methods:");
            Method methods[] = c.getDeclaredMethods();
            for(int i = 0; i < methods.length; i++) {
                int modifiers = methods[i].getModifiers();
                if(Modifier.isPublic(modifiers)) {
                    System.out.println(" " + methods[i].getName());
                }
            }
        }
    }
}
```

```

        catch(Exception e) {
            System.out.println("Exception: " + e);
        }
    }

class A {
    public void a1() {
    }
    public void a2() {
    }
    protected void a3() {
    }
    private void a4() {
    }
}

```

Вывод этой программы:

Public Methods:

a1

a2

**Таблица 24.3.** Методы класса *Modifier*, проверяющие характеристики модификаторов доступа

Метод	Описание
<code>static boolean isAbstract(int val)</code>	Возвращает true, если в val установлен флагок abstract, иначе — false
<code>static boolean isFinal(int val)</code>	Возвращает true, если в val установлен флагок final, иначе — false
<code>static boolean isInterface(int val)</code>	Возвращает true, если в val установлен флагок interface, иначе — false
<code>static boolean isNative(int val)</code>	Возвращает true, если в val установлен флагок native, иначе — false
<code>static boolean isPrivate(int val)</code>	Возвращает true, если в val установлен флагок private, иначе — false
<code>static boolean isProtected(int val)</code>	Возвращает true, если в val установлен флагок protected, иначе — false

**Таблица 24.3 (окончание)**

Метод	Описание
<code>static boolean isPublic(int val)</code>	Возвращает true, если в val установлен флагок public, иначе — false
<code>static boolean isStatic(int val)</code>	Возвращает true, если в val установлен флагок static, иначе — false
<code>static boolean isStrict(int val)</code>	Возвращает true, если в val установлен флагок strict, иначе — false
<code>static boolean isSynchronized(int val)</code>	Возвращает true, если в val установлен флагок synchronized, иначе — false
<code>static boolean isTransient(int val)</code>	Возвращает true, если в val установлен флагок transient, иначе — false
<code>static boolean isVolatile(int val)</code>	Возвращает true, если в val установлен флагок volatile, иначе — false

## Вызов удаленных методов (RMI)

Вызов удаленных методов (RMI, Remote Method Invocation) позволяет Java-объекту, который выполняется на одной машине, вызвать метод объекта Java, выполняющийся на другой машине. Это свойство является важным из-за того, что позволяет строить распределенные приложения. Хотя полное обсуждение RMI — вне рамок этой книги, основные принципы описаны в следующем примере.

### Простое RMI-приложение клиент-сервер

Данный раздел содержит пошаговое руководство для построения с помощью RMI простого клиент-серверного приложения. Сервер принимает запрос от клиента, обрабатывает его и возвращает результат. В этом примере запрос специфицирует два числа. Сервер складывает их и возвращает сумму.

#### Шаг 1. Ведите и компилируйте исходный код

Это приложение использует четыре исходных файла. Первый файл, AddServerIntf.java, определяет удаленный интерфейс, который обеспечива-

ется сервером. Он содержит один метод, принимающий два параметра и возвращающий их сумму. Все удаленные интерфейсы должны расширять Remote-интерфейс, который является частью `java.rmi`. `Remote` не определяет никаких членов. Его цель — просто указать, что интерфейс использует удаленные методы. Все удаленные методы могут выбрасывать исключения `RemoteException`.

```
import java.rmi.*;
public interface AddServerIntf extends Remote {
    double add(double d1, double d2) throws RemoteException;
}
```

Второй исходный файл, `AddServerImpl.java`, реализует удаленный интерфейс. Реализация метода `add()` — прямая. Все удаленные объекты должны расширять `UnicastRemoteObject`, который обеспечивает функциональные возможности объектов на удаленных машинах. (В настоящее время поддерживается только односторонние (unicast) серверы, т. е. RMI не поддерживает конфигураций, включающих реплицированные серверы.)

```
import java.rmi.*;
import java.rmi.server.*;
public class AddServerImpl extends UnicastRemoteObject
    implements AddServerIntf {

    public AddServerImpl() throws RemoteException {
    }

    public double add(double d1, double d2) throws RemoteException {
        return d1 + d2;
    }
}
```

Третий исходный файл, `AddServer.java`, содержит программу `main()` для машины сервера. Ее первичная функция должна обновить RMI-реестр на этой машине. Это делается с помощью метода `rebind()` класса `Naming` (из `java.rmi`). Указанный метод связывает имя с объектной ссылкой. Первый аргумент метода — строка, которая именует сервер как "AddServer". Его второй аргумент — ссылка на экземпляр `addServerImpl`.

```
import java.net.*;
import java.rmi.*;
public class AddServer {
    public static void main(String args[]) {
        try {
            AddServerImpl addServerImpl = new AddServerImpl();
            Naming.rebind("AddServer", addServerImpl);
        }
    }
}
```

```
    catch(Exception e) {
        System.out.println("Exception: " + e);
    }
}
```

Четвертый исходный файл, `AddClient.java`, реализует сторону клиента этого распределенного приложения. `AddClient.java` требует трех параметров командной строки. Первый — IP-адрес или имя машины сервера. Второй и третий параметры — два числа, которые нужно суммировать.

Приложение начинается с формирования строки, которая следует синтаксису URL. Этот URL использует протокол `rmi`. Стока включает IP-адрес или имя сервера и строку "AddServer". Затем программа вызывает метод `lookup()` класса `Naming`. Этот метод принимает один аргумент, `rmi` URL, и возвращает ссылку на объект типа `AddServerIntf`. Все удаленные вызовы методов можно направить к этому объекту.

Программа продолжается отображением ее параметров и затем вызывает удаленный метод `add()`, который возвращает и затем распечатывает сумму чисел.

```
import java.rmi.*;
public class AddClient {
    public static void main(String args[]) {
        try {
            String addServerURL = "rmi://" + args[0] + "/AddServer";
            AddServerIntf addServerIntf =
                (AddServerIntf)Naming.lookup(addServerURL);
            System.out.println("The first number is: " + args[1]);
            double d1 = Double.valueOf(args[1]).doubleValue();
            System.out.println("The second number is: " + args[2]);

            double d2 = Double.valueOf(args[2]).doubleValue();
            System.out.println("The sum is: " + addServerIntf.add(d1, d2));
        }
        catch(Exception e) {
            System.out.println("Exception: " + e);
        }
    }
}
```

После ввода всего кода используйте Java-компилятор (`javac`), чтобы компилировать четыре исходных файла, которые вы создали.

## Шаг 2. Генерируйте заглушки и скелеты

Перед использованием клиента и сервера нужно сгенерировать необходимую заглушку, а также скелетную схему. В контексте RMI заглушка (stub) —

это Java-объект, который постоянно находится на машине клиента. Его функция состоит в представлении тех же интерфейсов, что и на удаленном сервере. Вызовы удаленного метода, инициированные клиентом, фактически направляются к заглушке. Заглушка работает с другими частями RMI системы для формулировки запроса, который посыпается к удаленной машине.

Удаленный метод может принимать параметры, которые являются простыми типами или объектами. В последнем случае объект может иметь ссылки к другим объектам. Вся эта информация должна быть отправлена удаленной машине. Объект, передаваемый как параметр вызова удаленного метода, должен быть сериализован (преобразован в последовательную форму) и послан на удаленную машину. Напомним (см. гл. 17), что средства сериализации рекурсивно обрабатывают также все ссылочные объекты.

Скелетные схемы (скелеты) в Java 2 не используются. Однако они требуются для RMI модели Java 1.1. Поэтому скелеты все еще требуются для совместимости между Java 1.1 и Java 2. *Скелет* (skelton) — это объект Java, который постоянно находится на машине сервера. Он работает с другими частями RMI системы Java 1.1, чтобы принимать запросы, выполнять десериализацию и вызывать соответствующий код на сервере. Подчеркнем еще раз, что для кода Java 2 не требуется не только скелетный механизм, но и совместимость с Java 1.1. Поскольку большинство читателей захочет сгенерировать скелет, один из них используется в данном примере.

Если клиенту должен быть возвращен ответ, выполняется обратный процесс. Обратите внимание, что если объекты возвращаются клиенту, то средства сериализации и десериализации тоже используются.

Чтобы генерировать заглушки и скелеты, нужно использовать инструмент, называемый *компилятором RMI*, который вызывается из командной строки так:

```
rmic AddServerImpl
```

Данная команда генерирует два новых файла: `AddServerImplSkel.class` (скелет) и `AddServerimplStub.class` (заглушка). При использовании `rmic` убедитесь, что `CLASSPATH` включает текущий каталог. По умолчанию `rmic` генерирует как заглушки, так и скелетный файл. Если вы не нуждаетесь в скелете, существует возможность подавить его.

### Шаг 3. Установите файлы на машинах сервера и клиента

Скопируйте `AddClient.class`, `AddServerImpl_Stub.class` и `AddServerIntf.class` в каталог на машине клиента, а `AddServerIntf.class`, `AddServerImpl.class`, `AddServerImpl_Skel.class`, `AddServerImpl_Stub.class` и `AddServer.class` — в каталог на машине сервера.

### Замечание

RMI имеет методики для динамической загрузки класса, но они не применяются рассматриваемым примером. Вместо этого все файлы, которые используются клиентскими и серверными приложениями, должны быть установлены на этих машинах вручную.

## Шаг 4. Запустите RMI-реестр на машине сервера

JDK обеспечивает программу, называемую `rmiregistry`, которая выполняется на машине сервера. Она отображает имена объектных ссылок. Сначала, проверьте, чтобы переменная среды `CLASSPATH` включала каталог, в котором расположены ваши файлы. Затем запустите RMI-реестр из командной строки:

```
start rmiregistry
```

Когда эта команда возвратит управление, вы должны увидеть, что было создано новое окно. Пока вы выполняете эксперименты с RMI-примером, нужно оставить это окно открытым.

## Шаг 5. Запустите сервер

Программа сервера запускается следующей командной строкой:

```
java AddServer
```

Напомним, что программа `AddServer` создает экземпляр `AddServerImpl` и регистрирует этот объект с именем "AddServer".

## Шаг 6. Запустите клиента

Программное обеспечение `AddClient` требует три параметра: имя или IP-адрес машины сервера и два числа, которые должны быть суммированы. Вы можете вызывать его из командной строки, используя один из двух форматов:

```
Java AddClient server1 8 9  
Java AddClient 11.12.13.14 8 9
```

В первой строке указывается имя сервера. Вторая строка использует его IP-адрес (11.12.13.14).

Вы можете проверить данный пример, не имея удаленного сервера. Чтобы сделать это, просто установите все программы на той же самой машине, запустите `rmiregistry`, потом запустите `AddServer` и затем — `AddClient`, используя следующую командную строку:

```
java AddClient 127.0.0.1 8 9
```

Здесь, адрес 127.0.0.1 — это адрес "обратной петли" для локальной машины. Использование этого адреса позволяет осуществлять полный RMI-механизм без фактической необходимости установки сервера на удаленном компьютере.

В любом случае пример вывода этой программы выглядит так:

```
The first number is: 8  
The second number is: 9  
The sum is: 17
```

## Текстовое форматирование

Пакет `java.text` позволяет форматировать, отыскивать и обрабатывать текст. В этом разделе кратко рассматриваются наиболее часто используемые классы, которые форматируют информацию даты и времени.

### Класс `DateFormat`

`DateFormat` — абстрактный класс, который обеспечивает способность форматировать и синтаксически анализировать дату и время. Метод `getDateInstance()` возвращает экземпляр класса `DateFormat`, который может форматировать информацию даты. Он доступен в следующих формах:

```
static final DateFormat getDateInstance()  
static final DateFormat getDateInstance(int style)  
static final DateFormat getDateInstance(int style, Locale locale)
```

Параметр `style` принимает одно из следующих значений: `DEFAULT`, `SHORT`, `MEDIUM`, `LONG` или `FULL`. Это `int`-константы, определенные в `DateFormat`. Они предоставляют различные подробности относительно формата даты. Параметр `locale` — одна из статических ссылок, определенных в классе `Locale` (см. гл. 16). Если `style` и/или `locale` не определены, используются значения по умолчанию.

Один из наиболее часто используемых методов в этом классе — `format()`. Он имеет несколько перегруженных форм, одна из которых:

```
String format(Date d)
```

имеет в качестве параметра объект `d` класса `Date`, который должен быть отображен. Метод возвращает строку, содержащую отформатированную информацию.

Следующая программа иллюстрирует, как форматировать информацию даты. Она начинается с создания `Date`-объекта, который собирает текущую информацию даты и времени, а затем выводит эту информацию, используя различные стили и регионы.

```
// Демонстрирует форматы даты.  
import java.text.*;  
import java.util.*;
```

```
public class DateFormatDemo {  
    public static void main(String args[]) {  
        Date date = new Date();  
        DateFormat df;  
  
        df = DateFormat.getDateInstance(DateFormat.SHORT, Locale.JAPAN);  
        System.out.println("Japan: " + df.format(date));  
  
        df = DateFormat.getDateInstance(DateFormat.MEDIUM, Locale.KOREA);  
        System.out.println("Korea: " + df.format(date));  
  
        df = DateFormat.getDateInstance(DateFormat.LONG, Locale.UK);  
        System.out.println("United Kingdom: " + df.format(date));  
  
        df = DateFormat.getDateInstance(DateFormat.FULL, Locale.US);  
        System.out.println("United States: " + df.format(date));  
    }  
}
```

Пример вывода этой программы:

```
Japan: 99/02/19  
Korea: 1999-02-19  
United Kingdom: 19 February 1999  
United States: Friday, February 19, 1999
```

Метод `getTimeInstance()` возвращает объект типа `DateFormat`, который может форматировать информацию времени. Он доступен в следующих версиях:

```
static final DateFormat getTimeInstance()  
static final DateFormat getTimeInstance(int style)  
static final DateFormat getTimeInstance(int style, Locale locale)
```

Параметр `style` принимает одно из следующих значений: `DEFAULT`, `SHORT`, `MEDIUM`, `LONG` или `FULL`. Это int-константы, определенные в `DateFormat`. Они предоставляют различные подробности относительно формата времени. Параметр `locale` — одна из статических ссылок, определенных в классе `Locale`. Если `style` и/или `locale` не определены, используются значения по умолчанию.

Следующий пример программы иллюстрирует, как форматировать информацию времени. Он начинается с создания объекта типа `Date`, который собирает текущие сведения о дате и времени, а затем выводит информацию времени, используя различные стили и регионы.

```
// Демонстрирует форматы времени.  
import java.text.*;  
import java.util.*;  
public class TimeFormatDemo {  
    public static void main(String args[]) {
```

```

Date date = new Date();
DateFormat df;

df = DateFormat.getTimeInstance(DateFormat.SHORT, Locale.JAPAN);
System.out.println("Japan: " + df.format(date));

df = DateFormat.getTimeInstance(DateFormat.LONG, Locale.UK);
System.out.println("United Kingdom: " + df.format(date));

df = DateFormat.getTimeInstance(DateFormat.FULL, Locale.CANADA);
System.out.println("Canada: " + df.format(date));
}
}

```

Пример вывода этой программы:

```

Japan: 20:25
United Kingdom: 20:25:14 GMT-05:00
Canada: 8:25:14 o'clock PM EST

```

Класс `DateFormat` включает также метод `getDateTimeInstance()`, который может форматировать информацию как дат, так и времени.

## Класс `SimpleDateFormat`

`SimpleDateFormat` — конкретный подкласс `DateFormat`. Он позволяет определять ваши собственные образцы форматирования, которые используются для отображения даты и времени.

Один из его конструкторов:

```
SimpleDateFormat(String formatString)
```

Параметр `formatString` описывает, как отображается информация даты и времени. Пример его применения:

```
SimpleDateFormat sdf = SimpleDateFormat("dd MMM yyyy hh:iran:ss zzz");
```

Символы, используемые в строке форматирования, определяют информацию, которая отображается. Табл. 24.4 перечисляет эти символы и дает описание каждого.

**Таблица 24.4. Символы строк форматирования для `SimpleDateFormat`**

Символ	Описание
a	AM или PM (Ante Meridiem/Post Meridiem)
d	День месяца (1–31)
h	Час в AM/PM (1–12)

Таблица 24.4 (окончание)

Символ	Описание
k	Час в дне (1–24)
m	Минута в часе (0–59)
s	Секунда в минуте (0–59)
w	Неделя года (1–52)
y	Год
z	Временная зона
D	День года (1–366)
E	День недели (например, Четверг)
F	День недели в месяце
G	Эра (AD (Anno Domini, наша эра) или BC (Before Crist, до нашей эры))
H	Час в дне (0–23)
K	Час в AM/PM (0–11)
M	Месяц
S	Миллисекунда
W	Неделя месяца (1–5)
'	Escape-символ

В большинстве случаев количество повторений символа определяет, как это данные представляются. Текстовая информация отображается в сокращенной форме, если символ образца воспроизведен меньше чем четыре раза. Иначе используется несокращенная форма. Например, образец zzzz может отображать Pacific Daylight Time, а образец zzz — PDT.

Для чисел количество повторений символа образца определяет, сколько цифр представляется. Например, hh:mm:ss может представлять 01:51:15, но h:m:s отображает то же значение времени как 1:51:15.

Наконец, M. или MM заставляют отображать месяц как одну или две цифры. Однако три или большее количество повторений M. отображает месяц, как текстовую строку.

Следующая программа показывает, как этот класс используется:

```
// Демонстрирует SimpleDateFormat.
import java.text.*;
import java.util.*;
```

```
public class SimpleDateFormatDemo {  
    public static void main(String args[]) {  
        Date date = new Date();  
        SimpleDateFormat sdf;  
        sdf = new SimpleDateFormat("hh:mm:ss");  
        System.out.println(sdf.format(date));  
        sdf = new SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");  
        System.out.println(sdf.format(date));  
        sdf = new SimpleDateFormat("E MMM dd yyyy");  
        System.out.println(sdf.format(date));  
    }  
}
```

Пример вывода этой программы:

```
11:51:50  
19 Feb 1999 11:51:50 CST  
Fri Feb 19 1999
```

# **РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**



## **ЧАСТЬ III**

- 25. Компоненты Java Beans**
- 26. Система Swing**
- 27. Сервлеты**
- 28. Миграция из C++ в Java**



## ГЛАВА 25



# Компоненты Java Beans

Эта глава дает краткий обзор захватывающей технологии, которая находится на переднем крае Java-программирования, — технологии Java Beans. Beans<sup>1</sup>-компоненты важны, потому что они позволяют формировать комплексные системы из небольших программных компонентов. Эти компоненты вы можете создавать сами или покупать у одного или нескольких поставщиков. Java Beans определяет архитектуру, которая указывает, как эти строительные блоки могут работать вместе.

Чтобы лучше понять значение Beans-компонентов, приведем следующие соображения. Проектировщики аппаратуры имеют широкое разнообразие компонентов, которые могут быть интегрированы вместе, чтобы создать систему. Резисторы, конденсаторы, и катушки индуктивности — примеры простых стандартных блоков. Интегральные схемы обеспечивают более широкие функциональные возможности. Все эти части могут многократно использоваться. Нет необходимости или даже возможности перестраивать эти части каждый раз, когда нужна новая система. Кроме того, те же самые части могут использоваться в различных типах цепей. Это возможно потому, что поведение этих компонентов понятно и документировано.

К сожалению, программная индустрия не добилась успеха в достижении выгод от повторного использования и способности к взаимодействию. Большие приложения постоянно усложняются и становятся очень трудными для обслуживания и совершенствования. Часть проблемы состоит в том, что до недавнего времени не было стандартного, переносимого способа записывать программный компонент (на исходном языке). Чтобы достичь выгод от компонентного программного обеспечения, необходима компонентная архитектура, которая позволяет собирать программы из программных строительных блоков, возможно, поставляемых различными продавцами. Кроме

<sup>1</sup> От англ. beans — зёрна (любого рода — бобы, горох, фасоль, ореховые или просто маленькие шарики). — Примеч. пер.

того, разработчику должно быть предоставлено право выбирать компонент, понимать его возможности и включать в приложение. Когда новая версия компонента становится доступной, включение его функциональных возможностей в существующий код должно быть достаточно простым делом. К счастью, Java Beans является именно такой архитектурой.

Эта глава начинается с определения Java Bean-компонента и описания преимуществ, которые он обеспечивает для проектировщика программ. Вы увидите, как можно использовать комплект разработчика Bean-компонентов (BDK, Bean Developer Kit), который поставляется фирмой JavaSoft. Это инструментарий, который позволяет связывать несколько Bean-компонентов. Наконец, вы увидите пошаговый процесс, требуемый для записи простого Bean-компонента и использования его в BDK.

## Что такое Java Bean-компонент?

*Java Bean-компонент* является программным компонентом, который разработан так, чтобы многократно использоваться в различных средах. Нет никакого ограничения на возможности Bean-компонента. Он может выполнять простую функцию (например, проверку орфографии документа) или сложную (например, прогнозирование эффективности биржевого портфеля). Bean-компонент может быть видим конечному пользователю (например, как кнопка в графическом интерфейсе пользователя), но может быть также и невидим для него. Примером строительного блока подобного типа является программное обеспечение для декодирования потока мультимедийной информации в реальном времени. Наконец, Bean-компонент может быть разработан для автономной работы на рабочей станции пользователя или для работы в кооперации с набором других распределенных компонентов. Примером Bean-компонента, который может выполняться локально, является программное обеспечение для генерации круговой диаграммы из набора элементов данных. Однако Bean-компонент, обеспечивающий ценовую информацию в реальном масштабе времени с фондовой или товарной биржи, вынужден работать во взаимодействии с другим распределенным программным обеспечением (с целью получать его данные).

Далее мы увидим, какие специфические изменения разработчик программ должен сделать с классом, чтобы он стал пригодным для использования в качестве Bean-компонента. Одной из целей разработчиков Java было создание *простой в использовании* технологии. Поэтому изменения кода являются минимальными.

## Преимущества технологии Java Beans

Архитектура программных компонентов обеспечивает стандартные механизмы для работы с программными строительными блоками. Ниже перечислены

ны некоторые из специфических выгод, которые обеспечивает технология Java для разработчика компонентов.

- Bean-компонент получает все выгоды от Java-парадигмы "писать — однажды, выполнять — где угодно".
- Свойства, события и методы Bean-компонента, которые предъявляются инструменту построения приложений, могут быть управляемыми.
- Bean-компонент может быть спроектирован для правильной работы в различных языковых регионах, что делает его полезным для глобальных рынков.
- Для помощи в конфигурировании Bean-компонента возможно использование вспомогательного программного обеспечения. Это программное обеспечение необходимо только тогда, когда для того компонента устанавливаются параметры времени разработки. Его не требуется включать в среду времени выполнения.
- Параметры настройки конфигурации Bean-компонента могут быть сохранены в постоянной памяти и восстановлены в более позднее время.
- Bean-компонент можно регистрировать, чтобы принимать события от других объектов, и он может генерировать события, которые посылаются к другим объектам.

## Инструментарий построения приложений

При работе с компонентами Java Beans большинство разработчиков использует *инструментарий построения приложений*. Это утилита, которая дает возможность конфигурировать набор Bean-компонентов, соединять их вместе и создавать работающее приложение. Ее главные возможности:

- Обеспечение палитры, которая перечисляет все доступные Bean-компоненты. Когда разработан или куплен дополнительный Bean-компонент, его можно добавить к палитре.
- Отображение рабочего листа, который позволяет проектировщику располагать Bean-компонент в графическом интерфейсе пользователя. Проектировщик может перетащить Bean-компонент из палитры в этот рабочий лист.
- Использование специальных редакторов и настройщиков, позволяющих конфигурировать Bean-компонент. Это механизм, с помощью которого поведение Bean-компонента может быть адаптировано к специфической среде.
- Применение специальных команд, разрешающих проектировщику запрашивать состояние и поведение Bean-компонента. Эта информация ав-

томатически становится доступной, когда Bean-компонент добавляется к палитре.

- Связывание Bean-компонентов. Это означает, что события, сгенерированные одним компонентом отображаются в вызовы методов других компонентов.
- Сохранение Bean-компонентов в постоянной области памяти, когда их коллекция сконфигурирована и связана. Эта информация может использоваться в более позднее время для восстановления состояния приложения.

## Комплект разработчика Bean-компонентов

Комплект разработчика Bean-компонентов (BDK, Bean Developer Kit), доступный на сайте JavaSoft, является простым примером инструментария, который дает возможность создавать, конфигурировать и подключать набор Bean-компонентов. Существует также набор образцов Bean-компонентов с их исходным кодом. Этот раздел содержит пошаговые инструкции для установки и использования данного инструмента.

### Замечание

В текущей главе приводятся инструкции для среды Windows 95/98/NT. Процедуры для платформы UNIX аналогичны, но некоторые из команд различны.

## Установка BDK

Для работы BDK на вашей машине должен быть установлен JDK. Убедитесь, что инструментальные средства JDK доступны из вашей среды.

Затем можно загрузить BDK с сайта JavaSoft (<http://java.sun.com>). Он упакован в один файл, который является саморазворачивающимся архивом. Следуйте инструкциям для его установки на вашей машине.

## Запуск BDK

Чтобы запустить BDK:

1. Смените каталог на `c:\bdk\beanbox`.
2. Выполните пакетный файл `run.bat`. Это заставляет BDK отобразить три окна, показанных на рис. 25.1. Список **ToolBox** перечисляет все Bean-компоненты, которые были включены в BDK. **BeanBox** обеспечивает область для размещения и подключения Bean-компонентов, выбранных из **ToolBox**. Окно **Properties** (Свойства) дает возможность конфигурировать выбранный Bean-компонент.

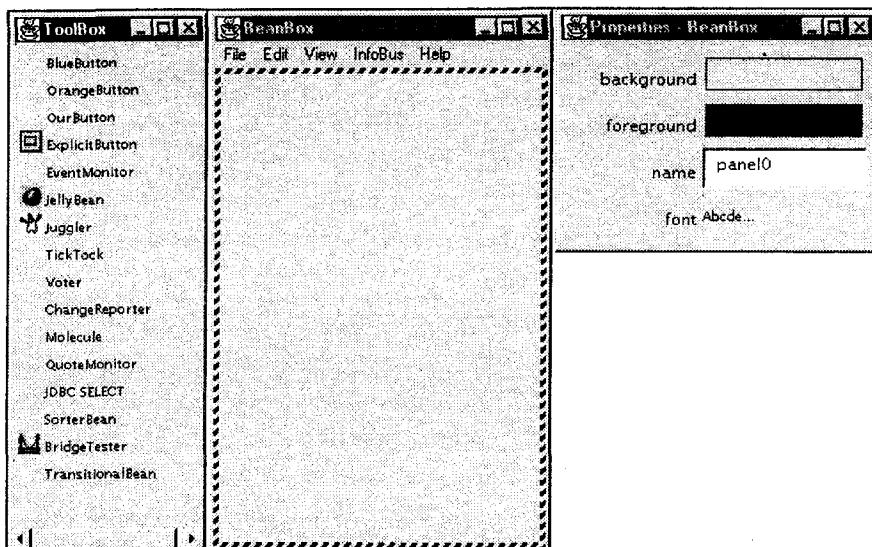


Рис. 25.1. Окна BDK

## Использование BDK

Этот раздел описывает, как создать приложение, используя некоторые Bean-компоненты из BDK. Во-первых, *Molecule Bean*<sup>1</sup> отображает трехмерное представление молекулы. Оно может быть сконфигурировано так, чтобы представить одну из следующих молекул: гиалуроновая кислота, бензол, бакминстерфуллерин, циклогексан, этан или вода. Этот компонент обладает методами, которые позволяют молекуле вращаться в пространстве вокруг оси X или Y декартовой плоскости. Во-вторых, *OurButton Bean* обеспечивает функциональные возможности кнопки. Мы будем иметь одну кнопку с меткой "Rotate X", чтобы вращать молекулу вокруг оси X, и другую — с меткой "Rotate Y" для вращения молекулы вокруг оси Y. Рис. 25.2 показывает, как это приложение выглядит.

### Создание и конфигурирование экземпляра *Molecule Bean*

Чтобы создать и конфигурировать экземпляр *Molecule Bean*, сделайте следующее:

1. Установите курсор на вход *ToolBox* и щелкните левой кнопкой мыши. Вы должны увидеть, что курсор становится перекрестием.

<sup>1</sup> Здесь и далее Bean-компоненты будут именоваться их типом из списка *ToolBox*. В этом случае слово *компонент* в словосочетании *Bean-компонент* будет опускаться. — Примеч. пер.

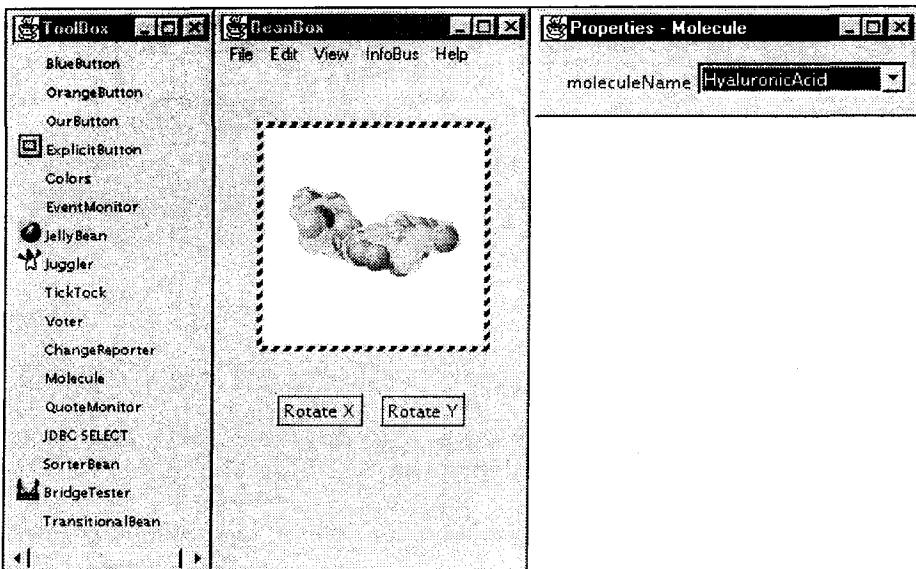


Рис. 25.2. Bean-компоненты Molecule и OurButton

2. Переместите курсор в область окна **BeanBox**, и щелкните левую кнопку мыши приблизительно в том месте, где вы хотите отобразить Bean-компонент. Вы должны увидеть появление прямоугольной области, которая содержит трехмерный вид молекулы. Область окружена заштрихованной границей, указывающей, что она в настоящее время выбрана.
3. Вы можете снова позиционировать Molecule Bean, устанавливая курсор над одной из заштрихованных границ и перетаскивая Bean-компонент.
4. Вы можете изменять отображенную молекулу, изменения выбор в окне **Properties**. Обратите внимание, что, когда вы изменяете свойства выбранной молекулы, Bean-отображение немедленно изменяется.

## Создание и конфигурирование экземпляра **OurButton Bean**

Чтобы создать и конфигурировать экземпляр **OurButton Bean** и присоединить его к **Molecule Bean**, выполните следующие действия:

1. Установите курсор на вход **OurButton** списка **ToolBox**, и щелкните левую кнопку мыши. Вы должны увидеть, что курсор изменяет свою форму на крестообразную.
2. Переместите курсор в область окна **BeanBox** и щелкните левую кнопку мыши приблизительно в той его части, где хотите отобразить Bean-компонент. Вы должны увидеть появление прямоугольной области, которая содержит кнопку. Эта область окружена заштрихованной границей, указывающей, что она в настоящее время выбрана.

3. Вы можете заново позиционировать OurButton Bean, устанавливая курсор над одной из заштрихованных границ и перетаскивая Bean-компонент.
4. Перейдите к окну **Properties** и измените метку Bean-компонента на "Rotate X". Когда это свойство изменяется, вид кнопки преобразуется немедленно.
5. Перейдите к строке меню окна **BeanBox** и выберите команду **Edit | Events | action | actionPerformed**. Теперь вы должны видеть строку, простирающуюся от кнопки до курсора. Обратите внимание, что один конец строки передвигается, когда передвигается курсор. Однако другой ее конец остается закрепленным на кнопке.
6. Передвиньте курсор так, чтобы он оказался внутри области показа Molecule Bean, и щелкните левой кнопкой мыши. Вы должны увидеть диалоговое окно **Event Target Dialog**.
7. Диалоговое окно позволяет вам выбирать метод, который должен быть вызван при нажатии этой кнопки. Выберите вход, маркированный как "rotateOnX", и нажмите кнопку **OK**. Вы должны увидеть окно сообщения, появляющееся на очень короткий момент и объявляющее, что инструмент "Generating and compiling adaptor class" (Генерация и компиляция класса адаптера).

Проверьте приложение. Каждый раз, когда вы нажимаете кнопку, молекула должна повернуться на несколько градусов вокруг одной из осей.

Теперь создайте другой экземпляр OurButton Bean. Маркируйте его "Rotate Y" и отобразите его action-событие на метод "rotateY" класса Molecule Bean. Эти шаги очень похожи на те, которые только что были описаны для кнопки с меткой "Rotate X".

Протестируйте приложение, нажимая эти кнопки и наблюдая, как молекула движется.

## JAR-файлы

Перед разработкой собственного Bean-компонента необходимо ознакомиться с понятием JAR-файлов (архивов Java), потому что инstrumentальные средства типа BDK предполагают упаковку Bean-компонентов в JAR-файлы. Файл JAR позволяет эффективно развертывать набор классов и связанных с ними ресурсов. Например, разработчик может построить мультимедийное приложение, которое использует различные звуковые и графические файлы. Набор Bean-компонентов может контролировать, как и когда представлять эту информацию. Все эти части могут быть помещены в один JAR-файл.

JAR-технология намного упрощает процесс поставки и установки программного обеспечения. Кроме того, элементы в JAR-файле сжаты, что делает загрузку файла подобного рода намного быстрее, чем раздельную за-

грузку нескольких несжатых файлов. С индивидуальными элементами в JAR-файле можно также связать цифровые подписи (сигнатуры). Они позволяют потребителю убедиться, что данные элементы были созданы определенной организацией или индивидуумом.

### Замечание

Классы, которые читают и записывают JAR-файлы, содержатся в пакете `java.util.zip`.

## Файлы описания

Чтобы указать, какие из компонентов в JAR-файле являются Java Bean-компонентами, разработчик должен обеспечить *файл описания* (manifest file). Пример файла описания показан в следующем листинге. Он определяет JAR-файл, содержащий четыре файла .gif и один файл .class. Последний вход — поле Java-Bean, содержащее значение типа Boolean, которое указывает, что данный файл содержит Java Bean-компонент.

```
Name: sunw/demo/slides/slides.class
Name: sunw/demo/slides/slidel.gif
Name: sunw/demo/slides/slidel.gif
Name: sunw/demo/slides/slidel.gif
Name: sunw/demo/slides/slidel.gif
Name: sunw/demo/slides/Slides.class
Java-Bean: True
```

Файл описания может ссылаться на несколько файлов с расширением .class. Если class-файл является Java Bean-компонентом, за его входом должна немедленно следовать строка "Java-Bean: True".

## Утилита jar

Утилита `jar` используется для генерации JAR-файла. Ее синтаксис выглядит так:

```
jar options files
```

где `options` — параметры команды, `files` — список (имен) файлов, включаемых в архив (разделенный пробелами, допустимы групповые имена).

Табл. 25.1 перечисляет возможные параметры и их значения. Следующие примеры показывают использование этой утилиты.

**Таблица 25.1. Параметры команды jar**

Параметр	Описание
c	Создать новый архив
C	Сменить каталоги во время выполнения команды

Таблица 25.1 (окончание)

Параметр	Описание
f	Первый элемент в списке файлов — имя архива, который должен быть создан или использован
m	Второй элемент в списке файлов — имя внешнего файла описания
M	Не создавать файл описания
t	Вывести содержимое указанного архива
u	Обновить существующий JAR-файл
v	Выводить сообщения о выполнении всех действий утилиты
x	Извлечь из архива файлы, указанные параметром <i>files</i> . (Если в списке <i>files</i> указан только один файл, который является именем архива, то из него извлекаются все файлы. Иначе, в первом элементе списка указывается имя архива, а в остальных — имена файлов, которые должны быть извлечены из архива.)
0	Не использовать сжатие

## Создание JAR-файла

Следующая команда создает JAR-файл с именем Xyz.jar, который содержит все файлы .class и .gif текущего каталога:

```
jar cf Xyz.jar *.class *.gif
```

Если существует файл описания, скажем Yxz.mf, его можно использовать следующей командой:

```
jar cfm Xyz.jar Yxz.mf *.class *.gif
```

## Вывод содержимого JAR-файла

Данная команда выводит список содержимого файла Xyz.jar:

```
jar tf Xyz.jar
```

## Извлечение файлов из JAR-файла

Следующая команда извлекает содержание Xyz.jar и помещает эти файлы в текущий каталог:

```
jar xf Xyz.jar
```

## Обновление существующего JAR-файла

Нижеуказанная команда добавляет файл file1.class в архив Xyz.jar:

```
jar -uf Xyz.jar file1.class
```

## Добавление файлов из подкаталогов

Следующая команда добавляет в Xyz.jar все файлы из подкаталога directoryX:

```
jar -uf Xyz.jar -C directoryX *
```

## Интроспекция

Интроспекция — это процесс анализа Bean-компонентта для определения его возможностей. Это существенное свойство Java Beans API, потому что оно позволяет инструменту построения приложений предоставить разработчику программ информацию о компоненте. Без интроспекции технология Java Beans не могла бы работать.

Существуют два способа, с помощью которых разработчик Bean-компонента может указывать, какие из его свойств, событий и методов должны быть показаны инструментом разработчика приложений. В первом способе используются простые соглашения об именах. Они позволяют механизмам интроспекции выводить информацию о Bean-компонентте. Во втором способе создается дополнительный класс, который явно поставляет эту информацию. Здесь рассматривается первый подход. Второй — описан позже.

Следующие разделы демонстрируют проектные шаблоны для свойств и событий, которые дают возможность определить функциональные возможности Bean-компонента.

## Проектные шаблоны для свойств

*Свойство* — это подмножество состояния Bean-компонента. Значения, назначенные свойствам, определяют поведение и внешний вид этого компонента. В данном разделе обсуждаются три типа свойств: простые, булевые и индексированные.

### Простые свойства

Простое свойство имеет одиночное значение. Оно может быть идентифицировано следующими *проектными шаблонами* (design patterns), где n — имя свойства, а t — его тип:

```
public T getN();  
public void setN(T arg);
```

Свойство *чтение/запись* (read/write) использует оба этих метода для доступа к своим значениям. Свойство *только для чтения* (read-only) использует только метод *get*, а свойство *только для записи* (write-only) — только метод *set*. Следующий листинг показывает класс, который имеет три простых свойства *чтение/запись*:

```

public class Box {
    private double depth, height, width;
    public double getDepth() {
        return depth;
    }
    public void setDepth(double d) {
        depth = d;
    }
    public double getHeight() {
        return height;
    }
    public void setHeight(double h) {
        height = h;
    }
    public double getWidth() {
        return width;
    }
    public void setWidth(double w) {
        width = w;
    }
}

```

## Булевые свойства

Булево свойство имеет значение true или false. Оно может быть идентифицировано следующими проектными шаблонами, где n — имя свойства:

```

public boolean isN();
public boolean getN();
public void setN(boolean value);

```

Чтобы извлечь значение булевого свойства, можно использовать первый или второй шаблон. Однако, если класс использует оба метода, применяется первый шаблон. Следующий листинг показывает класс, который имеет одно булево свойство:

```

public class Line {
    private boolean dotted = false;
    public boolean isDotted() {
        return dotted;
    }
    public void setDotted(boolean dotted) {
        this.dotted = dotted;
    }
}

```

## Индексированные свойства

Индексированное свойство состоит из множественных значений. Оно может быть идентифицировано следующими проектными шаблонами, где n — имя свойства, а t — его тип:

```
public T getN(int index), public void setN(int index, T value);
public T[ ]getN();
public void setN(T values[ ]);
```

Следующий пример программы показывает класс, который имеет одно индексированное свойство *чтение/запись*:

```
public class PieChart {
    private double data[ ];
    public double getData(int index) {
        return data[index];
    }
    public void setData(int index, double value) {
        data[index] = value;
    }
    public double[ ] getData() {
        return data;
    }
    public void setData(double[ ] values) {
        data = new double[values.length];
        System.arraycopy(values, 0, data, 0, values.length);
    }
}
```

## Проектные шаблоны для событий

Bean-компоненты используют модель делегирования событий, которая была обсуждена в главе 20. Они могут генерировать события и посыпать их другим объектам. События можно идентифицировать следующими проектными шаблонами, где *T* — тип события:

```
public void addTListener(TListener eventListener);
public void addTListener(TListener eventListener) throws TooManyListeners;
public void removeTListener(TListener eventListener);
```

Перечисленные методы используются блоками прослушивания событий, чтобы регистрировать интерес в событиях определенного типа. Первый шаблон указывает, что Bean-компонент может распространять событие многим слушателям. Второй шаблон указывает, что Bean-компонент может передавать событие только одному слушателю. Третий шаблон используется блоком прослушивания, когда он больше не желает принимать определенный тип уведомлений о событиях от Bean-компонента.

Представленный ниже фрагмент программы демонстрирует класс, который уведомляет другие объекты, когда температурное значение выходит из определенного диапазона. Два метода, обозначенные здесь, позволяют другим объектам, которые реализуют интерфейс *TemperatureListener*, принимать уведомления о событиях, когда они происходят.

```

public class Thermometer {
    public void addTemperatureListener(TemperatureListener t1) {
        ...
    }
    public void removeTemperatureListener(TemperatureListener t1) {
        ...
    }
}

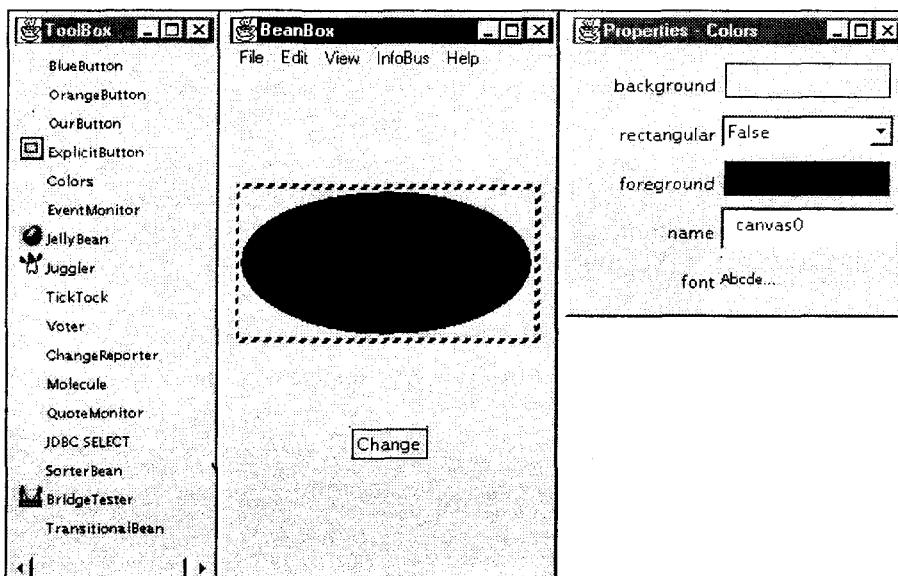
```

## Методы

Для именования неподходящих методов проектные шаблоны не используются. Механизм интроспекции находит все public-методы Bean-компонентта. Protected- и private-методы он не предоставляет.

## Разработка простого Bean-компонента

В этом разделе представлен пример, который показывает процедуру разработки простого Bean-компонента и соединения его с другими компонентами через BDK.



**Рис. 25.3.** Bean-компоненты Colors и OurButton

Новый компонент называется Colors Bean. Он появляется на экране либо как прямоугольник, либо как эллипс, который заполнен цветом. Цвет выби-

рается случайно, когда Bean-компонент начинает выполнение. Для его изменения можно вызвать общий метод. Каждый раз, когда выполняется щелчок мышью по Bean-компоненту, выбирается другой случайный цвет. Существует одно булевское свойство **чтение/запись**, которое определяет форму компонента.

Чтобы разместить в приложении один экземпляр Colors Bean и один экземпляр OurButton Bean, используется BDK. На кнопке имеется надпись **Change** (Изменить). Каждый раз, когда ее нажимают, цвет изменяется.

Рис. 25.3 показывает, как это приложение выглядит.

## Создание нового Bean-компонента

Алгоритм создания нового Bean-компонента выглядит так:

1. Создать каталог для нового Bean-компонента.
2. Создать исходный Java-файл (или несколько файлов).
3. Компилировать исходные файлы.
4. Создать файл описания.
5. Генерировать JAR-файл.
6. Запустить BDK.
7. Протестировать программу.

Следующие разделы обсуждают каждый из перечисленных шагов этого алгоритма в деталях.

## Создание каталога для нового Bean-компонента

Создайте каталог c:\bdk\demo\sunw\demo\colors и перейдите в него (командой cd (change directory), например).

## Создание исходного файла для нового Bean-компонента

Исходный код (текст) для Colors-компонента показан в следующем листинге (он хранится в файле Colors.java):

```
// Простой Bean-компонент.
package sunw.demo.colors;
import java.awt.*;
import java.awt.event.*;
public class Colors extends Canvas {
    transient private Color color;
    private boolean rectangular;
    public Colors() {
        addMouseListener(new MouseAdapter() {
```

```
public void mousePressed(MouseEvent me) {
    change();
}
});
rectangular = false;
setSize(200, 100);
change();
}
public boolean getRectangular() {
    return rectangular;
}
public void setRectangular(boolean flag) {
    this.rectangular = flag;
    repaint();
}
public void change() {
    color = randomColor();
    repaint();
}
private Color randomColor() {
    int r = (int)(255*Math.random());
    int g = (int)(255*Math.random());
    int b = (int)(255*Math.random());
    return new Color(r, g, b);
}
public void paint(Graphics g) {
    Dimension d = getSize();
    int h = d.height;
    int w = d.width;
    g.setColor(color);
    if(rectangular) {
        g.fillRect(0, 0, w-1, h-1);
    }
    else {
        g.fillOval(0, 0, w-1, h-1);
    }
}
```

Инструкция package в начале файла помещает его в пакет с именем sunw.demo.colors. Вспомните (см. гл. 9), что иерархия каталогов должна соответствовать иерархии пакетов. Поэтому этот файл нужно разместить в подкаталоге с именем sunw\demo\colors (не забудьте о соответствующей установке переменной среды CLASSPATH).

Цвет компонента определен частной Color-переменной color, а его форма определена частной boolean переменной rectangular.

Конструктор определяет анонимный внутренний класс, который расширяет `MouseAdapter` и переопределяет его метод `mousePressed()`. В ответ на нажатие кнопки мыши вызывается метод `change()`. Компонент инициализируется в форме прямоугольника 200×100 пикселов. Чтобы выбрать случайный цвет и перерисовать компонент, вызывается метод `change()`.

Методы `getRectangular()` и `setRectangular()` обеспечивают доступ к одному свойству этого Bean-компонентта. Метод `change()` вызывает `randomColor()`, чтобы выбрать цвет, и затем вызывает `repaint()`, чтобы сделать изменение видимым. Обратите внимание, что метод `paint()` использует переменные `rectangular` и `color` для определения внешнего вида Bean-компонента.

## Компиляция исходного кода для нового Bean-компонента

Откомпилируйте исходный код, чтобы создать файл `.class`. Для этого введите команду:

```
javac Colors.java
```

## Создание файла описания

Теперь нужно создать файл описания (*manifest file*). Сначала переключитесь в каталог `c:\bdk\demo`. Это каталог, в котором размещаются файлы описания для BDK-демонстраций. Поместите исходный код для вашего файла-описания в файл `colors.mft`. Это делается так:

```
Name: sunw/demo/colors/Colors.class  
Java-Bean: True
```

Данный файл указывает, что в JAR-файле имеется один class-файл, и что он является Java Bean-компонентом. Обратите внимание, что файл `Colors.class` находится в пакете `sunw.demo.colors` и в подкаталоге `sunw\demo\colors` (относительно текущего каталога).

## Генерация JAR-файла

Beans-компоненты включаются в окно **ToolBox** BDK, только если они находятся в JAR-файлах каталога `c:\bdk\jars`. Эти файлы генерируются утилитой `jar`. Введите следующую команду:

```
jar cfm ..\jars\colors.jar colors.mft sunw\demo\colors\*.class
```

Данная команда создает файл `colors.jar` и размещает это в каталог `c:\bdk\jars`. (Можно, при желании, поместить его в пакетный файл для будущего использования.)

## Запуск BDK

Перейдите в каталог `c:\bdk\beanbox` и введите команду `run`. Это приведет к запуску BDK. Вы должны увидеть три окна, озаглавленные как **ToolBox**,

**BeanBox и Properties.** Список окна ToolBox должен содержать вход с надписью "Colors" для нового Bean-компонентта.

## Создание экземпляра *Colors Bean*

После завершения предшествующих шагов создаете экземпляр Colors Bean в окне **BeanBox**. Проверьте ваш новый компонент, нажимая кнопку мыши где-нибудь в пределах его границ. Его цвет должен немедленно измениться. Используйте окно **Properties**, для изменения свойства `rectangular` с `false` на `true`. Его форма немедленно изменяется.

## Создание и конфигурация экземпляра *OurButton Bean*

Создайте экземпляр OurButton Bean в окне **BeanBox**. Затем выполните следующие действия:

1. Перейдите в окно **Properties**, и измените метку Bean-компонента на "Change". Когда это свойство модифицировано, вид кнопки должен немедленно измениться.
2. Перейдите к строке меню окна **BeanBox** и выберите команду **Edit | Events | action | actionPerformed**.
3. Переместите курсор так, чтобы он оказался внутри области показа Colors Bean, и щелкните левой кнопкой мыши. Вы должны видеть диалоговое окно **Event Target Dialog**.
4. Это диалоговое окно позволяет выбрать метод, который должен быть вызван при нажатии данной кнопки. Выберите вход, помеченный как "change", и нажмите кнопку **OK**. Вы должны увидеть окно сообщения, появляющееся на очень короткий момент и объявляющее, что инструмент "Generating and compiling adaptor class" (Генерация и компиляция класса адаптера).
5. Нажмите кнопку **Change**. Вы должны увидеть изменение цвета.

Перед переходом на кнопку можно немного поэкспериментировать с Colors Bean.

## Использование связанных свойств

Bean-компонент, который имеет связанное свойство , генерирует событие, когда свойство изменяется. Событие имеет тип `PropertyChangeEvent` и посыпается к объектам, которые предварительно зарегистрировали интерес в приеме таких уведомлений.

Компонент TickTock Bean поставляется с BDK. Он генерирует событие изменения свойства каждые N секунд. Интервал N — это тоже свойство Bean-компонента, которое может быть изменено через окно **Properties** BDK. Сле-

дующий пример показывает приложение, которое использует TickTock Bean, чтобы автоматически управлять компонентом Colors Bean. Рис. 25.4 показывает, как это приложение выглядит.

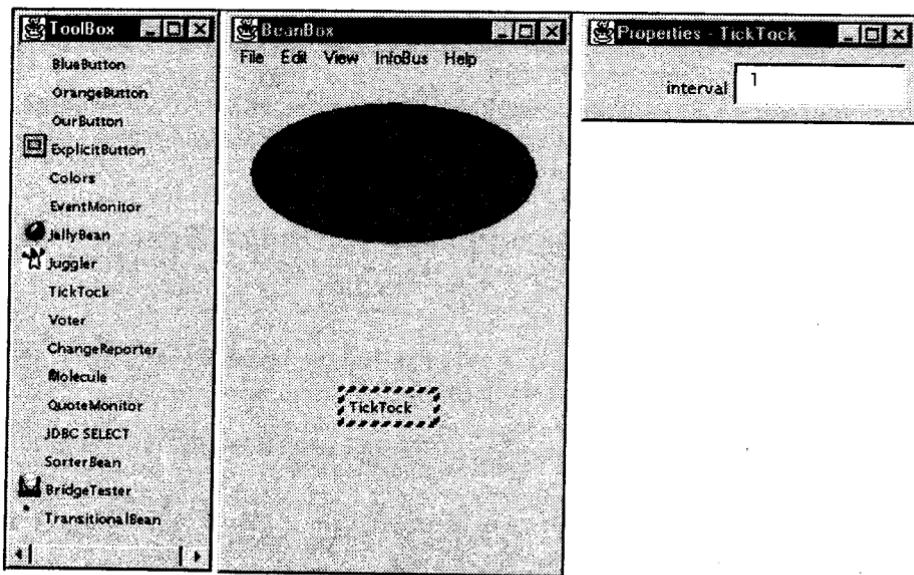


Рис. 25.4. Bean-компоненты Colors и TickTock

## Алгоритм

Для этого примера запустите BDK и создайте экземпляр Colors Bean в окне BeanBox.

Создайте экземпляр TickTock Bean. Окно Properties должно показать одно свойство для этого компонента — interval, а его начальное значение равно 5. Указанное свойство задает количество секунд, которые протекают между событиями изменения свойств, генерируемыми компонентом TickTock Bean. Измените это значение на 1.

Теперь нужно отобразить события, сгенерированные компонентом TickTock Bean, в вызовы методов из Colors Bean. Для этого выполните следующие действия:

- Перейдите к строке меню BeanBox и выберите команду **Edit | Events | propertyChange | propertyChange**. Вы должны увидеть строку, простирающуюся от кнопки до курсора.
- Переместите курсор так, чтобы он оказался внутри области показа Colors Bean, и щелкните левой кнопкой мыши. Вы должны увидеть диалоговое окно **Event Target Dialog**.

3. Данное диалоговое окно позволяет выбрать метод, который должен быть вызван при наступлении этого события. Выберите вход "change", и нажмите кнопку **OK**. Вы должны увидеть окно сообщения, появляющееся на очень короткий момент и объявляющее, что инструмент "Generating and compiling adaptor class" (Генерирует и компилирует класс адаптера).

Вы должны теперь видеть, что цвета вашего компонента изменяются каждую секунду.

## Использование интерфейса *BeanInfo*

В предыдущих примерах для определения информации, предназначенной пользователю Bean-компонентта, использовались проектные шаблоны. Текущий раздел описывает, как разработчик может использовать интерфейс *BeanInfo* для явного управления этим процессом.

Этот интерфейс определяет несколько методов, включая следующие:

```
PropertyDescriptor[ ] getPropertyDescriptors()
EventSetDescriptor[ ] getEventSetDescriptors()
MethodDescriptor[ ] getMethodDescriptors()
```

Они возвращают массивы объектов, которые обеспечивают информацию о свойствах, событиях и методах Bean-компонента. Реализуя эти методы, разработчик может точно определить, что предоставляется пользователю.

*SimpleBeanInfo* — это класс, который обеспечивает заданные по умолчанию реализации интерфейса *BeanInfo*, включая только что перечисленные три метода. Вы можете расширить этот класс и переопределить один из них (или несколько). Следующий листинг демонстрирует, как это сделано для *Colors Bean*, который был разработан ранее. *ColorsBeanInfo* — это подкласс *SimpleBeanInfo*. Он переопределяет метод *getPropertyDescriptors()*, чтобы указать, какие свойства предоставляются пользователю Bean-компонента. Этот метод создает объект *PropertyDescriptor* для свойства *rectangular*. Используется конструктор класса *PropertyDescriptor* в следующей форме:

```
PropertyDescriptor(String property, Class beanCls)
    throws IntrospectionException
```

Здесь первый параметр — название свойства, а второй — класс Bean-компонента.

```
// Класс Bean-информации.
package sunw.demo.colors;
import java.beans.*;
public class ColorsBeanInfo extends SimpleBeanInfo {
    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
```

```

PropertyDescriptor rectangular = new
    PropertyDescriptor("rectangular", Colors.class);
PropertyDescriptor pd[] = {rectangular};
return pd;
}
catch(Exception e) {
}
return null;
}
}

```

Вы должны компилировать этот файл из каталога BDK\demo или устанавливать переменную среды CLASSPATH так, чтобы она включила c:\bdk\demo. Если вы не сделаете этого, компилятор не найдет файл Colors.class. После успешной компиляции этого файла следует модифицировать файл colors.mft так:

```

Name: sunw/demo/colors/ColorsBeanInfo.class
Name: sunw/demo/colors/Colors.class
Java-Bean: True

```

Используйте JAR-программу, чтобы создать новый файл colors.jar. Перезапустите BDK и создайте экземпляр Colors Bean в окне BeanBox.

Для поиска класса BeanInfo используются средства интроспекции. Если он существует, его поведение явно определяет информацию, которая предоставляется пользователю Bean-компонентта. Иначе, чтобы вывести эту информацию, используются проектные шаблоны.

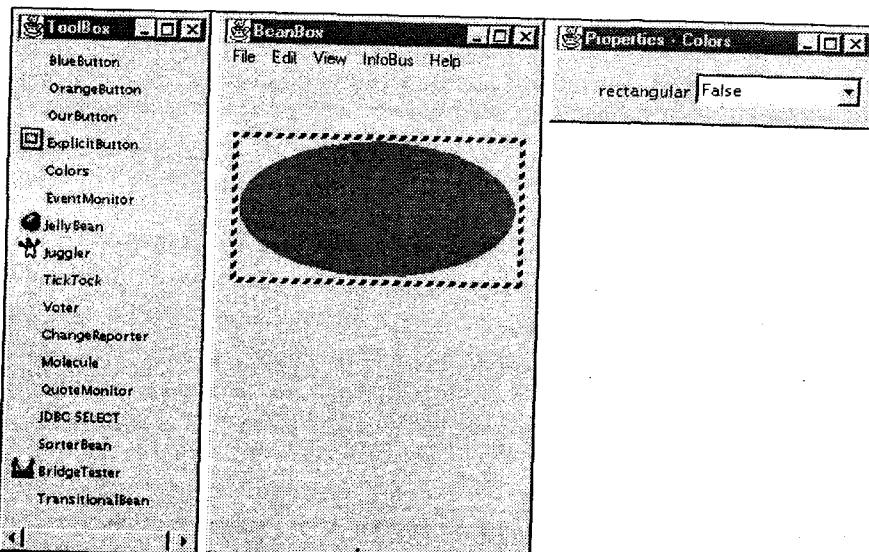


Рис. 25.5. Использование класса ColorsBeanInfo

Рис. 25.5 показывает, как теперь выглядит окно **Properties**. Сравните его с рис. 24.3. Можно видеть, что свойства, наследованные от класса Component, больше не предоставляются для Colors Bean. Появляется только свойство `rectangular`.

## Ограниченные свойства

Bean-компонент, который имеет *ограниченное* (constrained) свойство, генерирует событие, когда осуществляется попытка изменить значение этого свойства. Событие имеет тип `PropertyChangeEvent`. Оно посыпается объектам, которые предварительно зарегистрировали интерес в приеме таких уведомлений. Данные объекты обладают способностью отклонять предложенное изменение. Эта возможность позволяет Bean-компоненту работать по-разному, согласуясь со средой времени выполнения. Полное обсуждение ограниченных свойств выходит за рамки этой книги.

## Сохраняемость

*Сохраняемость* (persistance) — это способность сохранять Bean-компонент в энергонезависимой памяти и восстанавливать его в более позднее время. Особенно важно запоминать параметры конфигурации.

Посмотрим сначала, как BDK позволяет сохранять набор Bean-компонентов, которые были сконфигурированы и соединены вместе для формирования приложения. Вспомним предыдущий пример, включающий Bean-компоненты Colors и TickTock. Свойство `rectangular` в Colors Bean было изменено на `true`, а свойство `interval` в TickTock Bean — на одну секунду.

Чтобы сохранить приложение, перейдите к строке меню окна **BeanBox**, и выберите команду **File | Save**. Должно появиться диалоговое окно, позволяющее определить имя файла, в котором Bean-компоненты и их параметры конфигурации должны быть сохранены. Укажите имя файла и нажмите кнопку **OK** в этом диалоговом окне. Выйдите из BDK.

Снова запустите BDK. Чтобы восстановить приложение, перейдите к строке меню окна **BeanBox**, и выберите команду **File | Load**. Должно появиться диалоговое окно, позволяющее определить имя файла, из которого приложение должно быть восстановлено. Укажите имя файла, в котором приложение было сохранено, и нажмите кнопку **OK**. Ваше приложение должно теперь функционировать. Убедитесь, что свойство `rectangular` в Colors Bean установлено как `true` и что свойство `interval` для TickTock Bean равно одной секунде.

Чтобы обеспечить сохраняемость для Bean-компонента, используются возможности сериализации объектов, предоставленные библиотеками Java-клас-

сов. Если Bean-компонент наследуется непосредственно или косвенно от `java.awt.Component`, он сериализуется автоматически, поскольку этот класс реализует интерфейс `java.io.Serializable`. Если Bean-компонент не наследует реализацию интерфейса `Serializable`, вы должны обеспечить ее самостоятельно, иначе контейнеры не могут сохранять конфигурацию вашего компонента.

Чтобы обозначить члены данных Bean-компонента, которые не должны быть сериализованы, можно использовать ключевое слово `transient`. Примером такого элемента является `color`-переменная класса `Colors`.

## Конфигураторы

Разработчик может изменять свойства Bean-компонента в окне **Properties BDK**. Однако, это не лучший интерфейс пользователя для комплексного компонента с большим числом взаимодействующих свойств. Поэтому Bean-разработчик может создавать специализированный **конфигуратор** (*customizer*), который помогает другому разработчику конфигурировать данное программное обеспечение. Конфигуратор может обеспечить пошаговое руководство процессом, которому нужно следовать, чтобы использовать компонент в определенном контексте. Он может также обеспечить и оперативную (*online*) документацию. Разработчик должен обеспечить большую гибкость Bean-компонентов, чтобы разработать конфигуратор, который может дифференцировать его продукт в рыночном пространстве.

## Java Beans API

Функциональные возможности Java Bean-компонентов обеспечиваются набором классов и интерфейсов в пакете `java.beans`. Этот раздел дает краткий обзор его содержимого. В табл. 25.2 перечислены интерфейсы `java.beans` и приводится краткое описание их функциональных возможностей. В табл. 25.3 также кратко описываются классы `java.beans`.

**Таблица 25.2. Интерфейсы пакета `java.beans`**

Интерфейс	Описание
<code>AppletInitializer</code>	Методы этого интерфейса используются для инициализации Bean-компонентов, которые являются также и апллетами
<code>BeanInfo</code>	Этот интерфейс позволяет проектировщику определять информацию относительно свойств, событий и методов Bean-компонентов

Таблица 25.2 (окончание)

Интерфейс	Описание
Customizer	Этот интерфейс позволяет проектировщику обеспечивать графический интерфейс пользователя, через который Bean-компонент может быть конфигурирован
DesignMode	Методы этого интерфейса определяют, выполняется ли Bean-компонент в режиме проектирования
PropertyChangeListener	Метод этого интерфейса вызывается, когда изменено связанное свойство
PropertyEditor	Объекты, которые реализуют этот интерфейс, позволяют проектировщикам изменять и отображать значения свойств
VetoableChangeListener	Метод этого интерфейса вызывается, когда изменено ограниченное свойство
Visibility	Методы этого интерфейса позволяют Bean-компоненту выполнятся в средах, где графический интерфейс пользователя недоступен

Таблица 25.3. Классы пакета `java.beans`

Класс	Описание
BeanDescriptor	Этот класс обеспечивает информацию о Bean-компоненте. Он также позволяет связывать конфигуратор с Bean-компонентом
Beans	Этот класс используется для получения информации о Bean-компоненте
EventSetDescriptor	Экземпляры этого класса описывают событие, которое может быть сгенерировано Bean-компонентом
FeatureDescriptor	Это суперкласс классов PropertyDescriptor, EventSetDescriptor и MethodDescriptor
IndexedPropertyDescriptor	Экземпляры этого класса описывают индексированное свойство Bean-компонента
IntrospectionException	Исключение этого типа генерируется, если возникает проблема при анализе Bean-компонента
Introspector	Этот класс анализирует Bean-компонент и создает BeanInfo-объект, который описывает компонент
MethodDescriptor	Экземпляры этого класса описывают метод Bean-компонента

Таблица 25.3 (окончание)

Класс	Описание
ParameterDescriptor	Экземпляры этого класса описывают параметр метода
PropertyChangeEvent	Это событие генерируется, когда изменены связанные или ограниченные свойства. Оно посыпается объектам, которые зарегистрировали интерес в данных событиях и реализуют интерфейсы VetoableChangeListener или PropertyChangeListener
PropertyChangeSupport	Bean-компоненты, которые поддерживают связанные свойства, могут использовать этот класс, чтобы уведомлять объекты PropertyChangeListener
PropertyDescriptor	Экземпляры этого класса описывают свойство Bean-компонента
PropertyEditorManager	Этот класс размещает объект PropertyEditor для заданного типа свойств
PropertyEditorSupport	Этот класс обеспечивает функциональные возможности, которые могут использоваться при написании редакторов свойства
PropertyVetoException	Исключение этого типа генерируется, если на изменение ограниченного свойства накладывают вето
SimpleBeanInfo	Этот класс обеспечивает функциональные возможности, которые могут использоваться при написании классов BeanInfo
VetoableChangeSupport	Bean-компоненты, которые поддерживают ограниченные свойства, могут использовать этот класс, чтобы уведомлять объекты VetoableChangeListener

Полное обсуждение перечисленных классов и интерфейсов — вне рамок этой книги. Однако следующая программа иллюстрирует классы Introspector, BeanDescriptor, PropertyDescriptor и EventSetDescriptor и интерфейс BeanInfo. Она выводит свойства и события компонента Colors Bean, который был разработан ранее в текущей главе.

```
// Показывает свойства и события компонента.
package sunw.demo.colors;
import java.awt.*;
import java.beans.*;
public class IntrospectorDemo {
```

```
public static void main(String args[]) {  
    try {  
        Class c = Class.forName("sunw.demo.colors.Colors");  
        BeanInfo beanInfo = Introspector.getBeanInfo(c);  
        BeanDescriptor beanDescriptor = beanInfo.getBeanDescriptor();  
  
        System.out.println("Bean name = " + beanDescriptor.getName());  
  
        System.out.println("Properties:");  
        PropertyDescriptor[] propertyDescriptor =  
        beanInfo.getPropertyDescriptors();  
        for(int i = 0; i < propertyDescriptor.length; i++) {  
            System.out.println("\t" + propertyDescriptor[i].getName());  
        }  
  
        System.out.println("Events:");  
        EventSetDescriptor[] eventSetDescriptor =  
        beanInfo.getEventSetDescriptors();  
        for(int i = 0; i < eventSetDescriptor.length; i++) {  
            System.out.println("\t" + eventSetDescriptor[i].getName());  
        }  
    } catch(Exception e) {  
        System.out.println("Exception caught. " + e);  
    }  
}
```

Вывод этой программы:

```
Bean name = Colors  
Properties:  
    rectangular  
Events:  
    propertyChange  
    mouseMotion  
    focus  
    mouse  
    inputMethod  
    key  
    component
```

## Будущее Bean-технологии

Технология Java Beans находится на острие Java-программирования, а создание компонентного программного обеспечения будет важной частью большинства работ Java-программистов в ближайшем будущем. Кроме того,

Bean-компоненты формируют дополнение к ActiveX — архитектуре программных компонентов фирмы Microsoft. Программы типа Internet Explorer, Microsoft Office и Visual Basic могут служить контейнерами для этих компонентов. С сайта [java.sun.com](http://java.sun.com) можно загрузить инструментальную систему Bridge for ActiveX (мост для ActiveX). Данный инструмент позволяет использовать Java Bean-компоненты в контейнерах ActiveX. С этого же сайта можно загрузить и другой инструмент — Java Beans Migration Assistant for ActiveX. Эта программа анализирует элементы управления ActiveX и генерирует скелет Java Bean-компонента.



## ГЛАВА 26

# Система Swing

В Части II вы познакомились с возможностью построения интерфейсов пользователя с помощью AWT-классов. Здесь, мы сделаем обзор очень важной альтернативной технологии, получившей название *Swing-технология*. Swing API — это набор классов, который обеспечивает более мощные и гибкие компоненты, чем AWT. В дополнение к знакомым компонентам типа кнопок, флажков и меток Swing поставляет несколько интересных добавлений, включая панели со вкладками, панели с прокруткой, деревья и таблицы. Даже знакомые компоненты, такие как кнопки, имеют в Swing больше возможностей. Например, с кнопкой можно связать как изображение, так и текстовую строку. Кроме того, изображение может изменяться, когда изменяется состояние кнопки.

В отличие от AWT-компонентов, Swing-компоненты не реализованы специфическим для платформы кодом. Вместо этого они написаны полностью на Java и, поэтому, платформно-независимы. Для описания таких элементов используется термин *облегченный* (lightweight).

Число классов и интерфейсов в пакетах Swing достаточно велико, так что в текущей главе приводится краткий обзор только некоторых из них.

В табл. 26.1 показаны классы Swing-компонентов, которые используются в этой книге.

**Таблица 26.1. Классы Swing-компонентов**

Класс	Описание
AbstractButton	Абстрактный суперкласс для кнопок Swing
ButtonGroup	Инкапсулирует взаимоисключающий набор кнопок
ImageIcon	Инкапсулирует значок

Таблица 26.1 (окончание)

Класс	Описание
JApplet	Swing-версия класса Applet
JButton	Класс Swing-кнопок
JCheckBox	Класс Swing-флажков
JComboBox	Инкапсулирует combo box (комбинация раскрывающегося списка и текстового поля)
JLabel	Swing-версия метки
JRadioButton	Swing-версия переключателей
JScrollPane	Инкапсулирует прокручиваемую панель
JTabbedPane	Инкапсулирует панели с вкладками
JTable	Инкапсулирует таблицы или сетки
JTextField	Swing-версия текстового поля
JTree	Инкапсулирует деревья

Относящиеся к Swing классы содержатся в пакете javax.swing и его подпакетах, таких как javax.swing.tree<sup>1</sup>. Существует много других Swing-классов и интерфейсов, которые в данной главе не рассматриваются. Здесь мы разберем лишь некоторые Swing-компоненты и проиллюстрируем их на примерах апплетов.

## Класс JApplet

Фундаментальным для Swing является класс JApplet, который расширяет класс Applet. Апплеты, которые используют Swing-компоненты, должны быть подклассами JApplet. JApplet богат функциональными возможностями, которых нет в Applet. Например, JApplet поддерживает различные "панели", такие как панель содержания (content pane), прозрачная ("стеклянная") панель (glass pane) и корневая панель (root pane). В примерах этой главы мы не будем пользоваться большинством расширенных свойств JApplet. Однако одно различие между Applet и JApplet важно обсудить, потому что оно используется примерами апплетов текущей главы. При добавлении компонента к экземпляру JApplet не вызывайте метод add() для апплета. Вместо этого, вызовите add() для панели содержания JApplet-объекта. Панель содержания может быть получена с помощью следующего метода:

```
Container getContentPane()
```

<sup>1</sup> Полный перечень пакетов системы Swing можно найти в табл. 24.1 (глава 24). — Примеч. пер.

Чтобы добавить компонент в панель содержания, можно использовать метод `add()` класса `Container`. Его форма:

```
void add(comp)
```

где `comp` — компонент, который будет добавлен к панели содержания.

## Значки и метки

В Swing значки инкапсулированы классом `ImageIcon`, который рисует значок из изображения. Ниже показаны два его конструктора:

```
ImageIcon(String filename)
ImageIcon(URL url)
```

Первая форма использует изображение в файле с именем `filename`, а вторая форма — в ресурсе, расположенному по URL-адресу `url`.

Класс `ImageIcon` реализует интерфейс `Icon`, который объявляет методы, представленные в табл. 26.2.

**Таблица 26.2. Методы класса `ImageIcon`**

Метод	Описание
<code>int getIconHeight()</code>	Возвращает высоту значка в пикселях
<code>int getIconWidth()</code>	Возвращает ширину значка в пикселях
<code>void paintIcon(Component <i>comp</i>, Graphics <i>g</i>, int <i>x</i>, int <i>y</i>)</code>	Рисует значок в позиции ( <i>x</i> , <i>y</i> ) с графическим контекстом <i>g</i> . Дополнительная информация об операции рисования обеспечена в <i>comp</i>

Метки Swing — экземпляры класса `JLabel`, который расширяет `JComponent`. Он может отображать тексты и/или значки. Вот некоторые из его конструкторов:

```
JLabel(Icon i)
JLabel(String s)
JLabel(String s, Icon i, int align)
```

Здесь `s` и `i` — текст и значок, используемый для метки. Параметр `align` определяет выравнивание и имеет значения `LEFT`, `RIGHT` или `CENTER`. Эти константы определены в интерфейсе `SwingConstants`, наряду с несколькими другими, используемыми Swing-классами.

Значок и текст, связанный с меткой, можно считывать и записывать следующими методами:

```
Icon getIcon()
String getText()
void setIcon(Icon i)
void setText(String s)
```

Здесь *i* и *s* — значок и текст, соответственно.

Следующий пример показывает, как можно создать и отобразить метку, содержащую как значок, так и строку. Апплет начинается с получения панели содержания. Затем, создается объект ImageIcon для файла france.gif. Он используется как второй параметр конструктора JLabel. Первый и последний параметры для конструктора JLabel — текст метки и выравнивание. Наконец, метка добавляется к панели содержания.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JLabelDemo" width=250 height=150>
</applet>
*/
public class JLabelDemo extends JApplet {
    public void init() {
        // получить панель содержания
        Container contentPane = getContentPane();
        // создать значок
        ImageIcon ii = new ImageIcon("france.gif");
        // создать метку
        JLabel jl = new JLabel("France", ii, JLabel.CENTER);
        // добавить метку к панели содержания
        contentPane.add(jl);
    }
}
```

Вывод этого апплета представлен на рис. 26.1.

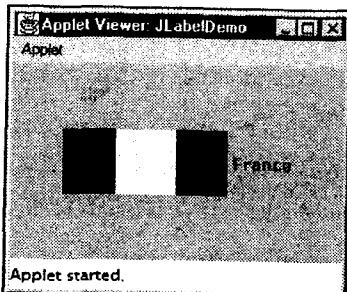


Рис. 26.1. Окно апплета JLabelDemo

## Текстовые поля

Поле текста Swing инкапсулировано классом JTextField, который расширяет JComponent. Он обеспечивает функциональные возможности, которые являются общими для текстовых Swing-компонентов. Один из его подклассов — JTextField, позволяет редактировать одну строку текста. Вот некоторые из его конструкторов:

```
JTextField()
JTextField(int cols)
JTextField(String s, int cols)
JTextField(String s)
```

Здесь *s* — строка, которая будет представлена; *cols* — число позиций в текстовом поле.

Следующий пример показывает, как можно создать текстовое поле. Апплет начинается с получения его панели содержания и затем для нее устанавливается поточное размещение в качестве менеджера компоновки. Далее, создается объект JTextField и добавляется к панели содержания.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300 height=50>
</applet>
*/
public class JTextFieldDemo extends JApplet {
    JTextField jtf;
    public void init() {
        // получить панель содержания
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        // добавить текстовое поле к панели содержания
        jtf = new JTextField(15);
        contentPane.add(jtf);
    }
}
```

Вывод этого апплета представлен на рис. 26.2.

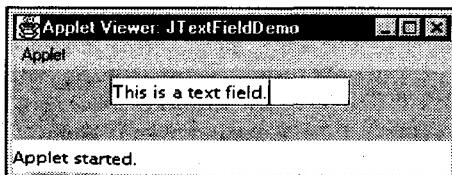


Рис. 26.2. Okno апплета JTextFieldDemo

## Кнопки

Кнопки Swing обладают свойствами, которых нельзя найти в классе Button, определенном в AWT. Например, с кнопкой Swing можно связать изображение. Кнопки Swing — это подклассы класса AbstractButton, который расширяет JComponent. AbstractButton содержит много методов, которые позволяют управлять поведением кнопок, флагков и переключателей. Например, можно определять различные пиктограммы для отображения компонента, когда он отжат (disabled), нажат (pressed), или выбран (selected). Некоторую пиктограмму можно использовать как значок "наезда" (rollover), который отображается, когда курсор мыши установлен поверх этого компонента ("наехал" на него). Ниже следуют описания форматов методов, которые управляют этим поведением:

```
void setDisabledIcon(Icon di)
void setPressedIcon(Icon pi)
void setSelectedIcon(Icon si)
void setRolloverIcon(Icon ri)
```

Здесь *di*, *pi*, *si* и *ri* — пиктограммы, которые нужно использовать для этих различных состояний.

Текст, связанный с кнопкой, можно читать и записывать с помощью следующих методов:

```
String getText()
void setText(String s)
```

Здесь *s* — текст, который нужно связать с кнопкой.

При нажатии кнопки конкретные подклассы AbstractButton генерируют action-события. Блоки прослушивания регистрируют и отменяют регистрацию для этих событий с помощью следующих методов:

```
void addActionListener(ActionListener al)
void removeActionListener(ActionListener al)
```

Здесь *al* — блок прослушивания событий действия.

AbstractButton — это суперкласс для кнопок, флагков и переключателей. Рассмотрим каждый из них.

## Класс JButton

Класс JButton обеспечивает функциональные возможности кнопки. JButton позволяет связать с кнопкой изображение, строку или и то и другое. Некоторые из его конструкторов:

```
JButton(Icon i)
JButton(String s)
JButton(String s, Icon i)
```

Здесь *s* и *i* — строка и изображение, используемые для кнопки.

Следующий пример демонстрирует четыре кнопки и текстовое поле. Каждая кнопка отображает пиктограмму, которая представляет флагок страны. Когда кнопка нажимается, в текстовом поле выводится название этой страны. Апплет начинается с получения панели содержания и установки для нее менеджера компоновки. Создаются четыре кнопки-изображения и добавляются к панели содержания. Затем апплет регистрируется, чтобы принимать генерируемые кнопками action-события. Далее, создается текстовое поле и добавляется к апплету. Наконец, обработчик action-событий отображает командную строку, которая связана с кнопкой. Для представления этой строки используется текстовое поле.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JButtonDemo" width=250 height=300>
</applet>
*/
public class JButtonDemo extends JApplet
implements ActionListener {
    JTextField jtf;
    public void init() {
        // Получить панель содержания
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        // Добавить кнопки в панель содержания
        ImageIcon france = new ImageIcon("france.gif");
        JButton jb = new JButton(france);
        jb.setActionCommand("France");
        jb.addActionListener(this);
        contentPane.add(jb);
        ImageIcon germany = new ImageIcon("germany.gif");
        jb = new JButton(germany);
        jb.setActionCommand("Germany");
        jb.addActionListener(this);
        contentPane.add(jb);
        ImageIcon italy = new ImageIcon("italy.gif");
        jb = new JButton(italy);
        jb.setActionCommand("Italy");
        jb.addActionListener(this);
        contentPane.add(jb);
    }
}
```

```

jb.setActionCommand("Italy");
jb.addActionListener(this);
contentPane.add(jb);

ImageIcon japan = new ImageIcon("japan.gif");
jb = new JButton(japan);
jb.setActionCommand("Japan");
jb.addActionListener(this);
contentPane.add(jb);

// Добавить текстовое поле в панель содержания
jtf = new JTextField(15);
contentPane.add(jtf);
}

public void actionPerformed(ActionEvent ae) {
    jtf.setText(ae.getActionCommand());
}
}
}

```

Вывод этого апплета представлен на рис. 26.3.

## Флажки

Класс `JCheckBox`, который обеспечивает функциональные возможности флагка, является конкретной реализацией класса `AbstractButton`. Некоторые из его конструкторов:

```

JCheckBox(Icon i)
JCheckBox(Icon i, boolean state)
JCheckBox(String s)
JCheckBox(String s, boolean state)
JCheckBox(String s, Icon i)
JCheckBox(String s, Icon z, boolean state)

```

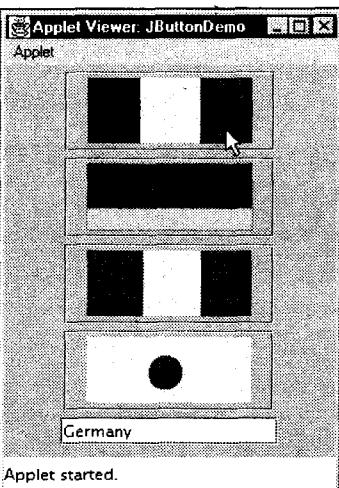
Здесь используются следующие параметры: *i* — изображение для кнопки, *s* — текст. Если *state* — `true`, флагок первоначально выбран. В противном случае — нет.

Состояние флагка может быть изменено с помощью следующего метода:

```
void (boolean state)
```

Здесь параметр *state* должен быть `true`, если нужно, чтобы флагок был установлен (помечен).

Следующий пример показывает, как можно создать апплет, отображающий четыре флагка и текстовое поле. Когда флагок помечается, его подпись



**Рис. 26.3.** Окно апплета JButtonDemo

отображается в текстовом поле. Сначала получена панель содержания для объекта JApplet, и в качестве ее менеджера компоновки устанавливается по-точное размещение. Затем к панели содержания добавлены четыре флашка, и назначены пиктограммы для нормального (без метки), rollover- (с "наездом" указателя мыши) и выбранного (с меткой) состояний. Далее апплет регистрируется, чтобы принимать item-события. Наконец, в панель содержания добавляется текстовое поле.

Когда флашок помечается (выбирается) или сбрасывается (отменяется выбор), генерируется item-событие. Оно обрабатывается методом itemStateChanged(). Внутри itemStateChanged() метод getItem() получает объект JCheckBox, который генерирует событие. Метод getText() получает подпись для этого флашка и использует его для вывода внутри текстового поля.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo" width=400 height=50>
</applet>
*/
public class JCheckBoxDemo extends JApplet
implements ItemListener {
    JTextField jtf;
    public void init() {
        // получить панель содержания
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        // создать пиктограммы
        ImageIcon normal = new ImageIcon("normal.gif");
        ImageIcon rollover = new ImageIcon("rollover.gif");
        ImageIcon selected = new ImageIcon("selected.gif");
        // добавить флашки в панель содержания
        JCheckBox cb = new JCheckBox("C", normal);
        cb.setRolloverIcon(rollover);
        cb.setSelectedIcon(selected);
        cb.addItemListener(this);
        contentPane.add(cb);
        cb = new JCheckBox("C++", normal);
        cb.setRolloverIcon(rollover);
        cb.setSelectedIcon(selected);
        cb.addItemListener(this);
        contentPane.add(cb);
```

```

cb = new JCheckBox("Java", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);

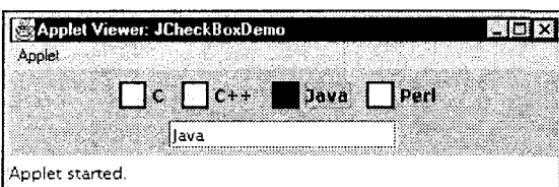
cb = new JCheckBox("Perl", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);

// добавить текстовое поле в панель содержания
jtf = new JTextField(15);
contentPane.add(jtf);
}

public void itemStateChanged(ItemEvent ie) {
    JCheckBox cb = (JCheckBox)ie.getItem();
    jtf.setText(cb.getText());
}
}

```

Вывод этого апплета представлен на рис. 26.4.



**Рис. 26.4.** Окно апплета JCheckBoxDemo

## Переключатели

Переключатели<sup>1</sup> поддерживаются классом `JRadioButton`, который является конкретной реализацией класса `AbstractButton`. Некоторые из его конструкторов:

```

JRadioButton(Icon i)
JRadioButton(Icon z, boolean state)
JRadioButton(String s)
JRadioButton(String s, boolean state)
JRadioButton(String s, Icon i)
JRadioButton(String s, Icon f, boolean state)

```

<sup>1</sup> Переключатели также называются радиокнопками (radio buttons). — Примеч. ред.

где используются параметры: *i* — изображение для кнопки; *s* — текст. Если *state* — true, кнопка первоначально выбрана. Иначе — нет.

Переключатели должны быть объединены в группу, где в каждый момент может быть выбран только один элемент. Например, если пользователь щелкает по переключателю, который находится в группе, любой предварительно нажатый переключатель в этой группе автоматически сбрасывается. Чтобы создать группу кнопок, строится экземпляр класса `ButtonGroup`. Для этой цели вызывается его умалчивающий конструктор. Элементы к группе кнопок добавляются с помощью следующего метода:

```
void add(AbstractButton ab)
```

Здесь *ab* — ссылка на кнопку, которая будет добавлена к группе.

Следующий пример иллюстрирует, как можно использовать переключатели. Создаются три переключателя и одно текстовое поле. Когда выбирается переключатель, его подпись отображается в текстовом поле. Сначала для объекта `JApplet` создается панель содержания, и в качестве ее менеджера компоновки устанавливается поточное размещение. Далее, в панель содержания добавляются три радиокнопки. Затем определяется группа кнопок, в которую добавляются кнопки. Наконец, к панели содержания добавляется текстовое поле.

Выбор переключателя генерирует `action`-событие, которое обрабатывается методом `actionPerformed()`. Метод `getActionCommand()` получает подпись, которая связывается с переключателем и используется для вывода в текстовое поле.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JRadioButtonDemo" width=300 height=50>
</applet>
*/
public class JRadioButtonDemo extends JApplet
implements ActionListener {
    JTextField tf;
    public void init() {
        // получить панель содержания
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        // добавить переключатели в панель содержания
        JRadioButton b1 = new JRadioButton("A");
        b1.addActionListener(this);
        contentPane.add(b1);
```

```

JRadioButton b2 = new JRadioButton("B");
b2.addActionListener(this);
contentPane.add(b2);

JRadioButton b3 = new JRadioButton("C");
b3.addActionListener(this);
contentPane.add(b3);

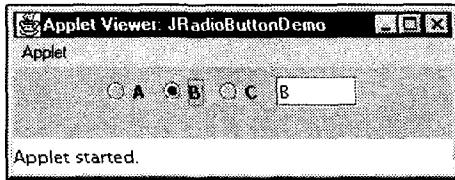
// определить группу кнопок
ButtonGroup bg = new ButtonGroup();
bg.add(b1);
bg.add(b2);
bg.add(b3);

// создать текстовое поле и добавить его
// в панель содержания
tf = new JTextField(5);
contentPane.add(tf);
}

public void actionPerformed(ActionEvent ae) {
    tf.setText(ae.getActionCommand());
}
}

```

Вывод этого апплета представлен на рис. 26.5.



**Рис. 26.5.** Окно апплета JRadioButtonDemo

## Поля со списком

Swing обеспечивает *комбинированное поле* (combo box) — комбинацию текстового поля и раскрывающегося списка, через класс `JComboBox`, который расширяет `JComponent`. Комбинированное поле обычно отображает один вход (элемент) списка. Однако оно может также отображать и раскрывающийся список, который дает возможность пользователю выбирать различные входы. Вы также можете ввести (с клавиатуры) свое значение элемента списка в текстовое поле. Ниже показаны два конструктора `JComboBox`:

```

JComboBox()
JComboBox(Vector v)

```

Здесь `v` — вектор, который инициализирует комбинированное поле.

Элементы добавляются к списку выборов с помощью метода `addItem()`, чья сигнатура имеет вид:

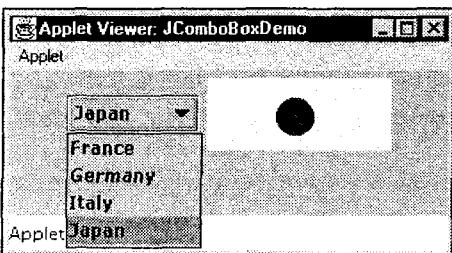
```
void addItem(Object obj)
```

Здесь `obj` — объект, который будет добавлен к комбинированному полю.

Следующий пример содержит комбинированное поле и метку. На метке отображается пиктограмма. Комбинированное поле содержит элементы "France", "Germany", "Italy" и "Japan". Когда выбирается название страны, метка обновляется так, чтобы отобразить флагок этой страны.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JComboBoxDemo" width=300 height=100>
</applet>
*/
public class JComboBoxDemo extends JApplet
implements ItemListener {
    JLabel jl;
    ImageIcon france, germany, italy, japan;
    public void init() {
        // получить панель содержания
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        // создать комбинированный список
        // и добавить его в панель
        JComboBox jc = new JComboBox();
        jc.addItem("France");
        jc.addItem("Germany");
        jc.addItem("Italy");
        jc.addItem("Japan");
        jc.addItemListener(this);
        contentPane.add(jc);
        // создать метку
        jl = new JLabel(new ImageIcon("france.gif"));
        contentPane.add(jl);
    }
    public void itemStateChanged(ItemEvent ie) {
        String s = (String)ie.getItem();
        jl.setIcon(new ImageIcon(s + ".gif"));
    }
}
```

Вывод этого апплета представлен на рис. 26.6.



**Рис. 26.6.** Окно апплета JComboBoxDemo

## Панели со вкладками

Панель со вкладками (tabbed pane) — компонент, который появляется как группа папок в САВ-файле (file cabinet). Каждая папка имеет заголовок. Когда пользователь выбирает папку, ее содержимое становится видимым. Только одна из папок может быть выбрана одновременно. Панель со вкладками обычно используется для установки параметров конфигурации.

Панель со вкладками инкапсулирована классом `JTabbedPane`, который расширяет `JComponent`. Мы будем использовать его умолчаемый конструктор. Вкладки определяются с помощью следующего метода:

```
void addTab(String str, Component comp)
```

Здесь `str` — заголовок вкладки; `comp` — компонент, который должен быть добавлен во вкладку. Как правило, добавляются объекты класса `JPanel` или его подклассов.

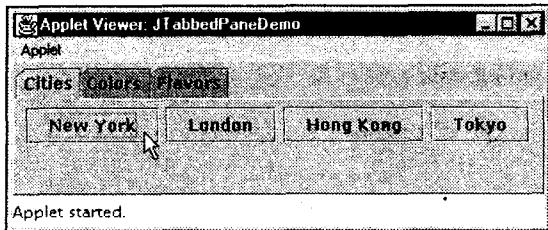
Общая процедура использования в апплете панели со вкладками:

1. Создать объект `JTabbedPane`.
2. Вызвать `addTab()` для добавления вкладки в панель. (Аргументы этого метода определяют заголовок вкладки и компонента, который она содержит.)
3. Повторить шаг 2 для каждой вкладки.
4. Добавить панель со вкладками в панель содержания апплета.

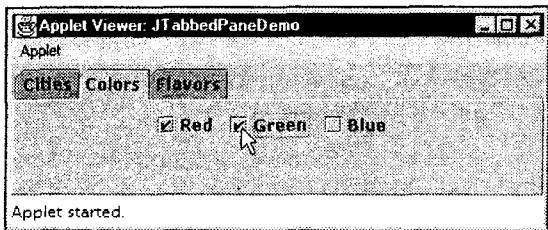
Следующий пример иллюстрирует, как можно создать панель со вкладками. Первая вкладка названа **Cities** (Города) и содержит четыре кнопки с названиями городов. Вторая вкладка названа **Colors** (Цвета) и содержит три флажка, отображающих название цвета. Третья вкладка названа **Flavors** (аромат, привкус) и содержит одно комбинированное поле. Оно дает возможность пользователю выбрать одну из трех разновидностей аромата.

```
import javax.swing.*;
/*
<applet code="JTabbedPaneDemo" width=400 height=100>
</applet>
*/
public class JTabbedPaneDemo extends JApplet {
    public void init() {
        JTabbedPane jtp = new JTabbedPane();
        jtp.addTab("Cities", new CitiesPanel());
        jtp.addTab("Colors", new ColorsPanel());
        jtp.addTab("Flavors", new FlavorsPanel());
        getContentPane().add(jtp);
    }
}
class CitiesPanel extends JPanel {
    public CitiesPanel() {
        JButton b1 = new JButton("New York");
        add(b1);
        JButton b2 = new JButton("London");
        add(b2);
        JButton b3 = new JButton("Hong Kong");
        add(b3);
        JButton b4 = new JButton("Tokyo");
        add(b4);
    }
}
class ColorsPanel extends JPanel {
    public ColorsPanel() {
        JCheckBox cb1 = new JCheckBox("Red");
        add(cb1);
        JCheckBox cb2 = new JCheckBox("Green");
        add(cb2);
        JCheckBox cb3 = new JCheckBox("Blue");
        add(cb3);
    }
}
class FlavorsPanel extends JPanel {
    public FlavorsPanel() {
        JComboBox jcb = new JComboBox();
        jcb.addItem("Vanilla");
        jcb.addItem("Chocolate");
        jcb.addItem("Strawberry");
        add(jcb);
    }
}
```

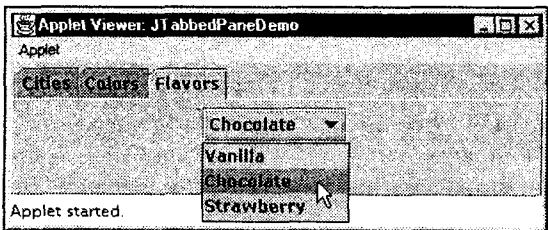
Вывод этого апплета показан на рис. 26.7—26.9.



**Рис. 26.7.** Вкладка **Cities** окна апллета JTabbedPaneDemo



**Рис. 26.8.** Вкладка **Colors** окна апллета JTabbedPaneDemo



**Рис. 26.9.** Вкладка **Flavors** окна апллета JTabbedPaneDemo

## Панели прокрутки

Панель прокрутки (scroll pane) — компонент, который представляет прямоугольную область, в которой компонент может быть просмотрен. В случае необходимости в панель можно добавить горизонтальную и/или вертикальную полосы прокрутки. Панели прокрутки реализованы в Swing классом JScrollPane, который расширяет JComponent. Вот некоторые из его конструкторов:

`JScrollPane(Component comp)`

`JScrollPane(int vsb, int hsb)`

`JScrollPane(Component comp, int vsb, int hsb)`

Здесь `comp` — компонент, который будет добавлен в панель прокрутки; `vsb` и `hsb` — int-константы, которые определяются, когда нужно показывать вертикальные и горизонтальные полосы прокрутки в панели прокрутки. Эти константы определены интерфейсом ScrollPaneConstants. Некоторые примеры этих констант описаны в табл. 26.3.

**Таблица 26.3.** Константы интерфейса `ScrollPaneConstants`

Константа	Описание
<code>HORIZONTAL_SCROLLBAR_ALWAYS</code>	Всегда обеспечивает горизонтальную полосу прокрутки
<code>HORIZONTAL_SCROLLBAR_AS_NEEDED</code>	Обеспечивает горизонтальную полосу прокрутки, если необходимо
<code>VERTICAL_SCROLLBAR_ALWAYS</code>	Всегда обеспечивает вертикальную полосу прокрутки
<code>VERTICAL_SCROLLBAR_AS_NEEDED</code>	Обеспечивает вертикальную полосу прокрутки, если необходимо

Для алгоритма создания апплета с панелью прокрутки необходимо:

1. Создать объект `JComponent`.
2. Создать объект `JScrollPane`. (Аргументы конструктора определяют компонент и установку вертикальных и горизонтальных полос прокрутки.)
3. Добавить панель прокрутки в панель содержания апплета.

Следующий пример иллюстрирует панель прокрутки. Сначала получена панель содержания объекта `JApplet`, и в качестве ее менеджера компоновки назначено граничное размещение. Потом создан объект `JPanel`, и к нему добавлены сотни кнопок, размещенных в двадцати столбцах. Затем создается панель прокрутки и добавляется в панель содержания. Это приводит к появлению вертикальной и горизонтальной полос прокрутки, которые вы можете использовать для листания набора кнопок в этом представлении.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JScrollPaneDemo" width=300 height=250>
</applet>
*/
public class JScrollPaneDemo extends JApplet {
    public void init() {
        // получить панель содержания
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        // добавить в панель 400 кнопок
        JPanel jp = new JPanel();
        jp.setLayout(new GridLayout(20, 20));
        int b = 0;
```

```

for(int i = 0; i < 20; i++) {
    for(int j = 0; j < 20; j++) {
        jp.add(new JButton("Button " + b));
        ++b;
    }
}

// создать панель прокрутки с полосами прокрутки
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(jp, v, h);

// добавить панель прокрутки в центр панели содержания
contentPane.add(jsp, BorderLayout.CENTER);
}
}

```

Вывод этого апплета представлен на рис. 26.10.

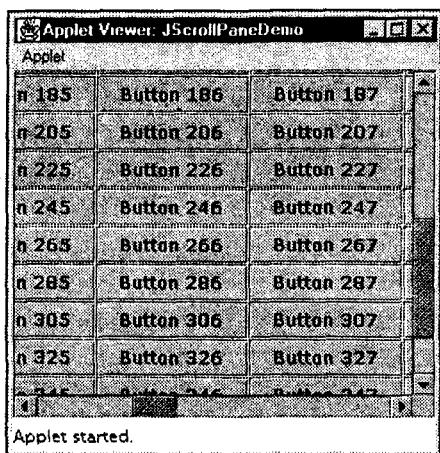


Рис. 26.10. Okno апплета JScrollPaneDemo

## Деревья

Дерево (tree) — компонент, который представляет иерархический вид данных. Пользователь имеет возможность развернуть или свернуть индивидуальные поддеревья в этом показе. Деревья реализованы в Swing классом JTree, который расширяет JComponent. Вот некоторые из его конструкторов:

```

JTree(Hashtable ht)
JTree(Object obj[])
JTree(TreeNode tn)
JTree(Vector v)

```

Первая форма создает дерево, в котором дочерней вершиной является каждый элемент хэш-таблицы `ht`. Во второй форме дочерней вершиной является каждый элемент массива `obj`. В третьей форме параметр `tn` указывает корневой узел дерева. Наконец, последняя форма использует элементы векторного параметра `v` как дочерние вершины.

Когда узел разворачивается или сворачивается, объект `JTree` генерирует события. Методы `AddTreeExpansionListener()` и `removeTreeExpansionListener()` позволяют блокам прослушивания регистрировать или отменять регистрацию для этих уведомлений. Сигнатуры этих методов:

```
void addTreeExpansionListener(TreeExpansionListener tel)  
void removeTreeExpansionListener(TreeExpansionListener tel)
```

Здесь `tel` — объект блока прослушивания.

Чтобы транслировать щелчок мыши на определенной точке дерева в ветвь (путь) дерева используется метод `GetPathForLocation()`. Его сигнатура:

```
TreePath getPathForLocation(int x, int y)
```

Здесь `x` и `y` — координаты указателя мыши, где выполнен щелчок. Возвращаемое значение — объект `TreePath`, который инкапсулирует информацию относительно узла дерева, выбранного пользователем.

Класс `TreePath` инкапсулирует информацию о пути к специальному узлу дерева. Он обеспечивает несколько конструкторов и методов. В этой книге используется только метод `toString()`. Он возвращает строковый эквивалент пути дерева.

Интерфейс `TreeNode` объявляет методы, которые получают информацию относительно узла дерева. Например, возможно получить ссылку к родительскому узлу или перечислению дочерних узлов. Интерфейс `MutableTreeNode` расширяет `TreeNode`. Он объявляет методы, которые могут вставлять и удалять дочерние узлы или изменять родительский узел.

Класс `DefaultMutableTreeNode` реализует интерфейс `MutableTreeNode`. Он представляет узел в дереве. Ниже показан один из его конструкторов:

```
DefaultMutableTreeNode(Object obj)
```

Здесь `obj` — объект, который будет включен в этот узел дерева. Новый узел дерева не имеет родительского или дочернего узла.

Чтобы создавать иерархию узлов дерева, можно использовать метод `add()` класса `DefaultMutableTreeNode`. Его сигнатурата:

```
void add(MutableTreeNode child)
```

Здесь `child` — изменяемый узел дерева, который должен быть добавлен как дочерний к текущему узлу.

События расширения дерева (tree expansion events) описаны классом TreeExpansionEvent в пакете javax.swing.event. Метод getPath() этого класса возвращает объект TreePath, который описывает путь к измененному узлу. Его сигнатура:

```
TreePath getPath()
```

Интерфейс TreeExpansionListener обеспечивает следующие два метода:

```
void treeCollapsed(TreeExpansionEvent tee)
void (TreeExpansionEvent tee)
```

Здесь tee — событие расширения дерева. Первый метод вызывается, когда поддерево сворачивается, а второй — когда поддерево становится видимым (разворачивается).

Шаги алгоритма создания апплета с деревом таковы:

1. Создать объект JTree.
2. Создать объект JScrollPane. (Аргументы конструктора определяют дерево и установку вертикальных и горизонтальных полос прокрутки.)
3. Добавить дерево к панели прокрутки.
4. Добавить панель прокрутки к панели содержания апплета.

Следующий пример иллюстрирует, как можно создавать дерево и распознавать щелчки мыши на нем.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;
/*
<applet code="JTreeEvents" width=400 height=200>
</applet>
*/
public class JTreeEvents extends JApplet {
    JTree tree;
    JTextField jtf;

    public void init() {
        // получить панель содержания
        Container contentPane = getContentPane();

        // установить менеджер компоновки
        contentPane.setLayout(new BorderLayout());

        // создать корневой узел дерева
        DefaultMutableTreeNode top = new DefaultMutableTreeNode("Options");
        tree = new JTree(top);
        contentPane.add(tree, "Center");
    }
}
```

```
// создать поддерево "A"
DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
top.add(a);
DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");
a.add(a1);
DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");
a.add(a2);

// создать поддерево "B"
DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");
top.add(b);
DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("B1");
b.add(b1);
DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("B2");
b.add(b2);
DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("B3");
b.add(b3);

// создать дерево
tree = new JTree(top);

// добавить дерево в панель прокрутки
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(tree, v, h);

// добавить панель прокрутки в панель содержания
contentPane.add(jsp, BorderLayout.CENTER);

// добавить текстовое поле к апплету
jtf = new JTextField("", 20);
contentPane.add(jtf, BorderLayout.SOUTH);

// анонимный внутренний класс для обработки щелчков мыши
tree.addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent me) {
        doMouseClicked(me);
    }
});

void doMouseClicked(MouseEvent me) {
    TreePath tp = tree.getPathForLocation(me.getX(), me.getY());
    if(tp != null)
        jtf.setText(tp.toString());
    else
        jtf.setText("");
}
```

Метод `init()` получает панель содержания для апплета. Затем создается объект `DefaultMutableTreeNode`, маркированный как **Options** (Параметры). Это — корневой узел иерархии дерева. Далее создаются дополнительные узлы дерева, и вызывается метод `add()`, чтобы подключить эти узлы к дереву. В качестве аргумента конструктора `JTree` используется ссылка к корневому узлу дерева. Затем дерево пересыпается как аргумент конструктору `JScrollPane`. Потом эта панель прокрутки добавляется к апплету. После чего создается и добавляется к апплету текстовое поле. В нем представлена информация относительно событий щелчка мыши. Чтобы принимать события мыши от дерева, вызывается метод `addMouseListener()` объекта `JTree`. Аргумент этого метода — анонимный внутренний класс, который расширяет `MouseAdapter` и переопределяет метод `mouseClicked()`.

Метод `doMouseClicked()` обрабатывает щелчки мыши. Он вызывает метод `getPathForLocation()`, чтобы транслировать координаты щелчка мыши в объект `TreePath`. Если кнопка мыши нажата в точке, которая не вызывает выбор узла, возвращаемое значение этого метода — `null`. В противном случае, путь дерева может быть конвертирован в строку и представлен в текстовом поле.

Вывод этого апплета представлен на рис. 26.11.

Строка, представленная в текстовом поле, описывает путь от вершины дерева к выбранному узлу.

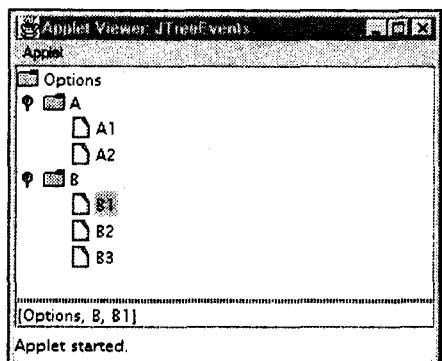


Рис. 26.11. Окно апплета JTreeEvents

## Таблицы

**Таблица** (`table`) — компонент, который отображает строки и столбцы данных. Для изменения размеров столбцов можно перемещать курсором их границы. Можно также перетаскивать столбцы в новую позицию. Таблицы реализованы классом `JTable`, который расширяет `JComponent`. Вот один из его конструкторов:

```
JTable(Object data[ ][ ], Object colHeads[ ])
```

Здесь **data** — двухмерный массив информации, которая будет представлена в форме таблицы; **colHeads** — одномерный массив с заголовками столбца.

Шаги алгоритма для создания таблицы в апплете таковы:

1. Создать объект **JTable**.
2. Создать объект **JScrollPane**. (Аргументы конструктора определяют таблицу и установку для вертикальных и горизонтальных полос прокрутки.)
3. Добавить таблицу в панель прокрутки.
4. Добавить панель прокрутки в панель содержания апплета.

Следующий пример показывает, как можно создать и использовать таблицу.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTableDemo" width=400 height=200>
</applet>
*/
public class JTableDemo extends JApplet {
    public void init() {
        // получить панель содержания
        Container contentPane = getContentPane();

        // установить менеджер компоновки
        contentPane.setLayout(new BorderLayout());

        // инициализировать заголовки столбцов
        final String[] colHeads = { "Name", "Phone", "Fax" };

        // инициализировать данные
        final Object[][] data = {
            { "Gail", "4567", "8675" },
            { "Ken", "7566", "5555" },
            { "Viviane", "5634", "5887" },
            { "Melanie", "7345", "9222" },
            { "Anne", "1237", "3333" },
            { "John", "5656", "3144" },
            { "Matt", "5672", "2176" },
            { "Claire", "6741", "4244" },
            { "Erwin", "9023", "5159" },
            { "Ellen", "1134", "5332" },
            { "Jennifer", "5689", "1212" },
            { "Ed", "9030", "1313" },
            { "Helen", "6751", "1415" }
        };

        // создать таблицу
        JTable table = new JTable(data, colHeads);
```

```

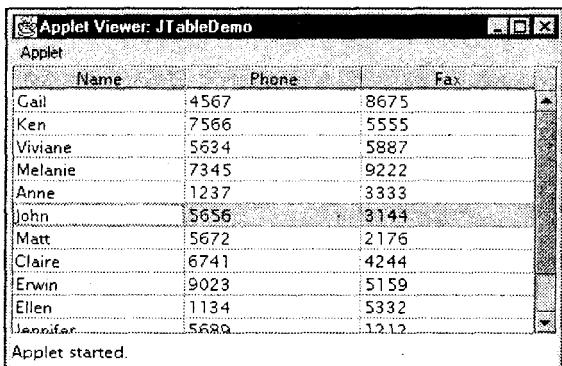
// добавить в панель прокрутки полосы прокрутки
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(table, v, h);

// добавить панель прокрутки в панель содержания
contentPane.add(jsp, BorderLayout.CENTER);
}
}

```

Здесь сначала считывается (с помощью метода `getContentPane()`) панель содержания объекта `JApplet` и в качестве ее менеджера компоновки назначается граничное размещение. Таблица содержит три столбца. Для заголовков столбцов используется одномерный строчный массив (`colHeads`), а для ячеек таблицы — двухмерный строчный массив (`data`). Не трудно видеть, что каждый элемент в этом массиве является, в свою очередь, массивом из трех строк. Эти массивы передаются конструктору `JTable`. Затем в таблицу добавляется полоса прокрутки, и панель прокрутки добавляется в панель содержания.

Вывод этого апплета представлен на рис. 26.12.



**Рис. 26.12.** Окно апплета JTableDemo

## Другие возможности и будущее Swing-технологии

Как говорилось ранее, Swing — это большая система. Она имеет еще очень много свойств, которые вы можете исследовать самостоятельно. Например, Swing обеспечивает инструментальные панели (toolbars), подсказки кнопочных команд (tooltips), и прогресс-полоски (progress bars). Компоненты Swing могут также обладать специфическим *pluf*<sup>1</sup>-свойством, которое "pluggable

<sup>1</sup> *Pluf* — от англ. *pluggable look and feel* (подключаемый вид и поведение). — Примеч. пер.

"look-and-feel" означает, что к Swing-компоненту можно подключить другой вид и поведение. Причем это может быть сделано динамически. Вы можете даже проектировать ваш собственный вид и поведение. Таким образом, Swing-подход к GUI-компонентам мог бы когда-то в будущем заменить AWT-классы, поэтому ознакомление с ним сейчас — хорошая идея.

Swing — только одна часть библиотеки классов Java Foundation Classes (JFC). Можно использовать и другие части JFC. Например, Accessibility API (пакет `java.awt.accessibility`) можно применять для построения программ, которые пригодны для использования людьми с физическими недостатками. Java 2D API (система улучшенного управления графикой), упомянутая в конце главы 21, обеспечивает расширенные возможности работы с формами, текстом и изображениями. Drag-and-Drop API (пакет `java.awt.dnd`) поддерживает обмен информацией между Java- и не Java-программами.



## ГЛАВА 27

# Сервлеты

В этой главе представлен краткий обзор сервлетов. *Сервлеты* (servlets) — это маленькие программы, которые выполняются на серверной стороне Web-соединения. Точно так же, как апплеты динамически расширяют функциональные возможности Web-браузера, сервлеты динамически расширяют функциональные возможности Web-сервера.

Библиотеки классов, которые необходимы для создания сервлетов, содержат специальный набор инструментальных средств Java Servlet Development Kit (JSDK). Несколько приведенных здесь примеров иллюстрируют, как можно использовать их функциональные возможности. Вы увидите, как нужно обрабатывать GET- и POST-запросы HTTP от браузера, использовать cookie-данные и организовывать сеансы.

## Предпосылки

Чтобы понимать преимущества сервлетов, нужно иметь основные представления о том, как кооперируются Web-браузеры и серверы, чтобы предоставить их содержимое пользователю. Рассмотрим запрос статической Web-страницы. Пользователь вводит *унифицированный локатор ресурса* (URL, Uniform Resource Locator) в браузер. Браузер генерирует HTTP-запрос к соответствующему Web-серверу. Web-сервер устанавливает соответствие этого запроса с определенным файлом. Данный файл возвращается браузеру в HTTP-ответе. HTTP-заголовок в этом ответе указывает тип своего содержимого в формате MIME<sup>1</sup>. Например, обычный ASCII-текст имеет MIME-тип text/plain. Исходный HTML-текст Web-страницы имеет MIME-тип text/html.

<sup>1</sup> Стандарт MIME (Multipurpose Internet Mail Extensions, *Многоцелевое расширение почты Internet*) используется для указания типа содержимого в заголовке HTTP-сообщений. — Примеч. перев.

Теперь рассмотрим динамическое содержание. Предположим, что сетевой книжный магазин использует базу данных для хранения информации относительно его бизнеса, включая книжные ценники, наличность, заказы и т. д. Требуется сделать эту информацию доступной для заказчиков через Web-страницы. Чтобы отражать самую последнюю информацию в базе данных, содержание этих Web-страниц должно быть генерировано динамически.

В ранних версиях Web-сервер мог динамически конструировать страницу, создавая отдельный процесс для обработки каждого запроса клиента. Чтобы получать необходимую информацию, процесс мог открывать соединения к одной или нескольким базам данных. Он общался с Web-сервером через интерфейс, известный как *Common Gateway Interface (CGI)*. CGI позволял отдельному процессу читать данные из HTTP-запроса и записывать данные в HTTP-ответ. Для построения CGI-программ использовался ряд различных языков, включая C, C++ и Perl.

Однако у CGI-программ отмечены серьезные проблемы с эффективностью. Создание отдельного процесса для каждого запроса клиента обходится дорого (в смысле требования больших ресурсов памяти и процессора). Дорого также открывать и закрывать соединения базы данных для каждого запроса клиента. Кроме того, программы CGI не являлись платформно-независимыми. Поэтому были разработаны другие методы общения, включая сервлеты.

Сервлеты обеспечивают несколько преимуществ по сравнению с CGI-интерфейсом.

- Повышена эффективность. Сервлеты выполняются в пределах адресного пространства Web-сервера. Создание отдельного процесса для обработки каждого запроса клиента не является необходимым.
- Сервлеты не зависят от платформы, потому что они написаны на Java. Несколько Web-серверов от таких поставщиков, как Sun, Netscape и Microsoft, предлагают Servlet API. Программы, разработанные для этого API, могут быть перемещены в любую из указанных сред без перетрансляции.
- Менеджер безопасности Java (Java Security Manager) на сервере поддерживает набор ограничений для защиты ресурсов на машине сервера. Вы увидите, что некоторые сервлеты надежны, а другие — нет.
- Сервлетам доступны полные функциональные возможности библиотек классов Java. Они могут связываться с апплетами, базами данных, или другим программным обеспечением через сокеты и RMI-механизмы, которые уже рассматривались ранее.

## Жизненный цикл сервлета

Центральное место в жизненном цикле сервлета занимают три метода: `init()`, `service()` и `destroy()`. Они реализуются каждым сервлетом и вызы-

ваются в определенные моменты сервером. Чтобы понять, когда эти методы вызываются, рассмотрим следующий типичный сценарий пользователя.

1. Предположим, что пользователь вводит в Web-браузер *унифицированный адрес ресурса* (URL, Uniform Resource Locator). Затем браузер генерирует HTTP-запрос для этого URL-адреса и посыпает его соответствующему серверу.
2. Web-сервер получает этот HTTP-запрос и направляет его к специальному сервлету, который динамически извлекается и загружается в адресное пространство сервера.
3. Сервер вызывает метод `init()` сервлета (метод вызывается, только когда сервлет предварительно загружен в память). Параметры инициализации можно переслать сервлету так, чтобы он мог сам себя конфигурировать.
4. Сервер обращается к методу `service()` сервлета, который вызывается для обработки HTTP-запроса. Сервлет может читать данные, которые были переданы в HTTP-запросе, а также формировать HTTP-ответ для клиента.
5. Сервлет остается в адресном пространстве сервера и может обрабатывать любые другие HTTP-запросы, полученные от клиентов. Метод `service()` вызывается для каждого HTTP-запроса.
6. Наконец, сервер может решить выгрузить сервлет из своей памяти. Алгоритмы, с помощью которых это намерение выполняется, специфичны для каждого сервера. Сервер вызывает метод `destroy()`, чтобы освободить любые ресурсы, такие как дескрипторы файлов, которые распределены для сервлета. Важные данные могут быть сохранены в постоянном хранилище. Затем может быть выполнена сборка мусора в памяти, распределенной для сервлета и его объектов.

## Java Servlet Development Kit

Набор инструментов разработки Java-сервлетов (JSDK, Java Servlet Development Kit) содержит библиотеки классов, которые нужны для создания сервлетов. Он включает также утилиту, известную как `servletrunner`, которая дает возможность проверить некоторые из создаваемых сервлетов. Мы будем использовать этот инструмент для выполнения примеров.

Вы можете загружать JSDK бесплатно с Web-сайта [java.sun.com](http://java.sun.com) фирмы Sun Microsystems. Для установки JSDK придерживайтесь следующих правил:

- Для Windows-машин JSDK Version 2 по умолчанию размещается в каталоге `c:\Jsdk2.0`.
- Каталог `c:\Jsdk2.0\bin` содержит утилиту `servletrunner.exe`. Обновите переменную среды PATH так, чтобы она включила этот каталог.

- Каталог c:\Jsdk2.0\lib содержит архивный файл jsdk.jar. Этот файл содержит классы и интерфейсы, которые необходимы для построения сервлетов. Обновите переменную среды CLASSPATH так, чтобы она включила каталог c:\Jsdk2.0\lib.

## Простой сервлет

Ознакомление с ключевыми концепциями сервлетов мы начнем с построения и проверки простого сервлета. Основные шаги этой процедуры таковы:

1. Создайте и скомпилируйте исходный код сервлета.
2. Запустите утилиту servletrunner.
3. Запустите Web-браузер и запросите сервлет.

Следующие разделы рассматривают каждый из этих шагов подробно.

## Создание и компиляция исходного кода сервлета

Для начала создайте файл с именем HelloServlet.java, который содержит следующую программу:

```
import java.io.*;
import javax.servlet.*;

public class HelloServlet extends GenericServlet {

    public void service(ServletRequest request,
                        ServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>Hello!" );
        pw.close();
    }
}
```

Обратите внимание, что эта программа импортирует пакет javax.servlet, который содержит классы и интерфейсы, требуемые для построения сервлетов. Затем, программа определяет HelloServlet как подкласс GenericServlet. Класс GenericServlet обеспечивает функциональные возможности, которые облегчают обработку запросов и ответов.

Внутри HelloServlet переопределяется метод service() (который унаследован из GenericServlet). Этот метод обрабатывает запросы клиента. Обратите внимание, что первый аргумент — это объект класса ServletRequest. Он дает возможность сервлету читать данные, которые получаются через запрос

клиента. Второй аргумент — объект класса `ServletResponse`. Он позволяет сервлету сформировать ответ для клиента.

Вызов `setContentType()` устанавливает MIME-тип HTTP-ответа. В этом вызове указан MIME-тип `text/html`, который заставляет браузер интерпретировать содержимое как исходный HTML-код.

Далее, метод `getWriter()` заполняет объект `PrintWriter`. Все записанное в этот поток посыпается клиенту как часть HTTP-ответа. Затем используется `println()`, чтобы записать некоторый простой исходный код HTML как HTTP-ответ.

Откомпилируйте этот исходный код, и поместите файл `HelloServlet.class` в каталог `c:\Jsdk2.0\examples`. Это гарантирует, что утилита `servletrunner` сможет его найти.

## Запуск утилиты `servletrunner`

Для запуска этой утилиты откройте окно с приглашением ввода команды и введите `servletrunner`. Этот инструмент прослушивает порт 8080 для получения входящих запросов клиента.

## Запуск Web-браузера и запрос сервлета

Запустите Web-браузер, и введите следующий URL-адрес:

`http://localhost:8080/servlet/HelloServlet`

В качестве альтернативы можно ввести такой URL-адрес:

`http://127.0.0.1:8080/servlet/HelloServlet`

Здесь вместо символьного DNS-адреса указан цифровой IP-адрес локальной машины (127.0.0.1).

Просмотрите вывод сервлета в области показа браузера. Он должен содержать строку `Hello!` (с полужирным шрифтом).

### Замечание

Примеры в этой главе предполагают, что `servletrunner` и Web-браузер выполняются на одной машине. Однако эти два приложения могут быть установлены на различных компьютерах. В этом случае URL-адреса должны быть изменены, чтобы идентифицировать машину, на которой выполняется `servletrunner`.

## Servlet API

Коды, которые требуются для построения сервлетов, содержат два пакета: `javax.servlet` и `javax.servlet.http`. Вместе они представляют Servlet API.

Имейте в виду, что (на момент написания данной книги) эти пакеты не являются частью пакетов ядра Java, потому что они не включены в JDK. Чтобы получить их функциональные возможности, следует загрузить JSDK.

Servlet API поддерживается большинством Web-серверов от фирм Sun, Microsoft и др. За дополнительной информацией обращайтесь к Web-страницам Sun Microsystems в <http://java.sun.com>.

## Пакет *javax.servlet*

Пакет *javax.servlet* содержит ряд интерфейсов и классов, устанавливающих обрамление, в котором работают сервлеты. Табл. 27.1 описывает интерфейсы, поддерживающиеся этим пакетом. Наиболее значительный из них — это *Servlet*. Все сервлеты должны реализовывать указанный интерфейс или расширять класс, который его реализует. Интерфейсы *ServletRequest* и *ServletResponse* тоже очень важны.

*Таблица 27.1. Интерфейсы пакета javax.servlet*

Интерфейс	Описание
<i>Servlet</i>	Объявляет методы цикла жизни для сервлета
<i>ServletConfig</i>	Позволяет сервлетам получать параметры инициализации
<i>ServletContext</i>	Активизирует возможности сервлетов для регистрации событий и доступа к информации об их среде
<i>ServletRequest</i>	Используется для чтения данных из запроса клиента
<i>ServletResponse</i>	Используется для записи данных в ответ клиента
<i>SingleThreadModel</i>	Указывает, что сервлет защищен от многопоточности

Табл. 27.2 описывает классы, которые поддерживаются этим пакетом.

*Таблица 27.2. Классы пакета javax.servlet*

Класс	Описание
<i>GenericServlet</i>	Реализует интерфейсы <i>Servlet</i> и <i>ServletConfig</i>
<i>ServletInputStream</i>	Обеспечивает входной поток для чтения запросов от клиента
<i>ServletOutputStream</i>	Обеспечивает выходной поток для записи ответов клиенту
<i>ServletException</i>	Указывает, что произошла ошибка сервлета
<i>UnavailableException</i>	Указывает, что сервлет постоянно или временно недоступен

Следующие разделы рассматривают эти интерфейсы и классы более подробно.

## Интерфейс *Servlet*

Все сервлеты должны реализовывать интерфейс *Servlet*. Он объявляет методы *init()*, *service()* и *destroy()*, которые вызываются сервером в течение цикла жизни сервлета. Существует также метод, который позволяет сервлету получать любые параметры инициализации. Методы, определенные в *Servlet*, показаны в табл. 27.3.

**Таблица 27.3. Методы, определенные в *Servlet***

Метод	Описание
<code>void destroy()</code>	Вызывается, когда сервлет выгружается
<code>ServletConfig getServletConfig()</code>	Возвращает объект <i>ServletConfig</i> , который содержит все параметры инициализации
<code>String getServletInfo()</code>	Возвращает строку, описывающую сервлет
<code>void init(ServletConfig sc) throws ServletException</code>	Вызывается, когда сервлет инициализируется. <i>sc</i> — определяет параметры его инициализации. Если сервлет не может быть инициализирован, выбрасывается исключение <i>UnavailableException</i>
<code>void service(ServletRequest req,                   ServletResponse res) throws ServletException, IOException</code>	Вызывается, чтобы обработать запрос от клиента. Запрос от клиента может читаться из <i>req</i> . Ответ клиенту может быть записан в <i>res</i> . Если случаются проблемы сервлета или ввода/вывода, генерируется исключение <i>ServletException</i> или <i>IOException</i>

Методы *init()*, *service()* и *destroy()* — это методы цикла жизни сервлета. Они вызываются сервером. Метод *GetServletConfig()* вызывается сервлетом, чтобы получить параметры инициализации. Разработчик сервлета переопределяет метод *getServletInfo()*, чтобы обеспечить строку с полезной информацией (например, с автором, версией, датой, авторским правом, и т. д.). Этот метод также вызывается сервером.

## Интерфейс *ServletConfig*

Интерфейс *ServletConfig* реализуется сервером. Он позволяет сервлету получать данные конфигурации (после своей загрузки). В табл. 27.4 описаны методы, объявленные этим интерфейсом.

**Таблица 27.4. Методы, определенные в *ServletConfig***

Метод	Описание
<code>ServletContext getServletContext()</code>	Возвращает контекст этого сервлета
<code>String getInitParameter(String param)</code>	Возвращает значение параметра инициализации с именем <code>param</code>
<code>Enumeration getInitParameterNames()</code>	Возвращает перечисление всех имен параметров инициализации

## Интерфейс *ServletContext*

Интерфейс *ServletContext* реализуется сервером. Он дает возможность сервлетам получать информацию об их среде. Описания этих методов приводятся в табл. 27.5. Одно важное замечание: методы `getServlet()` и `getServletNames()` опасны для использования, потому что они могут разрушить механизм, поддерживающий состояния сервера.

**Таблица 27.5. Методы, определенные в *ServletContext***

Метод	Описание
<code>Object getAttribute(String attr)</code>	Возвращает значение атрибута сервера с именем <code>attr</code>
<code>String getMimeType (String file)</code>	Возвращает MIME-тип файла
<code>String getRealPath(String vpath)</code>	Возвращает реальный путь, который соответствует виртуальному пути <code>vpath</code>
<code>String getServerInfo()</code>	Возвращает информацию о сервере
<code>Servlet getServlet(String sname) throws ServletException</code>	Возвращает сервлет с именем <code>sname</code>
<code>Enumeration getServletNames()</code>	Возвращает перечисление с именами сервлетов в том же пространстве имен на сервере
<code>void log(String s)</code>	Записывает строку <code>s</code> в файл регистрации сервера

Таблица 27.5 (окончание)

Метод	Описание
<code>void log(Exception e, String s)</code>	Записывает строку <i>s</i> и трассу стека <i>e</i> в файл регистрации сервера

## Интерфейс *ServletRequest*

Интерфейс *ServletRequest* реализуется сервером. Он дает возможность сервлету получить информацию о запросе клиента. Описания его методов приводятся в табл. 27.6.

Таблица 27.6. Методы, определенные в *ServletRequest*

Метод	Описание
<code>String getAttribute(String attr)</code>	Возвращает значение атрибута с именем <i>attr</i>
<code>String getCharacterEncoding()</code>	Возвращает шифрование символов запроса
<code>int getContentLength()</code>	Возвращает размер запроса. Если размер неизвестен, возвращается значение -1
<code>String getContentType()</code>	Возвращает тип запроса. Если тип не может быть определен, возвращается значение <i>null</i>
<code>ServletInputStream getInputStream() throws IOException</code>	Возвращает <i>ServletInputStream</i> , который можно использовать для чтения двоичных данных запроса. Если метод <code>getReader()</code> уже был вызван для этого запроса, выбрасывается исключение <i>IllegalStateException</i>
<code>String getParameter(String pname)</code>	Возвращает значение параметра с именем <i>pname</i>
<code>Enumeration getParameterNames()</code>	Возвращает перечисление имен параметров этого запроса
<code>String[] getParameterValues()</code>	Возвращает перечисление значений параметров этого запроса
<code>String getProtocol()</code>	Возвращает описание протокола

Таблица 27.6 (окончание)

Метод	Описание
<code>BufferedReader getReader() throws IOException</code>	Возвращает буферизированное считывающее устройство, которое может быть использовано, чтобы читать текст запроса. Если метод <code>getInputStream()</code> уже был вызван для этого запроса, выбрасывается исключение <code>IllegalStateException</code>
<code>String getRealPath(String vpath)</code>	Возвращает реальный путь, соответствующий виртуальному пути <code>vpath</code>
<code>String getRemoteAddr()</code>	Возвращает строковый эквивалент IP-адреса клиента
<code>String getRemoteHost()</code>	Возвращает строковый эквивалент имени хост-машины клиента
<code>String getScheme()</code>	Возвращает схему передачи URL, используемую для запроса (например "http", "ftp")
<code>String getServerName()</code>	Возвращает имя сервера
<code>int getServerPort()</code>	Возвращает номер порта

## Интерфейс *ServletResponse*

Интерфейс *ServletResponse* реализуется сервером. Он дает возможность сервлету формировать ответ для клиента. Его методы описаны в табл. 27.7.

Таблица 27.7. Методы, определенные в *ServletResponse*

Метод	Описание
<code>String getCharacterEncoding()</code>	Возвращает шифрование символов ответа
<code>ServletOutputStream getOutputStream() throws IOException</code>	Возвращает <i>ServletOutputStream</i> , который можно использовать для записи двоичных данных в ответ. Если метод <code>getWriter()</code> уже был вызван для этого запроса, выбрасывается исключение <code>IllegalStateException</code>

Таблица 27.7 (окончание)

Метод	Описание
<code>PrintWriter getWriter() throws IOException</code>	Возвращает объект PrintWriter, который можно использовать для записи символьных данных в ответ. Если метод <code>getOutputStream()</code> уже был вызван для этого запроса, выбрасывается исключение <code>IllegalStateException</code>
<code>void setContentLength(int size)</code>	Устанавливает длину содержания ответа к значению, полученному через параметр <code>size</code>
<code>void setContentType(String type)</code>	Устанавливает тип содержания ответа к значению, полученному через параметр <code>type</code>

## Интерфейс *SingleThreadModel*

Интерфейс *SingleThreadModel* поддерживает однопоточное выполнение метода `service()` сервлета. Он не определяет никаких констант и не объявляет никаких методов. Если используется данный интерфейс, то сервер создает несколько экземпляров сервлета. Когда прибывает запрос клиента, он посыпается доступному экземпляру сервлета.

## Класс *GenericServlet*

Класс *GenericServlet* обеспечивает реализацию основных методов жизненного цикла сервлета. Разработчики сервлетов обычно работают с его подклассами. *GenericServlet* реализует интерфейсы *ServletConfig* и *Servlet*. Кроме того, имеется метод для добавления строки в файл регистрации сервера. Сигнатура этого метода:

```
void (String s)
```

Здесь *s* — строка, которая будет добавлена к файлу регистрации.

## Класс *ServletInputStream*

Класс *ServletInputStream* расширяет *InputStream*. Он реализуется сервером и обеспечивает входной поток, который разработчик сервлетов может использовать для чтения данных запроса клиента. Он определяет умолчаемый конструктор. Кроме того, обеспечивается метод для чтения байтов из потока. Его сигнатура:

```
int readLine(byte[ ] buffer, int offset, int size) throws IOException
```

Здесь *buffer* — массив, в котором помещено *size* байтов, начиная с *offset*. Метод возвращает фактическое число считанных байтов, или *-1*, если встретилось условие конца потока.

## Класс *ServletOutputStream*

Класс *ServletOutputStream* расширяет *OutputStream*. Он реализуется сервером и обеспечивает выходной поток, который может использовать разработчик сервлетов для записи данных в ответ клиенту. Определен умолчиваемый конструктор. Он также определяет методы *print()* и *println()*, которые выводят данные в поток.

## Класс *ServletException*

Класс *ServletException* указывает, что в сервлете произошла какая-то проблема. Класс имеет следующие конструкторы:

```
ServletException()  
ServletException(String s)
```

Здесь *s* — строка, которая описывает проблему.

## Класс *UnavailableException*

Класс *UnavailableException* расширяет *ServletException*. Он указывает, что сервlet постоянно или временно недоступен. Класс имеет следующие конструкторы:

```
UnavailableException(Servlet servlet, String s)  
UnavailableException(int secs, Servlet servlet, String s)
```

Здесь параметр *servlet* указывает, какой сервлет является недоступным. Описание проблемы указывается в параметре *s*. Число секунд, в течение которых сервлет, как ожидается, будет недоступным, указывается в *secs*.

## Чтение параметров сервлета

Класс *ServletRequest* включает методы, позволяющие читать имена и значения параметров, которые включены в запрос клиента. Мы разработаем сервлет, иллюстрирующий их использование. Пример содержит два файла: *PostParameters.htm* — определяет Web-страницу и *PostParametersServlet.java* — определяет сервлет.

Исходный HTML-код для *PostParameters.htm* показан в следующем листинге. Он определяет таблицу, которая содержит две метки и два текстовых по-

ля. Одна из меток — **Employee** (Служащий), а другая — **Phone** (Телефон). Форма включает также предъявляющую кнопку. Обратите внимание, что параметр **action** тега формы (**<form>**) определяет URL-адрес. URL идентифицирует сервер для обработки POST-запроса HTTP.

```
<html>
  <body>
    <center>
      <form name="Form1"
            method="post"
            action="http://localhost:8080/servlet/PostParametersServlet">
        <table>
          <tr>
            <td><B>Employee</td>
            <td><input type= textbox name="e" size="25" value=""></td>
          </tr>
          <tr>
            <td><B>Phone</td>
            <td><input type= textbox name="p" size="25" value=""></td>
          </tr>
        </table>
        <input type=submit value="Submit">
      </form>
    </body>
  </html>
```

Исходный код для PostParametersServlet.java показан в следующем листинге. Метод **service()** переопределен для обработки запросов клиента. Метод **getParameterNames()** возвращает перечисление имен параметров. Они обрабатываются в цикле. Имя и значение параметра выводятся клиенту. Значение параметра получается с помощью метода **getParameter()**.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;

public class PostParametersServlet
extends GenericServlet {

  public void service(ServletRequest request,
                      ServletResponse response)
    throws ServletException, IOException {
    // получить запись печати
    PrintWriter pw = response.getWriter();

    // получить перечисление имен параметров
    Enumeration e = request.getParameterNames();
```

```
// показать имена и значения параметров
while(e.hasMoreElements()) {
    String pname = (String)e.nextElement();
    pw.print(pname + " = ");
    String pvalue = request.getParameter(pname);
    pw.println(pvalue);
}
pw.close();
}
```

Откомпилируйте сервлет, и выполните следующий алгоритм, чтобы проверить этот пример:

1. Запустите `servletrunner`.
2. Отобразите Web-страницу в браузере.
3. Введите в текстовые поля имя и телефонный номер служащего.
4. Предоставьте Web-страницу для просмотра.

После выполнения этих шагов, браузер отображает ответ, который динамически генерируется сервлетом.

## Чтение параметров инициализации

Можно обеспечить параметры инициализации для сервлета. Они могут использоваться для открытия файлов, создания подключений базы данных или выполнения других действий. К этой информации можно обращаться двумя способами:

- Метод `init()`, объявленный интерфейсом `Servlet`, принимает объект `ServletConfig` в качестве его параметра. Этот объект обеспечивает методы, которые дают возможность читать параметры инициализации.
- Метод `getServletConfig()`, объявленный интерфейсом `Servlet`, возвращает объект `ServletConfig`.

Манера, в которой параметры инициализации предоставляются сервлету, зависит от сервера. Утилита `servletrunner` определяет параметры инициализации сервлетов в файле с именем `servlet.properties`. По умолчанию этот файл располагается в каталоге `c:\jsdk2.0\examples`.

Теперь разработаем сервлет, который читает некоторые параметры инициализации. Пример содержит два файла: `servlet.properties` и `InitServlet.java`. В следующем листинге показан файл `servlet.properties`. Он определяет два свойства сервлета: `servlet.name.code` и `servlet.name.initargs`. Первое позволяет связать имя с классом, который содержит код сервлета. Второе дает возможность определить последовательность разграниченных запятой имен и

значений параметров. Для обоих свойств параметр *name* — строка имени сервлета. Синтаксис каждого имени и значения параметра следующий:

**pname = pvalue**

где **pname** — имя параметра, а **pvalue** — значение параметра.

В следующей версии servlet.properties можно видеть, что имя сервлета — *init servlet*. Его код расположен в классе *InitServlet*. Два параметра инициализации с именами *country* и *city* получают значения *Canada* и *Toronto* соответственно.

```
servlet.init servlet.code=InitServlet
servlet.init servlet.initArgs=\
    country=Canada, \
    city=Toronto
```

Далее показан исходный код *InitServlet.java*. Сначала вызывается метод *getServletConfig()*, чтобы получить объект *ServletConfig*. Метод *getInitParameter()* используется для получения значений для двух параметров инициализации. Затем они записываются в ответ HTTP.

```
import java.io.*;
import javax.servlet.*;

public class InitServlet extends GenericServlet {

    public void service(ServletRequest request,
        ServletResponse response)
    throws ServletException, IOException {

        // получить объект Get ServletConfig
        ServletConfig sc = getServletConfig();

        // вывести два параметра инициализации
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>Country: " +
            sc.getInitParameter("country"));
        pw.println("<br>City: " +
            sc.getInitParameter("city"));
        pw.close();
    }
}
```

Можно проверить этот пример, запрашивая адрес <http://localhost:8080/servlet/init servlet> в вашем браузере. Должен появиться следующий вывод:

```
Count ry: Canada
City: Toronto
```

Отметим, что URL должен содержать имя, которое назначено этому сервлету в файле `servlet.properties`, а не имя класса сервлета. Иначе параметры инициализации не могут быть считаны.

## Пакет `javax.servlet.http`

Пакет `javax.servlet.http` содержит несколько интерфейсов и классов, которые обычно используются разработчиками сервлетов. Вы увидите, что его функциональные возможности облегчают построение сервлетов, которые работают с HTTP-запросами и ответами.

Табл. 27.8 описывает интерфейсы, которые определяются в этом пакете.

**Таблица 27.8. Интерфейсы `javax.servlet.http`**

Интерфейс	Описание
<code>HttpServletRequest</code>	Разрешает сервлетам читать данные из HTTP-запроса
<code>HttpServletResponse</code>	Разрешает сервлетам записывать данные в HTTP-ответ
<code>HttpSession</code>	Позволяет читать и записывать данные сеанса
<code>HttpSessionBindingListener</code>	Информирует объект, что он связан или не связан с сеансом
<code>HttpSessionContext</code>	Обеспечивает управляемость сеансов

Табл. 27.9 описывает классы, которые определены в этом пакете. Наиболее важный из них — `HttpServlet`. Разработчики сервлетов обычно расширяют указанный класс, чтобы обработать запросы HTTP.

**Таблица 27.9. Классы `javax.servlet.http`**

Класс	Описание
<code>Cookie</code>	Позволяет сохранять информацию состояния на машине клиента
<code>HttpServlet</code>	Обеспечивает методы для обработки запросов и ответов HTTP
<code>HttpSessionBindingEvent</code>	Указывает на наличие или отсутствие связи блока прослушивания с сессионным значением
<code>HttpUtils</code>	Объявляет методы утилит для сервлетов

## Интерфейс *HttpServletRequest*

Интерфейс *HttpServletRequest* реализуется сервером. Он позволяет сервлету получить информацию о запросе клиента. Его методы показаны в табл. 27.10.

**Таблица 27.10.** Методы, определенные в *HttpServletRequest*

Метод	Описание
<code>String getAuthType()</code>	Возвращает схему аутентификации
<code>Cookie[] getCookies()</code>	Возвращает массив cookie-данных в этом запросе
<code>long getDateHeader(String field)</code>	Возвращает значение поля даты заголовка, передаваемого через параметр <code>field</code>
<code>String getHeader(String field)</code>	Возвращает значение поля заголовка, передаваемого через параметр <code>field</code>
<code>Enumeration getHeaderNames()</code>	Возвращает перечисление имен заголовка
<code>int getIntHeader(String field)</code>	Возвращает int-эквивалент поля заголовка получаемого в <code>field</code>
<code>String getMethod()</code>	Возвращает HTTP-метод этого запроса
<code>String getPathInfo()</code>	Возвращает информацию пути, которая расположена после пути сервлета и перед строкой запроса в URI <sup>1</sup>
<code>String getPathTranslated()</code>	Возвращает информацию пути, которая расположена после пути сервлета и перед строкой запроса в URI, после ее трансляции в реальный путь
<code>String getQueryString()</code>	Возвращает строку запроса в URI
<code>String getRemoteUser()</code>	Возвращает имя пользователя, который выдал этот запрос
<code>String getRequestedSessionId()</code>	Возвращает ID (идентификатор) сеанса

<sup>1</sup> URI (Universal Resource Identifier) — универсальный идентификатор ресурса. — Примеч. пер.

Таблица 27.10 (окончание)

Метод	Описание
<code>String getRequestURI()</code>	Возвращает часть URI слева от строки запроса
<code>String getServletPath()</code>	Возвращает часть URI, которая идентифицирует сервлет
<code>HttpSession getSession(boolean new)</code>	Если параметр <code>new</code> указывается как <code>true</code> , то метод создает и возвращает сеанс этого запроса. Иначе возвращает существующий сеанс этого запроса
<code>boolean isRequestedSessionIdFromCookie()</code>	Возвращает <code>true</code> , если cookie-данные содержат ID сеанса. Иначе возвращает <code>false</code>
<code>boolean isRequestedSessionIdFromUrl()</code>	Возвращает <code>true</code> , если URL-адрес содержит ID сеанса. Иначе возвращает <code>false</code>
<code>boolean isRequestedSessionIdValid()</code>	Возвращает <code>true</code> , если запрошенный ID сеанса является правильным в текущем контексте сеанса

## Интерфейс `HttpServletResponse`

Интерфейс `HttpServletResponse` реализуется сервером. Он дает возможность сервлету формировать HTTP-ответ клиенту. В нем определено несколько дюжин констант, соответствующих различным кодам состояния, которые могут быть назначены HTTP-ответу. Например, `SC_OK` указывает, что HTTP-запрос достиг цели, а `SC_NOT_FOUND` указывает, что требуемый ресурс недоступен. Методы этого интерфейса кратко описаны в табл. 27.11.

Таблица 27.11. Методы, определенные в `HttpServletResponse`

Метод	Описание
<code>void addCookie(Cookie cookie)</code>	Прибавляет cookie-данные к HTTP-ответу
<code>boolean containsHeader(String field)</code>	Возвращает <code>true</code> , если HTTP-заголовок ответа содержит поле с именем, которое указано в параметре <code>field</code>

Таблица 27.11 (продолжение)

Метод	Описание
<code>String encodeURL(String url)</code>	Определяет, нужно ли кодировать ID сеанса в URL-адресе, указанном в параметре <code>url</code> . Если нужно, то возвращает модифицированную версию адреса, указанного в <code>url</code> . Иначе возвращает немодифицированный <code>url</code> . Этим методом должны быть обработаны все URL-адреса, сгенерированные сервером
<code>String encodeRedirectUrl(String url)</code>	Определяет, нужно ли кодировать ID сеанса в URL-адресе, указанном в параметре <code>url</code> . Если нужно, то возвращает модифицированную версию адреса, указанного в <code>url</code> . Иначе возвращает немодифицированный <code>url</code> . Этим методом должны быть обработаны все URL-адреса, передаваемые в метод <code>sendRedirect()</code>
<code>void sendError(int c) throws IOException</code>	Посыпает клиенту код ошибки, указанный в параметре <code>c</code>
<code>void sendError(int c, String s) throws IOException</code>	Посыпает клиенту код ошибки, указанный в параметре <code>c</code> , и сообщение, указанное в <code>s</code>
<code>void sendRedirect(String url) throws IOException</code>	Перенаправляет клиента к адресу, указанному в <code>url</code>
<code>void setDateHeader(String field, long msec)</code>	Добавляет к заголовку поле, указанное в параметре <code>field</code> , со значением даты, равным величине, указанной в параметре <code>msec</code> (в миллисекундах, начиная с полуночи 1 января 1970 г., GMT)
<code>void setHeader(String field, String value)</code>	Добавляет (в ответ) заголовок (в виде поля с именем, указанным в параметре <code>field</code> , со значением, указанным в параметре <code>value</code> )
<code>void setIntHeader(String field, int value)</code>	Добавляет (в ответ) внутренний заголовок (в виде поля с именем, указанным в параметре <code>field</code> , со значением, указанным в параметре <code>value</code> )
<code>void setStatus(int code)</code>	Устанавливает код состояния данного ответа (со значением, указанным в параметре <code>code</code> )

**Таблица 27.11 (окончание)**

Метод	Описание
<code>void setStatus(int code, String s)</code>	Устанавливает код состояния и сообщение данного ответа (со значениями, указанными в параметрах <code>code</code> и <code>s</code> )

## Интерфейс HttpSession

Интерфейс HttpSession реализуется сервером. Он дает возможность сервлету читать и записывать информацию о состоянии, которая связана с сеансом HTTP. Его методы кратко описаны в табл. 27.12. Все эти методы выбрасывают исключение IllegalStateException, если сеанс был уже завершен.

**Таблица 27.12. Методы, определенные в HttpSession**

Метод	Описание
<code>long getCreationTime()</code>	Возвращает время создания сеанса (в миллисекундах, начиная с полуночи 1 января 1970 г., GMT)
<code>String getId()</code>	Возвращает ID (идентификатор) сеанса
<code>long getLastAccessedTime()</code>	Возвращает время последнего запроса клиента в этом сеансе (в миллисекундах, начиная с полуночи 1 января 1970 г., GMT)
<code>HttpSessionContext getSessionContext()</code>	Возвращает контекст, связанный с этим сеансом
<code>Object getValue(String name)</code>	Возвращает объект, связанный с именем, получаемым через параметр <code>name</code> . Возвращает null, если нет такой связи
<code>String[] getValueNames()</code>	Возвращает имена всех объектов, которые связаны с сеансом
<code>void invalidate()</code>	Завершает данный сеанс и удаляет его из контекста
<code>boolean isNew()</code>	Возвращает true, если сервер создал сеанс, но он еще не был доступен клиенту

Таблица 27.12 (окончание)

Метод	Описание
<code>void putValue(String name, Object obj)</code>	Связывает объект <code>obj</code> с именем <code>name</code> в данном сеансе
<code>void removeValue(String name)</code>	Удаляет из сеанса объект, связанный с именем через <code>name</code>

## Интерфейс *HttpSessionBindingListener*

Интерфейс *HttpSessionBindingListener* реализуется объектами, которые нуждаются в уведомлении о наличии или отсутствии связи с сеансом HTTP. Методы, которые вызываются для установления наличия или отсутствия связи объекта с сеансом, имеют следующие сигнатуры:

```
void valueBound(HttpSessionBindingEvent e)
void valueUnbound(HttpSessionBindingEvent e)
```

Здесь `e` — event-объект (объект события), который описывает событие связывания с сеансом.

## Интерфейс *HttpSessionContext*

Интерфейс *HttpSessionContext* реализуется сервером. Он дает возможность сервлету получить доступ к сеансу, который с ним связан. В нем объявлен метод, который возвращает перечисление всех сеансовых идентификаторов контекста. Его сигнатура:

```
Enumeration getIds()
```

Другой метод может отображать ID сеанса на объект класса *HttpSession*. Сигнатура этого метода имеет следующую форму:

```
HttpSession getSession(String id)
```

Здесь `id` — идентификатор (ID) сеанса.

## Класс *Cookie*

Класс *Cookie* инкапсулирует cookie<sup>1</sup>-элемент (или просто — cookie). Cookie-элементы хранятся на машине клиента и обычно содержат информацию со-

<sup>1</sup> Cookie — в Internet — небольшой фрагмент данных (имя и значение которого определяет разработчик сервлетов), автоматически создаваемый сервером на машине пользователя. Дословный перевод этого термина — *домашнее печенье, булочка*, а произносится он как "куки". В силу краткости и выразительности, а также из-за вхождения этого термина в имена классов, интерфейсов и методов, термин оставлен без перевода. — Примеч. пер.

стояния. Кстати, они полезны для прослеживания активности пользователя. Например, предположим, что пользователь посещает сетевой книжный магазин. Cookie-элемент может сохранять имя пользователя, его адрес и другую информацию о пользователе. Пользователю не нужно вводить эти данные каждый раз, когда он посещает этот магазин.

Сервлет может записывать cookie-элемент на машину пользователя с помощью метода `addCookie()` интерфейса `httpServletResponse`. Данные этого cookie-элемента включаются в заголовок HTTP-ответа, который посыпается браузеру клиента.

Имена и значения cookie-элементов сохраняются на машине пользователя. Cookie-элемент может содержать, например, следующую информацию:

- имя;                    дату хранения;
- значение;        домен и путь.

*Дата хранения* определяет, когда данный cookie-элемент нужно удалять с машины пользователя. Если дата хранения явно не назначена для cookie-элемента, он удаляется, когда заканчивается текущий сеанс браузера. В противном случае cookie-элемент сохраняется в дисковом файле на машине пользователя.

*Домен и путь* cookie-элемента определяют, когда он включается в заголовок HTTP-запроса. Если пользователь вводит URL, чей домен и путь соответствуют этим значениям, тогда cookie-элемент доставляются к Web-серверу. Иначе — нет.

Единственный конструктор для `Cookie`-объекта имеет следующую сигнатуру:

`Cookie(String name, String value)`

Здесь `name` и `value` — параметры, поставляющие конструктору имя и значение cookie-элемента. Методы класса `Cookie` кратко описаны в табл. 27.13.

**Таблица 27.13. Методы, определенные `Cookie`**

Метод	Описание
<code>Object clone()</code>	Возвращает копию этого объекта
<code>String getComment()</code>	Возвращает комментарий
<code>String getDomain()</code>	Возвращает домен
<code>int getMaxAge()</code>	Возвращает максимальный срок хранения (в секундах)
<code>String getName()</code>	Возвращает имя
<code>String getPath()</code>	Возвращает путь

Таблица 27.13 (окончание)

Метод	Описание
<code>boolean getSecure()</code>	Возвращает <code>true</code> , если cookie-элемент защищен. Иначе, возвращает <code>false</code>
<code>String getValue()</code>	Возвращает значение
<code>int getVersion()</code>	Возвращает версию
<code>void setComment(String c)</code>	Устанавливает комментарий, указанный в <code>c</code>
<code>void setDomain(String d)</code>	Устанавливает домен, указанный в <code>d</code>
<code>void setMaxAge(int secs)</code>	Устанавливает максимальное время хранения cookie-элемента, указанное в <code>secs</code> . Это число секунд, после которого cookie удаляется
<code>void setPath(String p)</code>	Устанавливает путь, указанный через параметр <code>p</code>
<code>void setSecure(boolean secure)</code>	Устанавливает флагок защиты, как указано в <code>secure</code>
<code>void setValue(String v)</code>	Устанавливает значение, указанное в <code>v</code>
<code>void setVersion(int v)</code>	Устанавливает версию, указанную в <code>v</code>

## Класс *HttpServlet*

Класс *HttpServlet* расширяет *GenericServlet*. Он обычно используется при разработке сервлетов, которые принимают и обрабатывают HTTP-запросы. Методы класса *HttpServlet* кратко описаны в табл. 27.14.

Таблица 27.14. Методы, определенные в *HttpServlet*

Метод	Description
<code>void doDelete(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</code>	Выполняет DELETE-запрос HTTP
<code>void doGet(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</code>	Выполняет GET-запрос HTTP
<code>void doOptions(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</code>	Выполняет OPTIONS-запрос HTTP

Таблица 27.14 (окончание)

Метод	Description
<code>void doPost(HttpServletRequest req,               HttpServletResponse res) throws IOException, ServletException</code>	Выполняет POST-запрос HTTP
<code>void doPut(HttpServletRequest req,               HttpServletResponse res) throws IOException, ServletException</code>	Выполняет PUT-запрос HTTP
<code>void doTrace(HttpServletRequest req,               HttpServletResponse res) throws IOException, ServletException</code>	Выполняет TRACE-запрос HTTP
<code>long getLastModified(HttpServletRequest req)</code>	Возвращает время (в миллисекундах с полуночи 1 января 1970 г., UTC), когда запрошенный ресурс был последний раз модифицирован
<code>void service(HttpServletRequest req,               HttpServletResponse res) throws IOException, ServletException</code>	Вызывается сервером, когда HTTP-запрос прибывает для этого сервлета. Параметры обеспечивают доступ к HTTP-запросу и ответу соответственно

## Класс *HttpSessionBindingEvent*

Класс расширяет *EventObject*. Его объект генерируется, когда организуется или ликвидируется связь блока прослушивания с объектом класса *HttpSession*. *HttpSessionBindingEvent* имеет следующий конструктор:

```
HttpSessionBindingEvent(HttpSession session, String name)
```

Здесь *session* — сеанс-источник события, а *name* — имя, связанное с объектом, для которого организуется или ликвидируется связь.

Метод *getName()* получает имя объекта, для которого устанавливается или ликвидируется связь. Его сигнатура имеет вид:

```
String getName()
```

Метод *getSession()* получает сеанс (точнее сеансовый объект), с которым слушатель (объект типа *EventListener*) устанавливает или ликвидирует связь. Его сигнатура:

```
HttpSession getSession()
```

## Класс *HttpUtils*

Класс *HttpUtils* обеспечивает три статических метода, которые являются полезными для серверт-разработчиков (табл. 27.15).

**Таблица 27.15.** Методы класса *HttpUtils*

Метод	Описание
<code>static StringBuffer getRequestURL(HttpServletRequest req)</code>	Возвращает URL, который был выдан клиентом
<code>static Hashtable parsepostData(int size, ServletInputStream sis)</code>	Возвращает хэш-таблицу, которая содержит пары ключ/значение. Этот метод полезен для синтаксического анализа HTTP-форм, которые подчиняются POST-методу. Первый параметр — число байтов во входном потоке, второй — сам входной поток
<code>static Hashtable parseQueryString(String s)</code>	Анализирует строку запроса и возвращает хэш-таблицу, содержащую пары ключ/значение

## Обработка запросов и ответов HTTP

Класс *HttpServlet* обеспечивает специализированные методы, которые обрабатывают различные типы HTTP-запросов. Сервлет-разработчик обычно переопределяет один из этих методов. Это методы *doDelete()*, *doGet()*, *doOptions()*, *doPost()*, *doPut()* и *doTrace()*. Полное описание различных типов HTTP-запросов — за пределами возможностей данной книги. Однако при обработке ввода для форм обычно используются методы GET и POST. Поэтому текущий раздел представляет примеры этих случаев. Чтобы больше узнать об HTTP, проконсультируйтесь с RFC 2068 на сайте фирмы Internet Engineering Task Force (<http://www.ietf.org>).

## Обработка GET-запросов HTTP

Этот раздел демонстрирует сервлет, который обрабатывает GET-запрос HTTP. Сервлет вызывается, когда на рассмотрение предъявляется форма на Web-странице.

Пример содержит два файла: *ColorGet.htm* — определяет Web-страницу, и *ColorGetServlet.java* — определяет сервлет. Исходный текст HTML (файл

ColorGet.htm) показан в следующем листинге. Он определяет форму, которая содержит элемент выбора и предъявляющую (submit) кнопку. Обратите внимание, что action-параметр тела формы (<form>) указывает URL, который идентифицирует сервлет для обработки GET-запроса HTTP.

```
<html>
<body>
<center>
<form name="Form1"
      action="http://localhost:8080/servlet/ColorGetServlet">
<B>Color:</B>
<select name="color" size="1">
<option value="Red">Red</option>
<option value="Green">Green</option>
<option value="Blue">Blue</option>
</select>
<br><br>
<input type=submit value="Submit">
</form>
</body>
</html>
```

Ниже показан исходный код для ColorGetServlet.java. Метод doGet() переопределен так, чтобы обрабатывать любые GET-запросы HTTP, которые посылаются этому сервлету. Чтобы получить выбор, который был сделан пользователем, используется метод getParameter() из HttpServletRequest. Затем формируется ответ.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ColorGetServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

        String color = request.getParameter("color");
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>The selected color is: " );
        pw.println(color);
        pw.close();
    }
}
```

Откомпилируйте сервлет, и затем исполните следующие шаги для проверки этого примера:

1. Запустите servletrunner.
2. Отобразите в браузере Web-страницу.
3. Выберите цвет.
4. Предъявите Web-страницу.

После завершения этих шагов браузер отображает ответ, который динамически генерируется сервлетом.

Одна тонкость: параметры для GET-запроса HTTP включены как часть URL-адреса, который посылается Web-серверу. Предположим, что пользователь выбирает red-опцию и предъявляет форму. Тогда URL, отправленный браузером на сервер, имеет вид:

`http://localhost:8080/servlet/ColorGetServlet?color=Red`

Символы справа от вопросительного знака известны как *строка запроса* (query string).

## Обработка POST-запросов HTTP

В этом разделе разрабатывается сервлет, который обрабатывает POST-запрос HTTP. Сервлет вызывается, когда на Web-странице предъявляется форма. Пример содержит два файла: ColorPost.htm — определяет Web-страницу, и ColorPostServlet.java — определяет сервлет.

Исходный текст HTML (ColorPost.htm) показан в следующем листинге. Он идентичен ColorGet.htm за исключением того, что параметр `method` для `<form>`-тега явно определяет использование POST-метода, а параметр `action` для того же тега определяет другой сервлет.

```
<html>
  <body>
    <center>
      <form name="Form1"
            method="post"
            action="http://localhost:8080/servlet/ColorPostServlet">
        <B>Color:</B>
        <select name="color" size="1">
          <option value="Red">Red</option>
          <option value="Green">Green</option>
          <option value="Blue">Blue</option>
        </select>
        <br><br>
        <input type=submit value="Submit">
      </form>
    </body>
  </html>
```

В следующем листинге показан исходный код ColorPostServlet.java. Метод doPost() переопределен для обработки любых POST-запросов HTTP, которые посылаются этому сервлету. Чтобы получить выбор, который был сделан пользователем, он использует метод getParameter() из HttpServletRequest. Затем формируется ответ.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ColorPostServlet extends HttpServlet {

    public void doPost(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException {
        String color = request.getParameter("color");
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>The selected color is: " );
        pw.println(color);
        pw.close();
    }
}
```

Откомпилируйте сервлет и затем испытайте его, выполняя те же шаги, что описаны в предыдущем разделе.

Обратите внимание, что параметры для POST-запроса HTTP не включены как часть URL-адреса, который посылается Web-серверу. В этом примере URL, посылаемый из браузера на сервер, выглядит так:

<http://localhost:8080/servlert/ColorGetServlet>

Имена и значения параметров посылаются в теле запроса HTTP.

## Использование cookie-данных

Теперь разработаем сервлет, который демонстрирует использование cookie-элементов. Сервлет вызывается, когда на Web-странице предъявляется форма. Пример включает три файла:

- AddCookie.htm. Дает возможность пользователю определить значение для cookie-элемента с именем MyCookie.
- AddCookieServlet.java. Обрабатывает предъявление из AddCookie.htm.
- GetCookiesServlet.java. Отображает cookie-значения.

Исходный текст HTML (AddCookie.htm) показан в следующем листинге. Эта страница содержит текстовое поле, в которое может быть введено некоторое

значение. Страница также включает предъявляющую кнопку. Когда эта кнопка нажимается, значение из текстового поля посыпается в AddCookieServlet через POST-запрос HTTP.

```
<html>
  <body>
    <center>
      <form name="Form1"
            method="post"
            action="http://localhost:8080/servlet/AddCookieServlet">
        <B>Enter a value for MyCookie:</B>
        <input type=textbox name="data" size=25 value="">
        <input type=submit value="Submit">
      </form>
    </body>
</html>
```

Эта страница содержит текстовое поле, в которое может быть введено некоторое значение. Страница также включает предъявляющую кнопку. Когда эта кнопка нажимается, значение из текстового поля посыпается в AddCookieServlet через POST-запрос HTTP.

Исходный код AddCookieServlet.java показан в следующем листинге:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class AddCookieServlet extends HttpServlet {

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        // получить параметр из запроса HTTP
        String data = request.getParameter("data");

        // создать cookie
        Cookie cookie = new Cookie("MyCookie", data);

        // добавить cookie к HTTP-запросу
        response.addCookie(cookie);

        // записать вывод в браузер
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>MyCookie has been set to");
        pw.println(data);
        pw.close();
    }
}
```

Он получает значение параметра, названного "data". Затем он создает Cookie-объект, который имеет имя "MyCookie" и содержит значение параметра "data". Далее, cookie добавляется к заголовку HTTP-ответа с помощью метода addCookie(). Потом в браузер записывается сообщение обратной связи.

Исходный код GetCookiesServlet.java показан в следующем листинге. Он вызывает метод getCookies() для чтения любых cookie-данных, которые включены в GET-запрос HTTP. Затем имена и значения этих cookie-данных записываются в ответ HTTP. Чтобы получить эту информацию, вызываются методы getName() и getValue().

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class GetCookiesServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        // Получить cookie-элементы из заголовка запроса HTTP
        Cookie[] cookies = request.getCookies();

        // Показать эти cookie-элементы
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>");
        for(int i = 0; i < cookies.length; i++) {
            String name = cookies[i].getName();
            String value = cookies[i].getValue();
            pw.println("name = " + name + "; value = " + value);
        }
        pw.close();
    }
}
```

Он вызывает метод getCookies() для чтения любых cookie-данных, которые включены в GET-запрос HTTP. Затем имена и значения этих cookie-данных записываются в ответ HTTP. Для получения этой информации вызываются методы getName() и getValue().

Откомпилируйте сервлет и выполните следующие шаги:

1. Запустите servletrunner.
2. Отобразите AddCookie.htm в браузер.
3. Введите значение для MyCookie.
4. Предъявите Web-страницу.

После завершения этих шагов понаблюдайте за сообщением обратной связи, которое отображается браузером.

Затем, запросите следующий URL через браузер:

`http://localhost:8080/servlet/GetCookiesServlet`

Убедитесь, что имя и значение cookie-элемента отображаются в браузере.

В этом примере срок хранения явно не назначен cookie-данным через метод `setMaxAge()` из объекта типа `Cookie`. Поэтому данный срок истекает, когда сеанс браузера заканчивается. Экспериментируя с этим методом, вы можете убедиться, что cookie-данные сохраняются на диске машины клиента.

## Прослеживание сеанса

HTTP — протокол “без состояния” (stateless). Это означает, что каждый запрос независим от предыдущего. Однако в некоторых приложениях необходимо сохранить информацию о состоянии так, чтобы можно было собрать информацию от нескольких взаимодействий между браузером и сервером. Такой механизм обеспечивают *сеансы* (sessions).

Сеанс может быть создан с помощью метода `getSession()` из `HttpServletRequest`. Он возвращает объект `HttpSession`. Данный объект может хранить набор связей, которые ассоциируют имена с объектами. Этими связями управляют методы `putValue()`, `getValue()`, `getValueNames()` и `removeValue()` класса `HttpSession`. Важно обратить внимание, что состояние сеанса разделяется всеми сервлетами, которые связаны со специфическим клиентом.

Следующий сервлет иллюстрирует использование состояния сеанса. Метод `getSession()` получает текущий сеанс. Новый сеанс создается, если он еще не существует. Чтобы получить объект, который связан с именем “date”, вызывается метод `getValue()`. Это объект `Date`, который инкапсулирует дату и время последнего обращения к данной странице. (Конечно, нет никакой связи с первым обращением к странице.) Затем создается объект `Date`, инкапсулирующий текущую дату и время. Наконец, вызывается метод `putValue()`, чтобы связать имя “date” с этим объектом.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DateServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

```

```
// получить HttpSession-объект
HttpSession hs = request.getSession(true);

// получить объект PrintWriter
response.setContentType("text/html");
PrintWriter pw = response.getWriter();
pw.print("<B>");

// показать дату/время последнего доступа
Date date = (Date)hs.getValue("date");
if(date != null) {
    pw.print("Last access: " + date + "<br>");
}

// показать текущую дату/время
date = new Date();
hs.putValue("date", date);
pw.println("Current date: " + date);
}
}
```

Когда вы впервые запрашиваете этот сервлет, браузер отображает одну строку с текущей датой и временем. При последующих вызовах отображаются две строки. Первая строка показывает дату и время последнего обращения к сервлету. Вторая строка — текущую дату и время.

## Проблемы защиты

В предыдущих главах этой книги вы узнали, что ненадежные апплеты вынуждены работать в "песочнице"<sup>1</sup> (sandbox). Они не могут выполнять операции, которые являются потенциально опасными для машины пользователя, включая чтение и запись файлов, открытие сокетов к произвольным машинам, вызов native<sup>2</sup>-методов и создание новых процессов. Применяются и другие ограничения.

Подобные ограничения существуют также для ненадежных сервлетов. Код, который загружается из удаленной машины, ненадежен. Однако *надежные сервлеты*, загружаемые из локальной машины, не имеют подобных ограничений.

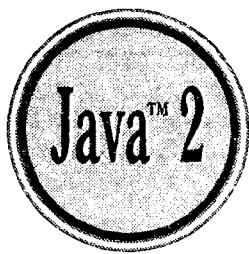
<sup>1</sup> Sandboxing — механизм обеспечения безопасности подкачанных из сети или полученных по электронной почте программ, предусматривающий изоляцию на время выполнения загружаемого кода в ограниченную среду — "песочницу" (sandbox). — *Примеч. пер.*

<sup>2</sup> Native — "родной", зависящий от платформы. — *Примеч. пер.*

## Исследование сервлетов

В данной главе приведено только краткое введение в возможности сервлетов. Консультируйтесь с сетевой документацией фирмы Sun, чтобы больше узнать о `servletrunner` и различных Web-серверах, которые поддерживают Servlet API.

Одним из наиболее общих приложений сервлетов является обращение к базе данных и динамическое создание ответов HTTP из найденной информации. Для этой цели можно использовать Java Database Connectivity (JDBC) API. Знание его функциональных возможностей полезно для формирования организационных приложений. Это область, которую вы можете исследовать самостоятельно.



## ГЛАВА 28

# Миграция из C++ в Java

В этой главе обсуждаются несколько проблем, возникающих при переходе от C++ к Java. Поскольку многие (если не большинство) Java-программисты приходят из среды C++, естественно желание перенести в Java навыки, методики и даже способы кодирования, приобретенные на этом языке. Хотя C++ и Java были разработаны для удовлетворения потребности программирования двух очень различных сред, многие методы кодирования, алгоритмы и оптимизации применимы к обоим языкам. Однако, как объяснялось в Части I, Java не является "Internet-версией C++". В то время как существует много подобий между этими двумя языками, имеется также и несколько различий. В данной главе сделан обзор этих различий и показано, как можно справиться с наиболее сложными из них.

## Различия между C++ и Java

Прежде чем рассматривать специфические ситуации, сделаем обзор основных различий между C++ и Java. Их можно разделить на три категории:

- Свойства C++, которые в Java не поддерживаются.
- Свойства, уникальные для Java.
- Общие свойства, которые различаются в C++ и Java.

## Что Java исключил из C++

Существует ряд свойств C++, которые Java не поддерживает. В некоторых случаях специфические свойства C++ просто не относятся к среде Java. В других ситуациях проектировщики Java устранили некоторые из дубликатов свойств, которые есть в C++. В остальных случаях то или иное свойство C++ не поддерживается в Java из-за того, что оно считалось слишком опасным для Internet-апплетов.

Возможно, единственное самое большое различие между Java и C++ заключается в том, что Java не поддерживает указатели. Как программист C++, вы знаете, что указатель — это один из наиболее мощных и важных средств языка C++. Но он также и одно из его наиболее опасных средств при неправильном использовании. Указатели не существуют в Java по двум причинам.

- Указатели опасны по своей сути. Например, используя указатель в стиле C++, можно получить доступ к адресам памяти вне кода и данных программы. Злонамеренная программа могла бы использовать этот факт, чтобы повредить систему, выполнять неправомочные доступы (типа получения паролей), или как-то иначе нарушать ограничения защиты.
- Если бы даже указатели были ограничены исполнительной системой (что теоретически возможно, т. к. программы Java *интерпретируются*), разработчики Java полагали, что они опасны.

### Замечание

Так как указатели не существуют в Java, не используйте операцию доступа по указателю (->).

Ниже перечислено несколько наиболее важных исключений.

- Java не включает структуру или объединений. Они показались избыточными, т. к. классы их полностью заменяют.
- Java не поддерживает перегрузки (overloading) операций. Данный механизм иногда является источником неоднозначности в программе C++, и группа разработчиков Java чувствовала, что она вызывает больше неприятностей, чем выгод.
- Java не содержит препроцессора и не поддерживает его директивы. В C++ препроцессор играет менее важную роль, чем в С. Разработчики Java решили, что пришло время устраниć его полностью.
- Java не выполняет никаких автоматических преобразований типов, которые приводят к потере точности. Например, преобразование длинного целого (`long`) в целый (`int`) тип должно быть выполнено явным приведением типов (`cast`).
- Весь код в программе Java инкапсулирован в одном или большем количестве классов. Поэтому в Java нет ничего похожего на глобальные переменные или глобальные функции.
- Java не допускает умалчиваемых аргументов. В C++ можно определить значение, которое будет принимать параметр, при вызове функции без аргумента, соответствующего этому параметру. В Java подобная ситуация не допускается.
- Java не поддерживает наследования подклассом множественных суперклассов.

- Хотя Java поддерживает конструкторы, он не имеет деструкторов. Однако в нем добавлена функция завершения `finalize()`.
- Java не поддерживает спецификатор `typedef`.
- В Java невозможно объявить целый тип без знака (`unsigned`).
- Java не допускает оператора `goto`.
- Java не имеет операции `delete`.
- Операции `<<` и `>>` в Java не перегружаются для операций ввода/вывода.
- Объекты в Java пересыпаются только *по ссылке*. В C++ объекты можно пересыпать *по значению* или по ссылке.

## Новые свойства, добавленные в Java

В Java существует ряд свойств, которые не имеют эквивалентов в C++. Вероятно, три из наиболее важных — это многопоточность (параллельные процессы), пакеты и интерфейсы, но имеется и несколько других, которые также обогащают среду Java-программирования.

- Как объяснялось в главе 11, многопоточность позволяет двум или большему количеству частей одной и той же программы выполняться одновременно. Кроме того, данный подход к параллелизму поддерживается на уровне языка. В C++ нет никакой параллели для этого свойства. Если необходимо распараллелить программу C++, то это нужно будет делать вручную, используя функции операционной системы. Хотя оба метода допускают конкурентное выполнение двух или нескольких потоков, подход Java яснее и легче для использования.
- В C++ нет свойства, которое прямо соответствует *пакету* Java. Самое близкое подобие — набор библиотечных функций, которые содержат общий файл заголовка. Однако создание и использование библиотеки в C++ полностью отличается от построения и применения пакета в Java.
- Интерфейс (*interface*) Java в чем-то подобен абстрактному классу в C++. (*Абстрактный класс* в C++ — это класс, который содержит, по крайней мере, одну чистую виртуальную функцию.) Невозможно, например, создать экземпляр абстрактного класса C++ или Java-интерфейса. Оба используются для определения непротиворечивого интерфейса, который реализуется в подклассах. Главное различие в том, что интерфейс более ясно представляет эту концепцию.
- Java имеет упрощенный подход к распределению памяти. Подобно C++ он поддерживает ключевое слово `new`, однако не имеет операции `delete`. Вместо этого, когда последняя ссылка на объект разрушается, сам объект автоматически удаляется последующей сборкой "мусора".
- В Java "удалена" стандартная библиотека C++ и заменена собственным набором классов API. В то время как существует заметное функциональ-

ное подобие, есть существенные различия в именах и параметрах. Кроме того, т. к. вся API-библиотека Java объектно-ориентирована, а библиотека C++ — только частично, имеются различия в способе вызова библиотечных подпрограмм.

- В Java были усовершенствованы операторы `break` и `continue` (они теперь могут иметь метки в качестве адресующих параметров).
- Тип `char` в Java использует 16-разрядные символы Unicode. Это делает его подобным C++ типу `wchar_t`. Использование Unicode обеспечивает мобильность (переносимость) Java-кода.
- В Java добавлена операция `>>>`, которая выполняет правый беззнаковый сдвиг..
- В дополнение к поддержке однострочных и многострочных комментариев, Java добавляет третью форму — *документационный комментарий* (*documentation comment*). Документационные комментарии начинаются последовательностью символов `/**` и заканчиваются последовательностью `*/`.
- Java содержит встроенный строчный тип с именем `String`. Тип `String` в чем-то подобен стандартному классу `string`, поддерживаемому в C++. Конечно, `string`-тип в C++ доступен только тогда, когда вы включаете в свою программу объявление его класса, т. е. это не встроенный тип.

## Отличающиеся свойства

Имеются некоторые свойства, общие для C++ и Java, которыми каждый язык управляет немного по-разному.

- В то время как в C++ и Java поддерживают булевский тип данных, Java не реализует `true`- и `false`-значения таким же образом, как C++. В C++ `true`-значение — это любое значение, отличное от нуля, а `false`-значение — это нуль. В Java, `true` и `false` — это предопределенные литералы, и они являются единственными значениями, которые может иметь булево выражение. В то время как C++ тоже определяет `true` и `false`, которые можно назначить `bool`<sup>1</sup>-переменной, C++ автоматически конвертирует ненулевые значения в `true`, а нулевые значения — в `false`. Этого не происходит в Java.
- Когда вы создаете класс C++, спецификаторы доступа применяются к группам утверждений. В Java спецификаторы доступа применяются только к объявлению, которым они непосредственно предшествуют.

---

<sup>1</sup> Авторы не точны: спецификатор булевского типа `bool` или `boolean` в C++ отсутствует. — Примеч. пер.

- C++ поддерживает обработку исключений, которая очень похожа на обработку в Java. Однако в C++ нет требования, чтобы выброшенное исключение было выловлено.

С этими добавлениями, исключениями и различиями в качестве фона, в остальной части главы будут подробно рассмотрены некоторые ключевые вопросы, с которыми вы столкнетесь при преобразовании кода из C++ в Java.

## Исключение указателей в C++

Когда вы конвертируете программу из C++ в Java, вероятно, самое большое число изменений будет вызвано указателями. Немалая часть кода сильно зависит в своей работе именно от них. В C++ вы не можете программировать нечто очень существенное без использования указателя.

Существует четыре общие категории указателей, с которыми вы сталкиваетесь в коде C++.

- Указатели, используемые в качестве параметров функций. Хотя C++ поддерживает ссылочный параметр, имеется большая база унаследованного кода, который был первоначально написан на С. Однако С не поддерживает ссылочных параметров. В С, если функция должна изменить значение аргумента, необходимо явно передать указатель этому аргументу. Поэтому в коде C++ еще можно найти параметры типа указателя, которые были первоначально перенесены из С. Кроме того, в некоторых случаях одна и та же функциональная библиотека должна одновременно использоваться как в С, так и в коде C++, что не допускает использования ссылочных параметров.
- Многие из стандартных библиотечных функций, поддерживаемых C++, унаследованы из С. Когда одна из этих С-функций требует адрес аргумента, используется указатель на аргумент. К такому аргументу внутри функции обращаются через его указатель.
- Указатели, используемые в качестве более эффективных средств реализации некоторых конструкций — особенно для индексации массивов. Например, часто более эффективно последовательно двигаться через массив, используя указатель, а не индекс массива. В то время как современные компиляторы реализуют высокоеффективную оптимизацию, указатели все еще могут обеспечивать существенное увеличение производительности. Таким образом, использование указателей для доступа к массивам — повсеместно в коде C++.
- Указатели, используемые для поддержки распределения памяти. В C++ при распределении памяти возвращается адрес (т. е. указатель) ячейки памяти. Адрес должен быть назначен переменной типа указателя. Как только это сделано, указатель может указывать на любую часть распределенной памяти (или куда-нибудь еще) с помощью арифметических опе-

раций над указателями. В Java, когда объект распределяется (в памяти) операцией `new`, возвращается ссылка на объект. Эта ссылка должна быть назначена ссылочной переменной совместимого типа. В Java ссылочные переменные, неявно указывающие на объект, который был распределен операцией `new`, не могут управляться таким же образом, как указатели C++. И они не могут указывать на память вне контекста времени выполнения.

- Указатели, используемые для обеспечения доступа к любому произвольному машинному адресу, возможно, чтобы вызвать подпрограмму ROM или читать/записывать прямо в память. Так как Java преднамеренно отвергает такие действия, это использование указателей не имеет никакой прямой параллели. Если вы пишете приложения, а не апплеты, то вы всегда можете использовать "родные" (native) возможности Java (описанные в первой части книги), чтобы получить доступ к подпрограммам "родного" кода, которые разрешили бы доступ к таким системным ресурсам.

Рассмотрим две ситуации, в которых использующий указатели C++ код преобразуется в код Java.

## Преобразование параметров типа указателя

Конвертировать функцию C++, которая использует параметры типа указателя (pointer parameters), в эквивалентный Java-метод — достаточно просто. Так как Java передает все объекты ссылкой, преобразование иногда просто требует удаления операций C++ над указателями. Например, рассмотрим следующую программу на C++, которая меняет знаки `Coord`-объекта, хранящего пару декартовых координат. Функция `reverseSign()` передается указатель на `Coord`-объект, который будет изменяться. Чтобы выполнить нужные действия, используются операции C++ (над указателями) `*`, `&` и `->`.

```
// Реверсирование знаков координат — версия C++.
#include <iostream>
using namespace std;

class Coord {
public:
    int x;
    int y;
};

// реверсировать знак координат
void reverseSign(Coord *ob) {
    ob->x = -ob->x;
    ob->y = -ob->y;
}
```

```
int main()
{
    Coord ob;

    ob.x = 10;
    ob.y = 20;

    cout << "Исходные значения для ob: ";
    cout << ob.x << ", " << ob.y << "\n";

    reverseSign(&ob);

    cout << "Значения с реверсированным знаком для ob: ";
    cout << ob.x << ", " << ob.y << "\n";

    return 0;
}
```

Эта программа может быть перекодирована в следующую Java-версию. Большинство преобразований состоит просто в удалении операций C++ над указателями. Поскольку Java передает объекты ссылкой, изменения параметров автоматически влияют на аргумент.

```
// Реверсирование знаков координаты - Java-версия.
class Coord {
    int x;
    int y;
};

class DropPointers {
    // реверсировать знак координат
    static void reverseCoord(Coord ob) {
        ob.x = -ob.x;
        ob.y = -ob.y;
    }
}

public static void main(String args[]) {
    Coord ob = new Coord();

    ob.x = 10;
    ob.y = 20;

    System.out.println("Исходные значения для ob: " +
        ob.x + ", " + ob.y);

    reverseCoord(ob);

    System.out.println("Значения с реверсированным знаком для ob: " +
        ob.x + ", " + ob.y);
}
```

Вывод этих двух программ — один и тот же:

Исходные значения для ob: 10, 20

Значения с реверсированным знаком для ob: -10, -20

## Преобразование указателей, работающих на массивах

Концептуально, преобразование доступа к массиву через указатели C++ в эквивалентную Java-индексацию массива выполняется прямо — простой заменой утверждений, индексирующих массив. Однако на практике это может потребовать некоторых усилий. Доступу к массиву через указатели может быть, немного труднее следовать, т. к. нормальный стиль C++ программирования поощряет довольно плотные, сложные выражения. Например, следующая короткая программа C++ копирует содержимое одного массива в другой. В ней используется 0 (числовой нуль) для отметки концов массивов. Обратите особое внимание на выражения с указателями. Даже в этом простом примере, если бы вы не знали, что эта программа скопировала содержимое nums в copy (и позже отобразила массивы на экране), потребуется довольно тщательное обдумывание, прежде чем вы полностью убедитесь, что понимаете, как работает код.

```
// Копирование массива в C++ с использованием указателей.
#include <iostream>
using namespace std;

int main()
{
    int nums[] = {10, 12, 24, 45, 23, 19, 44,
                  88, 99, 65, 76, 12, 89, 0};
    int copy[20];

    int *p1, *p2; // целые указатели

    // копировать массив
    p1 = nums; // p1 указывает на начало массива nums
    p2 = copy;
    while(*p1) *p2++ = *p1++;
    *p2 = 0; // завершить копирование нулем

    // отобразить содержимое каждого массива
    cout << "Это исходный массив:\n";
    p1 = nums;
    while(*p1) cout << *p1++ << " ";
    cout << endl;

    cout << "Это копия массива:\n";
    p1 = copy;
```

```
while(*p1) cout << *p1++ << " ";
cout << endl;

return 0;
}
```

Даже при том, что она весьма проста для кода C++, на первый взгляд строка

```
while(*p1) *p2++ = *p1++;
```

все еще требует напряжения мысли, чтобы декодировать ее точную работу. Одно из преимуществ Java заключается в том, что он не поощряет создания таких выражений. Ниже показана Java-версия программы. Ее цель и результаты — прозрачны.

```
// Копирование массива без указателей на Java.

class CopyArray {
    public static void main(String args[]) {
        int nums[] = {10, 12, 24, 45, 23, 19, 44,
                      88, 99, 65, 76, 12, 89, 0};
        int copy[] = new int[14];

        int i;

        // копировать массив
        for(i=0; nums[i]!=0; i++)
            copy[i] = nums[i];
        nums[i] = 0;                                // завершить копию нулем

        // Отобразить содержимое каждого массива.
        System.out.println("Это исходный массив:");
        for(i=0; nums[i]!=0; i++)
            System.out.print(nums[i] + " ");
        System.out.println();

        System.out.println("Это копия массива:");
        for(i=0; nums[i]!=0; i++)
            System.out.print(copy[i] + " ");
        System.out.println();
    }
}
```

Обе версии программы выдают следующие результаты:

Это исходный массив:

10 12 24 45 23 19 44 88 99 65 76 12 89

Это копия массива:

10 12 24 45 23 19 44 88 99 65 76 12 89

Многие программы на C++ усеяны неясными, трудными для понимания выражениями, содержащими указатели. И хотя эти выражения имеют тен-

денцию увеличивать скорость выполнения, они являются одной из самых неприятных проблем, противостоящих поддержке программ на C++. Они также представляют определенные трудности для конвертирования кода в Java. Когда вы сталкиваетесь со сложным выражением с указателями, иногда полезно начать с его разбиения на подвыражения, чтобы его точная работа стала более ясной.

## Ссылочные параметры C++ в сравнении со ссылочными параметрами Java

В предыдущем разделе был показан пример программы на C++, которая использовала параметр-указатель (pointer parameter). В Java он становится параметром-ссылкой (reference parameter). Конечно, C++ также поддерживает параметры-ссылки. Как уже говорилось, большинство параметров-указателей в коде C++ — просто пережитки С. Когда функции нужен доступ непосредственно к аргументу, почти весь новый код C++ будет использовать параметры-ссылки. (По существу, параметры-указатели все еще и часто используются, все-таки являются фактически анахронизмами в большинстве кодов C++.) Хотя и Java и C++ поддерживают параметры-ссылки, конверсия функции C++, которая использует параметры-ссылки, в метод Java, включает несколько изменений. К сожалению, это не всегда так. Чтобы понять — почему, давайте конвертировать следующую программу на C++, которая обменивает содержимое двух Coord-объектов, используя параметры-ссылки:

```
// Обмен координат -- версия C++.
#include <iostream>
using namespace std;

class Coord {
public:
    int x;
    int y;
};

// обменять содержимое двух Coord-объектов
void swap(Coord &a, Coord &b) {
    Coord temp;

    // обменять содержимое объектов
    temp = a;
    a = b;
    b = temp;
}

int main()
{
    Coord obj1, obj2;
```

```
ob1.x = 10;
ob1.y = 20;

ob2.x = 88;
ob2.y = 99;

cout << "Исходные значения:\n";
cout << "ob1: " << ob1.x << ", " << ob1.y << "\n";
cout << "ob2: " << ob2.x << ", " << ob2.y << "\n";
cout << "\n";

swap(ob1, ob2);

cout << "Значения после обмена:\n";
cout << "ob1: " << ob1.x << ", " << ob1.y << "\n";
cout << "ob2: " << ob2.x << ", " << ob2.y << "\n";

return 0;
}
```

Вывод, выполненный этой программой (как можно видеть, содержимое `ob1` и `ob2` было обменено):

Исходные значения:

ob1: 10, 20

ob2: 88, 99

Значения после обмена:

ob1: 88, 99

ob2: 10, 20

В Java доступ ко всем объектам выполняется через объектную ссылочную переменную (*object reference variable*). Таким образом, когда объект передается методу, то передается только его ссылка. Это означает, что все объекты автоматически передаются Java-методу *по ссылке*. Не задумываясь глубже относительно того, что фактически происходит, кто-то мог бы первоначально попробовать следующее (неправильное) преобразование предыдущей программы:

```
// Программа обмена, неправильно преобразованная в Java.
class Coord {
    int x;
    int y;
};

class SwapDemo {
    static void swap(Coord a, Coord b) {
        Coord temp = new Coord();

        // здесь не будет обмена содержимого a и b!
        temp = a;
```

```
a = b;
b = temp;
}

public static void main(String args[]) {
    Coord ob1 = new Coord();
    Coord ob2 = new Coord();

    ob1.x = 10;
    ob1.y = 20;

    ob2.x = 88;
    ob2.y = 99;

    System.out.println("Исходные значения:");
    System.out.println("ob1: " + ob1.x + ", " + ob1.y);
    System.out.println("ob2: " + ob2.x + ", " + ob2.y + "\n");

    swap(ob1, ob2);

    System.out.println("Значения после обмена:");
    System.out.println("ob1: " + ob1.x + ", " + ob1.y);
    System.out.println("ob2: " + ob2.x + ", " + ob2.y + "\n");
}
}
```

Вывод, выполненный этой некорректной программой:

Исходные значения:

ob1: 10, 20

ob2: 88, 99

Значения после обмена:

ob1: 10, 20

ob2: 88, 99

Значения `ob1` и `ob2` в `main()` не были обменены. Хотя сначала это немного противоречит интуиции, но причина фактически очевидна, если только вы точно понимаете, что случается, когда объектную ссылку передают методу. Java передает все аргументы методам, используя передачу "по значению". Это означает, что делается копия аргумента, а что происходит с копией внутри метода, не имеет никакого влияния на аргумент, использованный при вызове метода. Однако эта ситуация немного расплывчата в случае объектных ссылок.

Когда объектная ссылка передается методу, делается копия ссылочной переменной, как было только что объяснено. Это означает, что параметр внутри метода будет ссылаться на тот же объект, на который ссылается и ссылочная переменная, используемая как параметр вне метода. Поэтому операции на объекте, указанном через параметр, затронут объект, указанный аргументом (т. к. это один и тот же объект). Но операции непосредственно

на ссылочном параметре, затрагивают только этот параметр. Таким образом, когда предшествующая программа пытается обменивать объекты, помеченные указателями `a` и `b`, то все, что происходит — это обмен тех объектов, на которые ссылаются параметры (точнее копии аргументов), но это не изменяет того, на что ссылаются `ob1` и `ob2` в методе `main()`.

Чтобы скорректировать программу, нужно переписать метод `swap()` так, чтобы было обменено содержимое объектов, а не то, на что параметры ссылаются. Исправленная версия `swap()`:

```
// Исправленная версия swap().  
static void swap(Coord a, Coord b) {  
    Coord temp = new Coord();  
  
    // обмен содержимого объектов  
    temp.x = a.x;  
    temp.y = a.y;  
    a.x = b.x;  
    a.y = b.y;  
    b.x = temp.x;  
    b.y = temp.y;  
}
```

Если вы замените этой версией `swap()` предшествующую программу, то будут получены правильные результаты.

## Преобразование абстрактных классов C++ в Java-интерфейсы

Одним из наиболее новаторских аспектов Java является интерфейс (*interface*). Как объяснялось ранее в этой книге, *интерфейс* определяет *форму* своих методов без определения каких-либо деталей их реализации. Каждый класс, который реализует интерфейс, создает фактические методы, объявленные интерфейсом. Таким образом, в Java интерфейс — это средство, с помощью которого можно определять общую форму класса при гарантии того, что все специфические версии класса соответствуют одному и тому же набору правил. Интерфейс — один из способов, с помощью которого Java обеспечивает поддержку полиморфизма.

В C++ нет никакой прямой параллели интерфейсу Java. Вместо этого в C++, если вы хотите определить форму класса без определения деталей реализации, нужно использовать абстрактный класс. Абстрактные классы в C++ подобны абстрактным классам в Java: они не содержат полного набора деталей реализации. В C++ абстрактный класс содержит по крайней мере одну чистую виртуальную функцию. Чистая виртуальная функция не определяет никакой реализации, а назначает только прототип функции. Таким

образом, чистая виртуальная функция в C++ — по существу то же самое, что абстрактный метод в Java. В C++ абстрактные классы обслуживаются функцию, подобно интерфейсам в Java. По этой причине они являются одним из элементов, за которыми вам нужно наблюдать при преобразовании кода в Java. Не все абстрактные классы C++ подлежат преобразованию в интерфейсы Java, но для некоторых все-таки это можно сделать. Рассмотрим два примера.

Ниже показана короткая программа на C++, которая использует абстрактный класс с именем `IntList`, определяющий форму целочисленного списка. Реализация этого класса создается классом `IntArray`, который для создания массива использует список целых чисел.

```
// Абстрактный класс C++-стиля и его реализация.
#include <iostream>
#include <cstdlib>
using namespace std;

// Абстрактный класс, который определяет форму целочисленного списка.
class IntList {
public:
    virtual int getNext() = 0;    // чистая виртуальная функция
    virtual void putOnList(int i) = 0;
};

// создать реализацию целочисленного списка
class IntArray : public IntList {
    int storage[100];
    int putIndex, getIndex;
public:
    IntArray() {
        putIndex = 0;
        getIndex = 0;
    }

    // возвратить следующее целое в список
    int getNext() {
        if(getIndex >= 100) {
            cout << "List Underflow";
            exit(1);
        }
        getIndex++;
        return storage[getIndex-1];
    }

    // поместить целое в список
    void putOnList(int i) {
        if(putIndex < 100) {
```

```
    storage[putIndex] = i;
    putIndex++;
}
else {
    cout << "Список переполнен";
    exit(1);
}
};

int main()
{
    IntArray nums;
    int i;

    for(i=0; i<10; i++) nums.putOnList(i);

    for(i=0; i<10; i++)
        cout << nums.getNext() << endl;

    return 0;
}
```

В этой программе, абстрактный класс `IntList` определяет только форму целочисленного списка. Он содержит лишь чистые виртуальные функции и не объявляет никаких данных. По этим причинам, когда программа преобразуется в код Java, его можно встроить в интерфейс:

```
// Здесь IntList встроен в интерфейс, который реализует IntArray.
// Определить интерфейс для целочисленного списка.
interface IntListIF {
    int getNext();
    void putOnList(int i);
}

// Создать реализацию целочисленного списка.
class IntArray implements IntListIF {
    private int storage[];
    private int putIndex, getIndex;

    IntArray() {
        storage = new int[100];
        putIndex = 0;
        getIndex = 0;
    }

    // Создать реализацию целочисленного списка.
    public int getNext() {
        if(getIndex >= 100) {
```

```
System.out.println("Список пуст");
System.exit(1);
}
getIndex++;
return storage[getIndex-1];
}

// Поместить целое число в список.
public void putOnList(int i) {
    if(putIndex < 100) {
        storage[putIndex] = i;
        putIndex++;
    }
    else {
        System.out.println("Список переполнен");
        System.exit(1);
    }
}

class ListDemo {
    public static void main(String args[]) {
        IntArray nums = new IntArray();
        int i;

        for(i=0; i<10; i++) nums.putOnList(i);

        for(i=0; i<10; i++)
            System.out.println(nums.getNext());
    }
}
```

Существует почти взаимно однозначное соответствие между абстрактным классом C++ `IntList`, и Java интерфейсом `IntListIF`. Можно преобразовать `IntList` в `IntListIF`, потому что первый содержал только чистые виртуальные функции. В этом суть. Если бы `IntList` содержал любые данные или реализации функций, он не был бы пригоден для преобразования в интерфейс.

Когда вы конвертируете или адаптируете код C++ в Java, ищете примеры абстрактных классов, которые содержат только чистые виртуальные функции. Они — первые кандидаты на преобразование в Java-интерфейсы. Но не пропускайте и абстрактные классы C++, которые содержат небольшое число реализованных функций или данных. Возможно, что эти элементы сначала реально не принадлежат абстрактному классу и должны быть определены индивидуальными реализациями. Так как C++ не определяет конструкцию интерфейса, у программистов C++ не было никакой причины мыслить в таких терминах.

Иногда конкретный член содержится в абстрактном классе просто из-за целесообразности, а не потому, что это для него наиболее логичное место. Например, рассмотрим следующий абстрактный класс C++:

```
// Абстрактный класс C++.
class SomeClass {
    bool isOK;
public:
    virtual int f1() = 0;
    virtual void f2(int i) = 0;
    virtual double f3() = 0;
    virtual int f4(int a, char ch) = 0;
};
```

Единственная причина, по которой данный класс не может быть встроен в Java-интерфейс, заключается в существовании переменной `isOK`. Возможно, `isOK` используется для индикации некоторого состояния, связанного с классом. Однако на самом деле нет никакой причины определять `isOK` как переменную. Вместо этого, можно определить метод, называемый `isOk()`, который возвращает состояние. При этом подходе `isOk()` будет определен, наряду с другими методами, любым реализующим классом. Таким образом, можно конвертировать предыдущий абстрактный класс C++ в следующий Java-интерфейс:

```
interface SomeClass {
    int f1();
    void f2(int i);
    double f3();
    int f4(int a, char ch);
    boolean isOk();
}
```

При переводе кода в Java многие абстрактные классы C++ могут (и должны) быть преобразованы в интерфейсы. Поступая так, вы, вероятно, найдете, что это проясняет структуру иерархии классов.

## Преобразование умалчиваемых аргументов

Одно широко используемое свойство C++, которое Java не поддерживает, — это аргументы функций, заданные по умолчанию. Например, функция `area()`, показанная в следующей программе C++, вычисляет площадь прямоугольника, если вызывается с двумя аргументами, или площадь квадрата, если вызывается с одним аргументом.

```
// Программа C++, которая использует умалчиваемые аргументы.
#include <iostream>
using namespace std;
```



```

        System.out.println("Площадь квадрата 3.0 на 3.0: " +
                           area(3.0));
    }
}

```

## Преобразование иерархий множественного наследования C++

C++ позволяет одному классу наследовать два или больше базовых класса одновременно, а Java — нет. Чтобы понять это различие, рассмотрим две иерархии, представленные на рис. 28.1.

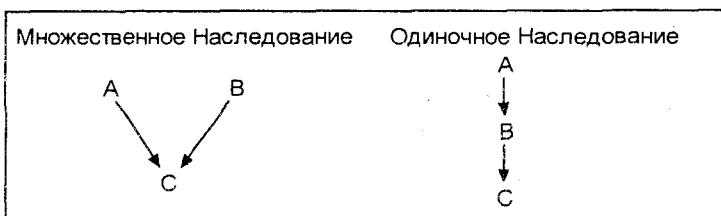


Рис. 28.1. Схема наследования в иерархии классов

В обоих случаях, подкласс C наследует классы A и B. Однако в иерархии слева C наследует как A, так и B одновременно. В иерархии справа класс B наследует A, а C наследует B. Не допуская наследования множественных базовых классов одиночным подклассом, Java сильно упрощает модель наследования. Множественное наследование несет с собой несколько частных случаев, которые должны быть обработаны. Оно добавляет накладные расходы как компилятору, так и исполнительной системе, обеспечивая только минимальную выгоду для программиста.

Так как C++ поддерживает множественное наследование, а Java — нет, вам, вероятно, придется иметь дело с этой проблемой при перенесении приложений C++ на язык Java. Хотя каждая ситуация различна, можно предложить два общих совета. Во-первых, во многих случаях множественное наследование используется в программе C++ тогда, когда на самом деле в этом нет необходимости. Когда же это необходимо, просто преобразуют структуру класса в иерархию с одиночным наследованием. Например, рассмотрим следующую иерархию классов C++, которая определяет класс с именем House:

```

class Foundation {
    // ...
};

```

```
class Walls {
    // ...
};

class Rooms {
    // ...
};

class House : public Foundation, Walls, Rooms {
    // ...
};
```

Обратите внимание, что класс House множественно наследует классы Foundation, Walls и Rooms. Хотя нет ничего неправильного в указанном структурировании иерархии C++, она не является необходимой. Например, тот же набор классов, структурированный для Java, выглядит так:

```
class Foundation {
    // ...
}

class Walls extends Foundation {
    // ...
}

class Rooms extends Walls {
    // ...
}

class House extends Rooms {
    // ...
}
```

Здесь каждый класс расширяет предшествующий, а класс House завершает последовательность расширений.

Иногда множественную иерархию наследований лучше преобразовать включением объектов множества наследованных классов в конечный объект. Например, можно предложить другой способ построения класса House в Java:

```
class Foundation {
    // ...
}

class Walls{
    // ...
}

class Rooms {
    // ...
}

/* Теперь тип House включает в качестве типов объектных членов Foundation,
Walls и Rooms . */

class House {
    Foundation f;
```

```
Walls w;
Rooms r;
// ...
}
```

Здесь Foundation, Walls и Rooms — это объекты, которые являются скорее частью класса House, а не унаследованы им.

Во-вторых, иногда программа C++ содержит иерархию со множественным наследованием просто из-за слабого начального проектирования. Хороший способ исправить этот тип недостатка проекта — это перейти к Java.

## Деструкторы в сравнении с методом *finalize()*

При переходе от C++ к Java один из наиболее тонких, но все же важных моментов, заключается в различии между деструктором C++ и Java-методом `finalize()`. Хотя они подобны во многих отношениях, их фактические действия заметно различны. Начнем с рассмотрения целей и результатов работы деструктора C++ и Java-метода `finalize()`.

В C++ объект разрушается после выхода из области действия своего идентификатора. Буквально перед разрушением вызывается функция деструктора (если она существует). Это жесткое правило, и нет никаких исключений. Рассмотрим подробнее каждую часть этого правила:

- **Каждый объект разрушается, когда он выходит из области действия.** Таким образом, если вы объявляете локальный объект внутри функции, то он автоматически разрушается, когда эта функция возвращает управление. То же самое происходит с параметрами и возвращаемыми объектами функции.
- **Сразу же перед разрушением вызывается деструктор объекта.** Это происходит немедленно и перед выполнением любых других инструкций программы. Таким образом, деструктор C++ будет всегда выполняться детерминированным способом. Вы сможете всегда узнать, когда и где деструктор будет выполнен.

В Java тесной связи между разрушением объекта и вызовом его метода `finalize()` не существует. В Java объекты разрушаются неявно, когда они выходят из области действия. Точнее, объект отмечается как неиспользуемый, когда больше нет никаких указывающих на него ссылок. Метод `finalize()` не будет вызываться до тех пор, пока не выполнится сборщик мусора. Таким образом, вы не можете знать точно, когда или где произойдет вызов `finalize()`. Даже, если вы выполняете обращение к `gc()` (сборщику мусора), нет никакой гарантии, что `finalize()` будет немедленно выполнен.

В то время как детерминированное поведение конструктора C++ и несколько вероятностный аспект "финализации" не сильно влияют на большинство ситуаций, существуют случаи, когда их влияние весьма заметно. Например, рассмотрим следующую программу C++:

```
// Эта программа C++ может вызывать f() бесконечно.
#include <iostream>
#include <cstdlib>
using namespace std;

const int MAX = 5;
int count = 0;

class X {
public:
    // конструктор
    X() {
        if(count<MAX) {
            count++;
        }
        else {
            cout << "Ошибка – не возможно применить конструктор";
            exit(1);
        }
    }

    // деструктор
    ~X() {
        count--;
    }
};

void f()
{
    X ob;          // распределить объект
                    // разрушить при выходе
}

int main()
{
    int i;

    for(i=0; i < (MAX*2); i++) {
        f();
        cout << "Текущее значение count равно: " << count << endl;
    }

    return 0;
}
```

Вывод, сгенерированный этой программой:

```
Текущее значение count равно: 0
```

Просмотрите внимательно на конструктор и деструктор класса x. Конструктор инкрементирует (увеличивает на 1) значение переменной count, пока она меньше чем константа max. Деструктор декрементирует (уменьшает на 1) count. Таким образом, count инкрементируется, когда объект x создается, и декрементируется, когда объект x разрушается. Но в любой момент могут существовать не больше чем max объектов. Однако в main() функция f() вызывается max\*2 раз без возникновения ошибки. И вот почему. Внутри f() создается объект типа x, выполняющий инкремент переменной count, и затем функция возвращает управление. Это приводит к немедленному выходу объекта из области действия и вызову его деструктора, который декрементирует count. Таким образом, вызов f() не имеет никакого результирующего влияния на значение count. Это означает, что она может вызываться бесконечно, не изменяя своего значения. Однако дело обстоит не так, когда эта программа преобразуется в Java-версию.

Java-версия предшествующей программы:

```
// Эта Java-программа даст сбой после 5 вызовов f().

class X {
    static final int MAX = 5;
    static int count = 0;

    // конструктор
    X() {
        if(count<MAX) {
            count++;
        }
        else {
            System.out.println("Ошибка – не возможно применить конструктор ");
            System.exit(1);
        }
    }

    // финализация
    protected void finalize() {
```

```

        count--;
    }

static void f()
{
    X ob = new X();           // распределить память объекту
                            // разрушить при выходе
}

public static void main(String args[])
{
    int i;

    for(i=0; i < (MAX*2); i++) {
        f();
        System.out.println("Текущее значение count равно: " + count);
    }
}
}

```

Эта программа дает сбой после пяти вызовов `f()`, как показывает следующий вывод:

```

Текущее значение count равно: 1
Текущее значение count равно: 2
Текущее значение count равно: 3
Текущее значение count равно: 4
Текущее значение count равно: 5
Ошибка — не возможно применить конструктор

```

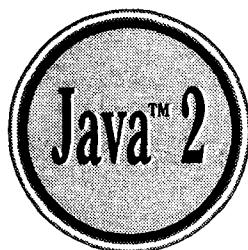
Причина сбоя программы в том, что сборка "мусора" не выполняется каждый раз, когда происходит возврат из `f()`. Таким образом, `finalize()` не вызывается, и значение `count` не декрементируется<sup>1</sup>. После пяти обращений к методу, `count` достигает своего максимального значения, и программа дает сбой.

Важно подчеркнуть, что точный момент сборки "мусора" зависит от реализации. Возможно, что для некоторой реализации Java на какой-нибудь платформе, предшествующая программа будет функционировать подобно ее версии на C++. Однако суть примера остается: в C++ вы знаете, когда и где деструктор будет вызываться, а в Java не знаете, когда или где будет выполнен `finalize()`. Поэтому при перенесении кода из C++ в Java нужно наблюдать за экземплярами, от которых зависит точный момент выполнения деструктора.

---

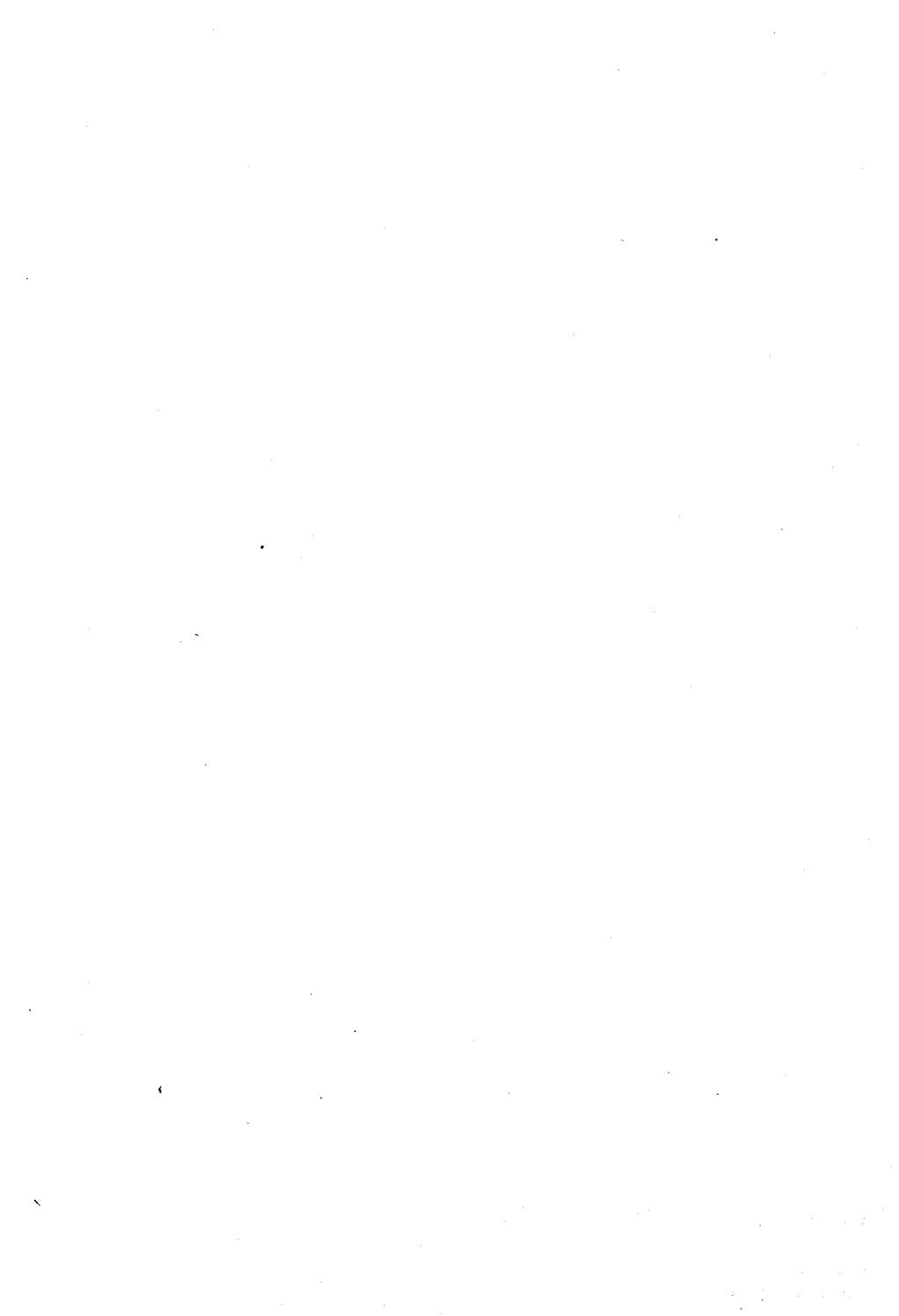
<sup>1</sup> Обратите внимание, что в сравнении с вариантом на C++ `count` начинает менять свои значения. — Примеч. пер.

# ПРИМЕНЕНИЕ JAVA



## ЧАСТЬ IV

29. Апплет *DynamicBillboard*
30. *ImageMenu*: Web-меню изображений
31. Апплет *Lavatron*: дисплей для спортивной арены
32. *Scrabblet*: многопользовательская игра в слова





## ГЛАВА 29

### Апплет *DynamicBillboard*

Роберт Тэмпл (Robert Temple), программный инженер из Starwave Corporation, разработал несколько аплетов, широко используемых во всем мире. Его работа включает аплеты ESPNET SportsZone "HitCharts" и "Batter vs. Pitcher". Апплетом, который впечатлил работников Starwave настолько, чтобы просить его присоединиться к компании, был *DynamicBillboard*, который он написал в то время, когда был в Университете Аэронавтики (Embry-Riddle Aeronautical University) во Флориде.

Апплет *DynamicBillboard* отображает последовательность изображений, повторно сменяющих друг друга на экране (после некоторой задержки). Переход между текущим и следующим изображением сопровождается одним из некоторых специальных эффектов. Пример одного из переходов, *SmashTransition*, — когда новое изображение выпадает сверху на старое и "сминает" его. Апплет связывается с другими страницами через URL-адрес, ассоциированный с каждым изображением. Когда пользователь нажимает кнопку мыши с курсором поверх аплета, браузер будет переходить к новой странице, связанной с текущим изображением. Апплет *DynamicBillboard* обеспечивает Web-сайты изящным способом вращать объявления, заголовки, или "доски объявлений" (billboards) на одиночной статической странице.

Робертом включено в аплет много интересных оптимизаций. Этот аплет не был бы функционален без тщательных изменений, которые он добавил. Имеются достаточно советов и уловок в этом исходном тексте, чтобы помочь сделать ваши аплеты действительно "летящими".

#### Тег *<applet>*

Законченный тег *<applet>* для аплета *DynamicBillboard* (со всеми переходами, обсуждаемыми в этой главе) выглядит так:

```

<applet code=DynamicBillboard width=392 height=72>
<param name="bgcolor" value="#ffffff">
<param name="delay" value="5000">
<param name="billboards" value="5"><param name="bill10"
    value="robert.jpg,http://www.starwave.com/people/robertt/">
<param name="bill11"
    value="kenna.jpg,http://www.starwave.com/people/naughton/kenna/">
<param name="bill12"
    value="lavallee.jpg,http://www.starwave.com/people/lavallee/">
<param name="bill13"
    value="handbook.jpg,http://www.starwave.corn/people/naught on/book/">
<param name="bill14"
    value=" family. jpg, http: / /www. starwave. com/people/naughton/ family/" >
<param name="transitions"
    value="5/ColumnTransition,FadeTransition,TearTransition,
          SmashTransition,UnrollTransition">
</applet>

```

Данный тег `<applet>` сконфигурирован довольно просто:

```
<applet code=DynamicBillboard width=392 height=72>
```

Главный класс (`DynamicBillboard`) указан в его параметре `code`, а ширину и высоту окна апплета определяют параметры `width` и `height`.

Для правильного функционирования в теге определено несколько параметров, без которых апплет ничего не выполняет. Обратите также внимание, что, если вы делаете какие-нибудь ошибки (например, при именовании файлов или вызывающие не очень благоприятное поведение программы), то или ничего не происходит, или некоторые из "досок объявлений" будут пустыми. Параметры определены в формате HTML тега `<param>`:

```
<param name=имя_параметра value="значение_параметра">
```

Вот их краткое описание:

- `bgcolor`. Этот параметр используется для установки цвета фона апплета перед первой загрузкой изображений. Вы можете использовать его, чтобы быстро избавиться от серого фона в окне апплета.
- `delay`. Этот параметр определяет число миллисекунд между показом каждого изображения. Обычно он имеет значение из диапазона от 5000 до 10 000, что означает 5 или 10 секунд.
- `billboards`. Этот параметр определяет число изображений, которое вы желаете просматривать циклически.
- `bill#`. Это пронумерованные имена досок объявлений (`billboards`) — `bill10`, `bill11`, `bill12` и т. д. до номера, на 1 меньшего, чем число, которое вы определили в параметре `billboards`. (Роберт — типичный программист, который начинает считать с 0.) Этих имен, как мы видим, столько,

сколько определено в параметре billboards. Значением каждого из этих имен является пара строк, разделенных запятой. Первая — это имя файла изображения данной доски объявлений. Вторая — URL-адрес перехода при щелчке мышью по данной доске объявлений. Например:

```
<param name="bill0" value="starwave.jpg,http://www.starwave.com/">
```

- transitions. Это список, начинающийся с целого значения, определяющего количество элементов в списке, за которым следует перечисление имен подклассов класса Transition. Например:

```
<param name="transitions" value="2,TearTransition,SmashTransition">
```

(В апплете DynamicBillboard, как мы видим, в этом списке пять элементов.)

## Обзор исходного кода

Роберт разработал апплед с быстрым временем загрузки в память. Для пересылки меньшего кода по сети он пробует сохранить минимальный размер аппледа. Он также пытается несколько задержать загрузку и инициализацию аппледа, пока не будет отображено первое изображение. Поскольку в этом заинтересован пользователь, то апплед запускается только после того, как первое изображение полностью отображено, хотя для этого необходимо проделать намного больше работы.

Апплед состоит из трех главных классов и любого числа классов перехода. Три главных класса — это DynamicBillboard, BillData и BillTransition. Класс DynamicBillboard — это высокоуровневый подкласс класса Applet, который использует все другие классы. Класс BillData инкапсулирует ряд атрибутов доски объявлений, включая изображение и URL, связанный с изображением. Класс BillTransition — это абстрактный класс, который содержит методы и атрибуты, общие для всех переходов. Ниже описаны три главных класса, наряду с пятью популярными переходами.

### **DynamicBillboard.java**

Это главный класс аппледа. Он реализует интерфейс Runnable, чтобы включить поток, который управляет непрерывным процессом создания и анимации переходов. Массив transition\_classes хранит имена классов перехода в форме строк. Он использует строки, потому что загружает эти классы динамически, используя метод java.lang.Class.forName(String), что позволяет аппледу откладывать загрузку этих классов, пока для них не будет создан первый объект.

#### **Метод init()**

Когда апплед впервые загружается, автоматически вызывается метод init(). Большинство аппледов использует этот метод, чтобы выполнить всю необ-

ходимую инициализацию. Роберт, однако, решил разделить инициализацию на два метода — `init()` и `finishInit()`. Идея, стоящая за расщеплением инициализации, заключается в том, чтобы попытаться отображать первое изображение в апплете наименьшее количество времени, минимизируя время, которое апплед показывает пустой серый прямоугольник во время своей загрузки и инициализации. Единственная обработка, которая выполняется в методе `init()`, состоит в том, чтобы получить начальное содержимое для экрана, потому что браузер не будет вызывать `paint()` до возврата из `init()`.

Первое, что Роберт делает с `init()`, это изменяет цвет фона аппледа и родительского фрейма, в который внедрен апплед. Обычно место, которое использует апплед на экране, показывается как сплошной серый прямоугольник (во время загрузки и инициализации аппледа). Этот прямоугольник склонен выделяться на страницах, которые используют иной цвет фона, по сравнению с тем серым, который есть почти на каждой странице, созданной после 1994 г. Роберт нашел способ обойти эту проблему. Он обнаружил, что аппледы всегда имеют родительский контейнер, в который они внедрены. Как в Netscape Navigator, так и в Internet Explorer этот контейнер является производным от класса ядра Java `java.awt.Container`. Роберт использует методы, наследованные от `java.awt.Component` (`setBackground()` и `repaint()`), чтобы заменить цвет фона значением параметра аппледа `bgcolor`. Это делает область аппледа менее выделяющейся, чем, если бы она оставалась серой. Все это делается перед тем, как апплед начинает загружать первое изображение.

### Замечание

С некоторыми из более новых браузеров на рынке, начиная с Netscape 3.0, этот фрейм по умолчанию больше не серый, а использует цвет фона страницы. В будущем, когда эти браузеры будут широко использоваться, аппледы не станут извлекать выгоду из изменения цвета фона.

После изменения цветов фона апплед Роберта считывает параметр, который сообщает, сколько будет различных досок объявлений, и затем распределяет массив `BillData`-объектов, основанных на этом параметре.

С помощью метода `Math.random()` начинается выбор случайной доски объявлений. Затем вызывается метод `ParseBillData()`, чтобы анализировать параметры этой доски.

### Метод `parseBillData()`

Данный метод создает и инициализирует объект следующей доски объявлений (`BillData`), которую апплед будет использовать. Он вызывается только в том случае, если соответствующий объект еще не был создан (элемент, соответствующий объекту следующей доски объявлений в массиве `billboards[]`, будет содержать `null`).

Обычно после создания нового объекта `parseBillData()` вызывает `BillData`-метод `initPixels()`, чтобы инициализировать массив пикселов внутри `BillData`-объекта. Однако, когда этот метод вызывается первый раз, апплет еще концентрируется на получении первого изображения (с такой скоростью, с какой это возможно). Он знает об этом, потому что ссылка на изображение, которая используется для рисования апплета, все еще имеет значение `null`. Поэтому он устанавливает `image`-переменную и ждет, пока загрузится первое изображение, чтобы затем вызвать интенсивный (по использованию процессора) метод `initPixels()`.

## Метод `finishInit()`

После того как первое изображение появится на экране, апплет может заканчивать остальную часть своей инициализации. Это включает инициализацию имен всех классов перехода, инициализацию массива пикселов для первой доски объявлений и чтение целевого параметра.

Метод `finishInit()` вызывается из метода `run()` апплета. Метод `run()` перезапускается сначала каждый раз, когда пользователь покидает страницу и возвращается на нее. Когда это случается, `finishInit()` вызывается заново. Так как апплет уже закончил свою инициализацию, Роберт не хочет, чтобы все повторно инициализировалось. Вот почему апплет проверяет, была ли переменная `delay` уже инициализирована. Если была, то апплет может пропустить остальную часть инициализации.

## Методы `start()` и `stop()`

Методы `start()` и `stop()` вызываются, когда пользователь переходит на страницу или покидает ее. Они гарантируют включение и выключение потока апплета, который выполняет переходы.

Если `stop()` вызывается в то время, когда апплет находится в середине выполнения перехода, то некоторые данные могут остаться в неподходящем состоянии. Определенные переменные переустанавливаются в `start()`, чтобы удостовериться, что апплет повторно стартует с новым переходом.

В методе `start()` курсор мыши изменяет свою форму на значок руки, так что, когда курсор мыши находится над апплетом, будет видно, что между ними существует связь.

## Метод `run()`

Метод `run()` начинается с цикла, который ждет (перед продолжением), чтобы первое изображение было полностью загружено. Затем он заканчивает инициализацию апплета, вызывая `finishInit()`, и входит в главный цикл программы.

Главный цикл управляет переходами между досками объявлений. Используя параметр `delay`, переданный из HTML, апплет вычисляет, когда предполага-

ется выполнить следующий переход. Во время ожидания он готовится к переходу. Подготовка начинается с определения, какая доска объявлений должна быть отображена следующей, и анализа данных доски из параметров HTML, если это еще не было сделано для данной доски. Затем он случайным образом выбирает, какой переход выполнить следующим, заботясь о том, чтобы не позволить апплету выполнять повторно тот же самый переход.

Как только апплет определил, какой переход будет далее выполнен, он создает новый экземпляр этого класса перехода (динамически загружая класс, использующий имя типа `String`, и затем создавая новый экземпляр класса). Динамическая загрузка классов перехода имеет большое влияние на время загрузки апплета в целом. Вместо загрузки каждого одиночного класса перед запусками апплета, первоначально посылаются только три класса: `DynamicBillboard`, `BillData` и `BillTransition`. Другие классы перехода загружаются апплетом только в тот момент, когда они необходимы. Это значительно сокращает начальную загрузку апплета. Если пользователь быстро покидает страницу, то некоторые файлы классов, возможно, даже не нуждаются в пересылке.

Наконец, апплет вызывает метод `init()` для объекта перехода, передавая в качестве параметров пиксели апплета и изображения для текущей и следующей досок объявлений. Он создает все фреймы ячеек, которые используются, чтобы оживить переход. С переходом, готовым к запуску, апплету нужно только ждать в течение надлежащего времени, чтобы запустить переход.

Апплет выполняет переход, используя простую фреймовую анимацию — рисуя на экране каждую ячейку по порядку, с короткой задержкой между каждым фреймом. Апплет вызывает инструментальный метод `sync()` только для того, чтобы убедиться, что рисование ячейки не было выполнено прежде, чем предыдущая ячейка показалась на экране. После того как отображена последняя ячейка, апплет выводит на экран изображение следующей доски объявлений, чем и завершает переход.

После этого проверяется флагок `mouse_over_applet`, чтобы увидеть, находится ли курсор мыши в настоящее время над апплетом. Если это так, то URL предыдущей доски объявлений, выводится на строку состояния и должен быть модифицирован таким образом, чтобы отобразить URL новой доски. Это делается с помощью вызова апплет-метода `showStatus()`. Апплет завершил этот переход и теперь готов начать следующий.

## Методы `mouseMoved()` и `mouseExited()`

Чтобы изменять текст, который появляется в строке состояния, используются методы `mouseMoved()` и `mouseExited()`. Когда курсор мыши находится над панелью апплета, предполагается, что строка состояния показывает URL, с которым связана текущая доска объявлений. Итак, когда вызывается `mouseMoved()`, апплет выводит URL в строке состояния. Когда вызывается

mouseExited(), текст URL удаляется из нее. Оба метода также устанавливают подходящее значение булевой переменной mouse\_inside\_applet. Указанная переменная используется в методе run() после того, как выполняется переход. Если указатель мыши установлен над апплетом после завершения перехода, апплет знает, что нужно отобразить в строке состояния URL новой доски объявлений.

### Метод *mouseReleased()*

Когда кнопка мыши нажата во время позиционирования курсора над окном апплета и затем отпущена, вызывается метод mouseReleased(). Для передачи браузеру URL-адреса, на который указывает текущая доска объявлений, апплет использует метод getAppletContext().showDocument(). Как выяснил Роберт, иногда браузеру требуется много времени, чтобы отобразить новую страницу. Чтобы предохранить апплет от запуска большего количества переходов, в то время как новая страница ожидает загрузки, вызывается метод stop() для завершения выполнения главного потока. Для предоставления пользователям сведений о процессе загрузки новой страницы апплетом изменяется курсор мыши.

Важно помнить, что пользователи могут возвращаться к данной странице после перехода к новой. Курсор ожидания будет все еще присутствовать на апплете при подобном возвращении. Когда пользователь возвращается к странице с апплетом, всегда вызывается метод start(), так что апплет переустанавливает курсор к форме руки.

### Код

Исходный код класса DynamicBillboard:

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.awt.*;
import java.awt.image.*;

public class DynamicBillboard
    extends java.applet.Applet
    implements Runnable {

    BillData[] billboards;
    int current_billboard;
    int next_billboard;

    String[] transition_classes;
    Thread thread = null;
    Image image = null;
    long delay = -1;
```

```
boolean mouse_inside_applet;
String link_target_frame;
boolean stopFlag;

public void init() {
    String s = getParameter("bgcolor");
    if(s != null) {
        Color color = new Color(Integer.parseInt(s.substring(1), 16));
        setBackground(color);
        getParent().setBackground(color);
        getParent().repaint();
    }
    billboards = new
        BillData[Integer.parseInt(getParameter("billboards"))];
    current_billboard = next_billboard
        = (int)(Math.random() *billboards.length);
    parseBillData();
}

void parseBillData() {
    String s = getParameter("bill" + next_billboard);
    int field_end = s.indexOf(",");
    Image new_image = getImage(getDocumentBase(),
                                s.substring(0, field_end));
    URL link;
    try {
        link = new URL(getDocumentBase(), s.substring(field_end + 1));
    }
    catch (java.net.MalformedURLException e) {
        e.printStackTrace();
        link = getDocumentBase();
    }
    billboards[next_billboard] = new BillData(link, new_image);
    if(image == null) {
        image = new_image;
    }
    else {
        prepareImage(new_image, this);
        billboards[next_billboard].initPixels(getSize().width,
                                              getSize().height);
    }
}

void finishInit() {
    if(delay != -1) {
        return;
    }
    delay = Long.parseLong(getParameter("delay"));}
```

```
link_target_frame = getParameter("target");
if(link_target_frame == null) {
    link_target_frame = "_top";
}

String s = getParameter("transitions");
int field_end = s.indexOf(",");

int trans_count = Integer.parseInt(s.substring(0, field_end));
transition_classes = new String[trans_count];
for(--trans_count; trans_count > 0; --trans_count) {
    s = s.substring(field_end + 1);
    field_end = s.indexOf(",");
    transition_classes[trans_count] = s.substring(0, field_end);
}
transition_classes[0] = s.substring(field_end + 1);
billboards[next_billboard].initPixels(getSize().width,
                                         getSize().height);
mouse_inside_applet = false;
}

public void paint(Graphics g) {
    g.drawImage(image, 0, 0, this);
}

public void update(Graphics g) {
    paint(g);
}

public void start() {
    next_billboard = current_billboard;
    image = billboards[current_billboard].image;
    setCursor(new Cursor(Cursor.HAND_CURSOR));
    if(thread == null) {
        thread = new Thread(this);
        thread.start();
    }
}

public void stop() {
    if(thread != null) {
        stopFlag = true;
    }
}

public void run() {
    while((checkImage(image, this) & ImageObserver.ALLBITS) == 0) {
        try { Thread.sleep(600); } catch (InterruptedException e) {}
    }
    finishInit();
```

```
addMouseListener(new MyMouseAdapter());
addMouseMotionListener(new MyMouseMotionAdapter());

int last_transition_type = -1;
BillTransition transition;
long next_billboard_time;
while(true) {
    if(stopFlag)
        return;
    next_billboard_time = System.currentTimeMillis() + delay;
    current_billboard = next_billboard;
    if(++next_billboard >= billboards.length) {
        next_billboard = 0;
    }
    if(billboards[next_billboard] == null) {
        parseBillData();
        try { Thread.sleep(120); } catch (InterruptedException e) {}
    }
    int transition_type = (int)(Math.random() *
                                (transition_classes.length - 1));
    if(transition_type >= last_transition_type) {
        ++transition_type;
    }
    last_transition_type = transition_type;

    try {
        String trans = transition_classes[last_transition_type];
        transition = (BillTransition)Class.forName(trans).newInstance();
    }
    catch(Exception e) {
        e.printStackTrace();
        continue;
    }

    transition.init(this, billboards[current_billboard].image_pixels,
                    billboards[next_billboard].image_pixels);

    if(System.currentTimeMillis() < next_billboard_time) {
        try {
            Thread.sleep(next_billboard_time - System.currentTimeMillis());
        } catch (InterruptedException e) {}
    }
    Graphics g = getGraphics();
    for(int c = 0; c < transition.cells.length; ++c) {
        image = transition.cells[c];
        g.drawImage(image, 0, 0, null);
        Toolkit.sync();
        try { Thread.sleep(transition.delay); }
    }
}
```

```
        catch(InterruptedException e) { };
    }
    image = billboards[next_billboard].image;
    g.drawImage(image, 0, 0, null);
    getToolkit().sync();
    g.dispose();
    if(mouse_inside_applet == true) {
        showStatus(billboards[next_billboard].link.toExternalForm());
    }
    transition = null;
    try { Thread.sleep(120); } catch (InterruptedException e) {}
}

}

public class MyMouseAdapter extends MouseAdapter {
    public void mouseExited(MouseEvent me) {
        mouse_inside_applet = false;
        showStatus("");
    }
    public void mouseReleased(MouseEvent me) {
        stop();
        setCursor(new Cursor(Cursor.WAIT_CURSOR));
        getAppletContext().showDocument(billboards[current_billboard].link,
                                         link_target_frame);
    }
}

public class MyMouseMotionAdapter extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent me) {
        mouse_inside_applet = true;
        showStatus(billboards[current_billboard].link.toExternalForm());
    }
}
}
```

## BillData.java

Класс BillData — это, главным образом, только структура данных для инкапсуляции атрибутов, связанных с индивидуальными досками объявлений. Он содержит три переменных. Первая переменная хранит URL, с которым связана доска объявлений. Вторая переменная имеет тип Image, который апплет использует, чтобы рисовать на экране. Третья переменная включает массив пикселов изображения в формате RGB.

Массив пикселов используется переходами в комбинации с другим BillData-массивом пикселов, чтобы создать ячейки для анимации перехода. Это одномерный массив. Пиксели в нем размещены таким образом, что первый

элемент является верхним левым углом изображения. Второй элемент — пиксел, следующий непосредственно справа от этого углового. Следующие элементы — это пиксели справа от него и т. д., пока не будет достигнут самый правый пиксел в строке изображения. Затем используется самый левый пиксел на следующей строке изображения. Это продолжается до последнего индекса в массиве, который соответствует пикселу в правом нижнем углу изображения.

Обратите внимание, что Роберт сделал все переменные в этом классе общими (`public`). Обычная практика программирования скрывает члены данных, которые должны читаться другими классами. Это делается объявлением их защищенными (`protected`) или частными (`private`) и затем созданием функций для возврата ссылок на такие переменные. К сожалению, в Java это увеличивает размер откомпилированного байт-кода, даже когда односторонняя функция объявляется с модификатором `final` и код компилируется с оптимизацией. Итак, чтобы сделать апплет меньше и, следовательно, быстрее загружать его, Роберт сделал компоненты данных общими.

## Конструктор

Конструктор объекта `BillData` просто инициализирует URL- и `Image`-переменные с двумя передаваемыми ему параметрами. Инициализация массива пикселов выполнена в отдельном методе, т. к. он очень интенсивно использует процессор. Это дает апплету шанс инициализировать массив пикселов только тогда, когда это требуется.

## Метод `initPixels()`

Метод `initPixels()` создает массив пикселов из изображения, используя класс `java.awt.image.PixelGrabber` ядра Java.

## Код

Исходный код класса `BillData`:

```
import java.net.*;
import java.awt.*;
import java.awt.image.*;

public class BillData {
    public URL link;
    public Image image;
    public int[] image_pixels;

    public BillData(URL link, Image image) {
        this.link = link;
        this.image = image;
    }
}
```

```
public void initPixels(int image_width, int image_height) {
    image_pixels = new int[image_width * image_height];
    PixelGrabber pixel_grabber = new
    PixelGrabber(image.getSource(), 0, 0,
                 image_width, image_height, image_pixels, 0, image_width);
    try {
        pixel_grabber.grabPixels();
    }
    catch (InterruptedException e) {
        image_pixels = null;
    }
}
```

## ***BillTransition.java***

Класс `BillTransition` используется как базовый класс для других классов перехода. Другие классы создают ячейки перехода между двумя индивидуальными изображениями доски объявлений. Этот абстрактный класс содержит переменные и методы, которые являются общими для всех переходов.

В классе `BillTransition` нет никаких конструкторов, из-за того что апплет не использует операции `new` для создания новых экземпляров, а вместо этого применяет производственный метод `java.lang.Class.newInstance()`. Объекты, созданные этим способом, не имеют возможности прямо инициализировать себя, используя параметры в конструкторах. Этот производственный метод косвенно создает объекты, используя умалчиваемый конструктор без каких-либо параметров. Класс `BillTransition` обеспечивает несколько перегруженных методов `init()` для инициализации экземпляров с параметрами.

В предыдущих версиях `DynamicBillboard` Роберт использовал статические переменные в различных классах перехода для хранения данных, которые нуждаются только в одноразовой инициализации. Однако было обнаружено, что при запуске Web-сервером больше одного экземпляра апплета, им нужно было совместно использовать (разделять) статические переменные. Это приводило к некоторым проблемам, когда апплеты имели различные размеры (т. к. один апплет нуждался в иных статических значениях, чем другой). Примером является класс `FadeTransition`, использовавшийся для создания массива, чей размер зависел от размеров апплета. Если бы создавался другой `DynamicBillboard` с размерами, которые были бы меньше, чем у предыдущего апплета, он переписывал бы массив первого апплета, слишком маленьким собственным массивом. Это привело бы первый апплет к аварийному завершению.

Для решения данной проблемы Роберт ввел в этой версии апплета статическую хэш-таблицу с именем `object_table`. Теперь классы перехода могут

запоминать данные внутри этой хэш-таблицы, используя название перехода вместе с размером апплета в качестве ключа. При возникновении необходимости использовать данные апплет может посмотреть, существуют ли они в хэш-таблице. Если нет, то он может создать данные и сохранить их там для более позднего использования. Теперь, если на Web-сервере могут существовать несколько апплетов одинакового размера, то только один должен инициализировать данные.

## Метод *init()*

Метод *init()* перегружен три раза. Первый метод, который имеет три параметра, является абстрактным и должен быть переопределён своими производными классами. Другие два инициализируют компоненты данных в пределах данного класса. Намерение Роберта состояло в том, чтобы иметь метод *init()* для классов, которые являются производными от данного класса, и вызывать один из этих двух методов для инициализации компонентных данных класса *BillTransition*.

## Метод *createCellFromWorkPixels()*

Метод *createCellFromWorkPixels()* используется для выполнения общей задачи преобразования массива *work\_pixels* в *Image*-объект. Обратите внимание, что для завершения этой задачи он использует переменную *owner*. Это единственная причина, по которой переменная *owner* необходима классам перехода. Когда переход завершил трансляцию новой ячейки в массив *work\_pixels*, он должно вызвать данный метод.

## Код

Исходный код класса *BillTransition*:

```
import java.util.*;
import java.awt.*;
import java.awt.image.*;

public abstract class BillTransition {
    static Hashtable object_table = new Hashtable(20);

    public Image[] cells;
    public int delay;

    Component owner;
    int cell_w;
    int cell_h;
    int pixels_per_cell;
    int[] current_pixels;
    int[] next_pixels;
    int[] work_pixels;
```

```
public abstract void
init(Component owner, int[] current_pixels, int[] next_pixels);

final protected void
init(Component owner, int[] current_pixels, int[] next_pixels,
      int number_of_cells, int delay) {
    this.delay = delay;
    this.next_pixels = next_pixels;
    this.current_pixels = current_pixels;
    this.owner = owner;

    cells = new Image[number_of_cells];
    cell_w = owner.getSize().width;
    cell_h = owner.getSize().height;
    pixels_per_cell = cell_w * cell_h;
    work_pixels = new int[pixels_per_cell];
}

final protected void
init(Component owner, int[] current_pixels, int[] next_pixels,
      int number_of_cells) {
    init(owner, current_pixels, next_pixels, number_of_cells, 120);
}

final void createCellFromWorkPixels(int cell) {
    cells[cell] = owner.createImage(
        new MemoryImageSource(cell_w, cell_h, work_pixels, 0, cell_w));
    owner.prepareImage(cells[cell], null);
}
}
```

## ***ColumnTransition.java***

Класс ColumnTransition заменяет одно изображение на другое, рисуя на старом изображении все больше и больше колонок нового изображения. Ширина колонок увеличивается влево, и одни и те же пиксели всегда рисуются на левой стороне каждой колонки. Это вызывает появление текущей доски объявлений из-за старой доски в виде расширяющихся вертикальных полосок.

Чтобы создать ячейки для этого перехода, пространство доски объявлений разбивается на ряд колонок шириной по 24 пикселя каждый. Левые пиксели в каждой колонке — от старого изображения, а правые — от нового. Переход создает семь ячеек таких изображений. Первая создаваемая ячейка запускается только с тремя правыми пикселями из нового изображения в каждой колонке. Следующая ячейка заполняется еще тремя пикселями из нового изображения. Последняя ячейка имеет только три левых пикселя в каждой колонке от старого изображения.

Поскольку ширина пространства изображения, вероятнее всего, не полностью делится на 24, справа на изображении будет несколько остаточных пикселов. Эти пиксели считаются в каждой ячейке с помощью переменных `rightmost_columns_max_width` и `rightmost_columns_x_start`.

## Метод `init()`

Функция `init()` запускается вызовом метода `init()` базового класса, чтобы инициализировать переменные, содержащиеся в этом базовом классе. Она поддерживает инициализацию переменных, связанных с самой правой колонкой, и затем копирует все пиксели из текущей доски объявлений в рабочие пиксели. Следующий за этим цикл создает все фреймы ячеек. Метод `nextCell()` изменяет рабочие пиксели (массив `work_pixels`), а метод `createCellFromWorkPixels()`, унаследованный от класса `BillTransition`, используется для того, чтобы конвертировать эти пиксели в изображение. Поскольку процесс создания ячеек может очень интенсивно использовать CPU, Роберт просит поток время от времени бездействовать ("засыпать"), чтобы позволить выполнять другим потокам.

## Метод `nextCell()`

Метод `nextCell()` модифицирует массив `work_pixels` для следующей ячейки. Он организует циклы для каждой строки изображения, начиная с нижней, и наполняет часть каждой колонки, копируя пиксели из следующей доски объявлений в массив `work_pixels`. Копировать пиксели из старой доски объявлений не нужно, потому что они были уже скопированы в массив методом `init()`.

Стоит повторить, что массивы пикселов, используемые для формирования изображений, только одномерны. Каждый `width`-пиксель представляет одну горизонтальную строку изображения.

## Код

Исходный код класса `ColumnTransition`:

```
import java.awt.*;
import java.awt.image.*;

public class ColumnTransition extends BillTransition {
    final static int CELLS = 7;
    final static int WIDTH_INCREMENT = 3;
    final static int MAX_COLUMN_WIDTH = 24;

    int rightmost_columns_max_width;
    int rightmost_columns_x_start;
    int column_width = WIDTH_INCREMENT;
```

```

public void init(Component owner, int[] current, int[] next) {
    init(owner, current, next, CELLS, 200);
    rightmost_columns_max_width = cell_w % MAX_COLUMN_WIDTH;
    rightmost_columns_x_start = cell_w - rightmost_columns_max_width;
    System.arraycopy(current_pixels, 0, work_pixels, 0, pixels_per_cell);
    for(int c = 0; c < CELLS; ++c) {
        try { Thread.sleep(100); } catch (InterruptedException e) {}
        NextCell();
        try { Thread.sleep(100); } catch (InterruptedException e) {}
        createCellFromWorkPixels(c);
        column_width += WIDTH_INCREMENT;
    }
    work_pixels = null;
}

void NextCell() {
    int old_column_width = MAX_COLUMN_WIDTH - column_width;
    for(int p = pixels_per_cell - cell_w; p >= 0; p -= cell_w) {
        for (int x = 0; x < rightmost_columns_x_start; x +=
            MAX_COLUMN_WIDTH) {
            System.arraycopy(next_pixels, x + p, work_pixels,
                old_column_width + x + p, column_width);
        }
        if(old_column_width <= rightmost_columns_max_width) {
            System.arraycopy(next_pixels, rightmost_columns_x_start + p,
                work_pixels, rightmost_columns_x_start +
                    old_column_width + p - 1,
                    rightmost_columns_max_width -
                    old_column_width + 1);
        }
    }
}

```

На рис. 29.1 видно, как колоночный (column) переход выглядит до, в течение и после смены изображения:



**Рис. 29.1.** Переход колоночным методом до смены изображения (вверху), в течение (посередине) и после (внизу)

## FadeTransition.java

Класс `FadeTransition` заменяет одно изображение на другое, беспорядочно включая ряд новых пикселов от следующей доски объявлений в каждый последующий фрейм ячейки. Это заставляет очередную доску объявлений постепенно проявляться на старой доске.

Основа этого перехода — двухмерный массив коротких (`short`) целых чисел с именем `random`. Массив содержит индекс (номер) каждого элемента в массиве пикселов изображения следующей доски объявлений. Эти индексы беспорядочно (случайно) распределены в двухмерном массиве. Восемь элементов в первом измерении этого массива будут использоваться при создании ячейки, по одному для каждой новой ячейки. Последний элемент никогда фактически не используется, потому что имеется только семь ячеек. Он включается, когда массив `random` создается, чтобы гарантировать, что случайное распределение индексов корректно.

`FadeTransition` использует этот массив, чтобы выбирать пиксели из следующей доски объявлений для перезаписи пикселов старой доски. Для первой ячейки массив `work_pixels` содержит только пиксели из старой доски объявлений. Восьмая часть этих пикселов заменяется на пиксели следующей доски объявлений. Для очередной ячейки используется тот же массив `work_pixels`, но на одну восьмую большее количество пикселов заполнено из следующей доски. В результате эта ячейка имеет четвертую часть пикселов из следующей доски объявлений, в то время как остальные — из старой доски. Это продолжается до последней ячейки (ячейки с номером 7), которая имеет семь-восьмых своих пикселов из новой доски объявлений. Чтобы завершить данный переход, апплет `DynamicBillboard` просто использует в качестве последней ячейки целое изображение следующей доски.

Поскольку размер двухмерного массива зависит от размера апплета, он должен быть уникальным в каждом апплете. Использование статической переменной для хранения этого массива недопустимо, потому что апплеты различных размеров будут совместно его использовать. Так как инициализация массива занимает много времени, не имеет смысла вновь создавать его каждый раз, когда будет использоваться этот переход.

Здесь впервые вступает в игру статическая переменная суперкласса `object_table`. Как только массив создан, он может быть сохранен в этой хэш-таблице с ключом, содержащим размер апплета. Когда нужно использовать массив, апплет может получить его подходящий вариант из хэш-таблицы. Если он там не существует, тогда апплет может создать массив и сохранить его в хэш-таблице для будущего применения. Новые апплеты того же размера, что и текущий, просто извлекут подходящий массив из хэш-таблицы. Это требует много усилий, но на практике Web-сайты имеют тенденцию использовать такой апплет на большом количестве страниц со стандартным размером размещения для каждой рекламной шапки. Так что

кэширование этой таблицы сохраняет огромное количество памяти и процессорного времени.

## Метод *createRandomArray()*

Статический метод *createRandomArray()* создает двумерный массив случайных значений. Он имеет два параметра, которые описывают размер апплета. Метод сильно оптимизирован, потому что первоначально был слишком медленным. Метод имеет собственный генератор случайных чисел, который очень быстр, но имеет короткий период. Из-за этого метод довольно сложен и в этой книге подробнее не рассматривается. Основная идея состоит в том, что встроенный генератор случайных чисел языка Java лучше для генерации действительно случайного распределения, но он слишком медленный для данного приложения. Кроме того, пользователь не будет замечать, насколько случаен этот переход, так что самодельный случайный генератор Роберта вполне достаточен.

## Метод *init()*

Метод *init()* для этого перехода начинается, подобно всем другим переходам, с обращения к методу *init()* базового класса. Затем, подобно некоторым другим переходам, он копирует все пиксели старой доски объявлений в массив *work\_pixels*.

Случайный двумерный массив извлекается из *object\_table* для апплета этого размера. Если он еще не существует, то вновь создается и сохраняется в *object\_table*. С помощью массива случайных чисел метод выполняет циклы по каждой ячейке и каждому индексу в нем, копируя в рабочие пиксели из следующей доски объявлений.

## Код

Исходный код класса *FadeTransition*:

```
import java.awt.*;
import java.awt.image.*;

public class FadeTransition extends BillTransition {
    private static final int CELLS = 7;
    private static final int MULTIPLIER = 0x5D1E2F;

    private static short[][] createRandomArray(int number_pixels,
                                                int cell_h) {
        int total_cells = CELLS + 1;
        int new_pixels_per_cell = number_pixels / total_cells;
        short[][] random = new short[total_cells][new_pixels_per_cell];
        int random_count[] = new int[total_cells];
        ...
```

```
for(int s = 0; s < total_cells; ++s) {
    random_count[s] = 0;
}
int cell;
int rounded_new_pixels_per_cell =
    new_pixels_per_cell * total_cells;
int seed = (int)System.currentTimeMillis();
int denominator = 10;
while((new_pixels_per_cell % denominator > 0 ||
    cell % denominator == 0) && denominator > 1) {
    --denominator;
}
int new_randoms_per_cell = new_pixels_per_cell / denominator;
int new_randoms = rounded_new_pixels_per_cell / denominator;
for(int p = 0; p < new_randoms_per_cell; ++p) {
    seed *= MULTIPLIER;
    cell = (seed >>> 29);
    random[cell][random_count[cell]++] = (short)p;
}
seed += 0x5050;
try { Thread.sleep(150); } catch (InterruptedException e) {}
for(int p = new_randoms_per_cell; p < new_randoms; ++p) {
    seed *= MULTIPLIER;
    cell = (seed >>> 29);
    while(random_count[cell] >= new_randoms_per_cell) {
        if(++cell >= total_cells) {
            cell = 0;
        }
    }
    random[cell][random_count[cell]++] = (short)p;
}
for(int s = 0; s < CELLS; ++s) {
    for(int ps = new_randoms_per_cell; ps < new_pixels_per_cell;
        ps += new_randoms_per_cell) {
        int offset = ps * total_cells;
        for(int p = 0; p < new_randoms_per_cell; ++p) {
            random[s][ps + p] = (short)(random[s][p] + offset);
        }
    }
    try { Thread.sleep(50); } catch (InterruptedException e) {}
}
random[CELLS] = null;
return random;
}

public void init(Component owner, int[] current, int[] next) {
    init(owner, current, next, CELLS);
    System.arraycopy(current_pixels, 0, work_pixels, 0, pixels_per_cell);
```

```

short random[][][] = (short[][][])object_table.get(
    getClass().getName() + pixels_per_cell);
if(random == null) {
    random = createRandomArray(pixels_per_cell, cell_h);
    object_table.put(getClass().getName() + pixels_per_cell, random);
}
for(int c = 0; c < CELLS; ++c) {
    try { Thread.sleep(100); } catch (InterruptedException e) {}
    int limit = random[c].length;
    for(int p = 0; p < limit; ++p) {
        int pixel_index = random[c][p];
        work_pixels[pixel_index] = next_pixels[pixel_index];
    }
    try { Thread.sleep(50); } catch (InterruptedException e) {}
    createCellFromWorkPixels(c);
}
work_pixels = null;
}
}

```

Как выглядит постепенно возникающий (fade) переход до, в течение и после смены изображения, представлено на рис. 29.2.



**Рис. 29.2.** Переход беспорядочным методом до смены изображения (вверху), в течение (посередине) и после (внизу)

## SmashTransition.java

Класс SmashTransition заменяет одно изображение на другое, опуская новое изображение на старое. Старое изображение представляется разрушающимся ("сминающимся") под весом нового изображения.

Для создания фреймов используются две экземплярные переменные `drop_amount` и `location`. Переменная `location` хранит положение пикселя, на котором начинается смятое изображение. Переменная `drop_amount` хранит число пикселов нового изображения, которое выпадает на смятое изображение

каждого фрейма. Другими словами, это число, которое нужно добавлять к переменной `location` каждого фрейма. Статический массив с именем `fill_pixels` используется для того, чтобы выкрасить в белый цвет целую строку массива `work_pixels`.

Эффект смятия выполняется рисованием старого изображения в форме, похожей на "меха аккордеона"<sup>1</sup>. Оно начинается с рисования первых строк старого изображения со смещением вправо. Каждая следующая строка смещена вправо немного больше предыдущей. Это продолжается, пока не будет достигнут некоторый максимум левого смещения. С этой точки левое смещение каждой строки сокращается, пока не достигнет нуля. Процедура продолжается до тех пор, пока все строки "ломаного" изображения не будут нарисованы.

Строки старого изображения не рисуются полностью. В рисунке используется длина строки, которая немного короче, чем фактическая длина.

Число строк для рисования ломаного изображения уменьшается с каждым фреймом, поскольку старая доска объявлений становится все более компактной. Этот переход использует строки, которые равномерно распределены по старому изображению, и гарантирует также, что ломаное изображение не представляется выпадающим из низа апплета или скользящим под новым изображением.

## Метод `setupFillPixels()`

Чтобы гарантировать, что массив `fill_pixels` инициализирован и содержит по крайней мере одну целую строку этого апплета, используется статический метод `setupFillPixels()`. Если массив еще не был инициализирован или имеет недостаточный размер для этого апплета, то метод вновь создает или создает и заполняет этот массив. Если имеются больше одного выполняющегося экземпляра этого апплета, оба могут совместно использовать (разделять) массив `fill_pixels`, но он должен быть, по крайней мере, того же размера, что самый большой апплет.

## Метод `init()`

Метод `init()` для этого перехода начинается, подобно всем другим переходам, с обращения к методу `init()` базового класса. За этим следует обращение к методу `setupFillPixels()`, описанному ранее. Затем рассчитываются начальные значения переменных `drop_amount` и `location`. После этого метод `init()` входит в цикл, чтобы создать каждый фрейм. В действительности он образует их в обратном порядке, создавая сначала последний фрейм. Фреймы, на самом деле, не должны создаваться в обратном порядке. Однако выполнение циклов в обратном порядке сокращает размер результирующего

<sup>1</sup> То есть изображается смятым "в гармошку". — Примеч. пер.

файла .class. После того как каждая ячейка создана, переменная location инкрементируется к следующему подходящему положению.

## Метод *Smash()*

Метод *Smash()* изменяет массив *work\_pixels* для следующей ячейки. Он создает искаженное изображение старой доски объявлений в массиве *work\_pixels* и выводит пиксели в новое изображение. Данный метод получает один параметр, *max\_fold*, используемый как максимальное правое смещение, которое будут иметь строки в "смятом" изображении. Этот параметр используется также для вычитания из ширины строки при определении длины искаженных строк.

Метод начинается копированием пикселов из нового изображения в *work\_pixels*. Затем он инициализирует ряд переменных, которые использует для вычерчивания ломаного изображения. Процесс выполняется в цикле, строка за строкой. В пределах цикла текущая строка сначала делается полностью белой. Затем над этой строкой копируется порция правильных строк из старой доски объявлений. Чтобы получить эффект гармошки, вычерчивание не начинается с той же пиксельной позиции, с которой вычерчивалась белая строка. Вместо этого целевые пиксели смещаются вправо на несколько пикселов. После вычерчивания строки к счетчику смещения прибавляется число. Затем следует проверка границ, чтобы выяснить, не вышло ли смещение за пределы своего минимума или максимума. Если это имеет место, то меняется знак числа, которое добавляется к счетчику смещения каждой строки. В результате направление смещения изменяется.

## Код

Исходный код класса *SmashTransition*:

```
import java.awt.*;
import java.awt.image.*;

public class SmashTransition extends BillTransition {
    final static int CELLS = 8;
    final static float FOLDS = 8.0f;
    static int[] fill_pixels;

    static void setupFillPixels(int width) {
        if(fill_pixels != null && fill_pixels.length <= width) {
            return;
        }
        fill_pixels = new int[width];
        for(int f = 0; f < width; ++f) {
            fill_pixels[f] = 0xFFFFFFFF;
        }
    }
}
```

```

int drop_amount;
int location;

public void init(Component owner, int[] current, int[] next) {
    init(owner, current, next, CELLS, 160);
    setupFillPixels(cell_w);
    drop_amount = (cell_h / CELLS) * cell_w;
    location = pixels_per_cell - ((cell_h / CELLS) / 2) * cell_w;
    for(int c = CELLS - 1; c >= 0; --c) {
        try { Thread.sleep(100); } catch (InterruptedException e) {}
        Smash(c + 1);
        try { Thread.sleep(150); } catch (InterruptedException e) {}
        createCellFromWorkPixels(c);
        location -= drop_amount;
    }
    work_pixels = null;
}

void Smash(int max_fold) {
    System.arraycopy(next_pixels, pixels_per_cell - location,
                    work_pixels, 0, location);
    int height = cell_h - location / cell_w;
    float fold_offset_adder = (float)max_fold * FOLDS / (float)height;
    float fold_offset = 0.0f;
    int fold_width = cell_w - max_fold;
    float src_y_adder = (float)cell_h / (float)height;
    float src_y_offset = cell_h - src_y_adder / 2;
    for(int p = pixels_per_cell - cell_w; p >= location; p -=
        cell_w) {
        System.arraycopy(fill_pixels, 0, work_pixels, p, cell_w);
        System.arraycopy(current_pixels, (int)src_y_offset * cell_w,
                        work_pixels, p + (int)fold_offset, fold_width);
        src_y_offset -= src_y_adder;
        fold_offset += fold_offset_adder;
        if(fold_offset < 0.0 || fold_offset >= max_fold) {
            fold_offset_adder *= -1.0f;
        }
    }
}
}

```

Как выглядит смятый<sup>1</sup> (smash) переход до, в течение и после смены изображения, представлено на рис. 29.3.

---

<sup>1</sup> Переход, похожий на удар новым изображением сверху по старому такой, что старое изображение сжимается "в гармошку". — Примеч. пер.



**Рис. 29.3.** Переход сминающимся методом до смены изображения (вверху), в течение (посередине) и после (внизу)

## TearTransition.java

Переход `TearTransition` создает иллюзию разворачивающегося обрывка в левом верхнем углу изображения.

В этом переходе используется только одна компонентная переменная `x_cross`. Она играет роль множителя для формирования эффекта обрывка (в левых частях верхних строк изображения). Чем больше значение этой переменной, тем меньше будет проявляться эффект обрывка в начале верхних строк<sup>1</sup>.

Код данного перехода содержит много оптимизаций. Одна из оптимизаций создает фреймы ячеек в обратном порядке, переставляя местами первый и последний фреймы. При нормальном порядке каждый последующий фрейм отображается в нижней части чуть больше нового изображения. Если бы фреймы создавались в обычном порядке, эффект обрывка должен был бы рисоваться вместе с пикселями нового изображения, выведенными в текущий фрейм так, что он прикрыл бы большую часть изображения предыдущего фрейма<sup>2</sup>. При построении фреймов в обратном порядке нужно пересыпывать только область эффекта обрывка во фрейме каждой ячейки. Например, последний фрейм, созданный первым, начинается эффектом обрывка, использующим только маленькую часть левого верхнего угла изображения, в то время как остальные пиксели берутся из изображения новой доски объявлений. В следующем фрейме (который по отношению к последнему фрейму создается вторым) новый эффект обрывка рисуется на не-

<sup>1</sup> Новое изображение появляется поверх предыдущего со свернутым верхним левым углом, и этот угол постепенно разворачивается, в итоге, полностью закрывая остатки прежнего изображения (в верхнем левом углу окна апплета). — Примеч. пер.

<sup>2</sup> То есть верхний левый угол текущего изображения сворачивался бы, вместо того, чтобы разворачиваться. — Примеч. пер.

сколько бит выше левого верхнего угла изображения, в то время как остальная часть изображения остается прежней. Следующие фреймы ячеек рисуют все больше и больше частей нового изображения (в его верхнем левом углу) над изображением предыдущего фрейма.

## Метод *init()*

Метод *init()* для рассматриваемого перехода начинается, подобно всем другим переходам, с обращения к методу *init()* базового класса. Затем он копирует пиксели всей новой доски объявлений в массив *work\_pixels* и размещает первую строку пикселов старой доски объявлений на первой строке массива *work\_pixels*. После того как переменная *x\_cross* инициализирована, метод *init()* организует обратный цикл через фреймы всех ячеек. Внутри цикла он создает каждый фрейм и уменьшает значение переменной *x\_cross*.

## Метод *Tear()*

Метод *Tear()* модифицирует массив *work\_pixels* для следующей ячейки. Он вычерчивает эффект обрывка на рабочих пикселях. Эффект обрывка выводится строка за строкой. Чтобы отобразить одну строку, метод копирует пиксели старого изображения в массив *work\_pixels*. При этом он использует два счетчика: один является индексом в массиве *work\_pixels* (*адресате*), а второй — ссылкой на индекс в массиве пикселов старой доски объявлений (*источнике*). Оба счетчика стартуют с нуля. Счетчик адресата всегда инкрементируется единицей. Счетчик источника, однако, инкрементируется числом с плавающей точкой, которое всегда больше чем единица. Цикл выполняется, пока индекс адресата не становится больше ширины строки, причем индекс источника растет быстрее, чем индекс адресата. Полный эффект заключается в том, что в адресате только несколько пикселов на левом краю изображения будут скопированы из источника. Некоторые пиксели, принятые из источника, будут пропущены, а остальные — равномерно распределены по строке.

Каждая строка текущего фрейма будет использовать большее значение числа с плавающей точкой, по сравнению с предыдущей строкой. Это приводит к тому, что чем ближе строка располагается к основанию изображения, тем меньше пикселов эффекта обрывка в ней выводится.

Данный метод выполняет серьезную оптимизацию цикла для устранения медленной индексации массивов (что свойственно для самого языка Java). Всякий раз, когда используется элемент в массиве, выполняется проверка границ для гарантии, что индекс находится в пределах границ массива. Имеется бит эффективности, включенный в эту проверку границ. Стандартный Java-класс *System* обеспечивает метод, который позволяет копировать разделы массивов из одного массива в другой так же (или почти так же) бы-

стро, как производится копирование одного элемента массива в другой. Такой механизм используется для того, чтобы ускорить создание строк во фреймах ячейки, и только тогда, когда апплет знает, что некоторые из исходных пикселов будут смежны с друг другом. Если апплет пропускает, по крайней мере, каждый следующий пиксел из исходного изображения, то будет применен стандартный циклический метод. Значение `x_increment`, меньшее чем 0.5, указывает, что к индексному счетчику источника каждый раз будет добавляться число, меньшее чем 1.5, и для специфической строки будет иметь место выигрыш по скорости от использования метода копирования массива.

## Код

Исходный код класса `TearTransition`:

```
import java.awt.*;
import java.awt.image.*;

public class TearTransition extends BillTransition {
    static final int CELLS = 7;
    static final float INITIAL_X_CROSS = 1.6f;
    static final float X_CROSS_DIVISOR = 3.5f;
    float x_cross;

    public void init(Component owner, int[] current, int[] next) {
        init(owner, current, next, CELLS);
        System.arraycopy(next_pixels, 0, work_pixels, 0, pixels_per_cell);
        System.arraycopy(current_pixels, 0, work_pixels, 0, cell_w);

        x_cross = INITIAL_X_CROSS;

        for(int c = CELLS - 1; c >= 0; --c) {
            try { Thread.sleep(100); } catch (InterruptedException e) {}
            Tear();
            try { Thread.sleep(150); } catch (InterruptedException e) {}
            createCellFromWorkPixels(c);
            x_cross /= X_CROSS_DIVISOR;
        }
        work_pixels = null;
    }

    final void Tear() {
        float x_increment;
        int p, height_adder;

        p = height_adder = cell_w;
        for (int y = 1; y < cell_h; ++y) {
            x_increment = x_cross * y;
```

```

if(x_increment >= 0.50f) {
    float fx = 0.0f;
    x_increment += 1.0f;
    int x = 0;
    do {
        work_pixels[p++] = current_pixels[height_adder + x];
        x = (int)(fx += x_increment);
    } while(x < cell_w);
}
else {
    float overflow = 1.0f / x_increment;
    float dst_end = overflow / 2.0f + 1.49999999f;
    int dst_start = 0, src_offset = 0, length = (int)dst_end;
    while(dst_start + src_offset + length < cell_w) {
        System.arraycopy(current_pixels, p + src_offset,
                         work_pixels, p, length);
        ++src_offset;
        dst_end += overflow;
        p += length;
        dst_start += length;
        length = (int)dst_end - dst_start;
    }
    length = cell_w - src_offset - dst_start;
    System.arraycopy(current_pixels, p + src_offset,
                     work_pixels, p, length);
}
p = height_adder += cell_w;
}
}
}

```

Рис. 29.4 демонстрирует, как выглядит обрывочный (tear) переход до, в течение и после смены изображения.



Рис. 29.4. Переход методом разворачивающегося обрывка до смены изображения (вверху), в течение (посередине) и после (внизу)

## ***UnrollTransition.java***

Переход `UnrollTransition` проявляется (на экране) так, будто скатанный (в рулон) плакат, помещенный в основание окна апплете, раскатывается вверх, постепенно показывая новое и закрывая старое изображение. Чтобы расширить иллюзию развертывания, рулон постепенно уменьшается в диаметре (по пути к верху окна апплете).

Во время создания такого перехода используются две экземплярные переменные. Переменная `location` указывает на элементы в массиве пикселов. Она хранит текущую позицию начального пикселя рулона в этом массиве. Массив `unroll_amount` сообщает классу, на сколько пиксельных позиций должен перемещаться рулон в любом фрейме.

Наиболее трудная часть создания фрейма каждой ячейки — это рисование рулона. Вторая задача — это рисование пикселов нового изображения на освободившееся (при перемещении рулона) место в предыдущем фрейме.

В каждом из восьми создаваемых фреймов рулон, перемещаясь вверх, рисует (под собой) несколько строк нового изображения. Поскольку рулон разворачивается во фрейме снизу вверх, порции строк нового изображения прорисовываются в обратном порядке. Например, если рулон расположен на десятой строке фрейма, то в строку, находящуюся сразу же под ним, будет выводиться *девятая* строка нового изображения. Следующей строкой под перемещающимся вверх рулоном рисуется *восьмая* строка нового изображения, потом *седьмая* и т. д. Этот процесс повторяется до тех пор, пока не будет заполнена новым изображением часть текущего фрейма, находящаяся под рулоном.

Сам рулон отображается в объемном виде (каждая строка рисунка рулона вычерчивается с небольшим левым смещением). При этом строки, близкие к центру рулона, рисуются с большим смещением, чем линии, близкие к его верхнему и нижнему краям. Затем верхние и нижние строки рулона оттеняются так, чтобы создать иллюзию подсветки над апплетеом. Эффект к тому, что верхние линии в изображении рулона выглядят немного ярче, чем остальные, а нижние — немного темнее.

### ***Метод `createUnrollAmountArray()`***

В этом переходе каждый следующий фрейм разворачивает рулон по окну апплете на несколько бит меньше<sup>1</sup>, чем предыдущий. Для вычисления значений элементов массива, указывающего, на сколько пикселов каждый фрейм должен продвинуть рулон (в массиве пикселов), используется статический метод `createUnrollAmountArray()`.

<sup>1</sup> Смещение рулона по вертикали отсчитывается от верхнего края фрейма, поэтому при разворачивании рулона снизу вверх это смещение в каждом последующем фрейме будет уменьшаться. — Примеч. пер.

## Метод *init()*

Метод *init()* этого перехода начинается аналогично всем другим переходам с обращения к методу *init()* базового класса. Затем инициализируется переменная *location* (в ней устанавливается индекс, выходящий за пределы индексов массива пикселов). За этим следует копирование всех пикселов старой доски объявлений в массив *work\_pixels*.

Далее, из хэш-таблицы *object\_table* извлекается массив, который хранит набор пикселов, необходимых для разворачивания каждого фрейма. Если подходящий массив не существует, он создается заново и сохраняется в *object\_table*.

Затем метод *init()* выполняет цикл, создающий и подготавливающий фрейм каждой ячейки. При подготовке фрейма *init()* вызывает прерывание текущего потока до и после создания<sup>1</sup> фрейма ячейки (что позволяет выполняться другим потокам в многозадачной среде Java). После того как создается фрейм (из массива *work\_pixels*, с помощью вызова метода *createCellFromWorkPixels()*), на нем рисуется область с рулоном и пикселями из нового изображения. Подчеркнем, что здесь происходит только подготовка массива *work\_pixels* для фрейма следующей ячейки.

## Метод *Unroll()*

Метод *Unroll()* модифицирует массив *work\_pixels* следующей ячейки. Он также рисует рулон на рабочих пикселях. Данный метод сначала вычисляет смещение, которое необходимо использовать для рисования каждой строки рулона. Затем он выполняет циклы через каждую строку рулона, копируя строки из нового изображения в массив *work\_pixels*. Пиксели, которые используются для левого смещения каждой строки рулона, заполняются пикселями из статического массива *fill\_pixels*.

Наконец, последний цикл (с шагом через каждый пикセル на верхних и нижних строках рулона) освещает пиксели на верхних строках рулона и затемняет — на нижних.

## Код

Исходный код класса *UnrollTransition*:

```
import java.awt.*;
import java.awt.image.*;
public class UnrollTransition extends BillTransition {
```

<sup>1</sup> Процесс создания фрейма требует интенсивной работы процессора, поэтому для его выполнения необходимо приостанавливать текущий поток. — Примеч. пер.

```
final static int CELLS = 9;
static int fill_pixels[] = { 0xFFFFFFFF, 0x00000000,
                            0xFF000000, 0xFFFFFFF0 };
private static int[] createUnrollAmountArray(int cell_h) {
    float unroll_increment =((float)cell_h / (float)(CELLS + 1)) /
        ((float)(CELLS + 2) / 2.0f);
    int total = 0;
    int unroll_amount[] = new int[CELLS + 1];
    for(int u = 0; u <= CELLS; ++u) {
        unroll_amount[u] = (int)(unroll_increment * (CELLS - u + 1));
        total += unroll_amount[u];
    }
    if(total < 0) {
        unroll_amount[0] -= 1;
    }
    return unroll_amount;
}
int location;
int[] unroll_amount;
public void init(Component owner, int[] current, int[] next) {
    init(owner, current, next, CELLS, 220);
    location = pixels_per_cell;
    System.arraycopy(current_pixels, 0, work_pixels, 0, pixels_per_cell);
    unroll_amount = (int[])object_table.get(getClass().getName() +
        cell_h);
    if(unroll_amount == null) {
        unroll_amount = createUnrollAmountArray(cell_h);
        object_table.put(getClass().getName() + cell_h, unroll_amount);
    }
    for(int c = 0; c < CELLS; ++c) {
        location -= unroll_amount[c] * cell_w;
        try { Thread.sleep(150); } catch (InterruptedException e) {}
        Unroll(c);
        try { Thread.sleep(100); } catch (InterruptedException e) {}
        createCellFromWorkPixels(c);
        System.arraycopy(next_pixels, location, work_pixels, location,
                        unroll_amount[c] * cell_w);
    }
    work_pixels = null;
}
void Unroll(int c) {
    int y_flip = cell_w;
    int offset[] = new int[unroll_amount[c]];
    for(int o = 0; o < unroll_amount[c]; ++o) {
        offset[o] = 4;
    }
    offset[0] = 2;
```

```

if(unroll_amount[c] > 1) {
    offset[1] = 3;
}
if(unroll_amount[c] > 2) {
    offset[unroll_amount[c] - 1] = 2;
}
if(unroll_amount[c] > 3) {
    offset[unroll_amount[c] - 2] = 3;
}
int offset_index = 0;
int end_location = location + unroll_amount[c] * cell_w;
for(int p = location; p < end_location; p += cell_w) {
    System.arraycopy(next_pixels, p - y_flip + offset[offset_index],
                    work_pixels, p, cell_w - offset[offset_index]);
    System.arraycopy(fill_pixels, 0, work_pixels,
                    p + cell_w - offset[offset_index],
                    offset[offset_index]);
    ++offset_index;
    y_flip += cell_w + cell_w;
}
for(int x = location + cell_w - 1; x > location; --x) {
    work_pixels[x] |= 0xFFAAAAAA;
    work_pixels[x + unroll_amount[c]] &= 0xFF555555;
}
}
}
}

```

Рис. 29.5 демонстрирует, как выглядит разворачивающийся (unroll) переход до, в течение и после замены изображения.

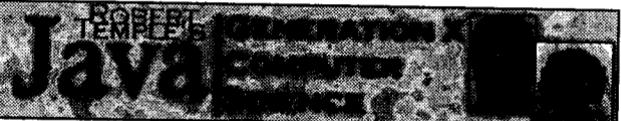


Рис. 29.5. Переход методом раскатывающегося рулона до смены изображения (вверху), в течение (посередине) и после (внизу)

## Динамический код

Роберт показал, как можно создавать интерактивную высокоэффективную графику, работая с многими из очевидных ограничений языка Java. Он также показал, как нужно использовать метод `System.arraycopy()`, чтобы эффективно перетасовывать окружающие пиксельные данные, и продемонстрировал, как можно использовать кооперативную многопоточность для выполнения вычислений и сетевых передач в фоновом режиме, а также динамически загружать файлы класса, не навлекая на себя синдром начального "ожидания серого блока", часто связываемый с Java-апплетами. Роберт доказал, что на языке Java можно писать высокоэффективные алгоритмы с прямой манипуляцией пикселями.

В дополнение к интересному коду апплет `DynamicBillboard` неотразим для непрограммистов и прочих пользователей. Он легко конфигурируется с помощью редакторов HTML, расширенных Java-программистами, и интересен Web-пользователям. В веке рекламных норм, управляемых "щелчком (мыши)", где рекламодатели хотят оплачивать только передачи от сайта, содержащего нужную им информацию, к их собственному сайту, апплет Роберта можно использовать для увеличения трафика и, в конечном счете, для увеличения дохода.

На рис. 26.6—26.16 показаны все переходы (в начальных и конечных состояниях).



Рис. 29.6. Исходное изображение



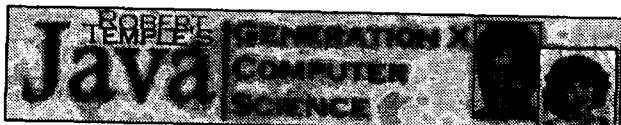
Рис. 29.7. Переход беспорядочным методом



Рис. 29.8. Второе изображение



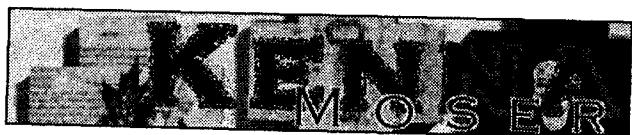
Рис. 29.9. Переход методом раскатывающегося рулона



**Рис. 29.10.** Третье изображение



**Рис. 29.11.** Переход сминающимся методом



**Рис. 29.12.** Четвертое изображение



**Рис. 29.13.** Переход колоночным методом



**Рис. 29.14.** Пятое изображение



**Рис. 29.15.** Переход методом разворачивающегося обрывка



**Рис. 29.16.** Последнее изображение



## ГЛАВА 30

# *ImageMenu:* Web-меню изображений

Апплет *ImageMenu* — это простая программа, которая представляет меню изображений (image-based menu) с произвольным числом выбираемых элементов в вертикальном списке. Когда пользователь перемещает указатель мыши по этим элементам, то тот, над которым размещается указатель, изменяет свой вид, сообщая тем самым, что на нем можно выполнить щелчок мыши. Когда пользователь щелкает по выбранному элементу, Web-браузер переходит к новому документу<sup>1</sup>, указанному этим элементом.

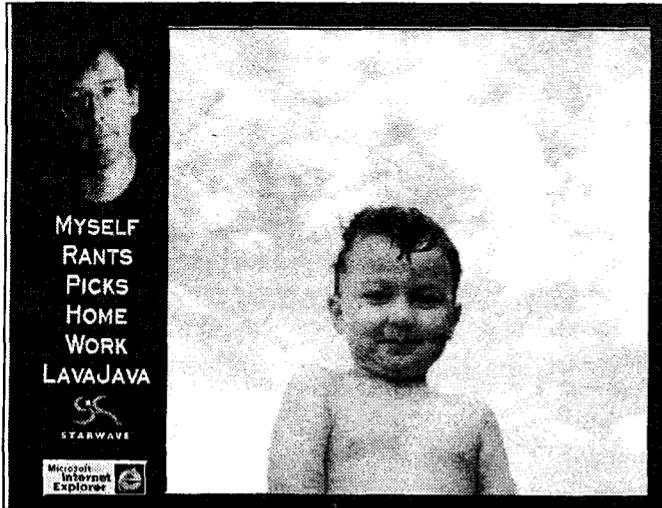
Апплет *ImageMenu* был создан Дэвидом Лавалли (David LaVallee) — старшим научным сотрудником Starwave Corporation. Дэвид написал несколько забавных и удивительных апплетов, которые он поддерживает в области, называемой "LavaJava". Он писал UI-код (код интерфейса пользователя) на языке Oak, предшественнике Java, в 1991 г. Такой подход кажется сомнительным. Посетите его домашнюю страницу в <http://www.starwave.com/people/lavallee>, изображенную на рис. 30.1. Как можно видеть, она включает экземпляр *ImageMenu*.

*ImageMenu* использует функцию `showDocument()` из `AppletContext`, чтобы делать гипертекстовый переход к новым страницам. Новинкой *ImageMenu* является то, что для изображения меню на экране он разместил различные части одиночного исходного изображения. Базирование меню на изображении, а не на тексте, освобождает вас от проектирования меню, которое использует какой-либо шрифт или изображение. Вы можете также обеспечивать различные типы выборочной обратной связи. Вам не нужно больше полагаться на налагающие различные ограничения AWT-функции.

Апплет *ImageMenu* был навеян апплетом с именем *Navigation*, созданным первоклассным Java-программистом Скэном Велчем (Scan Welch). *Navigation*

<sup>1</sup> Как выбранный, так и заменяющий его документы являются изображениями. — Примеч. пер.

и ImageMenu различаются эффективностью (повышенной пропускной способностью) и спецификациями тега апплена. Апплена Navigation использует исходное изображение, содержащее последовательный набор изображений меню во всех их возможных состояниях. Ширина исходного изображения равна ширине элемента меню (т. е. окна апплена), умноженной на число состояний. Оба апплена загружают одиночное изображение, что намного эффективнее в Internet, чем загрузка множественных файлов. Семь меню-изображений размером 100×140 (пикселов) в апплете Navigation образуют исходное изображение размера 700×140. Описанный здесь апплена (ImageMenu) использует исходное изображение с удвоенной шириной апплена (200×140 пикселов). Большинство Web-проектировщиков ненавидят печатание (на клавиатуре), тем более что они не должны этого делать. Это приводит ко второму существенному различию между Navigation и ImageMenu — сокращению параметров апплена.



**Рис. 30.1.** Домашняя страница Дэвида с ImageMenu

В то время как ImageMenu намного эффективнее, т. к. использует меньшее исходное изображение и меньшее количество байтов параметров, Navigation Велча имеет одну неподражаемую черту — он может отображать индивидуально выбранные "состояния", которые стекают сверху в пространство следующего пункта меню. Апплена ImageMenu требует, чтобы каждый пункт меню содержался в отдельной прямоугольной области, которая не может накладываться со смежными элементами. Это запрещает, например, букве с асцентом (возвышающейся частью, такой, например, как у буквы h) перекрываться с буквой с десцентом (спускающейся частью, такой как у буквы j), находящейся в той же позиции в вышележащей строке.

## Исходное изображение

Хотя вы и не будете видеть здесь код апплета *Navigation*, просмотр его GIF-изображения и тега <applet> ясно показывает, что *ImageMenu* собирается делать. Исходное изображение для *Navigation* на рис. 30.2 содержит семь столбцов, каждый из которых обеспечивает визуальное представление возможного выбора. Однако любой выбираемый элемент имеет только два состояния, так что каждая строка располагает пятью<sup>1</sup> избыточными копиями невыбранного состояния.

Исходное изображение для *ImageMenu* показано на рис. 30.3. По нему просто представить любое из семи возможных состояний меню с шестью выборами (выбираемыми элементами). Сначала *drawImage()* отображает левую половину исходного изображения. Это состояние, где нет выбранных элементов. Если выбрать любой из элементов, то вдоль контура выбранного элемента устанавливается ограничивающий прямоугольник и для отображения правой стороны изображения используется метод *drawImage()*. Он будет рисовать только ячейку, выбранную через ограничивающий прямоугольник.



Рис. 30.2. Исходное изображение для апплета *Navigation*

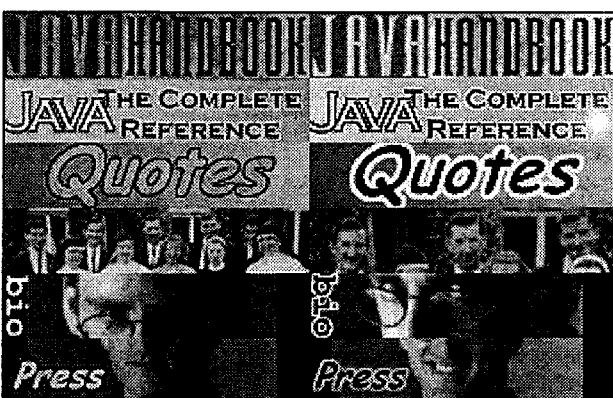


Рис. 30.3. Исходное изображение для апплета *ImageMenu*

<sup>1</sup> Левое изображение отражает состояние меню-изображения "без выборки", поэтому, вероятно, в счет избыточных элементов оно не входит. — Примеч. пер.

## Тег <applet>

Ниже показан тег <applet> для программы Navigation. Параметры апплета детализируют источник class-файла, размер апплета на экране, источник изображения и несколько пар параметров для прямоугольников и URL-адресов, которые указывают, куда послать поддерживающий Java-браузер, когда пользователь щелкает мышью в прямоугольнике.

```
<applet codebase="/java/Classes" code="Navigation" width=87 height=131>
<param name=color value="#ffffff">
<param name=image value="http://www.suck.com/nav/waynewstrip.gif">
<param name=zone_1_coord value="0 0 87 32">
<param name=zone_1_url value="http://www.suck.com/">
<param name=zone_2_coord value="0 33 87 53">
<param name=zone_2_url value="http://www.suck.com/vacuum/">
<param name=zone_3_coord value="0 54 87 72">
<param name=zone_3_url value="http://www.suck.com/pitch/">
<param name=zone_4_coord value="0 73 87 90">
<param name=zone_4_url value="http://www.suck.com/filler/">
<param name=zone_5_coord value="0 91 87 110">
<param name=zone_5_url value="http://www.suck.com/zerobaud/">
<param name=zone_6_coord value="0 109 87 131">
<param name=zone_6_url value="http://www.suck.com/netmoguls/">
</applet>
```

Тег <applet> для ImageMenu имеет так же много частей информации, как и тег для Navigation, но он требует меньше половины строк и намного легче читается. Мы будем использовать java.util.StringTokenizer для чтения подпараметров urlList и targetList, чьи значения представляют собой списки значений, разграниченных знаком "плюс" (+). Мы также подсчитываем координаты каждого пункта меню делением высоты апплета на число URL, анализируемых в urlList. Для читабельности тега <applet> мы допускаем префикс и суффикс, которые будут сцепляться с URL, когда наступит время перемещения на новую страницу.

```
<applet code="ImageMenu" width=140 height=180 hspace=0 vspace=0>
<param name="img" value="menu.jpg">
<param name="urlPrefix"
       value="http://www.starwave.com/people/naughton/">
<param name="urlList"
       value="book/index+book2/index+quotes+family/index+bio+press">
<param name="targetList"
       value="_self+_self+_self+_self+_self+_self">
<param name="urlSuffix" value=".html">
</applet>
```

## Методы

Здесь будет представлен маленький аплет — точно 100 исходных строк Java. Мы пройдемся через все восемь методов и затем, в конце главы, покажем весь исходный код целиком.

### Метод *init()*

Когда аплет инициализируется, *init()* сохраняет размер в переменной *Dimension d* и анализирует теги *<param>* аплета. Затем он использует  *StringTokenizer* для анализа строк, разграниченных знаком "плюс", и создания строчных массивов *url[]* и *target[]*. Число анализируемых URL используется для деления вертикального пространства на ячейки меню. Из этих вычислений *init()* сохраняет номер и высоту ячеек в переменных *cells* и *cellH*.

### Метод *update()*

Мы обнуляем метод *update()* класса *Applet*, чтобы избежать мерцания. Как было упомянуто в главе 23, метод *update()* суперкласса *Applet* перед вызовом *paint()* наполняет прямоугольную панель аплета цветом фона. Так как мы не собираемся использовать *repaint()*, то можно полностью исключить обновление.

### Метод *lateInit()*

Частный метод *lateInit()* создает внеэкранный объект *Image*, который будет применяться для двойной буферизации образа меню. Данный метод также использует класс *MediaTracker* для синхронного получения исходного изображения.

### Метод *paint()*

Метод *paint()* весьма прост. Сначала он проверяет, был ли уже создан внеэкранный буфер. Если нет, вызывается *lateInit()*, чтобы создать буфер и загрузить изображение меню.

После чего он выводит левую половину изображения во внеэкранный буфер. Конечно, для этого требуется, чтобы изображение меню было вдвое шире аплета. Таким образом, когда будет вызываться *drawImage(img, 0, 0, null)*, аплет просто отсечет правую половину изображения. Далее, если выбирается какая-либо ячейка (*selectedCell >= 0*), то по границам подсвеченного пункта меню устанавливается ограничивающий прямоугольник. Обратите внимание, что *paint()* каждый раз получает графический контекст внеэк-

ранного изображения. Это создает эффект переустановки ограничивающего прямоугольника по контуру изображения. Без AWT-метода `resetClip()` данный метод навязывает несколько странный стиль кодирования.

Далее, с помощью вызова `drawImage(img, -d.width, 0, null)` полное изображение рисуется снова, но на сей раз оно смешено на ширину апплета влево. Это создает эффект помещения правого высвеченного пункта меню в ограничивающий прямоугольник. Наконец, внеэкранный буфер копируется в окно апплета.

### Замечание

Быстродействие большинства графических дисплеев в значительной степени сдерживается скоростью прямого доступа CPU к экрану. Кроме того, много современных видеокарт оптимизированы для копирования прямоугольных областей из памяти на дисплей с целью поддержки систем управления окнами. Поэтому намного лучше выполнять все рисование во внеэкранном буфере, а не копировать биты на экран. На подобных PC-системах мы наблюдали от 10 до 400 изменений буфера в секунду, в зависимости от глубины пикселя и архитектуры видеокарты.

## Метод *mouseExited()*

Для метода `mouseExited()` необходима специальная обработка, потому что он выполняет выбор всех пунктов меню. Все, что нужно сделать — только установить переменные `selectedCell` и `oldCell` в `-1`. Это вынуждает последующий вызов `paint()` показывать все пункты как невыбранные. Установка `oldCell` в `-1` означает, что следующий раз, когда указатель мыши перейдет на апплет и вызовет `mouseMoved()`, первый пункт будет нарисован должным образом.

## Метод *mouseDragged()*

Метод `mouseDragged()` вызывается, когда мышь перемещается с нажатой кнопкой (любой). В этом апплете при перетаскивании или передвижении мыши мы хотим делать одно и то же, поэтому просто вызываем метод `mouseMoved()`, передавая ему те же самые параметры, которые приняли.

## Метод *mouseMoved()*

Всякий раз, когда мышь передвигается, метод `mouseMoved()` проверяет координату `y`, чтобы видеть, какая из ячеек была выбрана. Если значение `selectedCell` отличается от `oldCell`, то значит пользователь передвинулся из одной ячейки в другую, и меню перерисовывается. Это — оптимизация, которая избегает постоянного перерисовывания идентичных экраных битов каждый раз, когда мышь передвигается. Обратите внимание, что `repaint()`

здесь не вызывается. Мы сокращаем нормальный протокол апплета, напрямую вызывая `paint()` после выборки контекста `Graphics` из `getGraphics()`. Это обеспечивает существенно более быстрый ответ. После рисования меню выполняется отображение вновь выбранного пункта в строке состояния, кроме того, он сохраняется в переменной `oldCell`.

## Метод `mouseReleased()`

Метод `mouseReleased()` отсылает браузер к URL-адресу, который соответствует выбранному в настоящее время пункту меню. Сначала создается желательный URL-объект. Если в теге `<applet>` URL-адрес был отформатирован ненадлежащим образом, то на строке состояния отображается сообщение об ошибке и выполняется возврат без попытки переключать документы. Метод `ShowDocument()` помещает документ, описанный в URL, во фрейм, указанный в массиве `target`. И, наконец, проверяется состояние клавиши `<Shift>` (с помощью вызова метода `isShiftDown()` класса `MouseEvent`). Если клавиша `<Shift>` была нажата, то документ, описанный в URL, открывается в новом пустом окне браузера (а не в том, что указан в `target`).

## Код

Исходный код `ImageMenu`:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.util.*;
import java.net.*;

public class ImageMenu extends Applet {
    Dimension d;

    Image img, off;
    Graphics offg;
    int MAXITEMS = 64;
    String url[] = new String[MAXITEMS];
    String target[] = new String[MAXITEMS];
    String urlPrefix, urlSuffix;
    int selectedCell = -1;
    int oldCell = -1;
    int cellH;
    int cells;

    public void init() {
        d = getSize();
        urlPrefix = getParameter("urlPrefix");
        urlSuffix = getParameter("urlSuffix");
```

```
 StringTokenizer st;
st = new StringTokenizer(getParameter("urlList"), "+");
int i=0;
while(st.hasMoreTokens() && i < MAXITEMS)
    url[i++] = st.nextToken();
cells = i;
cellH = d.height/cells;
st = new StringTokenizer(getParameter("targetList"), "+");
i=0;
while(st.hasMoreTokens() && i < MAXITEMS)
    target[i++] = st.nextToken();
addMouseListener(new MyMouseAdapter());
addMouseMotionListener(new MyMouseMotionAdapter());
}

private void lateInit() {
off = createImage(d.width, d.height);
try {
    img = getImage(getDocumentBase(), getParameter("img"));
    MediaTracker t = new MediaTracker(this);
    t.addImage(img, 0);
    t.waitForID(0);
} catch(Exception e) {
    showStatus("error: " + e);
}
}

public void update(Graphics g) {}
public void paint(Graphics g) {
if(off == null)
    lateInit();

offg = off.getGraphics();
offg.drawImage(img, 0, 0, this);
if (selectedCell >= 0) {
    offg.clipRect(0, selectedCell * cellH, d.width, cellH);
    offg.drawImage(img, -d.width, 0, this);
}
g.drawImage(off, 0, 0, this);
}

class MyMouseMotionAdapter extends MouseMotionAdapter {
    public void mouseDragged(MouseEvent me) {
        mouseMoved(me);
    }
    public void mouseMoved(MouseEvent me) {
        int y = me.getY();
        selectedCell = (int)(y/(double)d.height*cells);
    }
}
```

```
if (selectedCell != oldCell) {
    paint(getGraphics());
    showStatus(urlPrefix + url[selectedCell] + urlSuffix);
    oldCell = selectedCell;
}
}

class MyMouseAdapter extends MouseAdapter {
    public void mouseExited(MouseEvent me) {
        selectedCell = oldCell = -1;
        paint(getGraphics());
        showStatus("");
    }

    public void mouseReleased(MouseEvent me) {
        // mouseMoved(me);
        URL u = null;
        try {
            u = new URL(urlPrefix + url[selectedCell] + urlSuffix);
        } catch(Exception e) {
            showStatus("error: " + e);
        }
        if (me.isShiftDown())
            getAppletContext().showDocument(u, "_blank");
        else
            getAppletContext().showDocument(u, target[selectedCell]);
    }
}
}
```

## Резюме

При использовании апплет ImageMenu может выглядеть **большим**, и это сильно влияет на очень маленькую программу. Применение метода `showDocument(URL u, String target)`

в этом апплете выполняет тонкую оптимизацию в проекте Web-страницы. Если вы помещаете апплет ImageMenu во фрейм набора HTML-фреймов и используете его для посылки документов во второй фрейм, апплет никогда не должен перезагружаться.



## ГЛАВА 31

# Апплет *Lavatron*: дисплей для спортивной арены

Lavatron — "лампочный" дисплей для спортивной арены. Как правило, апплет не имеет длительной хронологии, но этот — исключение. Дэвид Лавалли (David LaVallee), автор апплета *ImageMeli* (см. предыдущую главу), давно стремился достичь эффекта подобного вида. Хронология Lavatron начинает путь с 1974 г., когда Лавалли был игроком калифорнийской команды НХЛ *Golden Seals* (Золотой Тюлень). В 1979 г. Лавалли захватила идея графического программируемого табло, когда он был специалистом по ремонту мини-компьютера PDP 11/34 фирмы Digital Equipment Corporation (DEC), которая выполнила табло на Канадском Национальном Выставочном Стадионе (где имела обыкновение играть команда Toronto Blue Jays). Это табло было построено на простых старых 100-ваттных электролампах, подобных тем, что вы используете дома. В 1991 г. в Торонто было установлено табло Sony Jumbotron HDTV (в Скайдоме), имевшее натуральный цвет, изображения, видео, и вид Hard Rock Cafe тройной высоты. В 1992 г. Лавалли записал первую версию апплета *Lavatron* на Objective-C и PostScript. Наконец, в 1995 г. *Lavatron* был переписан заново, чтобы выполняться в среде Java. С тех пор он претерпел несколько итераций и улучшений эффективности. Версия, представленная здесь, была модифицирована для Java 2.

Существует много возможных усовершенствований апплета *Lavatron* (рис. 31.1), которые вы могли бы протестировать, например, рисование исходного изображения динамически, в памяти, а не загрузка его, или прокрутка анимированной последовательности. Но и в существующем виде — это интересный анимированный дисплейный апплет, который может оказаться весьма полезным для вас.

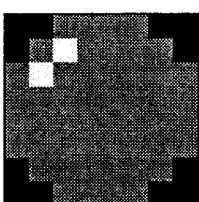
## Как работает *Lavatron*

В апплете *Lavatron* использован интересный и довольно хитрый способ построения экранного изображения горящей лампочки, а его побочный эф-

фект позволяет апплету очень быстро загружаться. Причина, по которой он загружается так быстро, — небольшое количество данных, передаваемых по сети. Исходное изображение имеет сжатый формат JPEG, так что оно в 64 раза меньшее, чем отображаемое. Каждый пиксель в исходном изображении масштабируется<sup>1</sup> в квадрате размером 8×8 пикселов. Это и есть та уловка, которую использует Lavatron, чтобы произвести эффект светящейся лампочки. Над цветными пикселями каждой масштабной клетки исходного изображения рисуется прозрачный кружок (размером 8×8 пикселов), окруженного черной границей с белой подсветкой для стилевых штрихов (имитирующих отблеск осветителя в верхней левой части лампочки). В целях оптимизации "лампочки" заранее смонтированы в изображение, которое может быть быстро нарисовано (колонка за колонкой). Рис. 31.2 показывает, что маска лампочки выглядит как "надутая". Два белых пикселя — это подсветка. Черные пиксели в углах непрозрачны. Наконец, все серые пиксели в середине прозрачны, что позволяет показывать через них цвет.



**Рис. 31.1.** Апплет Lavatron в действии — исходное изображение показано ниже апплета



**Рис. 31.2.** Изображение "раздутой" лампочки

<sup>1</sup> Масштабирование — преобразование с изменением размеров. В данном случае речь идет о нанесении каждого пикселя исходного изображения на двухмерную шкалу с размером ячейки 8×8 пикселов. — Примеч. пер.

Lavatron рисует так быстро, потому что он не должен перерисовывать то, что уже видно. Методика *копирования* области экрана, что — хорошо, и *рисование* только части, что — ново, используется во многих общих операциях, включая прокрутку. Функция `copyArea()` класса `awtGraphics` берет порцию изображения, определенную прямоугольником, и перемещает ее (с помощью  $(x, y)$ -смещения) из его начального положения. По оптимизации графического быстродействия, метод `copyArea()` трудно превзойти. Он существенно лучше по скорости работы любых других методик построения изображений, таких как использование `drawImage()` или `drawImage()` с `clipRect()`.

Очень быстрая методика построения изображений в Java-исполнении включает:

- построение изображения, много большего, чем апллет, который содержит несколько исходных изображений, склеенных в единое изображение;
- использование `copyArea()`, чтобы перемещать их на место;
- отсечение результирующего экрана.

## Исходный код

Lavatron начинает работу с инициализации данных, которая включает загрузку исходного изображения и создание столбца лампочек. Последняя стадия инициализации — рисование внеэкранного изображения (двойного буфера), заполненного неактивными (черным) лампочками (он используется для запуска дисплея с "пустым" изображением). Последующее рисование внеэкранного изображения начинается с использования `copyArea()` для передвижения существующей порции изображения влево на ширину столбца лампочек и добавления ее к правому краю апллета. Затем считаются пиксельные значения следующего столбца и применяются в качестве цвета для заполнения столбца из прямоугольников  $8 \times 8$  на правом краю апллета. Далее рисуется прозрачный столбец лампочек, и затем все изображение заднего плана выводится на экран. Так как этот апллет не должен ничего делать, кроме прокрутки изображения, он отказывается от нормального цикла `repaint()` в пользу следующей процедуры: поток, который неоднократно вызывает `paint()`, разветвляется и приостанавливается только для вызова метода `yield()`, разрешающего выполнение других потоков.

## Тег `<applet>`

Исходный код апллета Lavatron начинается с тега `<applet>`, показанного ниже. Апллет смотрится лучше всего, когда его ширина равна размеру лампочки (8) с четным множителем, а высота — размеру лампочки, умножен-

ному на высоту исходного изображения. Единственный параметр — имя файла исходного изображения (именуется в параметре `img`):

```
<applet code=Lavatron.class width=560 height=128>
<param name="img" value="swsm.jpg">
</applet>
```

## Lavatron.java

Главный апплет — небольшой, приблизительно 100 строк исходного кода Java. Однако требуется также класс поддержки, который описан в следующем разделе.

### Метод `init()`

Метод `init()` сначала определяет размер апплета, используя `getSize()`, и затем округляет его до размера, кратного размеру лампочки, указанному в `bulbs`, и сохраняет в `offw, offh`. Далее он создает изображение с размерами, указанными в `offscreen`, для использования в качестве двойного буфера для дисплея. Объект `Graphics`, применяемый для рисования на `offscreen`, сохраняется в `offGraphics`. Размер апплета, измеряемый в единицах лампочек, а не в пикселях, сохраняется в `bulbsW, bulbsH`.

Далее, с помощью вызова метода `createBulbs()`, в который передаются необходимые размеры, создается изображение столбца лампочек. Затем загружается изображение, именованное в параметре апплета `img`. Это делается передачей результата `getImage()` методу `addImage()` класса `MediaTracker` и последующим вызовом `waitForID()`, который ждет (перед возвратом управления), пока изображение не будет полностью загружено.

Чтобы нарисовать "лампочную" версию этого изображения, `init()` должен извлечь цветовую информацию каждого его пикселя. Сначала он получает размер изображения, используя `getWidth()` и `getHeight()`, и сохраняет ширину в переменной `pixscan`. Далее он выделяет массиву `pixels` память для хранения `pixscan*h` целых чисел. Затем создается объект класса `PixelGrabber` и вызывается его метод `grabPixels()`, заполняющий массив `pixels` цветными значениями.

Заключительный шаг в `init()` — рисование черных лампочек на внеэкранном изображении, которое делает эффект более ощущимым, когда изображение прокручивается с правой стороны, показывая освещенные лампочки.

### Метод `createBulbs()`

Метод `createBulbs()` возвращает `Image`-объект (т. е. модель изображения) стека изображений лампочек, который можно использовать для маскировки

столбца цветных квадратиков, придавая ему вид светящихся<sup>1</sup> лампочек. Это немного хитро, но довольно изящно.

Сначала он выделяет необходимый объем памяти для int-массива `pixels[]`, который хранит пиксели изображения лампочек. Затем он объявляет другой int-массив (`bulbBits[]`), который является цифровым изображением пикселов одной лампочки. Каждый пиксель в этом изображении представлен числами 0, 1 и 2. 0 представляет черный пиксель, 1 — прозрачный, а 2 — пиксель белой подсветки. Затем объявляется короткий (всего три элемента) линейный int-массив `bulbCLUT` (Color Look Up Table, таблица цветового вида лампочки). Он содержит цифры цветов пикселов лампочки (которые в `bulbBits` представлены числами 0, 1 и 2) в виде полных 32-разрядных значений, в которых старший байт соответствует алфавитному символу или прозрачности. Значение `xffff000000` представляет непрозрачный черный цвет, `0x00c0c0c0` — полностью прозрачный светло-серый а `0xffffffff` — непрозрачный белый.

Цикл `for` пробегает через все пиксели в массиве `pixels[]` и загружает в каждую пикセルную позицию подходящее значение из массива `bulbBits`. Для поиска цвета в массиве `bulbCLUT` используется операция сравнения по модулю `mod (%)`. При заданном массиве пикселов метод `createBulbs()` возвращает объект класса `Image`, который создается методом `createImage()`. В качестве аргумента данный метод получает вновь создаваемый объект `MemoryImageSource`, который конструируется из только что построенного (в предшествующем цикле) массива `pixels[]`.

## Метод `color()`

Метод `color()` возвращает цвет пикселя в (x, y)-позиции исходного изображения (в виде `Color`-объекта). Так как этот апплет выполняется непрерывно, мы решили не просто создавать новый объект типа `Color` каждый раз, когда рисуется одиночная лампочка, что было бы обременительно для динамической памяти сборщика мусора. Вместо этого, уникальные `Color`-объекты сохраняются в хэш-таблице. Максимальное число `Color`-объектов в хэш-таблице может быть равным произведению (ширина × высота) исходного изображения, но на практике число таких объектов обычно намного меньше.

## Метод `update()`

`Lavatron` переопределяет `update()` так, чтобы он ничего не делал, поскольку мы не хотим использовать AWT-методы, вызывающие мерцание.

---

<sup>1</sup> После наложения такой маски на изображение каждая ее лампочка будет не просто светиться ровным светом, а начнет показывать маленький круглый участок находящегося под ней исходного изображения. — Примеч. пер.

## Метод *paint()*

Метод *paint()* весьма прост. Первый его шаг вызывает *copyArea()*, чтобы переместить все столбцы влево на ширину одного столбца. Затем используется цикл *for* для заполнения самого правого столбца прямоугольниками с цветом соответствующего пикселя (используя *color()*). Далее, по новому столбцу рисуется полоса *bulb*-изображений (лампочек). Потом текущая позиция прокрутки *scrollX* модифицируется так, чтобы переместиться еще на одну ширину столбца вправо (для вычисления *scrollX* используется операция сравнения по модулю *pixscan*).

## Методы *start()*, *stop()* и *run()*

После запуска апплета создается и стартует новый Thread-поток *t*. Этот поток вызывает метод *run()*, который будет поддерживать вызов *paint()* с такой скоростью, с какой возможно, при одновременной поддержке своеевременного вызова *yield()*, так чтобы могли выполняться другие потоки. Когда апплетом вызывается метод *stop()*, *stopFlag* устанавливается в *true*. Эта переменная проверяется бесконечным циклом в методе *run()*. Программное управление преждевременно прерывает цикл, когда *stopFlag* имеет значение *true*.

Интересным усовершенствованием было бы введение порога фреймовой скорости (скорости передачи кадров изображения), скажем 30 fps (фреймов в секунду), и замена вызова *yield()* на соответствующий вызов *sleep()*, если выполнение становится слишком быстрым. С ростом скорости выполнения апплетов следует задуматься об их адаптивном поведении, чтобы достигнуть постоянных скоростей передачи кадров.

## Код

Исходный код класса *Lavatron*:

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class Lavatron extends Applet implements Runnable {
    int scrollX;
    int bulbsW, bulbsH;
    int bulbS = 8;
    Dimension d;
    Image offscreen, bulb, img;
    Graphics offgraphics;
    int pixels[];
    int pixscan;
    IntHash clut = new IntHash();
    boolean stopFlag;
```

```
public void init() {
    d = getSize();
    int offw = (int) Math.ceil(d.width/bulbS) * bulbS;
    int offh = (int) Math.ceil(d.height/bulbS) * bulbS;
    offscreen = createImage(offw, offh);
    offgraphics = offscreen.getGraphics();
    bulbS = offw/bulbS;
    bulbsh = offh/bulbS;

    bulb = createBulbs(bulbS, bulbsh*bulbS);
    try {
        img = getImage(getDocumentBase(), getParameter("img"));
        MediaTracker t = new MediaTracker(this);
        t.addImage(img, 0);
        t.waitForID(0);
        pixscan = img.getWidth(null);
        int h = img.getHeight(null);
        pixels = new int[pixscan * h];
        PixelGrabber pg = new PixelGrabber(img, 0, 0, pixscan, h,
                                           pixels, 0, pixscan);
        pg.grabPixels();
    } catch (InterruptedException e) { };
    scrollX = 0;
    // рисование черных лампочек на внеэкранном изображении
    offgraphics.setColor(Color.black);
    offgraphics.fillRect(0, 0, d.width, d.height);
    for (int x=0; x<bulbS; x++)
        offgraphics.drawImage(bulb, x*bulbS, 0, null);
}

Image createBulbs(int w, int h) {
    int pixels[] = new int[w*h];
    int bulbBits[] = {
        0,0,1,1,1,1,0,0,
        0,1,2,1,1,1,1,0,
        1,2,1,1,1,1,1,1,
        1,1,1,1,1,1,1,1,
        1,1,1,1,1,1,1,1,
        1,1,1,1,1,1,1,1,
        0,1,1,1,1,1,1,0,
        0,0,1,1,1,1,0,0
    };
    int bulbCLUT[] = { 0xff000000, 0x00c0c0c0, 0xffffffff };
    for (int i=0; i<w*h; i++)
        pixels[i] = bulbCLUT[bulbBits[i%bulbBits.length]];
    return createImage(new MemoryImageSource(w, h, pixels, 0, w));
}
```

```

public final Color color(int x, int y) {
    int p = pixels[y*pixscan+x];
    Color c;
    if ((c=(Color)clut.get(p)) == null)
        clut.put(p, c = new Color(p));
    return c;
}

public void update() {}

public void paint(Graphics g) {
    offgraphics.copyArea(bulbsS, 0, bulbsW*bulbsS-bulbsS, d.height,
                         -bulbsS, 0);
    for (int y=0; y<bulbsH; y++) {
        offgraphics.setColor(color(scrollX, y));
        offgraphics.fillRect(d.width-bulbsS, y*bulbsS, bulbsS, bulbsS);
    }
    offgraphics.drawImage(bulb, d.width-bulbsS, 0, null);
    g.drawImage(offscreen, 0, 0, null);
    scrollX = (scrollX + 1) % pixscan;
}

Thread t;
public void run() {
    while (true) {
        paint(getGraphics());
        try{t.yield();} catch(Exception e) { };
        if(stopFlag)
            break;
    }
}

public void start() {
    t = new Thread(this);
    t.setPriority(Thread.MIN_PRIORITY);
    stopFlag = false;
    t.start();
}

public void stop() {
    stopFlag = true;
}
}

```

## Класс *IntHash*

Как сказано в предыдущем разделе, Color-объекты хранятся в хэш-таблице, а не создаются заново много раз. В качестве дальнейшей оптимизации мы

написали нашу собственную версию Java-класса `Hashtable`, которая использует нормальный `int`-массив ключей, а не `Object`-дескрипторы.

Целым данным нужно намного меньше памяти для хранения в массиве пикселов, чем `Color`-объектам, так что мы используем хэш-таблицу как механизм для поиска `Color`-объектов по целочисленному значению любого индивидуального пикселя. Создание объектов типа `Color` "на лету" из целого значения каждого пикселя является очень дорогим, потому что это создает много мусора в памяти, который нужно собирать. Одним из возможных решений было бы использование Java-класса `Hashtable`, за исключением того, что это создаст столько же мусора, т. к. в стандартной хэш-таблице Java в качестве ключей могут использоваться только объекты. Таким образом, чтобы хранить данные типа `int` в стандартной хэш-таблице Java, вам следовало бы создать в качестве ключей новые `Integer`-объекты. В высокопроизводительном циклическом апплете, подобном апплете `Lavatron`, мусорные `Integer`-объекты создавались бы тысячами в секунду. Это — неудачное решение.

Надлежащее решение состояло в том, чтобы строить собственную хэш-таблицу `IntHash`, которая в качестве ключей использует значения данных целого типа, а не `Integer`-объекты. Класс `IntHash` содержит около 60 программных строк. Он дублирует интерфейс класса `java.util.Hashtable`, за исключением того, что аргумент, передаваемый в `put()` и `get()`, имеет `int`-тип, а не `Object`. В этой главе не нужно объяснять, как работает хэш-таблица, но достаточно сказать, что `put(42, "Hello") == get(42)`.

## Код

Исходный код класса `IntHash`:

```
class IntHash {  
    private int capacity;  
    private int size;  
    private float load = 0.7F;  
    private int keys[];  
    private Object vals[];  
  
    public IntHash(int n) {  
        capacity = n;  
        size = 0;  
        keys = new int[n];  
        vals = new Object[n];  
    }  
  
    public IntHash() {  
        this(101);  
    }  
}
```

```
private void rehash() {
    int newcapacity = capacity * 2 + 1;
    Object newvals[] = new Object[newcapacity];
    int newkeys[] = new int[newcapacity];
    for (int i = 0; i < capacity; i++) {
        Object o = vals[i];
        if (o != null) {
            int k = keys[i];
            int newi = (k & 0x7fffffff) % newcapacity;
            while (newvals[newi] != null)
                newi = (newi + 1) % newcapacity;
            newkeys[newi] = k;
            newvals[newi] = o;
        }
    }
    capacity = newcapacity;
    keys = newkeys;
    vals = newvals;
}

public void put(int k, Object o) {
    int i = (k & 0x7fffffff) % capacity;
    while (vals[i] != null && k != keys[i])      // хэш-коллизия
        i = (i + 1) % capacity;
    if (vals[i] == null)
        size++;
    keys[i] = k;
    vals[i] = o;
    if (size > (int)(capacity * load))
        rehash();
}

public final Object get(int k) {
    int i = (k & 0x7fffffff) % capacity;
    while (vals[i] != null && k != keys[i])      // хэш потерян
        i = (i + 1) % capacity;
    return vals[i];
}

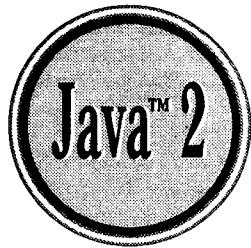
public final boolean contains(int k) {
    return get(k)!=null;
}

public int size() {
    return size;
}

public int capacity() {
    return capacity;
}
```

## Апплет *Hot Lava*

Этот апплет — другой небольшой пример удивительной эффективности, которую можно выжать из Java, если быть осторожным и прилежным. Дэвид Лавалли использует много уловок, чтобы избежать чрезмерного выделения памяти и ненужных обращений к рисующим функциям AWT. Создание изображения маски лампочки, состоящей из небольшого массива целых чисел, а не из загруженных GIF-изображений, сохраняет время загрузки и увеличивает гибкость. Использование `paint(getGraphics())`, а не `repaint()` значительно увеличивает фреймовую скорость. Эффективность происходит от использования `copyArea()`, иначе повторное выполнение изображения или вызовы `drawImage()` становятся непомерными. Наконец, создание и использование `IntHash` делается для того, чтобы повышение эффективности не вынуждало систему собирать мусор слишком часто.



## ГЛАВА 32

# *Scrabblet:* многопользовательская игра в слова

Scrabblet — это законченная сетевая многопользовательская игра, построенная по технологии клиент-сервер. Это наиболее сложный апплет в книге. Он имеет дело с некоторыми из самых трудных проблем в Java-программировании. Scrabblet содержит более 1400 строк кода в одиннадцати классах. Два из них — часть серверной стороны апплета. Другие девять загружены в Web-браузер и действуют как модель игры. Все элементы кода, используемые в игре, были описаны подробно в этой книге. В текущей главе, мы проанализируем каждый класс и покажем, насколько просто строить такую мультиигру.

## Вопросы сетевой безопасности

На сегодня большинство сетевых апплетов мало что делают с сетью после своей загрузки. Дело в том, что без специальной защиты работа Java-программ в сети становится более трудной. Многие среды, такие как Netscape Navigator и Microsoft Internet Explorer, строго ограничивают использование сети апплетами. Данная ситуация создана отсутствием аутентификации в основных протоколах TCP/IP. Это неотъемлемое ограничение Internet тщательно управляется корпорациями, которые хотят защитить свои частные данные при помощи использования межсетевой защиты. *Межсетевая защита* (firewall) — это компьютер, который находится между частной сетью и остальной частью Internet. Все подключения к Internet проходят через него, и он способен фильтровать и отклонять подключения и пакеты как входящие, так и выходящие. Таким образом, если программа на внешней стороне межсетевой защиты пытается присоединиться к внутреннему порту сети, межсетевая защита может блокировать ее. Если бы не межсетевая защита, системные администраторы должны были бы ревизовать защиту каждой машины на их внутренней сети. В случае сети, защищенной межсетевой

защитой, только межсетевая защита должна быть безопасной, а каждая машина внутри рассматривается как "дружественная" и оставляется незащищенной от любой другой внутренней машины.

Покажем, как Java-апплет мог бы создать угрозу безопасности. Если бы поддерживающие Java-браузеры разрешили апплетам соединяться с произвольным Internet-адресом, то апплет мог бы действовать как сетевой посредник (proxy) с некоторой злонамеренной программой на внешней стороне межсетевой защиты. Как только апплет был бы загружен и автоматически запущен Web-браузером, он мог бы соединяться с соседними компьютерами и серверами. Эти компьютеры не ожидали бы ничего враждебного, что могло бы исходить от внутреннего компьютера, и принимали бы подключения. Апплет был бы тогда способен захватить засекреченные данные и передать их назад через межсетевую защиту к злонамеренному хост-компьютеру.

Из-за этого сценария апплетам позволено делать сетевые подключения только к одному хост-компьютеру — к тому, из которого они были загружены. Это не позволяет апплету подглядывать за окружением внутренней сети. Одним из многих хорошо рекламированных "Java нападений на защиту" (разработанных исследователями в Принстонском Университете) было разрешение апплету открывать сетевые сокеты на машинах, доступ к которым иным способом для него невозможен. К счастью, эта проблема была достаточно трудна для воспроизведения и впоследствии была кому-то переадресована.

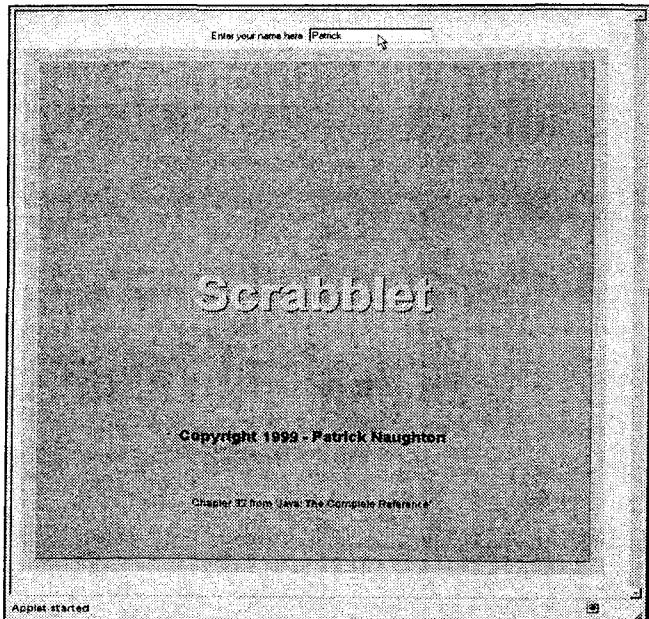
Что защита должна делать с мультиигрой? Многое. Самой простой для программирования мультиигры была бы возможность игроков напрямую связываться друг с другом в "одноранговой" сети. Таким образом, выполнение игры не зависело бы от выполнения любого программного обеспечения на сервере. К сожалению, апплет способен соединиться только обратно с сервером, из которого он был загружен. Это означает, что два игрока должны передавать все свои сообщения друг другу через сервер.

В этой главе вы увидите исходный код для простого сервера, который управляет списком подсоединенных клиентов и пересыпает сообщения между этими клиентами. По большому счету данный сервер ничего не знает о выполняемой игре. Он только пересыпает сообщения из пункта A в пункт B. Эта функция управляет двумя классами: `Server` и `ClientConnection`. Они будут описаны в конце этой главы.

## Игра

Прежде чем пользователи могут запустить мультиигру, они должны выбрать соперника. Вместо форсирования телефонного вызова для организации игры, этот апплет использует иной подход. При первом запуске он просит пользователя ввести свое имя (рис. 32.1). Имя передается серверу, который

рассыпает его всем другим потенциальным соперникам. Затем пользователь видит список всех игроков (рис. 32.2), выбирает одного и щелкает по кнопке вызова (**Challenge**). В настоящее время, не существует никакого способа подтвердить или отклонить вызов — они принимаются автоматически. Как только вызов сделан, оба игрока обнаружат появление игрового поля (доски), а все другие соперники просто увидят, как имена обоих игроков исчезают из списка играющих.



**Рис. 32.1.** Пользователь должен напечатать свое имя, чтобы начать игру

Это совсем простая игра, но очень трудно победить искусственного противника. Игрокам предоставляется сетка  $15 \times 15$ , и задается набор из семи квадратных фишек с выгравированными на них буквами алфавита (рис. 32.3). Фишку выбираются наугад из пакета с сотней вариантов. По каждой из них можно щелкнуть мышью и перетащить на любой квадрат сетки. Если ячейка уже занята, фишку возвращается на свою исходную позицию. Фишки могут быть установлены на поле во время хода<sup>1</sup>, но не когда ход закончен.

Первый игрок начинает с размещения нескольких фишек в линию на игровом поле, чтобы сформировать слово на английском языке. Первое слово должно размещаться в центре квадрата. Последующие слова обязаны прикасаться, по крайней мере к одной фишке, уже играющей на поле. Чтобы закончить ход, игрок нажимает кнопку **Done**. Если правильное слово не

<sup>1</sup> Под *ходом* здесь понимается вся последовательность перемещений фишек из лотка. — Примеч. перев.

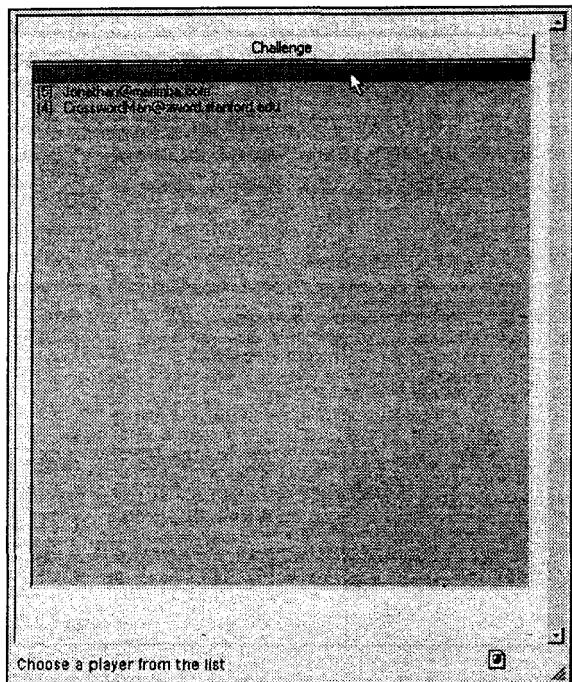


Рис. 32.2. Список участников

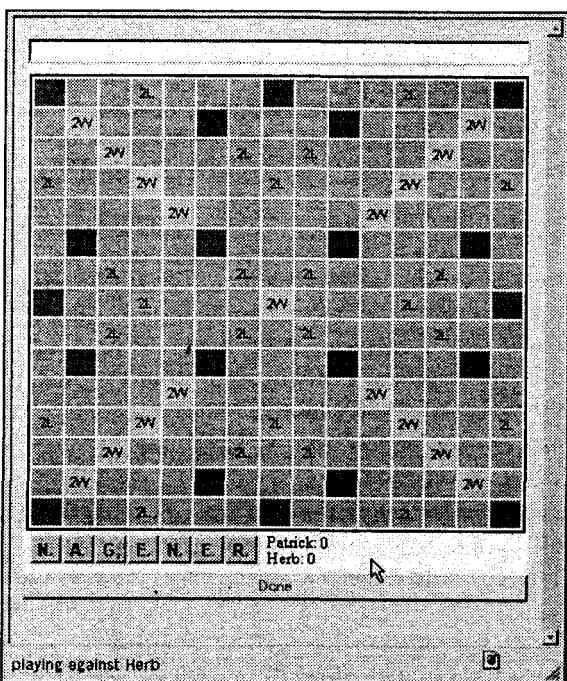


Рис. 32.3. Патрик и Герберт готовы играть против друга друга

найдено, игрок может пропустить ход, нажимая **Done** дважды, не имея какой-либо фишки на поле. Два игрока делают ходы, размещающие слова, до тех пор, пока все фишки не будут использованы.

Поле, показанное на рис. 32.3, предназначено для меньших дисплеев (по сравнению с размерами панели апплета), и поэтому квадраты множителя отмечены простыми символами: 2L удваивает значение (очки) буквы на этом квадрате, 3L — утраивает. 2W означает, что целое слово удваивает очки; 3W означает утраивание очков всего закрывающего этот квадрат слова. Если вы создаете апплет достаточно большим, он будет использовать более описательные метки для этих квадратиков, как показано на рис. 32.4.

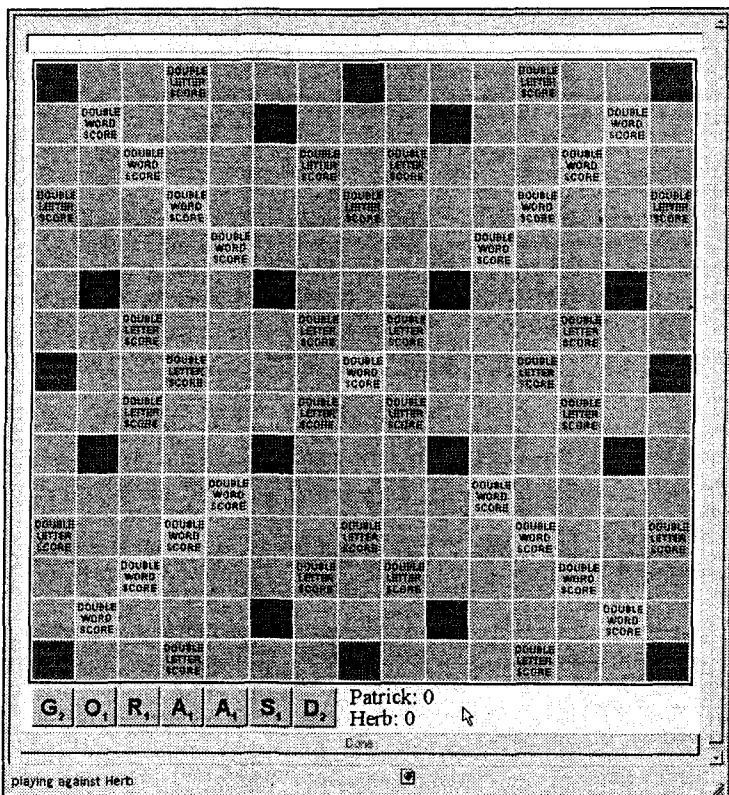


Рис. 32.4. Все становится яснее с большим размером апплета (650×700)

## Подсчет очков

Очки подсчитываются в конце каждого хода. Любая фишка имеет маленький номер, выгравированный на ее поверхности вслед за буквой. Этот счет может быть умножен на два или три, в зависимости от значения (цвета)

квадрата, на котором она (фишка) была помещена. Полная сумма для слова может также быть умножена на два или три, если буква в слове закрывает соответствующий квадрат. Если слово входит в контакт с другими фишками, чтобы сформировать дополнительные слова, они подсчитываются отдельно. При использовании игроком всех семи фишек в одном ходе присуждается дополнительно 50 пунктов (очков). В конце игры побеждает игрок с самым большим количеством очков.

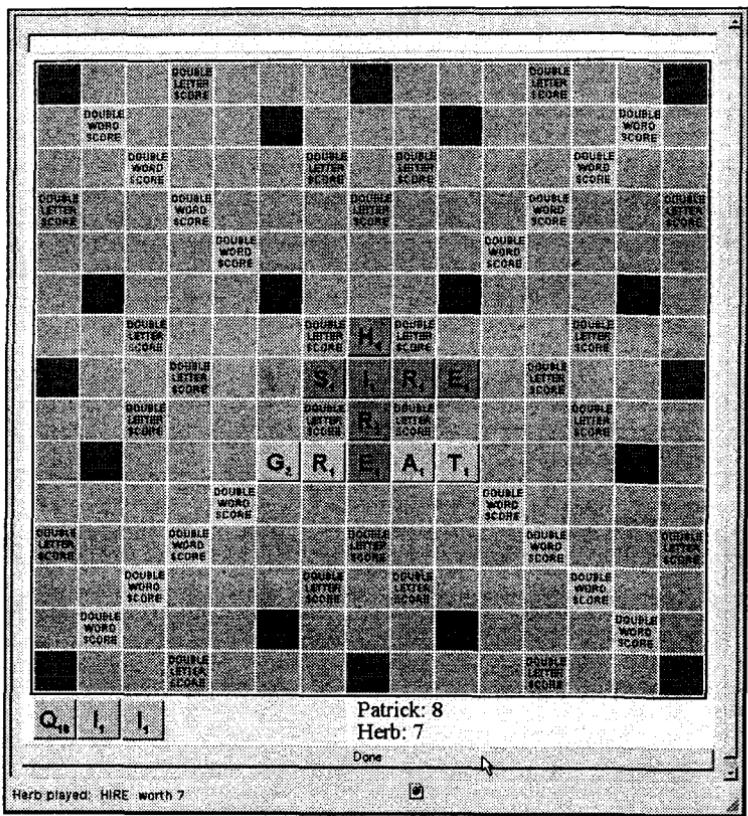


Рис. 32.5. Scrabble на ранней стадии игры

Рис. 32.5 показывает пример поля после нескольких сделанных шагов. Патрик начал со слова SIRE, стоимостью восемь очков. Эта оценка исходит из четырех однопунктовых фишек и удвоивания очков из-за размещения слова на центральной фишке. Далее, Герберт сыграл HIRE, используя букву i из слова SIRE. Это стоило семь очков — суммы на четырех включенных фишках. Обратите внимание, что Герберт получил кредит на повторное использование буквы i Патрика, но не очки двойного слова под ней. В точке, показанной на рис. 32.5, Патрик сыграл GREAT и собирается нажать кнопку

**Done** для завершения своего хода. Обратите внимание, что фишки находящиеся в игре — более яркие, чем те, которые уже играли (рис. 32.6).

В любое время в течение раунда игроки могут поддерживать контакт, общаясь через область текстового ввода наверху апплета (рис. 32.7). Эти сообщения будут по-одному появляться в строке состояния браузера соперника, обычно внизу браузера (рис. 32.8).

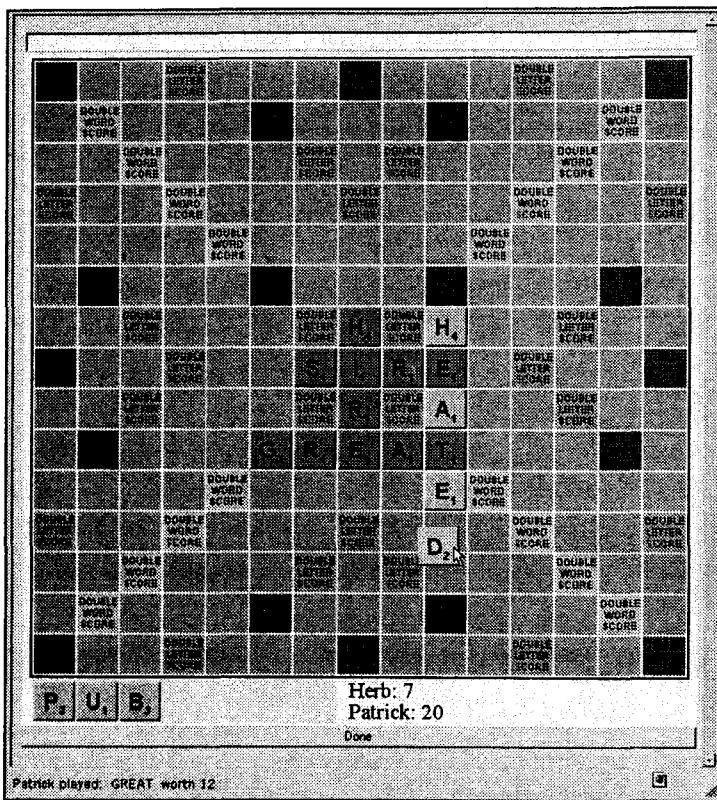
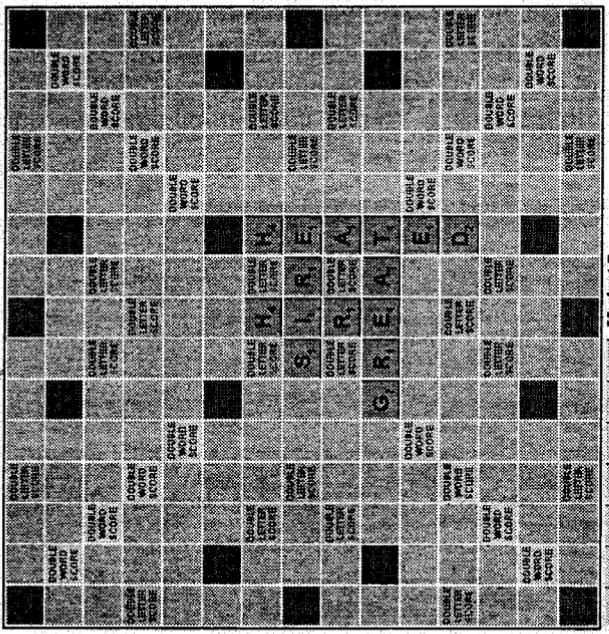


Рис. 32.6. Герберт собирается размещать букву D, чтобы собрать слово HEATED

И, наконец, последнее замечание относительно выполнения игры, прежде чем мы перейдем к исходному коду. Победа приходит со словами, которые набирают очки в одном направлении и, одновременно, участвуют в словах, построенных в другом направлении. Эти вторичные слова имеют тенденцию быть короткими, двухбуквенными словами, но они добавляются к первичным. На рис. 32.9 Патрик размещает Y в DEITY, что приведет к счету в 21 очко, потому что он получает на лицевой стороне фишек значение 9, удвоенное до 18, плюс 3 очка за слово AD, которое набирает по вертикали. Помните, что все слова, с которыми играет каждый ход, должны быть

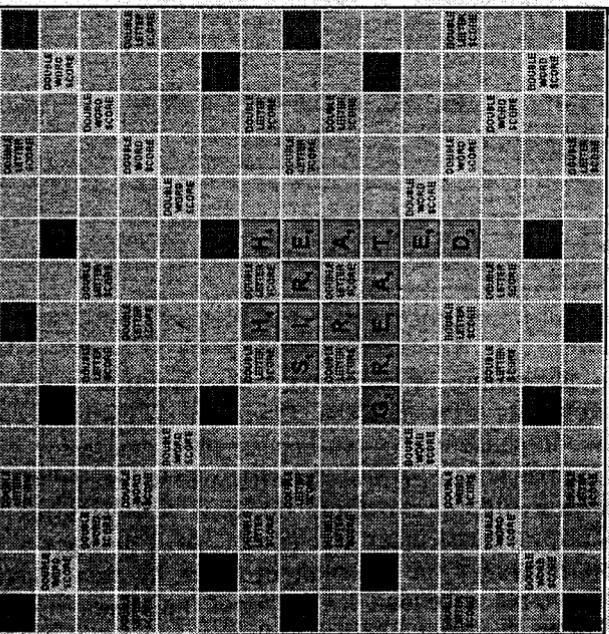


P, U, B, O, A, Y, E  
Herb: 17  
Patrick: 20

Done

Patrick: I have one... you stuck with that Q!

**Рис. 32.7.** Патрик жалуется, что ему всучили букву Q, а ему нужна U



P, U, B, O, A, Y, E  
Herb: 17  
Patrick: 20

Done

Patrick: I have one... you stuck with that Q!

**Рис. 32.8.** Герберт отвечает. Обратите внимание на последнее сообщение Патрика внизу

настоящими (осмысленными) словами. Со временем, эта игра будет нуждаться либо в операции отмены (undo) для спорных слов, либо в автоматической проверке словаря, чтобы разрешать конфликты.

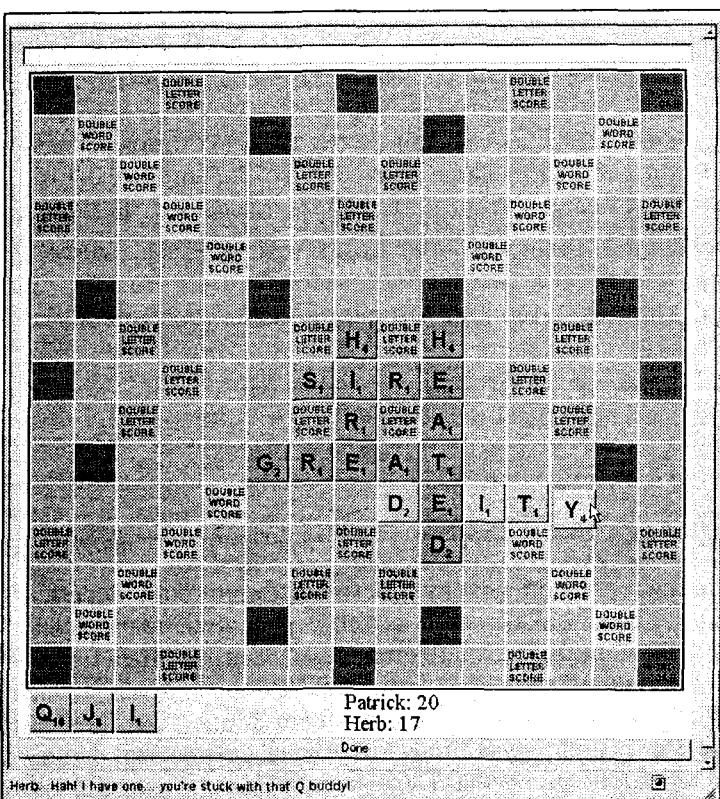


Рис. 32.9. Патрик набирает очки в двух направлениях!

## Исходный код

Теперь, когда вы знаете, как играть в эту игру, пришло время исследовать ее исходный код. Так как некоторые из классов очень длинные, мы будем просто вставлять комментарии по всему коду, прежде чем дойдем до его конца.

## Тег <applet>

Тег <applet> для игры прост. Он только именует главный класс и устанавливает размер (панели) апплета. В Scrabble нет никаких тегов <param>. Напомним, что чем больше размер панели апплета, тем лучше выглядит поле

игры. По соотношению размеров высота должна быть немного больше ширины.

```
<applet code=Scrabblet.class width=400 height=450>
</applet>
```

## **Scrabblet.java**

Главный класс апплета — Scrabblet.java. При почти трехстах строках текста это довольно сложный апплет-класс, даже притом, что большинство игровой логики оставлено классу Board, который вы найдете далее в этой главе.

Класс начинается с обычной совокупности import-утверждений, загружая почти все стандартные пакеты Java. Затем объявляется, что Scrabblet — это подкласс класса Applet, который реализует интерфейс ActionListener.

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Scrabblet extends Applet implements ActionListener {
```

Затем следуют объявления большой совокупности экземплярных переменных. Переменная server — это подключение к Web-серверу, выполняющему наш игровой сервер. Имя машины сохраняется в serverName. Переменная bag представляет общедоступный (для всех игроков) набор букв для нашей игры. Соперник имеет свою собственную копию пакета фишек bag, которая инициализируется той же самой случайной последовательностью фишек, так что эти два набора синхронизированы. Переменная board — наша копия игрового поля. Противник также имеет копию board, и игра синхронно сохраняет их после каждого хода.

Если к сетевому серверу нет доступа, устанавливается флагок single, и апплет запускает игру в режиме одиночного игрока. Переменная boolean ourturn имеет значение true всякий раз, когда ход в игре — наш. Если игрок не может найти правильное слово, он может пропустить ход, нажимая кнопку Done дважды подряд, не имея при этом каких-либо фишек на поле. Переменная seen\_pass используется для того, чтобы отметить, была ли кнопка Done нажата первой (в начале хода).

Для управления синхронизацией игрового поля удаленного игрока мы храним копию выбранных фишек в массиве theirs. Наблюдение за тем, что другой человек имеет в своем лотке, является жульничеством, так что не будьте хакером, не просматривайте содержимого theirs! Две строки, name и others\_name, содержат наше имя и имя оппонента, соответственно.

```
private ServerConnection server;
private String serverName;
private Bag bag;
private Board board;
private boolean single = false;
private boolean ourturn;
private boolean seen_pass = false;
private Letter theirs[] = new Letter[7];
private String name;
private String others_name;
```

Далее, мы объявляем восемь переменных, используемых для управления интерфейсом пользователя. Все они — компоненты AWT, которыми апплет должен некоторым образом манипулировать. Переменная `topPanel` — это объект класса `Panel`, а `prompt` и `namefield` — тоже объекты, которые используются для получения имен пользователей при запуске игры. Кнопка `done` (объект класса `Button`) служит для ввода команды завершения хода. Переменная `TextField chat` используется для ввода сообщения интерактивного разговора. Переменная `idList` применяется для отображения доступных оппонентов. Кнопка `challenge` — для прикрепления нас к оппоненту. Переменная `Canvas ican` содержит заметки с именем и авторским правом для начальной заставки апплета.

```
private Panel topPanel;
private Label prompt;
private TextField namefield;
private Button done;
private TextField chat;

private List idList;
private Button challenge;
private Canvas ican;
```

## Метод `init()`

Метод `init()` вызывается однажды и просто устанавливает `BorderLayout`, выясняет, из какой хост-машины Internet пришел апплет, и создает полотно (пустую панель) для экрана-заставки.

```
public void init() {
    setLayout(new BorderLayout());
    serverName = getCodeBase().getHost();
    if (serverName.equals(""))
        serverName = "localhost";
    ican = new IntroCanvas();
}
```

## Метод start()

Метод `start()` вызывается всякий раз, когда браузер восстанавливает изображение страницы, в которой находится апплет. В начале используется большой `try`-блок, чтобы выловить ситуацию, когда сетевое соединение терпит неудачу. Если мы преуспеваем в создании нового соединения (объекта `ServerConnection`) и никогда прежде не выполняли `start()`, тогда устанавливаем экран с подсказкой имени пользователя. Пока мы в нем, мы помещаем экран заставки (`ican`) в центр окна. В случае, когда `name` — не `null`, что означает, что пользователь покинул страницу и теперь возвратился, мы предполагаем, что уже получили имя пользователя и выполняем переход прямо к методу `nameEntered()`. Данный метод вызывается, когда пользователь вводит "return" в поле ввода имени. Вызов `validate()` в конце `try`-блока удостоверяет, что все компоненты AWT модифицированы должным образом.

При выбросе исключения мы предполагаем, что сетевое подключение потерпело неудачу, и входим в режим одиночного игрока. Вызов `start_game()` запускает игру.

```
public void start() {
    try {
        showStatus("Connecting to " + serverName);
        server = new ServerConnection(this, serverName);
        server.start();
        showStatus("Connected: " + serverName);
        if (name == null) {
            prompt = new Label("Enter your name here:");
            namefield = new TextField(20);
            namefield.addActionListener(this);
            topPanel = new Panel();
            topPanel.setBackground(new Color(255, 255, 200));
            topPanel.add(prompt);
            topPanel.add(namefield);
            add("North", topPanel);
            add("Center", ican);
        } else {
            if (chat != null) {
                remove(chat);
                remove(board);
                remove(done);
            }
            nameEntered(name);
        }
        validate();
    } catch (Exception e) {
```

```

    single = true;
    start_Game((int)(0x7fffffff * Math.random()));
}
}

```

## Метод **stop()**

Метод `stop()` вызывается всякий раз, когда пользователь покидает страницу с апплетом. Здесь мы просто сообщаем серверу, что покинули страницу. Мы вновь создаем сетевое подключение в методе `start()`, если пользователь позже возвращается к странице.

```

public void stop() {
    if (!single)
        server.quit();
}

```

## Метод **add()**

Метод `add()` вызывается объектом типа `ServerConnection` всякий раз, когда новый игрок включается в игру. Мы добавляем имя игрока к нашему `List`-объекту. Обратите особое внимание на форматирование строки в `add()`. Мы используем ее позже, чтобы извлечь из списка некоторые идентификаторы.

```

void add(String id, String hostname, String name) {
    delete(id); // в случае, когда имя уже есть в списке.
    idList.add("(" + id + " " + name + "@" + hostname);
    showStatus("Choose a player from the list");
}

```

## Метод **delete()**

Метод `delete()` вызывается тогда, когда игрок не хочет больше быть идентифицирован как участник игры. Это случается, когда игрок выходит из игры или решает играть с кем-то еще. Здесь мы вылавливаем идентификатор строки (`id`) из нашего списка, извлекая значения внутри скобок. Если в списке нет больше имен (и мы уже не играем, т. е. `bag == null`), то мыываем пользователю специальное сообщение с просьбой подождать прибытия каких-нибудь других игроков.

```

void delete(String id) {
    for (int i = 0; i < idList.getItemCount(); i++) {
        String s = idList.getItem(i);
        s = s.substring(s.indexOf("(") + 1, s.indexOf("")));
        if (s.equals(id)) {
            idList.remove(i);
        }
    }
}

```

```

        break;
    }
}
if (idList.getItemCount() == 0 && bag == null)
    showStatus("Wait for other players to arrive.");
}

```

## Метод *getName()*

Метод *getName()* очень похож на *delete()* за исключением того, что он просто извлекает часть name-элемента списка имен игроков и возвращает его в вызывающую программу. Если элемент списка не найден, то возвращается *null*.

```

private String getName(String id) {
    for (int i = 0; i < idList.getItemCount(); i++) {
        String s = idList.getItem(i);
        String id1 = s.substring(s.indexOf("(") + 1, s.indexOf("")));
        if (id1.equals(id)) {
            return s.substring(s.indexOf(" ") + 3, s.indexOf("@"));
        }
    }
    return null;
}

```

## Метод *challenge()*

Метод *challenge()* вызывается объектом типа *ServerConnection* всякий раз, когда другой игрок направляет нам вызов к игре. Мы могли бы сделать метод более сложным, например, он мог бы посыпать пользователю запрос на подтверждение или отказ от вызова, но вместо этого вызов принимается автоматически. Обратите внимание, что случайное начальное число, которое мы используем для запуска игры, посыпается обратно сопернику в метод *accept()*. Оно используется обеими сторонами для инициализации случайного состояния набора фишек, чтобы обеспечить синхронную игру. Мы вызываем *server.delete()*, чтобы удостовериться, что нас больше не спрашивают игроки, желающие сражаться против нас. Обратите также внимание, что мы уступаем начальный ход партнеру, устанавливая в переменной *ourturn* значение *false*.

```

// Нас вызвал на игру "id".
void challenge(String id) {
    ourturn = false;
    int seed = (int)(0x7fffffff * Math.random());
    others_name = getName(id);           // Кто это был?
    showStatus("challenged by " + others_name);

    // поместить сюда запрос подтверждения...
}

```

```

server.accept(id, seed);
server.delete();
start_Game(seed);
}
}

```

## Метод *accept()*

Метод *accept()* вызывается на удаленной стороне в ответ на только что упомянутый вызов *server.accept()*. Точно так же, как другой игрок удалил себя из списка доступных соперников, мы должны вызывать метод *server.delete()*. Кроме того, мы берем первый ход, устанавливая в *ourturn* значение *true*.

```

// Наш вызов был принят.
void accept(String id, int seed) {
    ourturn = true;
    others_name = getName(id);
    server.delete();
    start_Game(seed);
}

```

## Метод *chat()*

Метод *chat()* вызывается сервером всякий раз, когда противник печатает текст в своем разговорном (*chat*<sup>1</sup>-) окне. В этой реализации, метод просто показывает *chat*-сообщение в строке состояния браузера. В будущем было бы неплохо зарегистрировать их в специальной текстовой области.

```

void chat(String id, String s) {
    showStatus(others_name + ": " + s);
}

```

## Метод *move()*

Метод *move()* вызывается однажды для каждого перемещения фишкой вашим противником. Он следит за буквами, сохраненными в *theirs*, чтобы найти используемую. Если квадрат уже занят, фишка возвращается в лоток игрока. Иначе буква противника перемещается на игровое поле навсегда. Затем фишка в *theirs* замещается на очередную из пакета фишек (методом *bag.takeOut()*). Если пакет пуст, появляется специальное сообщение. Игровая доска перерисовывается для показа на ней новых фишек. Обратите внимание, что никакой подсчет очков не делается при размещении этих фишек. Апплет ожидает момента вызова метода *turn()*, чтобы выдать счет.

---

<sup>1</sup> От англ. *chat* — болтовня. — Примеч. пер.

```
// Противник перемещает фишку и размещает ее в (x, y).
void move(String letter, int x, int y) {
    for (int i = 0; i < 7; i++) {
        if (theirs[i] != null && theirs[i].getSymbol().equals(letter)) {
            Letter already = board.getLetter(x, y);
            if (already != null) {
                board.moveLetter(already, 15, 15);           // на лоток
            }
            board.moveLetter(theirs[i], x, y);
            board.commitLetter(theirs[i]);
            theirs[i] = bag.takeOut();
            if (theirs[i] == null)
                showStatus("No more letters");
            break;
        }
    }
    board.repaint();
}
```

## Метод *turn()*

Метод *turn()* вызывается после того, как все фишки противника перемещены. Удаленный экземпляр *Scrabblet* определяет счет и посыпает его нам, так что наша копия не должна вычислять его повторно. Затем счет сообщается в строке состояния, и метод *setEnabled* позволяет нам выполнить ход. Метод *othersTurn()* сообщает счет игровому полю *board*.

```
void turn(int score, String words) {
    showStatus(others_name + " played: " + words + " worth " + score);
    done.setEnabled(true);
    board.othersTurn(score);
}
```

## Метод *quit()*

Когда противоположная сторона аккуратно завершает работу, вызывается метод *quit()*. Он удаляет AWT-компоненты игры и переходит обратно в *nameEntered()*, описанный ниже, чтобы вернуться обратно к списку игроков.

```
void quit(String id) {
    showStatus(others_name + " just quit.");
    remove(chat);
    remove(board);
    remove(done);
    nameEntered(name);
}
```

## Метод *nameEntered()*

Метод *nameEntered()* вызывается из *actionPerformed()* всякий раз, когда в исходном вводе имени пользователя нажимается клавиша <Enter>. Любые AWT-компоненты, которые могли бы присутствовать в этом вводе, удаляются, и затем создается новый *List*-объект *idList* для хранения имен других игроков. Метод также добавляет в верхнюю часть панели кнопку с именем *challenge* и затем уведомляет сервер, что мы на месте, вызывая *setName()*.

```
private void nameEntered(String s) {
    if (s.equals(""))
        return;
    name = s;
    if (ican != null)
        remove(ican);
    if (idList != null)
        remove(idList);
    if (challenge != null)
        remove(challenge);
    idList = new List(10, false);
    add("Center", idList);
    challenge = new Button("Challenge");
    challenge.addActionListener(this);
    add("North", challenge);
    validate();
    server.setName(name);
    showStatus("Wait for other players to arrive.");
    if (topPanel != null)
        remove(topPanel);
}
```

## Методы *wepick()* и *theypick()*

Методы *wepick()* и *theypick()* предназначены для начала игры, чтобы выбрать семь фишек для каждого игрока. Важно, что вызывающая программа делает это в нужном порядке на каждой стороне вызова в зависимости от того, кто ходит первым. Обращение к *bag.takeOut()* постоянно получает одиночный символ из общего игрового набора. Обращение к *board.addLetter()* помещает фишку в наш лоток. Для противоположной стороны метод *theypick()* просто сохраняет буквы в массив *theirs*.

```
private void wepick() {
    for (int i = 0; i < 7; i++) {
        Letter l = bag.takeOut();
        board.addLetter(l);
    }
}
```

```

private void theypick() {
    for (int i = 0; i < 7; i++) {
        Letter l = bag.takeOut();
        theirs[i] = l;
    }
}

```

## Метод *start\_Game()*

В режиме одного игрока, *start\_Game()* выводит заставочный экран во Frame-окно. Затем он создает игровое поле *board*, не пересыпая в конструктор никаких параметров, что и указывает на монорежим.

В парном режиме мы удаляем компоненты списка выбора и добавляем к апплету *chat*-окно, а затем — игровое поле *board* и кнопку **Done**. Далее, мы создаем общий набор фишек *bag*, и если ход — наш (*ourturn* имеет значение *true*), то первым вызывается *wepick()*, затем *theypick()*. В случае, когда мы не имеем первого хода, мы отключаем *board* и кнопку **Done**, и первым вызываем метод *theypick()*. Затем выполняем перерисовку игрового поля *board*, что и инициализирует его.

```

private void start_Game(int seed) {
    if (single) {
        Frame popup = new Frame("Scrabblet");
        popup.setSize(400, 300);
        popup.add("Center", ican);
        popup.setResizable(false);
        popup.show();
        board = new Board();
        showStatus("no server found, playing solo");
        ourturn = true;
    } else {
        remove(idList);
        remove(challenge);
        board = new Board(name, others_name);
        chat = new TextField();
        chat.addActionListener(this);
        add("North", chat);
        showStatus("playing against " + others_name);
    }

    add("Center", board);
    done = new Button("Done");
    done.addActionListener(this);
    add("South", done);
    validate();

    bag = new Bag(seed);
    if (ourturn) {

```

```

wepick();
if (!single)
    theypick();
} else {
    done.setEnabled(false);
    theypick();
    wepick();
}
board.repaint();
}
}

```

## Метод *challenge\_them()*

Метод *challenge\_them()* вызывается, когда нажимается кнопка *challenge*. Он просто берет игрока, которого вы выбрали в *idList* и посыпает ему сообщение *challenge()*. Затем метод удаляет список и кнопку, чтобы быть готовым к началу игры.

```

private void challenge_them() {
    String s = idList.getSelectedItem();
    if (s == null) {
        showStatus("Choose a player from the list then press Challenge");
    } else {
        remove(challenge);
        remove(idList);
        String destid = s.substring(s.indexOf('(')+1, s.indexOf(')'));
        showStatus("challenging: " + destid);
        server.challenge(destid);
        validate();
    }
}

```

## Метод *our\_turn()*

Когда нажимается кнопка **Done**, вызывается метод *our\_turn()*. Сначала он проверяет, поместили ли мы фишку в правильное положение, вызывая *board.findwords()* и сохраняя результат в *word*. Если *word* имеет значение *null*, то что-то неладно с фишками, и метод показывает это в строке состояния. Если *word* равен "" (т. е. содержит пустую строку), то метод знает, что в это время не было никаких фишек. В режиме одиночного игрока метод завершает свою работу. В режиме соревнования, если мы щелкаем по **Done** подряд дважды, когда в игре нет никаких фишек, мы передаем наш ход противнику.

Если у вас есть играющие фишки, и они находятся в правильных позициях, то вы закончили свой ход так, что *ourturn()* передает буквы на игровое поле. Обратите внимание, что метод *commit()* получает *server* как параметр. Он будет использовать его, чтобы сообщать удаленной стороне о позиции

каждой новой буквы. Затем метод заменяет буквы, которые вы уже разместили на поле. В режиме мультиигрока мы деактивизируем себя и вызываем `server.turn()`, чтобы сообщить другому игроку, что его ход — следующий.

```

private void our_turn() {
    String word = board.findwords();
    if (word == null) {
        showStatus("Illegal letter positions");
    } else {
        if ("".equals(word)) {
            if (single)
                return;
            if (seen_pass) {
                done.setEnabled(false);
                server.turn("pass", 0);
                showStatus("You passed");
                seen_pass = false;
            } else {
                showStatus("Press done again to pass");
                seen_pass = true;
                return;
            }
        } else {
            seen_pass = false;
        }
        showStatus(word);
        board.commit(server);
        for (int i = 0; i < 7; i++) {
            if (board.getTray(i) == null) {
                Letter l = bag.takeOut();
                if (l == null)
                    showStatus("No more letters");
                else
                    board.addLetter(l);
            }
        }
        if (!single) {
            done.setEnabled(false);
            server.turn(word, board.getTurnScore());
        }
        board.repaint();
    }
}

```

## **Метод `actionPerformed()`**

Метод `actionPerformed()` служит для того, чтобы захватить ввод от различных компонентов, которые использует апллет. Он обрабатывает кнопки

**Challenge** и **Done**, а также поле ввода имени и поле ввода сетевых (chat-) сообщений.

```
public void actionPerformed(ActionEvent ae) {
    Object source = ae.getSource();
    if(source == chat) {
        server.chat(chat.getText());
        chat.setText("");
    }
    else if(source == challenge) {
        challenge_them();
    }
    else if(source == done) {
        our_turn();
    }
    else if(source == namefield) {
        TextComponent tc = (TextComponent)source;
        nameEntered(tc.getText());
    }
}
```

## **IntroCanvas.java**

IntroCanvas, подкласс Canvas, очень прост. Он только переопределяет paint(), чтобы вывести имя апплета и краткое замечание об авторском праве. Он создает некоторые заказные цвета и шрифты. Строки дисплея содержатся в статических переменных просто для ясности.

```
import java.awt.*;
import java.awt.event.*;

class IntroCanvas extends Canvas {
    private Color pink = new Color(255, 200, 200);
    private Color blue = new Color(150, 200, 255);
    private Color yellow = new Color(250, 220, 100);

    private int w, h;
    private int edge = 16;
    private static final String title = "Scrabblet";
    private static final String name ="Copyright 1999 - Patrick Naughton";
    private static final String book =
        "Chapter 32 from 'Java: The Complete Reference'";
    private Font namefont, titlefont, bookfont;

    IntroCanvas() {
        setBackground(yellow);
        titlefont = new Font("SansSerif", Font.BOLD, 58);
```

```

namefont = new Font("SansSerif", Font.BOLD, 18);
bookfont = new Font("SansSerif", Font.PLAIN, 12);
addMouseListener(new MyMouseAdapter());
}
}

```

## Метод d()

Частный метод `d()` — метод удобства (convenience method), который рисует размещенный по центру текст с необязательным изометрическим смещением. Он используется для создания эффекта подсветки/тени у основного заголовка, рисуя белую строку выше и левее на 1 пиксель, черную строку — ниже и правее на 1 пиксель, и затем выводя строку в последний раз в розовом цвете, без смещения.

```

private void d(Graphics g, String s, Color c, Font f, int y,
    int off) {
    g.setFont(f);
    FontMetrics fm = g.getFontMetrics();
    g.setColor(c);
    g.drawString(s, (w - fm.stringWidth(s)) / 2 + off, y + off);
}

public void paint(Graphics g) {
    Dimension d = getSize();
    w = d.width;
    h = d.height;
    g.setColor(blue);
    g.fillRect(edge, edge, w - 2 * edge, h - 2 * edge, true);
    d(g, title, Color.black, titlefont, h / 2, 1);
    d(g, title, Color.white, titlefont, h / 2, -1);
    d(g, title, pink, titlefont, h / 2, 0);
    d(g, name, Color.black, namefont, h * 3 / 4, 0);
    d(g, book, Color.black, bookfont, h * 7 / 8, 0);
}
}

```

## Метод mousePressed()

В следующем кодовом фрагменте обратите внимание, что `MyMouseAdapter` — внутренний класс, который расширяет `MouseAdapter`. Он переопределяет метод `mousePressed()`, чтобы скрыть родителя этого полотна (пустого фрейма), если на нем выполняется щелчок мышью. Он полезен только в режиме одиночного игрока, чтобы отделаться от раскрывающегося фрейма.

```

class MyMouseAdapter extends MouseAdapter {
    public void mousePressed(MouseEvent me) {
        ((Frame)getParent()).setVisible(false);
    }
}
}

```

## Board.java

Класс Board инкапсулирует большую часть игровой логики, а также вид и стиль игрового поля. Это — самый большой класс в игре, содержащий более пятисот строк кода. Присутствуют несколько частных переменных, которые хранят состояние игры. Чтобы хранить фишки на каждом квадрате игрового поля, используется массив букв с именем board (размером 15×15). Массив tray содержит буквы, которые в настоящее время находятся на нашем лотке. Напомним, что класс апплета Scrabblet содержит семь букв от нашего противника. Чтобы помнить позиции символа, используются Point-объекты orig и here. Для отображения имен для табло используются переменные name и others\_name. В режиме одного игрока обе переменные будут иметь значение null. Очки двух игроков сохраняются в переменных total\_score и others\_score, в то время как результат нашего последнего хода сохраняется в turn\_score. Два конструктора устанавливают имена игроков, или оставляют их пустыми в режиме одного игрока.

```
import java.awt.*;
import java.awt.event.*;

class Board extends Canvas {
    private Letter board[][] = new Letter[15][15];
    private Letter tray[] = new Letter[7];
    private Point orig = new Point(0,0);
    private Point here = new Point(0,0);
    private String name;
    private int total_score = 0;
    private int turn_score = 0;
    private int others_score = 0;
    private String others_name = null;

    Board(String our_name, String other_name) {
        name = our_name;
        others_name = other_name;
        addMouseListener(new MyMouseAdapter());
        addMouseMotionListener(new MyMouseMotionAdapter());
    }

    Board() {
        addMouseListener(new MyMouseAdapter());
        addMouseMotionListener(new MyMouseMotionAdapter());
    }
}
```

### Методы othersTurn(), getTurnScore() и getTray()

Эти три метода используются для того, чтобы управлять доступом к некоторым частным переменным. Метод othersTurn() вызывается апплетом,

когда другой игрок заканчивает ход. Он увеличивает счет игрока и перерисовывает область игрового поля, в которой произошли изменения. Метод `getTurnScore()` просто возвращает счет сохраненного последнего хода, после того как убедится, что табло очков нарисовано с правильным значением. Апплет использует данный метод, чтобы передать счет нашему противнику, где он, в конечном счете, вызовет `othersTurn()` на удаленной машине. Метод `getTray()` просто обеспечивает доступ `read-only` (только для чтения) к частному массиву `tray`.

```
void othersTurn(int score) {
    others_score += score;
    paintScore();
    repaint();
}

int getTurnScore() {
    paintScore();
    return turn_score;
}

Letter getTray(int i) {
    return tray[i];
}
```

### **Метод *addLetter()***

Метод `addLetter()` используется для размещения символа в лотке. Символ помещается в первый пустой слот. Если метод не может найти пустой слот, он возвращает `false`.

```
synchronized boolean addLetter(Letter l) {
    for (int i = 0; i < 7; i++) {
        if (tray[i] == null) {
            tray[i] = l;
            moveLetter(l, i, 15);
            return true;
        }
    }
    return false;
}
```

### **Метод *existingLetterAt()***

Частный метод `existingLetterAt()` применяется для проверки позиции игрового поля на предмет того, не находится ли вне игры присутствующая в ней в настоящее время буква. Это используется следующим (за данным) методом `findwords()`, чтобы удостовериться, что по крайней мере одна буква в ходе соприкасается с уже существующим символом.

```
private boolean existingLetterAt(int x, int y) {
    Letter l = null;
    return (x >= 0 && x <= 14 && y >= 0 && y <= 14
        && (l = board[y][x]) != null && l.recall() == null);
}
```

## Метод *findwords()*

Очень большой метод *findwords()* служит для проверки состояния игрового поля правильного хода. Если правила для размещения буквы нарушены, то возвращается `null`. Если в игре не было никаких фишек, то возвращается `" "` (пустая строка). Если все фишки, которые играли в этом ходе, правильные, то возвращается список сформированных ими слов (в виде строки, содержащей отделенные пробелом слова). Экземплярные переменные `turn_score` и `total_score` модифицируются так, чтобы отражать значение слов, которые только что участвовали в игре.

Сначала *findwords()* считает фишки, находящиеся в игре (в переменной `ntiles`), сохраняя их в отдельном массиве `atplay`. Далее, он смотрит на ориентацию первых двух фишек (если в игре их больше, чем одна), чтобы определить их ориентацию (вертикальную или горизонтальную). Затем он осматривает все другие фишки, находящиеся в игре, чтобы удостовериться, что они находятся на той же строке. Если любая из фишек находится вне своей строки или столбца, метод возвращает `null`.

```
synchronized String findwords() {
    String res = "";
    turn_score = 0;

    int ntiles = 0;
    Letter atplay[] = new Letter[7];
    for (int i = 0; i < 7; i++) {
        if (tray[i] != null && tray[i].recall() != null) {
            atplay[ntiles++] = tray[i];
        }
    }
    if (ntiles == 0)
        return res;

    boolean horizontal = true;           // если есть хоть одна фишка,
                                         // назовем ее горизонтальной
    boolean vertical = false;
    if (ntiles > 1) {
        int x = atplay[0].x;
        int y = atplay[0].y;
        horizontal = atplay[1].y == y;
        vertical = atplay[1].x == x;
```

```

if (!horizontal && !vertical) . . . // диагональ...
    return null;
for (int i = 2; i < ntiles; i++) {
    if (horizontal && atplay[i].y != y
        || vertical && atplay[i].x != x)
        return null;
}
}
}

```

Далее, он смотрит на каждую фишку, чтобы убедиться, что по крайней мере одна из них касается существующей фишкой одной из своих четырех сторон. Для начала игры предусмотрен специальный случай: если фишкой закрыт центральный квадратик и в игре больше одной фишки, это законно.

```

// Убедиться, что по крайней мере одна играющая фишка
// касается хотя бы одной существующей фишки.
boolean attached = false;
for (int i = 0; i < ntiles; i++) {
    Point p = atplay[i].recall();
    int x = p.x;
    int y = p.y;
    if ((x == 7 && y == 7 && ntiles > 1) ||
        existingLetterAt(x-1, y) || existingLetterAt(x+1, y) ||
        existingLetterAt(x, y-1) || existingLetterAt(x, y+1)) {
        attached = true;
        break;
    }
}
if (!attached) {
    return null;
}

```

Следующий цикл выполняет итерации по каждой букве в главном слове ( $i == -1$ ), затем возвращается снова к каждой букве ( $i == 0..ntiles$ ), которая могла бы тоже создавать слово, ортогональное к главному направлению (управляемому с помощью переменной `horizontal`).

```

// Мы используем -1, что означает проверку сначала главного
// направления, затем 0..ntiles проверяет слова, ортогональные ему.
for (int i = -1; i < ntiles; i++) {
    Point p = atplay[i== -1? 0:i].recall(); // Где она?
    int x = p.x;
    int y = p.y;

    int xinc, yinc;
    if (horizontal) {
        xinc = 1;
        yinc = 0;
    }
    else {
        xinc = 0;
        yinc = 1;
    }
    . . .
}
}

```

```

} else {
    xinc = 0;
    yinc = 1;
}
int mult = 1;

String word = "";
int word_score = 0;

```

Затем метод выбирает каждую фишку и движется влево или вверх от нее, чтобы найти первую фишку в каждом слове. Оказавшись в начале слова, он перемещается вправо или вниз от него, просматривая каждую букву. Метод считает буквы в letters\_seen. Для каждой буквы он определяет ее вклад, основанный на премиальном множителе, показанном ниже буквы (справа). Если квадратик разыгрывается впервые, то применяется значение множителя; иначе — фишка подсчитывается в номинальном значении. Этот счет накапливается в word\_score.

```

// Здесь мы возвращаемся к самой верхней/левой букве.
while (x >= xinc && y >= yinc &&
       board[y-yinc][x-xinc] != null) {
    x -= xinc;
    y -= yinc;
}

int n = 0;
int letters_seen = 0;           // буквы, которые мы только что разыграли
Letter l;
while (x < 15 && y < 15 && (l = board[y][x]) != null) {
    word += l.getSymbol();
    int lscore = l.getPoints();
    if (l.recall() != null) {      // буква, разыгрываемая сейчас...
        Color t = tiles[y < 8 ? y : 14 - y][x < 8 ? x : 14 - x];
        if (t == w3)
            mult *= 3;
        else if (t == w2)
            mult *= 2;
        else if (t == 13)
            lscore *= 3;
        else if (t == 12)
            lscore *= 2;
        if (i == -1) {
            letters_seen++;
        }
    }
    word_score += lscore;
    n++;
}

```

```

x += xinc;
y += yinc;
}
word_score *= mult;
}

```

Последняя (и единственная) проверка ошибки выполняется только на главном слове. Так как цикл заканчивается всякий раз, когда он натыкается на пустой квадрат или край игрового поля, он должен охватить все недавно разыгранные фишкы так же, как и некоторые предварительно сыгранные. Если он видит меньшее количество фишек, то, должно быть, в них имелся промежуток, что является незаконной ситуацией, так что метод возвращает null. Если это испытание проходит успешно, оно выясняет, все ли семь фишек играли, предоставляемый выигрыш 50 премиальных очков (если он был). После осмотра главного слова findwords() инвертирует смысл horizontal и ищет ортогональные слова на последующих проходах.

```

// Здесь мы возвращаемся обратно к самой верхней/левой букве.
while (x >= xinc && y >= yinc &&
       board[y-yinc][x-xinc] != null) {
    x -= xinc;
    y -= yinc;
}

int n = 0;
int letters_seen = 0; // буквы, только что разыгранные
Letter l;
while (x < 15 && y < 15 && (l = board[y][x]) != null) {
    word += l.getSymbol();
    int lscore = l.getPoints();
    if (l.recall() != null) { // буквы, которые сейчас в игре...
        Color t = tiles[y < 8 ? y : 14 - y][x < 8 ? x : 14 - x];
        if (t == w3)
            mult *= 3;
        else if (t == w2)
            mult *= 2;
        else if (t == 13)
            lscore *= 3;
        else if (t == 12)
            lscore *= 2;
        if (i == -1) {
            letters_seen++;
        }
    }
    word_score += lscore;
    n++;
    x += xinc;
    y += yinc;
}
word_score *= mult;
}

```

Когда `findwords()` проходит по слову, ему нужна уверенность в том, что он считает очки только для букв, формирующих слова, по меньшей мере, с двумя буквами. В этом случае он добавляет `word_score` к `turn_score` и вставляет это слово в конец строки результата. Как только все буквы были просмотрены, он подытоживает очки и выполняет возврат.

```

if (n < 2)                                // не считать одиночные буквы дважды
    continue;

turn_score += word_score;
res += word + " ";
}

total_score += turn_score;
return res;
}

```

## Методы `commit()` и `commitLetter()`

Методы `commit()` и `commitLetter()` передают буквы, которые были экспериментально (для пробы) помещены в игровое поле. Эти буквы удаляются из лотка и рисуются в более темном цвете на игровом поле. Когда все символы переданы, `commit()` уведомляет сервер о позиции каждой буквы, вызывая метод `move()`, так что игровое поле противника можно обновить.

```

synchronized void commit(ServerConnection s) {
    for (int i = 0; i < 7; i++) {
        Point p;
        if (tray[i] != null && (p = tray[i].recall()) != null) {
            if (s != null)                // соединение с сервером существует
                s.move(tray[i].getSymbol(), p.x, p.y);
            commitLetter(tray[i]);        // помечает символ как "вне игры"
            tray[i] = null;
        }
    }
}

void commitLetter(Letter l) {
    if (l != null && l.recall() != null) {
        l.paint(offGraphics, Letter.DIM);
        l.remember(null);           // помечает символ как "вне игры"
    }
}

```

## Методы `update()` и `paint()`

Здесь объявлено много частных переменных, обеспечивающих легкий доступ к размерностям игрового поля. Этот код также объявляет два внеэкран-

ных буфера: один — для использования в качестве изображения игрового поля и всех постоянно установленных фишек, а другой — для применения в качестве двойного буфера дисплея. Метод update() просто вызывает paint(), чтобы избежать мерцания. Метод paint() делает быстрое обращение к checksize(), чтобы удостовериться, что все буферы были созданы, затем проверяет, перемещаем ли мы окружающие буквы (с помощью выражения pick != null). Если так, то paint() делает копию графического внеэкранного контекста и усекает его до границ буквы, которую он рисует (x0, y0, w0, h0). Затем, он усекает экранный графический контекст до такого же прямоугольника. Это минимизирует число пикселов, которое он должен будет передвинуть для каждого перемещения мыши.

Для рисования мы копируем фоновое изображение offscreen, затем вызываем paint() на каждом символе в лотке, с установкой NORMAL-яркости буквы. Перетаскиваемый символ рисуется в режиме BRIGHT. Наконец, мы копируем на экран изображение двойного буфера (offscreen2).

```
private Letter pick;           // перетаскиваемая буква
private int dx, dy;            // смещение к верхнему/левому углу выбора
private int lw, lh;            // ширина и высота буквы
private int tm, lm;            // верхний и левый край
private int lt;                // толщина линии(между фишками)
private int aw, ah;            // размер области буквы

private Dimension offscreensize;
private Image offscreen;
private Graphics offGraphics;
private Image offscreen2;
private Graphics offGraphics2;

public void update(Graphics g) {
    paint(g);
}

public synchronized void paint(Graphics g) {
    Dimension d = checksize();
    Graphics gc = offGraphics2;
    if (pick != null) {
        gc = gc.create();
        gc.clipRect(x0, y0, w0, h0);
        g.clipRect(x0, y0, w0, h0);
    }
    gc.drawImage(offscreen, 0, 0, null);

    for (int i = 0; i < 7; i++) {
        Letter l = tray[i];
        if (l != null && l != pick)
            l.paint(gc, Letter.NORMAL);
    }
}
```

```

    if (pick != null)
        pick.paint(gc, Letter.BRIGHT);

    g.drawImage(offscreen2, 0, 0, null);
}

```

## Метод *LetterHit()*

Метод *LetterHit()* возвращает букву, которая находится под точкой  $(x, y)$ , и возвращает `null`, если никакой буквы там нет.

```

Letter LetterHit(int x, int y) {
    for (int i = 0; i < 7; i++) {
        if (tray[i] != null && tray[i].hit(x, y)) {
            return tray[i];
        }
    }
    return null;
}

```

## Метод *unplay()*

Этот простой метод удаляет букву из игры, которая была помещена в игровое поле, но не была еще передана.

```

private void unplay(Letter let) {
    Point p = let.recall();
    if (p != null) {
        board[p.y][p.x] = null;
        let.remember(null);
    }
}

```

## Метод *moveToTray()*

Методом *moveToTray()* удобно вычислять экранную позицию буквы в данном слоте лотка.

```

private void moveToTray(Letter l, int i) {
    int x = lm + (lw + lt) * i;
    int y = tm + ah - 2 * lt;
    l.move(x, y);
}

```

## Метод *dropOnTray()*

Метод *dropOnTray()* используется всякий раз, когда мы опускаем символ на область лотка или где-нибудь вне игрового поля. Это позволяет нам перета-

совывать содержимое лотка, а также просто возвращать фишку из игрового поля.

```
private void dropOnTray(Letter l, int x) {
    unplay(l); // удалить букву

    // найти, в каком слоте была эта буква
    int oldx = 0;
    for (int i = 0; i < 7; i++) {
        if (tray[i] == l) {
            oldx = i;
            break;
        }
    }

    // Если слот, на который мы опустили фишку, был пуст,
    // найти самый правый занятый слот.
    if (tray[x] == null) {
        for (int i = 6; i >= 0; i--) {
            if (tray[i] != null) {
                x = i;
                break;
            }
        }
    }

    // Если слот, на который мы опустили фишку, уже содержит
    // игравшую на поле фишку, просто обменяться со слотом.
    if (tray[x].recall() != null) {
        tray[oldx] = tray[x];
    } else {
        // мы просто переупорядочиваем фишку, которая уже в лотке
        if (oldx < x) { // перемещение влево
            for (int i = oldx; i < x; i++) {
                tray[i] = tray[i+1];
                if (tray[i].recall() == null)
                    moveToTray(tray[i], i);
            }
        } else { // перемещение вправо
            for (int i = oldx; i > x; i--) {
                tray[i] = tray[i-1];
                if (tray[i].recall() == null)
                    moveToTray(tray[i], i);
            }
        }
    }
    tray[x] = l;
    moveToTray(l, x);
}
```

## Метод *getLetter()*

*getLetter()* — простая оболочка *read-only* массива *board*.

```
Letter getLetter(int x, int y) {
    return board[y][x];
}
```

## Метод *moveLetter()*

Метод *moveLetter()* обрабатывает случаи, когда мы хотим передвигать фишку к позициям игрового поля или устанавливаем их на лоток. Если позиция  $(x, y)$  находится вне диапазона игрового поля, то используется лоток. Когда символ перемещается в игровое поле, оно должно быть пустым слотом, иначе символ возвращается назад к значению, сохраненному в *orig*.

```
void moveLetter(Letter l, int x, int y) {
    if (y > 14 || x > 14 || y < 0 || x < 0) {
        // если мы вне игрового поля
        if (x > 6)
            x = 6;
        if (x < 0)
            x = 0;
        dropOnTray(l, x);
    } else {
        if (board[y][x] != null) {
            x = orig.x;
            y = orig.y;
        } else {
            here.x = x;
            here.y = y;
            unplay(l);
            board[y][x] = l;
            l.remember(here);

            // вернуться к писелям
            x = lm + (lw + lt) * x;
            y = tm + (lh + lt) * y;
        }
        l.move(x, y);
    }
}
```

## Метод *checksize()*

Данный метод имеет имя, вводящее в заблуждение. *checksize()* делает намного больше, чем проверку размера апплета, но этот вид инициализации удобно выполнять однажды, когда мы подтверждаем размер апплета. Метод

содержит код рисунка для главного образа игрового поля. Он рисует все квадраты, включая цветные, и текст премиальных очков.

```
private Color bg = new Color(175, 185, 175);
private Color w3 = new Color(255, 50, 100);
private Color w2 = new Color(255, 200, 200);
private Color l3 = new Color(75, 75, 255);
private Color l2 = new Color(150, 200, 255);
private Color tiles[][] = {
    {w3, bg, bg, l2, bg, bg, bg, w3},
    {bg, w2, bg, bg, bg, l3, bg, bg},
    {bg, bg, w2, bg, bg, bg, l2, bg},
    {l2, bg, bg, w2, bg, bg, bg, l2},
    {bg, bg, bg, bg, w2, bg, bg, bg},
    {bg, l3, bg, bg, bg, l3, bg, bg},
    {bg, bg, l2, bg, bg, bg, l2, bg},
    {w3, bg, bg, l2, bg, bg, bg, w2}
};

private Dimension checksize() {
    Dimension d = getSize();
    int w = d.width;
    int h = d.height;

    if (w < 1 || h < 1)
        return d;
    if ((offscreen == null) ||
        (w != offscreensize.width) ||
        (h != offscreensize.height)) {
        System.out.println("updating board: " + w + " x " + h + "\r");

        offscreen = createImage(w, h);
        offscreensize = d;
        offGraphics = offscreen.getGraphics();
        offscreen2 = createImage(w, h);
        offGraphics2 = offscreen2.getGraphics();

        offGraphics.setColor(Color.white);
        offGraphics.fillRect(0, 0, w, h);

        // lt - толщина белых линий между фишками
        // gaps - сумма всех пробелов
        // lw, lh - размеры фишек
        // aw, ah - размеры всего игрового поля
        // lm, tm - левое и верхнее краевые поля
        // для центральной (aw,ah)-области в апплете

        lt = 1 + w / 400;
        int gaps = lt * 20;
```

```
lw = (w - gaps) / 15;
lh = (h - gaps - lt * 2) / 16;      // компенсация для высоты лотка
aw = lw * 15 + gaps;
ah = lh * 15 + gaps;
lm = (w - aw) / 2 + lt;
tm = (h - ah - (lt * 2 + lh)) / 2 + lt;

offGraphics.setColor(Color.black);
offGraphics.fillRect(lm, tm, aw-2*lt, ah-2*lt);
lm += lt;
tm += lt;
offGraphics.setColor(Color.white);
offGraphics.fillRect(lm, tm, aw-4*lt, ah-4*lt);
lm += lt;
tm += lt;
int sfh = (lh > 30) ? lh / 4 : lh / 2;
Font font = new Font("SansSerif", Font.PLAIN, sfh);
offGraphics.setFont(font);
for (int j = 0, y = tm; j < 15; j++, y += lh + lt) {
    for (int i = 0, x = lm; i < 15; i++, x += lw + lt) {
        Color c = tiles[j < 8 ? j : 14 - j][i < 8 ? i : 14 - i];
        offGraphics.setColor(c);
        offGraphics.fillRect(x, y, lw, lh);
        offGraphics.setColor(Color.black);
        if (lh > 30) {
            String td = (c == w2 || c == 12) ? "DOUBLE" :
                (c == w3 || c == 13) ? "TRIPLE" : null;
            String wl = (c == 12 || c == 13) ? "LETTER" :
                (c == w2 || c == w3) ? "WORD" : null;
            if (td != null) {
                center(offGraphics, td, x, y + 2 + sfh, lw);
                center(offGraphics, wl, x, y + 2 * (2 + sfh), lw);
                center(offGraphics, "SCORE", x, y + 3 * (2 + sfh), lw);
            }
        } else {
            String td = (c == w2 || c == 12) ? "2" :
                (c == w3 || c == 13) ? "3" : null;
            String wl = (c == 12 || c == 13) ? "L" :
                (c == w2 || c == w3) ? "W" : null;
            if (td != null) {
                center(offGraphics, td + wl, x,
                    y + (lh - sfh) * 4 / 10 + sfh, lw);
            }
        }
    }
}
Color c = new Color(255, 255, 200);
offGraphics.setColor(c);
```

```

offGraphics.fillRect(lm, tm + ah - 3 * lt, 7 * (lw + lt),
                     lh + 2 * lt);

Letter.resize(lw, lh);

// если мы уже имеем несколько букв, разместим их
for (int i = 0; i < 7; i++) {
    if (tray[i] != null) {
        moveToTray(tray[i], i);
    }
}
paintScore();
}

return d;
}

```

## Метод *center()*

*center()* — подпрограмма удобства, которую использует *checksize()* для выравнивания по центру текста "Double Letter Score".

```

private void center(Graphics g, String s, int x, int y, int w) {
    x += (w - g.getFontMetrics().stringWidth(s)) / 2;
    g.drawString(s, x, y);
}

```

## Метод *paintScore()*

Метод *paintScore()* выводит счет (очки) двух игроков или только одного (в режиме моногрока).

```

private void paintScore() {
    int x = lm + (lw + lt) * 7 + lm;
    int y = tm + ah - 3 * lt;
    int h = lh + 2 * lt;
    Font font = new Font("TimesRoman", Font.PLAIN, h/2);
    offGraphics.setFont(font);
    FontMetrics fm = offGraphics.getFontMetrics();

    offGraphics.setColor(Color.white);
    offGraphics.fillRect(x, y, aw, h);
    offGraphics.setColor(Color.black);
    if (others_name == null) {
        int y0 = (h - fm.getHeight()) / 2 + fm.getAscent();
        offGraphics.drawString("Score: " + total_score, x, y + y0);
    } else {
        h/=2;
        int y0 = (h - fm.getHeight()) / 2 + fm.getAscent();
        offGraphics.drawString(others_name + " Score: " +

```

```

    offGraphics.drawString(name + ":" + total_score, x, y + y0);
    offGraphics.drawString(others_name + ":" + others_score,
                           x, y + h + y0);
}
}

private int x0, y0, w0, h0;

```

## Метод *selectLetter()*

Метод *selectLetter()* проверяет позицию мыши, чтобы узнать, находится ли курсор над символом. При удовлетворительном результате он сохраняет то, что находится в *pick*, и вычисляет, как далеко была мышь от левого верхнего угла буквы, что сохраняется в *dx*, *dy*. Он также запоминает исходное положение этого символа в *orig*.

```

private void selectLetter(int x, int y) {
    pick = LetterHit(x, y);
    if(pick != null) {
        dx = pick.x - x;
        dy = pick.y - y;
        orig.x = pick.x;
        orig.y = pick.y;
    }
    repaint();
}

```

## Метод *dropLetter()*

В *dropLetter()* пользователь опускает букву, если он ее вытащил (из лотка). Он определяет, над каким квадратом на игровом поле была буква, когда она была опущена. Затем вызывается *moveLetter()*, чтобы попытаться передвинуть букву к этому квадрату.

```

private void dropLetter(int x, int y) {
    if(pick != null) {
        // определить центр фишки
        x += dx + lw / 2;
        y += dy + lh / 2;
        // определить индекс фишки
        x = (x - lm) / (lw + lt);
        y = (y - tm) / (lh + lt);

        moveLetter(pick, x, y);

        pick = null;
        repaint();
    }
}

```

## Метод *dragLetter()*

Метод *dragLetter()* управляет иначе, чем другие связанные с мышью события (главным образом — по соображениям эффективности). Цель состоит в том, чтобы иметь как можно более гладкое взаимодействие с пользователем. *dragLetter()* использует некоторую величину, чтобы вычислить ограничивающую область вокруг того места, где фишка была перед перемещением, и того, где она находится теперь. Затем он напрямую вызывает *paint(getGraphics())*. Это нестандартное Java-программирование апплета, но метод работает намного надежнее.

```
private void dragLetter(int x, int y) {
    if (pick != null) {
        int ox = pick.x;
        int oy = pick.y;
        pick.move(x + dx, y + dy);
        x0 = Math.min(ox, pick.x);
        y0 = Math.min(oy, pick.y);
        w0 = pick.w + Math.abs(ox - pick.x);
        h0 = pick.h + Math.abs(oy - pick.y);
        paint(getGraphics());
    }
}
```

## Метод *mousePressed()*

В следующем фрагменте кода обратите внимание, что *MyMouseAdapter* — внутренний класс, который расширяет *MouseAdapter*. Он переопределяет методы *mousePressed()* и *mouseReleased()*.

Для необходимой обработки метод *mousePressed()* вызывает метод *selectLetter()*. Координаты (x, y) текущей позиции мыши получаются из аргумента, передаваемого методу *mousePressed()*.

```
class MyMouseAdapter extends MouseAdapter {
    public void mousePressed(MouseEvent me) {
        selectLetter(me.getX(), me.getY());
    }
}
```

## Метод *mouseReleased()*

Для выполнения необходимой обработки метод *mouseReleased()* вызывает метод *dropLetter()*. Координаты (x, y) текущей позиции мыши получаются из аргумента, передаваемого методу *mouseReleased()*.

```
public void mouseReleased(MouseEvent me) {
    dropLetter(me.getX(), me.getY());
}
```

## Метод *mouseDragged()*

В следующем фрагменте кода обратите внимание, что `MyMouseMotionAdapter` — внутренний класс, который расширяет `MouseMotionAdapter`. Он переопределяет метод `mouseDragged()`.

Для необходимой обработки метод `MouseDragged()` вызывает метод `dragLetter()`. Координаты (*x*, *y*) текущей позиции мыши получаются из аргумента, передаваемого методу `mouseDragged()`.

```
class MyMouseMotionAdapter extends MouseMotionAdapter {
    public synchronized void mouseDragged(MouseEvent me) {
        dragLetter(me.getX(), me.getY());
    }
}
```

## *Bag.java*

Класс `Bag` совершенно прозрачен по сравнению с `Board`. Это простая абстракция для набора (пакета) букв. Когда вы создаете `Bag`-объект, вам пересыпается начальное случайное число, которое позволяет создавать два случайных набора на базе одного и того же начального случайного числа. Генератор случайных чисел сохраняется в переменной `rand`. Имеются два несколько странных целых массива, названных `letter_counts` и `letter_points`. Оба массива имеют длину в 27 элементов. Они представляют пустую фишку в элементе 0, и буквы от *A* до *Z* — в элементах от 1 до 26. Массив `letter_counts` указывает, сколько каждой из букв находится в полном `bag`-наборе. Например, `letter_counts[1] = 9` означает наличие в наборе девяти фишек с буквой *A*. Аналогично, массив `letter_points` приписывает каждому символу некоторое значение (вес, стоимость), измеряемое в очках. Фишка *A* стоит только 1 очко, а фишка *Z* — 10. Существует 100 букв, хранящихся в массиве с именем `letters`. Количество букв, фактически оставшихся в `bag`-наборе за время проведения игры, сохраняется в `n`.

```
import java.util.Random;

class Bag {
    private Random rand;
    private int letter_counts[] = {
        2, 9, 2, 2, 4, 12, 2, 3, 2, 9, 1, 1, 4, 2,
        6, 8, 2, 1, 6, 4, 6, 4, 2, 2, 1, 2, 1
    };
    private int letter_points[] = {
        0, 1, 3, 3, 2, 1, 4, 2, 4, 1, 8, 5, 1, 3,
        1, 1, 3, 10, 1, 1, 1, 4, 4, 8, 4, 10
    };
}
```

```
private Letter letters[] = new Letter[100];
private int n = 0;
```

## Конструктор *Bag()*

Баг-конструктор берет начальное число и делает из него Random-объект. Затем он сканирует массив letter\_counts, создавая требуемое количество новых Letter-объектов, заботясь о том, чтобы заменить пробельную фишку звездочкой. Затем он вызывает putBack() для каждой буквы, чтобы поместить их в bag-набор.

```
Bag(int seed) {
    rand = new Random(seed);
    for (int i = 0; i < letter_counts.length; i++) {
        for (int j = 0; j < letter_counts[i]; j++) {
            Letter l = new Letter(i == 0 ? '*' : (char) ('A' + i - 1),
                                  letter_points[i]);
            putBack(l);
        }
    }
}
```

## Метод *takeOut()*

Следующий метод довольно остроумен и немного неэффективен, но не опасен. Метод takeOut() выбирает случайное число между 0 и n-1. Затем извлекает букву с таким смещением из массива letters. Используя System.arraycopy(), он "закрывает дырку" в этом элементе массива letters. Далее takeOut() выполняет декремент переменной n и возвращает букву.

```
synchronized Letter takeOut() {
    if (n == 0)
        return null;
    int i = (int) (rand.nextDouble() * n);
    Letter l = letters[i];
    if (i != n - 1)
        System.arraycopy(letters, i + 1, letters, i, n - i - 1);
    n--;
    return l;
}
```

## Метод *putBack()*

Метод putBack() используется конструктором для первоначального размещения фишки в bag-наборе. Его можно также использовать в будущем усовершенствовании игры, которое позволит соперникам торговаться фишками,

неудачно обмененными при потере хода. Он просто помещает букву назад в конец массива.

```
synchronized void putBack(Letter l) {
    letters[n++] = l;
}
```

## **Letter.java**

Класс Letter довольно ясен в том, что он ничего не знает об игре или игровом поле. Он просто инкапсулирует позицию и визуальное исполнение одиночной буквы. Класс использует несколько статических переменных, содержащих информацию относительно шрифтов и размеров. Подобный механизм не позволяет апплету закончиться с сотней шрифтов в памяти сразу. Однако это приводит к побочному эффекту, заключающемуся в том, что страница браузера не может содержать два экземпляра апплета Scrabblet, если каждый из них имеет отличающиеся размеры. Инициализация второго будет переписывать значения в этих статических переменных.

Переменные w и h содержат постоянную ширину и высоту каждой буквы. Переменные font и smfont — это AWT-объекты шрифта для большой буквы и маленького (весового) значка<sup>1</sup>. int-переменные y0 и ys0 хранят смещение базовой линии символа и веса, соответственно. Обеспечивается несколько констант для передачи обратно в paint(), чтобы указать, в каком цветовом состоянии следует рисовать (NORMAL (нормальный), DIM (серый) или BRIGHT (светлый) режимы).

```
import java.awt.*;

class Letter {
    static int w, h;
    private static Font font, smfont;
    private static int y0, ys0;
    private static int lasth = -1;
    static final int NORMAL = 0;
    static final int DIM = 1;
    static final int BRIGHT = 2;
```

## **Массив colors[], методы mix(), gain() и clamp()**

Массив colors инициализируется статически с девятью цветовыми объектами — три набора из трех цветов. Метод mix() берет набор RGB-значений, например, 250, 220, 100 и переводит его в три цвета, которые можно исполь-

<sup>1</sup> Здесь весовой значок соответствует количеству очков, приписанных букве. — Примеч. ред.

зователь для обеспечения трехмерной подсветки. Данный метод обращается к `gain()`, чтобы увеличить или уменьшить яркость цвета, и обращается к `clamp()`, чтобы удостовериться, что цвет остается в нужном диапазоне.

```
private static Color colors[][] = {
    mix(250, 220, 100),           // нормальный
    mix(200, 150, 80),           // серый
    mix(255, 230, 150)           // светлый
};

private static Color mix(int r, int g, int b) {
    Color arr[] = new Color[3];

    arr[NORMAL] = new Color(r, g, b);
    arr[DIM] = gain(arr[0], .71);
    arr[BRIGHT] = gain(arr[0], 1.31);
    return arr;
}

private static int clamp(double d) {
    return (d < 0) ? 0 : ((d > 255) ? 255 : (int) d);
}

private static Color gain(Color c, double f) {
    return new Color(clamp(c.getRed() * f), clamp(c.getGreen() * f),
                    clamp(c.getBlue() * f));
}
```

## Экземплярные переменные

Флажок `valid` используется для того, чтобы удостовериться, что все размерные переменные установлены точно так же, как в момент первой прорисовки данной буквы (`Letter`-объекта). Несколько переменных (`x0`, `w0`, `xs0`, `ws0` и `gap`) хранятся здесь во избежание большого количества вычислений, необходимых каждый раз, когда апллет рисует буквы и очки на фишках. Все они объясняются в следующих за ними комментариях. Объект `Point tile` применяется для запоминания, на каком квадрате игрового ( $15 \times 15$ )-поля включена данная буква (объект типа `Letter`). Если эта переменная — `null`, то `Letter` не находится на игровом поле. Пара `int x,y;` используется для того, чтобы точно располагать `Letter`-объект.

```
private boolean valid = false;  
  
// Квантованная позиция Letter на фишке. (Здесь только хранится.)  
private Point tile = null;  
  
int x, y; // позиция Letter  
private int x0; // смещение символа на фишке  
private int w0; // ширина символа в пикселях  
private int xs0; // смещение очков (веса) на фишке
```

```
private int ws0;           // ширина очков в пикселях
private int gap = 1;        // интервал (в пикселях) между символом и весом
```

## Методы *Letter()*, *getSymbol()* и *getPoints()*

Переменная *symbol* — строка, которая содержит отображаемую букву, а *points* — значение (*стоимость* или *вес*) этой буквы (в очках — целых числах). Обе они инициализируются только конструктором и возвращаются методами-оболочками *getSymbol()* и *getPoints()*, соответственно.

```
private String symbol;
private int points;

Letter(char s, int p) {
    symbol = "" + s;
    points = p;
}

String getSymbol() {
    return symbol;
}

int getPoints() {
    return points;
}
```

## Методы *move()*, *remember()* и *recall()*

Метод *move()* используется для сообщения данной фишке, где ее нужно нарисовать (на доске). Метод *remember()*, однако, более сложный. Он может вызываться с аргументом *null*, что означает, что эта фишка должна "забыть", где она была (в памяти). Данная ситуация означает, что символ не в игре. Иначе он сообщает, какие координаты на доске занимает эта фишка (с буквой). Состояние фишки просматривается вызовом *recall()*.

```
void move(int x, int y) {
    this.x = x;
    this.y = y;
}

void remember(Point t) {
    if (t == null) {
        tile = t;
    } else {
        tile = new Point(t.x, t.y);
    }
}
```

```
Point recall() {
    return tile;
}
```

## Метод *resize()*

Метод *resize()* вызывается однажды (игровым полем), чтобы восстанавливать размеры каждой буквы. Помните, что переменные *w* и *h* — статические, так что он воздействует на все экземпляры *Letter* сразу.

```
static void resize(int w0, int h0) {
    w = w0;
    h = h0;
}
```

## Метод *hit()*

Метод *hit()* возвращает *true*, если переданная ему пара *xp*, *yp* попадает внутрь границ заданного *Letter*-объекта.

```
boolean hit(int xp, int yp) {
    return (xp >= x && xp < x + w && yp >= y && yp < y + h);
}
```

## Метод *validate()*

Метод *validate()* используется для загрузки шрифтов, чтобы выяснить, насколько велики буквы, и решить, где их рисовать. Эта информация хранится в частных переменных, обсужденных ранее. Результаты данных вычислений применяются далее в *paint()*.

```
private int font_ascent;
void validate(Graphics g) {
    FontMetrics fm;
    if (h != lasth) {
        font = new Font("SansSerif", Font.BOLD, (int)(h * .6));
        g.setFont(font);
        fm = g.getFontMetrics();
        font_ascent = fm.getAscent();

        y0 = (h - font_ascent) * 4 / 10 + font_ascent;

        smfont = new Font("SansSerif", Font.BOLD, (int)(h * .3));
        g.setFont(smfnt);
        fm = g.getFontMetrics();
        ys0 = y0 + fm.getAscent() / 2;
        lasth = h;
    }
}
```

```
if (!valid) {
    valid = true;
    g.setFont(font);
    fm = g.getFontMetrics();
    w0 = fm.stringWidth(symbol);
    g.setFont(smfnt);
    fm = g.getFontMetrics();
    ws0 = fm.stringWidth(" " + points);
    int slop = w - (w0 + gap + ws0);
    x0 = slop / 2;
    if (x0 < 1)
        x0 = 1;
    xs0 = x0 + w0 + gap;
    if (points > 9)
        xs0--;
}
}
```

## Метод *paint()*

Метод *paint()* вызывается игровым полем. Ему пересыпается целое число *i*, которое является одной из статических переменных (констант) — *NORMAL*, *BRIGHT* или *DIM* — этого класса. Оно используется как индекс в массиве *colors* для выбора основного цвета. Чтобы создать внешний вид трехмерной высвеченной и затененной кнопки, заполняется последовательность прямоугольников. Если *points* больше, чем нуль, что указывает непустую букву, то рисуется основная буква и рядом с ней ее значение.

```
void paint(Graphics g, int i) {
    Color c[] = colors[i];
    validate(g);
    g.setColor(c[NORMAL]);
    g.fillRect(x, y, w, h);
    g.setColor(c[BRIGHT]);
    g.fillRect(x, y, w - 1, 1);
    g.fillRect(x, y + 1, 1, h - 2);
    g.setColor(Color.black);
    g.fillRect(x, y + h - 1, w, 1);
    g.fillRect(x + w - 1, y, 1, h - 1);
    g.setColor(c[DIM]);
    g.fillRect(x + 1, y + h - 2, w - 2, 1);
    g.fillRect(x + w - 2, y + 1, 1, h - 3);
    g.setColor(Color.black);
    if (points > 0) {
        g.setFont(font);
        g.drawString(symbol, x + x0, y + y0);
```

```
    g.setFont(smfont);
    g.drawString(""+points, x + xs0, y + ys0);
}
}
```

## **ServerConnection.java**

Последний класс на стороне клиента этого апплета — `ServerConnection`, который инкапсулирует связь с сервером и нашим противником. Существует несколько переменных, объявленных в начале класса. Номер порта сокета (`port`) для присоединения к серверу — 6564. `CRLF` — константная строка `Internet`, представляющая символ "конец строки". Потоки ввода/вывода сервера находятся в объектных переменных `in` и `out`, соответственно. Уникальный идентификатор (`ID`), под которым это подключение известно на сервере, сохраняется в `id`. Идентификатор того, кто подключается к нам как противник, сохраняется в `toid`. Апплет `Scrabblet`, с которым мы соединяемся, — это переменная `scrabblet`.

```
import java.io.*;
import java.net.*;
import java.util.*;

class ServerConnection implements Runnable {
    private static final int port = 6564;
    private static final String CRLF = "\r\n";
    private BufferedReader in;
    private PrintWriter out;
    private String id, toid = null;
    private Scrabblet scrabblet;
```

### Конструктор *ServerConnection()*

Конструктор `ServerConnection` получает имя Internet-сайта, к которому нужно присоединиться, и пытается открыть сокет кциальному порту на этой хост-машине. Если это проходит успешно, он создает `BufferedReader`-объект (с вложенным `InputStreamReader`-объектом, построенным вокруг объекта входного потока `InputStream`) и `PrintWriter`-объект, строящийся вокруг объекта выходного потока (`OutputStream`). Если подключение неудачно, в вызывающую программу выбрасывается исключение `IOException`.

```
public ServerConnection(Scrabblet sc, String site) throws IOException {
    scrabblet = sc;
    Socket server = new Socket(site, port);
    in = new BufferedReader(new InputStreamReader(server.getInputStream()));
    out = new PrintWriter(server.getOutputStream(), true);
}
```

## Метод `readline()`

Метод `readline()` — просто функция удобства, которая преобразует исключение `IOException` из `readLine()` в простое возвращаемое значение — `null` (пустая ссылка).

```
private String readline() {
    try {
        return in.readLine();
    } catch (IOException e) {
        return null;
    }
}
```

## Методы `setName()` и `delete()`

Метод `setName()` просит сервер ассоциировать указанное имя с нами, а метод `delete()` используется для удаления нашего имени из любых списков, хранящихся на сервере.

```
void setName(String s) {
    out.println("name " + s);
}

void delete() {
    out.println("delete " + id);
}
```

## Методы `setTo()` и `send()`

Метод `setTo()` устанавливает связку с идентификатором (ID) противника. Будущие вызовы `send()` будут направлять свои результаты этому игроку.

```
void setTo(String to) {
    toid = to;
}

void send(String s) {
    if (toid != null)
        out.println("to " + toid + " " + s);
}
```

## Методы `challenge()`, `accept()`, `chat()`, `move()`, `turn()` и `quit()`

Следующие небольшие методы посыпают короткие сообщения от данного клиента серверу, который в свою очередь пересыпает их нашему противнику. Чтобы инициализировать старт игры, используется сообщение `challenge`,

а accept посыпается в ответ на challenge. Для каждой буквы, которая перемещается, посыпается сообщение move, а в конце каждого хода — сообщение turn. Если клиент завершает игру или покидает страницу с апплетом, он отправляет сообщение quit.

```

void challenge(String destid) {
    setTo(destid);
    send("challenge " + id);
}

void accept(String destid, int seed) {
    setTo(destid);
    send("accept " + id + " " + seed);
}

void chat(String s) {
    send("chat " + id + " " + s);
}

void move(String letter, int x, int y) {
    send("move " + letter + " " + x + " " + y);
}

void turn(String words, int score) {
    send("turn " + score + " " + words);
}

void quit() {
    send("quit " + id);           // попросить другого игрока
    out.println("quit");          // завершить игру
}

```

## Метод start()

Следующий метод просто запускает поток, который управляет клиентской стороной сети.

```

// чтение с сервера...
private Thread t;

void start() {
    t = new Thread(this);
    t.start();
}

```

## Ключевые слова

Показанные здесь статические переменные и статический блок используются для того, чтобы инициализировать объектную переменную Hashtable keys

с последующим отображением строк в keystrings их позициями в массиве — например, keys.get("move") == MOVE. Метод lookup() заботится о распаковке Integer-объектов в правильные целые. При этом -1 означает, что ключевое слово не было найдено.

```
private static final int ID = 1;
private static final int ADD = 2;
private static final int DELETE = 3;
private static final int MOVE = 4;
private static final int CHAT = 5;
private static final int QUIT = 6;
private static final int TURN = 7;
private static final int ACCEPT = 8;
private static final int CHALLENGE = 9;
private static Hashtable keys = new Hashtable();
private static String keystrings[] = {
    "", "id", "add", "delete", "move", "chat",
    "quit", "turn", "accept", "challenge"
};
static {
    for (int i = 0; i < keystrings.length; i++)
        keys.put(keystrings[i], new Integer(i));
}
private int lookup(String s) {
    Integer i = (Integer) keys.get(s);
    return i == null ? -1 : i.intValue();
}
```

## Метод run()

run() — главный цикл подключения игры к серверу. Он начинается с блокирующего обращения к readLine(), который будет возвращать объект типа String всякий раз, когда от сервера приходит строка текста. Он использует StringTokenizer, чтобы разбить строку на слова. Оператор switch подключает нас кциальному коду, основанному на первом слове во входной строке. Каждое из ключевых слов в протоколе по-разному анализирует входную строку, и большинство из них для выполнения своей работы организуют вызовы методов класса Scrabblet.

```
public void run() {
    String s;
    StringTokenizer st;
    while ((s = readLine()) != null) {
        st = new StringTokenizer(s);
        String keyword = st.nextToken();
        switch (lookup(keyword)) {
            default:
```

```
System.out.println("bogus keyword: " + keyword + "\r");
break;
case ID:
    id = st.nextToken();
    break;
case ADD: {
    String id = st.nextToken();
    String hostname = st.nextToken();
    String name = st.nextToken(CRLF);
    scrabblet.add(id, hostname, name);
}
break;
case DELETE:
    scrabblet.delete(st.nextToken());
    break;
case MOVE: {
    String ch = st.nextToken();
    int x = Integer.parseInt(st.nextToken());
    int y = Integer.parseInt(st.nextToken());
    scrabblet.move(ch, x, y);
}
break;
case CHAT: {
    String from = st.nextToken();
    scrabblet.chat(from, st.nextToken(CRLF));
}
break;
case QUIT: {
    String from = st.nextToken();
    scrabblet.quit(from);
}
break;
case TURN: {
    int score = Integer.parseInt(st.nextToken());
    scrabblet.turn(score, st.nextToken(CRLF));
}
break;
case ACCEPT: {
    String from = st.nextToken();
    int seed = Integer.parseInt(st.nextToken());
    scrabblet.accept(from, seed);
}
break;
case CHALLENGE: {
    String from = st.nextToken();
    scrabblet.challenge(from);
}
```

```
        break;
    }
}
}
}
```

## Код сервера

Два последних класса не являются частью этого апплета. Они должны устанавливаться и выполняться отдельно на Web-сервере, откуда должны быть загружены классы апплета. Это потребует прав защиты, чтобы устанавливать и выполнять так называемые демон-процессы (daemon processes) на Web-сайте, которые имеют не многие пользователи. К счастью, большинство участников этой игры не будет устанавливать свои собственные серверы; более вероятно, что они будут только запускать игры, подключенные к существующим серверам.

### **Server.java**

Server — главный класс для серверной стороны Scrabblet. До установки на Web-сервере вы должны выполнять его, используя Java-интерпретатор своей системы, такой как java.exe, jview.exe или sj.exe. Пример для Windows 95/98/NT:

```
C:\java\Scrabblet> jview Server
```

После запуска Server ответит следующим сообщением:

```
Server listening on port 6564
```

Класс Server начинается объявлением нескольких переменных. Порт должен иметь такой же номер (6564), какой мы видели в ServerConnection. Чтобы хранить все подключения ко всем клиентам, используется hashtable idcon. Для управления частой вставки и удаления, мы используем хештаблицу, а не массив, который требует большого количества копирований самого себя. Переменная id увеличивается на 1 для каждого нового подключения. Она соответствует экземплярной переменной id, которую мы видели ранее у клиента.

```
import java.net.*;
import java.io.*;
import java.util.*;

public class Server implements Runnable {
private int port = 6564;
private Hashtable idcon = new Hashtable();
```

```
private int id = 0;
static final String CRLF = "\r\n";
```

## Метод **addConnection()**

Метод `addConnection()` вызывается каждый раз, когда новый клиент присоединяется к нашему апплету. Для управления клиентом метод создает новый экземпляр `ClientConnection`, описанный ниже. Ему передается ссылка на данный `Server` (сокет с присоединенным клиентом) и текущее значение `id`. Наконец, он инкрементирует переменную `id`, чтобы сделать ее готовой к следующему подключению.

```
synchronized void addConnection(Socket s) {
    ClientConnection con = new ClientConnection(this, s, id);
    // Мы будем ждать, пока ClientConnection не выполнит
    // квитированную установку своего "имени" перед вызовом
    // set()(см. ниже), который делает это соединение "живым".
    id++;
}
```

## Метод **set()**

Метод `set()` вызывается из `ClientConnection` в ответ на сообщение клиентом его имени. Он прослеживает все подключения в хэш-таблице `idcon` и сначала удаляет данный `id` из нее так, чтобы она не смогла получить дубликата, если клиент посыпает свое имя дважды. Чтобы показать, что это подключение доступно для запуска игры, метод вызывает `setBusy(false)`. Затем он проходит через все другие подключения, перечисляя ключи хэш-таблицы `idcon`. Для всех незанятых подключений (чье игроки ожидают противника) `set()` посыпает протокольное сообщение "add", так что они будут знать об этом подключении.

```
synchronized void set(String the_id, ClientConnection con) {
    idcon.remove(the_id);           // убедитесь, что мы здесь не дважды
    con.setBusy(false);
    // скажите об этом другим клиентам
    Enumeration e = idcon.keys();
    while (e.hasMoreElements()) {
        String id = (String)e.nextElement();
        ClientConnection other = (ClientConnection) idcon.get(id);
        if (!other.isBusy())
            con.write("add " + other + CRLF);
    }
    idcon.put(the_id, con);
    broadcast(the_id, "add " + con);
}
```

## Метод *sendto()*

Метод *sendto()* вызывается в ответ на протокольное сообщение "to". Он записывает то, что находится в строке *body* прямо в соединение, идентифицированное параметром *dest*.

```
synchronized void sendto(String dest, String body) {
    ClientConnection con = (ClientConnection) idcon.get(dest);
    if (con != null) {
        con.write(body + CRLF);
    }
}
```

## Метод *broadcast()*

Метод *broadcast()* используется для отправки одиночного сообщения (передаваемого в параметре *body*) на каждое отдельное соединение, исключая то, которое идентифицировано в параметре *exclude* (обычно это соединение отправителя).

```
synchronized void broadcast(String exclude, String body) {
    Enumeration e = idcon.keys();
    while (e.hasMoreElements()) {
        String id = (String)e.nextElement();
        if (!exclude.equals(id)) {
            ClientConnection con = (ClientConnection) idcon.get(id);
            con.write(body + CRLF);
        }
    }
}
```

## Метод *delete()*

Чтобы попросить всех соединенных клиентов забыть, что они когда-либо слышали об *the\_id*, применяется метод *delete()*. Он используется клиентами, которые заняты в игре, чтобы удалить себя из списков преемственности других игроков.

```
synchronized void delete(String the_id) {
    broadcast(the_id, "delete " + the_id);
}
```

## Метод *kill()*

Метод *kill()* вызывается всякий раз, когда клиент явно завершает игру, посыпая сообщение "quit", или когда клиент просто "умирает", если браузер завершает работу.

```

synchronized void kill(ClientConnection c) {
    if (idcon.remove(c.getId()) == c) {
        delete(c.getId());
    }
}

```

## Метод *run()*

Метод *run()* — главный цикл сервера. Он создает новый сокет на порте 6564 и входит в бесконечный цикл, принимающий сокет-соединения от клиентов. Он вызывает *addConnection()* для каждого принимаемого сокета.

```

public void run() {
    try {
        ServerSocket acceptSocket = new ServerSocket(port);
        System.out.println("Server listening on port " + port);
        while (true) {
            Socket s = acceptSocket.accept();
            addConnection(s);
        }
    } catch (IOException e) {
        System.out.println("accept loop IOException: " + e);
    }
}

```

## Метод *main()*

*main()* — это, конечно, метод, выполняемый Java-интерпретатором (через командную строку). Он создает новый экземпляр класса *Server* и запускает на выполнение новый поток (*Thread*).

```

public static void main(String args[]) {
    new Thread(new Server()).start();
    try {
        Thread.currentThread().join();
    } catch (InterruptedException e) { }
}

```

## *ClientConnection.java*

Данный класс — зеркальное отображение *ServerConnection* рассматриваемого апплета. Он создается для каждого клиента, и его работа должна управлять всем вводом/выводом клиента. Частные экземплярные переменные содержат все о состоянии данного клиента. *Socket* хранится в *sock*. Буферизированные считающий и выходной потоки хранятся в *in* и *out*. Имя

хост-машины клиента хранится в `host`. Ссылка на Server-экземпляр, которая создается этим клиентом, содержится в `server`. Имя игрока данного клиента хранится в `name`, в то время как автоматически назначаемый идентификационный номер игрока содержится в `id`. Булева переменная `busy` указывает, действительно ли этот клиент активно занят в игре.

```
import java.net.*;
import java.io.*;
import java.util.*;

class ClientConnection implements Runnable {
    private Socket sock;
    private BufferedReader in;
    private OutputStream out;
    private String host;
    private Server server;
    private static final String CRLF = "\r\n";
    private String name = null;           // для людей
    private String id;
    private boolean busy = false;
```

## **Конструктор *ClientConnection()***

Конструктор сохраняет ссылку на сервер и сокет и запоминает уникальный идентификатор. Он создает `BufferedReader`-объект (с вложенным `InputStreamReader`-объектом, организуемым на базе входного потока), так что он может вызывать `readLine()`. Затем записывает `id` обратно клиенту, чтобы позволить ему узнать этот номер. Наконец, он создает и запускает новый поток `Thread`, чтобы обработать соединение.

```
public ClientConnection(Server srv, Socket s, int i) {
    try {
        server = srv;
        sock = s;
        in = new BufferedReader(new InputStreamReader(s.getInputStream()));
        out = s.getOutputStream();
        host = s.getInetAddress().getHostName();
        id = "" + i;

        // сказать новичку, кто он такой...
        write("id " + id + CRLF);

        new Thread(this).start();
    } catch (IOException e) {
        System.out.println("failed ClientConnection " + e);
    }
}
```

## Метод *toString()*

Мы переопределяем *toString()* так, чтобы можно было иметь ясное представление этого соединения для регистрации.

```
public String toString() {
    return id + " " + host + " " + name;
}
```

## Методы *getHost()*, *getId()*, *isBusy()* и *setBusy()*

Мы упаковываем *host*, *id*, и *busy* в *public*-методы, чтобы разрешить для них доступ *read-only* (доступ только для чтения).

```
public String getHost() {
    return host;
}
public String getId() {
    return id;
}
public boolean isBusy() {
    return busy;
}
public void setBusy(boolean b) {
    busy = b;
}
```

## Метод *close()*

Метод *close()* вызывается, если клиент явно завершает работу, или если мы получаем от сокета сообщение об исключении. Мы вызываем метод *kill()* сервера, который удаляет нас из любых списков. Затем закрываем сокет, который в свою очередь закрывает потоки ввода и вывода.

```
public void close() {
    server.kill(this);
    try {
        sock.close();           // закрыть сокет и входные/выходные потоки
    } catch (IOException e) { }
}
```

## Метод *write()*

Чтобы записать строку в поток, мы должны преобразовать ее в массив байтов, используя *getBytes()*.

```
public void write(String s) {
    byte buf[];
    buf = s.getBytes();
```

```
    try {
        out.write(buf, 0, buf.length);
    } catch (IOException e) {
        close();
    }
}
```

## Метод `readline()`

Метод `readline()` просто преобразует строчный IOException из `readLine()` в возвращаемое значение `null`.

```
private String readline() {
    try {
        return in.readLine();
    } catch (IOException e) {
        return null;
    }
}
```

## Ключевые слова

Текущий раздел очень похож на аналогичную часть класса `ServerConnection`, который представляет другой конец соединения. Статические переменные и статический блок, показанные здесь, используются для инициализации `Hashtable keys` с отображением между строками в `keystrings` и их позициями в массиве — например, `keys.get("quit") == QUIT`. Метод `lookup()` заботится о распаковке `Integer`-объектов в правильные `int`-значения, а `-1` означает, что ключевое слово не было найдено.

```
static private final int NAME = 1;
static private final int QUIT = 2;
static private final int TO = 3;
static private final int DELETE = 4;

static private Hashtable keys = new Hashtable();
static private String keystrings[] = {
    "", "name", "quit", "to", "delete"
};
static {
    for (int i = 0; i < keystrings.length; i++)
        keys.put(keystrings[i], new Integer(i));
}

private int lookup(String s) {
    Integer i = (Integer) keys.get(s);
    return i == null ? -1 : i.intValue();
}
```

## Метод run()

run() содержит цикл, который управляет всей связью с данным клиентом. Он использует StringTokenizer, чтобы анализировать входные строки, исключая первое слово в каждой строке. Чтобы отыскивать эти первые слова в хэш-таблице, keys использует только что показанный метод lookup(). Затем выполняется переключение, основанное на целочисленном значении ключевого слова. Сообщение с NAME приходит от клиентов тогда, когда они впервые получают именную идентификацию. Чтобы установить это подключение, мы вызываем метод сервера set(). Сообщение QUIT посыпается тогда, когда клиент хочет закончить сеанс своего сервера. Сообщение TO содержит идентификатор адресата и тело сообщения, которое будет послано этому клиенту. Для передачи сообщения мы вызываем метод сервера sendto(). Сообщение DELETE отправляется клиентами, которые хотят продолжить соединение, но больше не желают, чтобы их имена были перечислены как доступные для игры. Метод run() устанавливает флагок busy и вызывает метод сервера delete(), который уведомляет клиентов о том, что мы не хотим, чтобы нас вызывали.

```
public void run() {
    String s;
    StringTokenizer st;

    while ((s = readline()) != null) {
        st = new StringTokenizer(s);
        String keyword = st.nextToken();
        switch (lookup(keyword)) {
            default:
                System.out.println("bogus keyword: " + keyword + "\r");
                break;
            case NAME:
                name = st.nextToken() +
                    (st.hasMoreTokens() ? " " + st.nextToken(CRLF) : "");
                System.out.println("[ " + new Date() + " ] " + this + "\r");
                server.set(id, this);
                break;
            case QUIT:
                close();
                return;
            case TO:
                String dest = st.nextToken();
                String body = st.nextToken(CRLF);
                server.sendto(dest, body);
                break;
            case DELETE:
                busy = true;
```

```
server.delete(id);
break;
}
}
close();
}
}
```

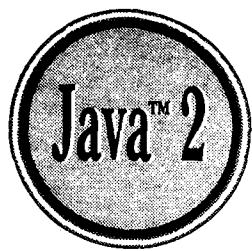
## Совершенствование Scrabble

Этот апплет представляет законченную многопользовательскую игру, построенную по технологии клиент-сервер. В будущем код в `Server` и `ServerConnection` мог бы быть расширен во многих направлениях. Он мог бы использоваться для поддержки других игр, основанных на игровых ходах. Он мог бы прослеживать и поддерживать список счетов (очков) для каждой игры и динамически расширяться для понимания новых глаголов протокола.

Например, для игры, описанной в этой главе, можно было бы иметь поисковую функцию, которая проверяла ряд представленных слов по словарю, хранящемуся на сервере. Сервер мог бы тогда быть арбитром для таких, например, споров, является ли *хужу* допустимым словом. Вы могли бы также создать слово *робот*, которое постоянно находится на сервере, но будет действовать подобно другому игроку и будет использовать словарь, чтобы генерировать лучшее размещение слов из своего текущего набора из семи букв. Он мог бы даже использовать список содержательных цитат для ввода в chat-окно после каждого перемещения фишк. Вы могли бы попытаться сделать некоторые из этих улучшений самостоятельно.

Рассмотренный апплет предназначен для развлечения и образовательных целей. Любое подобие каким-либо коммерческим продуктам — просто совпадение.





## ПРИЛОЖЕНИЕ

# Использование документационных комментариев Java

Как упоминалось в Части I, Java поддерживает три типа комментариев. Два первых (одно- и многострочные) используют символьные последовательности `//` и `/* */`. Третий тип называется *документационным комментарием*. Он начинается с символьной последовательности `/**`, а заканчивается последовательностью `*/`. Документационные комментарии позволяют внедрять информацию относительно вашей программы в саму программу. Затем можно использовать утилиту `javadoc`, чтобы извлечь информацию и поместить ее в HTML-файл. Документационные комментарии делают удобными пояснения к вашей программе. Вы, конечно, видели документацию, сгенерированную утилитой `javadoc`, потому что этим способом фирма Sun документировала библиотеку Java API.

### Теги `javadoc`

Утилита `javadoc` распознает теги, представленные в табл. П1.

**Таблица П1. Теги `javadoc`**

Тег	Значение
<code>@author</code>	Идентифицирует автора класса
<code>@deprecated</code>	Определяет, что класс или элемент исключен
<code>@exception</code>	Идентифицирует исключение, выброшенное методом
<code>(@link)</code>	Вставляет встроенную (in-line) ссылку к другому разделу. (Добавлен в Java 2)
<code>@param</code>	Документирует параметр метода

Таблица П1 (окончание)

Ter	Значение
@return	Документирует возвращаемое значение метода
@see	Определяет ссылку к другому разделу
@serial	Документирует умалчивающееся сериализуемое (serializable) поле. (Добавлен в Java 2)
@serialData	Документирует данные, записанные методом writeObject() или writeExternal(). (Добавлен в Java 2)
@serialField	Документирует компонент ObjectOutputStreamField. (Добавлен в Java 2)
@since	Объявляет выпуск (релиз), в котором было введено специфическое изменение
@throws	То же самое, что @exception
@version	Определяет версию класса

Как можно видеть, все теги документа начинаются с символа @. В документационном комментарии можно также использовать другие, стандартные теги HTML. Однако некоторые из них, такие как заголовки, не должны использоваться, потому что они прерывают просмотр HTML-файла, произведенного утилитой javadoc.

Вы можете использовать документационные комментарии для разъяснения классов, интерфейсов, полей, конструкторов и методов. Во всех случаях, документационный комментарий должен непосредственно предшествовать документируемому элементу. Когда вы документируете переменную, документационные теги, которые вы можете использовать — это @see, @since, @serial, @serialField и @deprecated. Для классов вы можете использовать теги @see, @author, @since, @deprecated и @version. Методы могут быть документированы с помощью @see, @return, @param, @since, @deprecated, @throws, @serialData и @exception. Тег {@link} можно использовать везде. Ниже рассматривается каждый тег.

## Ter @author

Ter @author документирует автора класса. Он имеет следующий синтаксис:

**@author description**

Здесь **description** — обычно имя того, кто написал класс. Тег @author можно использовать только в документации для класса. Для того чтобы включить в документацию HTML поле @author, вам может понадобиться определить опцию -author при выполнении javadoc.

## Тег `@deprecated`

Тег `@deprecated` определяет, что класс или член (*member*) исключены и не рекомендуются к применению. Не плохо, чтобы существовали также теги `@see` для информирования программиста о доступных альтернативах. Синтаксис:

`@deprecated description`

Здесь *description* — сообщение, которое описывает исключение. Информация, указанная в теге `@deprecated`, распознается компилятором и включается в генерируемый class-файл. Поэтому программисту эта информация может быть выдана при компиляции исходных Java-файлов. Тег `@deprecated` можно использовать в документации для переменных, методов и классов.

## Тег `@exception`

Тег `@exception` описывает исключение (исключительное состояние) для метода. Он имеет следующий синтаксис

`@exception exception-name explanation`

Здесь *exception-name* — полное составное имя исключения; *explanation* — строка, которая описывает, как исключение может происходить. Тег `@exception` можно использовать только в документации метода.

## Тег `{@link}`

Тег `{@link}` обеспечивает встроенную гиперссылку на дополнительную информацию. Он имеет следующий синтаксис:

`{@link name text}`

Здесь *name* — имя класса или метода, к которому добавляется гиперссылка; *text* — отображаемая строка.

## Тег `@param`

Тег `@param` документирует параметр метода. Он имеет следующий синтаксис:

`@param parameter-name explanation`

Здесь *parameter-name* определяет имя параметра метода. Значение этого параметра описывается в *explanation*. Тег `@param` можно использовать только в документации для метода.

## Тег `@return`

Тег `@return` описывает возвращаемое значение метода. Он имеет следующий синтаксис:

`@return explanation`

Здесь `explanation` описывает тип и значение, возвращаемые методом. Тег `@return` можно использовать только в документации метода.

## Тег `@see`

Тег `@see` обеспечивает ссылку к дополнительной информации. Он обычно использует следующие формы:

```
@see anchor
@see pkg.class#member text
```

В первой форме `anchor` — гиперссылка к абсолютному или относительному URL-адресу. Во второй форме `pkg.class#member` определяет имя элемента, а `text` — текст, отображаемый для этого элемента. Текстовый параметр — необязательный и если не используется, то отображается элемент, специфицированный в `pkg.class#member`. Имя члена также является необязательным. Таким образом, вы можете определить ссылку на пакет, класс или интерфейс в дополнение к ссылке на определенный метод или поле. Имя может быть квалифицировано полностью или частично. Однако точка, которая предшествует имени члена (если она существует), должна быть заменена хэш-символом (#).

## Тег `@serial`

Тег `@serial` определяет комментарий для умалчивающего поля сериализации (serializable field). Он имеет следующий синтаксис:

```
@serial description
```

Здесь `description` — комментарий для того поля.

## Тег `@serialData`

Тег `@serialData` документирует данные, записанные методами `writeObject()` или `writeExternal()`. Он имеет следующий синтаксис:

```
@serialData description
```

Здесь `description` — комментарий для этих данных.

## Тег `@serialField`

Тег `@serialField` обеспечивает комментарии для компонента `ObjectStreamField`. Он имеет следующий синтаксис:

```
@serialField description
```

Здесь `description` — комментарий для того поля.

## Тег @since

Тег `@since` заявляет, что класс или член был введен в определенном выпуске (системы Java). Он имеет следующий синтаксис:

```
@since release
```

Здесь `release` — строка, которая определяет выпуск (релиз) или версию, в которой это свойство стало доступным. Тег `@since` можно использовать в документации для переменных, методов и классов.

## Тег @throws

Тег `@throws` имеет такое же значение, как тег `@exception`.

## Тег @version

Тег `@version` определяет версию класса. Он имеет следующий синтаксис:

```
@version info
```

Здесь `info` — строка, которая содержит информацию о версии (обычно номер версии), например, 2.2. Тег `@version` можно использовать только в документации для класса. Для того чтобы поле `@version` было включено в документацию HTML, вам может понадобиться определить опцию `-version` при выполнении `javadoc`.

## Общая форма документационного комментария

После начальных символов `/**`, первая строка или строки становятся главным описанием вашего класса, переменной или метода. После этого вы можете включать один или несколько различных `@`-тегов. Каждый `@`-тег должен начинаться в начале новой строки или следовать за знаком `*`, указанным там же. Множественные теги того же типа должны быть сгруппированы вместе. Например, если вы имеете три тега `@see`, помешайте их один за другим.

Пример документационного комментария для класса:

```
/**  
 * Этот класс рисует столбиковую диаграмму.  
 * @author Herbert Schildt  
 * @version 3.2  
 */
```

## Что выводит *javadoc*

Программа *javadoc* берет в качестве входных данных ваш исходный файл Java-программы и выводит несколько HTML-файлов, которые содержат документацию программы. Информация о каждом классе будет в его собственном HTML-файле. Программа *javadoc* выведет также индекс и дерево иерархий. Могут быть сгенерированы и другие HTML-файлы. Так как различные реализации *javadoc* могут работать по-разному, вам нужно будет проверить инструкции, сопровождающие вашу систему разработки Java-программ (для выяснения специфических подробностей данной версии).

## Пример использования документационных комментариев

Ниже следует типовая программа, которая использует документационные комментарии. Обратите внимание, что любой комментарий непосредственно предшествует элементу, который он описывает. После обработки с помощью *javadoc*, документация относительно класса *SquareNum* будет найдена в файле *SquareNum.html*.

```
import java.io.*;  
  
/**  
 * Этот класс демонстрирует документационные комментарии.  
 * @author Herbert Schildt  
 * @version 1.2  
 */  
public class SquareNum {  
    /**  
     * Данный метод возвращает квадрат num.  
     * Это многострочное описание. Вы можете увидеть  
     * столько строк, сколько пожелаете.  
     * @param num The value to be squared.  
     * @return num squared.  
     */  
    public double square(double num) {  
        return num * num;  
    }  
  
    /**  
     * Этот метод вводит число от пользователя.  
     * @return The value input as a double.  
     * @exception IOException On input error.  
     * @see IOException  
     */
```

```
public double getNumber() throws IOException {
    // создание a BufferedReader, использующего System.in
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader inData = new BufferedReader(isr);
    String str;

    str = inData.readLine();
    return (new Double(str)).doubleValue();
}

/**
 * Метод демонстрирует square().
 * @param args Unused.
 * @return Nothing.
 * @exception IOException On input error.
 * @see IOException
 */

public static void main(String args[])
    throws IOException
{
    SquareNum ob = new SquareNum();
    double val;

    System.out.println("Введите значение для вычисления его квадрата: ");
    val = ob.getNumber();
    val = ob.square(val);

    System.out.println("Квадрат числа равен " + val);
}
}
```

# Предметный указатель

## A

Accessibility API 853  
Adapter classes 612, 636  
API (Applicatin Programming Interface) 559, 783  
API ядро 783, 301  
appletviewer (программа просмотра апллетов) 588  
AWT (Abstract Window Toolkit) 301, 316, 588

## B

BDK (Bean Developer Kit) 804, 806  
Bean-компонент 804  
big-endian (формат коротких целых чисел) 55

## C

CGI (Common Gateway Interface) 855  
Class members (члены класса) 134  
collections framework (строктура коллекций) 413  
collection-view (представление в виде коллекции) 415, 438  
Convenience routine (подпрограмма удобств) 570  
Cookie-элементы 874  
CORBA (Common Object Request Broker Architecture) 27  
CPU (Central Processor Unit) 20

## D

Daemon thread ("демонический" поток) 401  
Delegation event model (модель делегирования событий) 613  
Default  
    access level 173  
    constructor 150  
    package 220  
DLL (Dynamic Link Library) 324  
DNS (Domain Naming Service), служба доменных имен 552  
Drag-and-Drop API 853

## E

Endianness 55  
escape-последовательности символов, таблица 62  
Event loop with polling 264

## F

Factory methods (производственные методы) 554  
Finalization, завершение работы с объектом 154  
firewall (компьютер межсетевой защиты) 967

**G**

GMT (Greenwich Mean Time) 490  
 GUI (Graphical User Interface — графический интерфейс пользователя) 25, 642

**H**

HSB (Hue-Saturation-Brightness — цветовая модель "тон-насыщенность-яркость") 665  
 HTML (HyperText Markup Language)

## теги

<applet> 28, 317, 603, 604  
 <img> 604  
 <param> 604

файл 317, 589

HTTP (HyperText Transport Protocol) 854

**I**

IDL (Interface Definition Language) 27

image-based menu 947

instance variables (переменные экземпляра) 134

IP (Internet Protocol) 549

ISO (International Standardization Organization) 493

**J**

JAR (Java ARchive) 26, 809

Java

Internet 16, 17

JAR-файл (архивный файл Java) 809

апплет Java 18, 315

байт-код (bytecode) 19

библиотеки

Java 2D 26

доступности (Accessibility library) 26

ввод/вывод

консольный (текстовый) 301

оконный (графический) 301

версия 1.0 (исходная) 24

версия 1.1 24

список добавлений 25

версия 2 24

список добавлений 26

Всемирная Паутина (WWW) 16  
 единица компиляции (compilation unit) 38

исключенные свойства C++ 887

исполнительная (run-time)

система 20

история создания 15

коллекции 26

межсетевая защита 19

мобильность (переносимость) программ 19

независимость от платформ 17

новые свойства (по сравнению с C++) 889

приложение Java 18, 315

родной (native) код 20

свойства, различающиеся с C++ 890

связь с языками С и C++ 11

список базовых терминов 21

строгая типизация 52

файл(ы)

исходный (.java) 37

откомпилированный (.class) 38, 39

цели разработки 11

язык свободной формы 48

языки-предшественники 11, 13

Java Beans 25, 787

API 812

Bean-компоненты

ActiveX 828

булевы свойства 813

индексированные свойства 813

конфигуратор (customizer) 824

ограниченные (constrained)

свойства 823

простые свойства 812

связанные свойства (bound properties) 819

сохраняемость (persistence) 823

инструменты

Bridge for ActiveX (мост

для ActiveX) 828

Java Beans Migration Assistant for ActiveX 828

- Java Beans (*прод.*)  
 интерфейс BeanInfo 821  
   getEventSetDescriptors() 821  
   getMethodDescriptors() 821  
   getPropertyDescriptors() 821  
 интроспекция 812  
 класс SimpleBeanInfo 821  
 проектные шаблоны (design patterns) 812
- Java I/O system 302, 501
- Java Security Manager 855
- Java 2D API 853
- java.awt.event (пакет) 612, 637
- java.lang (пакет)  
 интерфейсы  
   Cloneable 389  
   Comparable 410  
   Runnable 399
- классы  
 Class 391–393, 787–789  
 ClassLoader 394, 395  
 Compiler 399  
 Component 589, 632, 633  
 Container 589  
 InheritableThreadLocal 407  
 Math 65, 395  
 Modifier 789  
 Object 217, 218, 388, 389  
 Panel 589  
 Package 408, 409  
 Process 379  
 Random 240, 398  
 Runtime 379–381  
 RuntimePermission 409  
 SecurityManager 410  
 System 384–386  
 Thread 399–402  
 ThreadGroup 402–404  
 Throwable 409  
 Void 378
- оболочки  
 Boolean 378  
 Byte 366, 367  
 Character 374–376  
 Double 360, 362–364  
 Float 360–362
- Integer 369–371  
 Long 371–373  
 Short 367–369
- java.lang.reflect (пакет ядра API)  
 Conctructor 787  
 Field 787  
 Method 787
- java.lang.rmi (пакет ядра API)  
 Naming 792, 793
- java.util  
 интерфейсы  
   Cloneable 479  
   Collection 416  
   Comparable 482  
   Comparator 445  
   Enumeration 477  
   Iterator 431  
   List 419  
   ListIterator 431  
   Map 437  
   Map.Entry 439, 440  
   Observer 496, 497  
   Set 421  
   SortedMap 439  
   SortedSet 421
- классы  
 AbstractMap 441  
 Arrays 452  
 BitSet 479  
 Calendar 484  
 Collections 448  
 Date 482  
 DateFormat 796–798  
 EventListener 413  
 EventObject 413  
 GregorianCalendar 488  
 HashMap 441  
 ListResourceBundle 413  
 Locale 492, 493, 796  
 Observable 496  
 PropertyPermission 413  
 PropertyResourceBundle 413  
 Random 493  
 ResourceBundle 413  
 SimpleDateFormat 798  
 SimpleTimeZone 491

**java.util (прод.)**  
  кlasses (прод.)  
    StringTokenizer 477, 478  
    TimeZone 490  
    TreeMap 443  
    WeakHashMap 441  
    Dictionary (наследованный) 465  
    Hashtable (наследованный) 466, 467  
    Properties (наследованный) 470  
    Stack (наследованный) 463  
    Vector (наследованный) 459  
  кlasses коллекций 422  
    ArrayList 423  
    HashSet 428  
    LinkedList 427  
    TreeSet 430  
  методы (для получения Properties-объектов среди)  
    System.getProperties() 470  
  методы (для работы с ArrayList)  
    ensureCapacity() 425  
    toArray() 426  
    toString() 425  
    trimToSize() 425  
  методы (для работы с Arrays)  
    AsList() 453  
    binarySearch() 453  
    equals() 453  
    fill() 454  
    sort() 454  
  методы (для работы с LinkedList)  
    addFirst() 427  
    addLast() 427  
    removeFirst() 427  
    removeLast() 427  
  методы (для работы с коллекциями)  
    add() 418  
    addAll() 418  
    clear() 418  
    contains() 418  
    containsAll() 418  
    equals() 419  
    isEmpty() 418  
    iterator() 419, 432  
    remove() 418  
    removeAll() 418  
    retainAll() 418  
    size() 418  
    subList() 421  
    toArray() 418  
  методы (для работы с сортированными наборами)  
    first() 421  
    headSet() 421  
    last() 421  
    subSet() 421  
    tailSet() 421  
  методы (для работы со списками)  
    get() 419  
    indexOf() 419  
    lastIndexOf() 419  
    set() 419  
  методы (интерфейса Comparator)  
    compare() 445  
    equals() 445  
  методы (интерфейса Map)  
    entrySet() 439  
    keySet() 438  
    values() 438  
  методы (интерфейса SortedMap)  
    firstKey() 439  
    headMap() 439  
    lastKey() 439  
    subMap() 439  
  методы (интерфейса Map)  
    entrySet() 438  
  методы (класса Collections)  
    synchronizedList() 451  
    synchronizedSet() 451  
  методы (наследованного интерфейса Enumeration)  
    hasMoreElements() 457  
    nextElement() 457  
JDBC (Java Database Connectivity) 25, 886  
JDBC API 886  
JDK (Java Developer's Kit) 27, 37, 588, 806  
  appletviewer (программа просмотра апплетов) 317  
утилиты  
  jar (генерация JAR-файлов) 810  
  java (запуск приложений, интерпретатор) 27, 38

**JDK (прод.)****утилиты (прод.)**

- javac (компилятор) 27, 38
- javadoc (создание документации) 27, 1027
- javah.exe (построение .h файлов) 324

JFC (Java Foundation Class) 853

JIT (Just In Time) 20, 27

JNI (Java Native Interface) 25, 323

JRE (Java Runtime Environment) 27

JSdk (Java Servlet Development Kit)  
854, 856

JVM (Java Virtual Machine) 19, 27

**L**Listener (блок прослушивания  
событий) 613**M**

Manifest file 810

MIME (Multipurpose Internet Mail  
Extensions) 854

Multicasting 614

Mutex (взаимоисключающая  
блокировка) 281**N**

Native-методы 885

NCSA 744

**O**OOP (Object-Oriented Programming)  
14, 29

ORB (Object Request Broker) 27

**P**

Pluggable look-and-feel 853

Preemptive multitasking (упреждающая  
многозадачность) 265

Protection domain (домен защиты) 393

Proxy (сетевой посредник) 968

**R**RGB (Red-Green-Blue — цветовая  
модель "красный-зелёный-синий")

666

RMI 24, 25, 783

rmiregistry 795

run-time-состояние 391

динамическая загрузка классов 795

интерфейс Remote 792

компилятор RMI 794

объекты

заглушка (stub) 793

скелет (skelton) 794

простое приложение клиент-сервер  
791

сериализация 539, 794

**S**Scope (область видимости  
идентификаторов) 65

Security manager 384

Servlet (сервлет) 854

Servlet API 855, 858

set-view (представление в виде набора)  
441, 469

skelton (скелет), RMI-объект 794

subclasser responsibility method (метод,  
находящийся на ответственности  
подкласса) 213

Swing 26

API 829

Swing-компоненты 846, 850

интерфейсы

Icon 831

MutableTreeNode 847

SwingConstants 831

TreeExpansionListener 848

TreeNode 847

ScrollPaneConstants 844

классы

AbstractButton 834

Container, метод Add() 831

DefaultMutableTreeNode 847

ImageIcon 831

JApplet 830

**Swing (прод.)****классы (прод.)**

JButton 834

JCheckBox 836

JComboBox 840

JComponent 831

JLabel 831

JRadioButton 838

JScrollPane 844

JTabbedPane 842

JTable 850

JTextFieldComponent 833

JTextField 833

JTree 846

TreeExpansionEvent 848

TreePath 847

**компоненты**

плагин-свойство (pluggable look-and-feel) 852

подсказки кнопочных команд (tooltips) 852

прогресс-полоски (progress bars) 852

**пакеты**

javax.swing 830

javax.swing.event 848

javax.swing.tree 830

**панели**

корневая (root pane) 830

прозрачная (glass pane) 830

прокрутки (scroll pane) 844

со вкладками (tabbed pane) 842

содержания (content pane) 830

таблица классов 829

System.in.read() 119

**T**

TCP (Transmission Control Protocol)

549

Thread-safe (поточно-безопасный) 451

Type wrappers (оболочки простых

типов) 359

**U**

UDP (User Datagram Protocol) 549

unicasting, 614

Unicode 25, 58, 302

URI (User Resource Identifier) 870

URL (Uniform Resource Locator) 559, 856

UTC (Coordinated Universal Time) 490

**W**

Web 28, 559

whitespace (пробельный символ) 48

**X**

XOR-режим рисования 668

**A**

Абстрактный класс 899  
 Абстрактный метод 900  
 Автоматическое преобразование типов 68, 160  
     расширяющее (widening conversion) 68  
     сужающее (narrowing conversion) 69  
 Алгоритмы (коллекций) 414  
 Алгоритмы синхронизации (коллекций) 451  
 Апплет(ы)  
     HTML-тег <applet> 589  
     базовые методы работы с апплетами 594  
         destroy() 595  
         init() 594  
         paint() 595  
         start() 594  
         stop() 595  
         update() 595  
     интерфейсы  
         AppletContext 609  
         AppletStub 611  
         AudioClip 611  
     класс Applet  
         методы 608—609  
     консольный вывод 611  
     ненадежные 885  
     пересылка параметров в 605  
     простые методы отображения 596  
         drawString() 596  
         getBackground() 597  
         getForeground() 597  
         getGraphics() 599  
         repaint() 598  
         setBackground() 596  
         setForeground() 596  
         цветовые константы 596  
     с бегущим заголовком 599, 606  
     скелетная схема 593  
 Аргументы  
     командной строки 188  
     метода 142, 146

**Б**

Библиотеки классов 51  
 Блоки кода (кодовые блоки) 46  
 Браузер Web 18  
  
**В**  
 Ввод/вывод  
     байтовые потоки 502  
         буферизованные 518  
     Externalizable (интерфейс) 540  
     FileFilter (интерфейс) 509  
     FilenameFilter (интерфейс) 507  
     ObjectInput (интерфейс) 542  
     ObjectOutput (интерфейс) 540  
     Serializable (интерфейс) 540  
     ArrayOutputStream (класс) 516  
     BufferedInputStream (класс) 518  
     BufferedOutputStream (класс) 518, 520  
     ByteArrayInputStream (класс) 515  
     File (класс) 503  
     FileInputStream (класс) 312, 511  
     FileOutputStream (класс) 312, 513  
     FilterInputStream (класс) 518  
     FilterOutputStream (класс) 518  
     InputStream (класс) 510  
     ObjectInputStream (класс) 543  
     ObjectInputStream.GetField (класс) 502  
     ObjectOutput.Stream.GetField (класс)  
         502  
     ObjectOutputStream (класс) 541  
     OnlyExt (класс) 507  
     OutputStream (класс) 510  
     OutputStream (класс) 303  
     PrintStream (класс) 309  
     PrintStream (класс) 523  
     PushbackInputStream (класс) 520  
     PushbackInputStream (класс) 518  
     RandomAccessFile (класс) 524  
     SequenceInputStream (класс) 522  
     System (класс) 303  
 исключения  
     FileNotFoundException 312, 527  
     IOException 307, 513, 528  
     SecurityException 513, 528

**Ввод/вывод (прод.)****поточный**

- байтовый поток 302
- буферизованный 306
- поток (stream) 302, 501
- поток ввода 302
- поток вывода 302
- символьный поток 302, 525

**сериализация объектов** 502**символьные потоки** 502

- `BufferedReader` (класс) 307, 308
  - `BufferedReader` (класс) 531
  - `BufferWriter` (класс) 533
  - `CharArrayReader` (класс) 529
  - `CharArrayWriter` (класс) 530
  - `FileReader` (класс) 527
  - `FileWriter` (класс) 528
  - `InputStreamReader` (класс) 306
  - `PrintWriter` (класс) 534
  - `PushbackReader` (класс) 533
  - `Reader` (класс) 525
  - `StreamTokenizer` (класс) 537
  - `Writer` (класс) 303, 526
- таблица классов символьного ввода/вывода 303

**Вызов удаленных методов (RMI)** 791**Вызовы методов**

- встроенные (inline) 217

**Выражения с авторасширением****типов** 70

- правила 71

**Г****Графический контекст** 658**Д****"Демонический" (demon) процесс**

- 1017

**Десериализация** 794, 539**Динамическая диспетчеризация**

- методов 208

**Домен защиты** 393**Е****Емкость (размер) массива** 424**Емкость загрузки хэш-набора** 429**Естественное упорядочение объектов** 410**З****Завершение** 154**И****Идентификаторы Java** 48**Интернационализация** 492**Интерфейс (interface)** 899

- `AWT`, `LayoutManager` 712

- `java.io.Serializable` 824

альтернатива множественного наследования 231

определение 231

реализация 232

**Исключения** 243

- `ArrayIndexOutOfBoundsException` 455

- `ArrayStoreException` 417, 418

- `ClassCastException` 362, 377, 416—455

- `ClassFormatError` 394

- `ClassNotFoundException` 392, 395

- `CloneNotSupportedException` 389

- `EmptyStackException` 463

- `IllegalAccessException` 393

- `IllegalArgumentException` 454, 455

- `IllegalStateException` 432

- `InstantiationException` 393

- `InterruptedException` 389, 401

- `NoSuchElementException` 421—439

- `NullPointerException` 421, 437, 439

- `NumberFormatException` 362

- `SecurityException` 392

- `SecurityException` 379, 384

- `UnsupportedOperationException` 416, 419, 451

контролируемые 258

- `ClassNotFoundException` 259

- `CloneNotSupportedException` 259

- `IllegalAccessException` 259

**Исключения (прод.)****контролируемые (прод.)**

InstantiationException 259

InterruptedException 259

NoSuchFieldException 259

NoSuchMethodException 259

таблица 258

**неконтролируемые 258**

ArithmetiException 258

ArrayIndexOutOfBoundsException  
258

ArrayStoreException 259

ClassCastException 259

IllegalArgumentExcepion 259

IllegalMonitorStateException 259

IllegalStateException 259

IllegalThreadStateException 259

IndexOutOfBoundsException 259

NegativeArraySizeException 259

NullPointerException 259

NumberFormatException 259

SecurityException 259

StringIndexOutOfBoundsException  
259UnsupportedOperationException  
259

таблица 258

**обработчик по умолчанию 245****объект(ы) 243****собственные методы 260**

String getLocalizedMessage() 260

String getMessage() 260

String toString() 260

Throwable fillInStackTrace() 260

void printStackTrace() 260

void printStackTrace(PrintStream  
stream) 260void printStackTrace(PrintWriter  
stream) 260**средства работы**

catch-блок 246

finally-блок 256

throw-оператор 253

throws-методы 255

try-блок 246

вложенные try-блоки 251

**множественные catch-блоки 249****общий формат 244****типы (классы)**

Error 245

Exception IllegalAccessException  
255

Exception 244

ArithmetiException 245

RuntimeException 244

RuntimeException

ArrayIndexOutOfBoundsException  
249

NullPointerException 254

Throwable 244

Итератор коллекции 414, 419, 431

**K****Карта отображений (map) 415, 436****Классы 133**

Applet 316, 588

AppletContext (интерфейс) 588

AppletStub (интерфейс) 588

AudioClip (интерфейс) 588

таблица методов 589

AppletContext 588

AppletStub 588

AudioClip 588

таблица методов 589

AWT 643

AWTEvent 736

Button 690

Canvas (окно) 648

Checkbox 693

CheckboxGroup 696

CheckboxMenuItem 724, 725

Choice 697, 698

Color 665, 666

Component 636—673, 736

Container 646, 687, 715—720

Dialog (модальный, немодальный) 729

Dimension 648, 664

Dimention 787

FileDialog 734, 735

Font 670

- Классы (*прод.*)  
AWT (*прод.*)  
    FontMetrics 676–678  
    Frame 647–649  
    Graphics 660–668  
    GraphicsEnvironment 671, 672  
    ItemEvent 725  
    Label 688  
    List 700, 701  
    Menu 723  
    MenuBar 723  
    MenuItem 723  
    Panel (окно) 647  
    PopupMenu 729  
    Scrollbar 704, 705  
    TextArea 710  
    TextField 707, 708  
    Window 647, 730  
работа с фреймовыми окнами 648  
режим рисования 668  
таблица 643  
Component 589  
Container 589  
java.util 796  
    java.util.DateFormat 796–798  
    java.util.SimpleDateFormat 798  
LANG  
    Class 787, 789  
    Modifier 789  
Math 65  
Object 217, 218  
Panel 589  
public 224  
Random 240  
REFLECT  
    Conctructor 787  
    Field 787  
    Method 787  
RMI, Naming 792, 793  
String 80, 185, 186, 331  
StringBuffer 185, 331  
System 51, 305  
абстрактный 213, 235  
вложенные 181  
    нестатические 182  
    статические 182  
внутренние 25, 182, 638  
    анонимные 185, 640  
иерархия 33  
    подклассы 33  
    суперклассы 33  
интерфейсы 230  
как новый тип данных 133  
как шаблон для объектов 133  
наследованные 457  
определение 32  
подкласс  
    определение 189  
просмотра (peer classes) 185  
реализация интерфейсов 232  
сетевые 548  
суперкласс  
    определение 189  
члены 32  
    методы 32  
    переменные (переменные экземпляра) 32  
Клон (clone) 389  
Ключевые слова 50  
    abstract 213, 235  
    class 39, 134  
    extends 189, 241  
    final 179, 216  
    interface 219, 230  
    native 323  
    static 40, 177  
    strictfp 322  
    super 177, 196  
    this 152  
    transient 319, 824  
    volatile 319  
для работы с исключениями  
    catch 244  
    finally 256  
    throw 244  
    throws 244, 255  
    try 244  
спецификаторы доступа  
    private 40, 173, 224  
    protected 173, 224  
    public 40, 173, 224  
Кодовые блоки (блоки кода) 46

**Коллекция (объектов)** 412  
**изменяемая (modifiable)** 416  
**неизменяемая (unmodifiable)** 416  
**несинхронизированная** 451  
**синхронизированная (поточно-безопасная)** 451

**Комментарий** 39  
**документационный** 49  
**использование** 1027  
**многострочный** 39  
**однострочный** 40

**Компаратор (comparator)** 444

**Компоненты Java Beans** 787

**Константы** 49

**Конструктор (constructor)** 139, 148

**super()** 196, 204

**по умолчанию** 139, 150

**Контейнер C++** 415

**Коэффициент заполнения хэш-набора (fillRatio)** 429

## Л

**Лексема (token)** 477, 537

**Лексический анализатор (сканер)** 477

**Литералы** 60

**Локализация** 493

## М

**Массив(ы)** 72

**многомерные** 75

**инициализация** 74, 78

**одномерные** 72

**альтернативный формат** 80

**общий формат** 72

**Менеджер безопасности (security manager)** 384, 410

**Менеджеры компоновки (layout manager)** 712

**BorderLayout** 714

**CardLayout** 719, 720

**FlowLayout** 713

**GridLayout** 718

**использование вставок** 716

**Меню изображений**

**(image-based menu)** 947

**Методы** 134

**clone()** 218

**equals()** 218

**finalize()** 154, 218

**getClass()** 218

**hashCode()** 218

**main()** 65

**notify()** 218

**notifyAll()** 218

**toString()** 218

**wait()** 218

**абстрактный** 213

**динамическая диспетчеризация** 208

**заглушка** 591

**как члены класса** 134

**на ответственности подкласса** 213

**перегруженные** 158

**переопределенные (overrided)** 206

**производственные (factory)** 554, 925

**getAllByName()** 554

**getLocalHost()** 554

**рекурсивные** 170

**удобств (convenience method)** 998

**makeMimeHeader()** 570

**toBytes()** 570

**writeTo()** 517

## Многозадачность

**блокировка задач** 291

**основанная на потоках** 263

**поток (как единица**

**диспетчеризации)** 263

**основанная на процессах** 263

**программа (как единица**

**диспетчеризации)** 263

**упреждающая** 265

## Многопоточность

### Модификатор

**abstract** 213

**transient** 319

**volatile** 319

## Н

**Начальное число (псевдослучайного генератора)** 494

## О

Облегченные (Swing) компоненты 829

Оболочки простых типов (type wrappers) 359

Обработка событий

  блок(и) прослушивания 613, 625

    метод отказа от регистрации,  
    формат 614

    методы регистрации 614, 632

  интерфейсы прослушивания 626

    ActionListener 627, 690

    AdjustmentListener 627

    ComponentListener 627

    ContainerListener 627

    FocusListener 627

    ItemListenere 628, 694, 698

    KeyListener 628

    MouseListener 628

    MouseMotionListener 628

    TextListener

    WindowListener 629

  таблица 626

классы-адаптеры 612, 636

  внутренние, анонимные 612

  таблица 637

модель делегирования событий

  613, 625

модель расширения

  AWT-компонентов 736

  таблица методов 737

мультивещание (multicasting)

  событий 614

обработчики

  keyPressed() 633

  keyReleased() 633

  keyTyped() 633

пакет поддержки

  java.awt.event 612, 637

унивещание (unicasting) событий 614

Обработка строк 331

  извлечение символов 338

классы

  Object 337

  String 331—350

  StringBuffer 331—357

конкатенация строк 335

  с другими типами данных 336

поиск строк 344

строковый литерал 335

Объект(ы) 133

  возврат методами 169

  как экземпляр класса 32, 133

  передача методу по ссылке 168

  состав 32

ООП (Объектно-Ориентированное

  Программирование)

  базовые принципы

  наследование 189

Оператор(ы)

  import 228

  null (пустой) оператор 116

  package 220

  управления 43

    выбора 106—109

    перехода 126—132

    повторения (циклов) 115, 117, 120

Операции

  () — круглые скобки 104

  — точка (dot) 104

  [ ] — квадратные скобки 104

  new — распределение памяти

  для объекта 138

арифметические

  таблица 82

беззнакового сдвига 96

дополнение до двух 89

дополнение до единицы 89

логические

  instanceof 82

  таблица 100

отношений

  таблица 98

поразрядные

  таблица 88

поразрядные логические

  таблица 90

постфиксная форма 86

префиксная форма 87

присваивания (назначения) 102

расширение знака 95

старшинство (таблица) 104

укороченные (short-circuit) 102

Отражение (reflection) 411, 783, 786

**П****Пакет(ы)**

AWT 642

- менеджер компоновки (layout manager) 686, 712
- работа с графикой 658
- работа с меню, методы 724
- работа с цветом 665
- работа со шрифтами 670
- строка меню (menu bar) 686
- элементы управления (controls) 686

java.applet 301, 316, 588

java.awt 588

java.awt.dnd 853

java.awt.event 612, 636

java.beans 824

таблица интерфейсов 824

таблица классов 824

java.io 301, 303, 501

список интерфейсов 502

список классов 501

java.lang 258, 305, 331, 358

Number (класс) 359

список интерфейсов 358

список классов 358

java.lang.ref 410

java.lang.reflect 411, 783

Member (интерфейс) 787

таблица методов 787

java.net (программирование для Internet) 548

java.rmi 783

java.text 783, 796

java.util 412, 477

таблица интерфейсов 413

таблица классов 412

java.util.jar 500

java.util.zip 500

javax.servlet 857, 858

javax.servlet.http 869

иерархия 221

именованные 228

как группа классов 173

определение 220

по умолчанию (default package) 220

типы меню

иерархическое 723

плавающее 723

ядро API (таблица) 783

**Параметр(ы)**

метода 40, 141, 146

**Параметры указателя (pointer parameters)** 892**Перегрузка (overloading)** методов 158**Передача аргумента**

по значению 167

по ссылке 167

**Переменные**

время жизни 65

выражение инициализации 64

динамическая инициализация 64

инициализация 64

массива (array variable) 72

область видимости идентификатора 65

определяемая классом (class scope) 65

определяемая методом (method scope) 65

**окружения**

CLASSPATH 221

определение 41, 63

ссылочные (объекта) 140

управления циклом 120

формат объявления 63

экземпляра

length 180

определение 134

**Переопределение (overriding)**

методов 593

**Песочница (sandbox)** 885**Полиморфизм** 899

перегруженных функций 158

**Потоки** 263

асинхронные 266

выполнения 267

главный 268

группа 269

не синхронизированные

состояние состязаний (гонок) 284

- Потоки (прод.)**
- приоритеты 265
  - синхронизация 281
  - нейвый монитор 286
  - синхронные 266
  - состояния 265
  - текущий 268
- Поточное программирование**
- межпоточные связи
    - использование 287
  - многопоточная многозадачность
    - в Java 264
  - многопоточное
    - Runnable (интерфейс) 267
    - synchronized (ключевое слово) 282
    - Thread (класс), таблица методов 267, 279
    - многопоточность 263
    - монитор (семафор) 266, 281
    - синхронизированные методы 284
    - синхронизированные операторы, блоки 285
    - правила переключения контекста 265
    - работа с приоритетами 278
    - сериализация (преобразование в последовательную форму) 284
  - однопоточное
    - цикл событий с опросом 264
- Представление в виде коллекции (collection-view) 415, 438
- Представление в виде набора (set-view) 441, 469
- Преобразование типов
- усечение (truncation) 69
- Приведение (явное преобразование)
- типов (cast)
  - формат 69
- Пробельные символы (whitespace)
- space, tab, newline 48
- Программа
- как модель, ориентированная на процесс 30
  - как управляемый данными доступ к коду 30
- Программирование**
- компонентное 803
  - объектно-ориентированное (ООП)
    - 14, 29, 30
    - абстракция 30
    - основные механизмы
      - (инкапсуляция, наследование, полиморфизм) 31, 33, 34
    - сущность 31
  - парадигмы
    - объектно-ориентированная 30
    - ориентированная на процессы 30
  - языки
    - процедурные 30
- Прототип функции 899
- Процесс (выполняющаяся программа) 379
- Псевдослучайные числа 493
- P**
- Работа в сети Internet 548
- DNS (Domain Naming Service)
- MIME
- заголовок 564
  - стандарт 564
  - строка User-Agent 573
  - типы 564
- адрес
- IP 549, 552
  - URL 559
  - групповой (multicast) 555
  - доменный адрес 552
  - определение 552
- браузер
- Web 550
- дейтаграммы
- определение 584
- интерфейсы 553
- классы 553
- DatagramPacket 584
  - DatagramSocket 584
  - InetAddress 553–555
  - ServerSocket 556, 562
  - Socket 556, 557
  - URL 559
  - URLConnection 561

**Работа в сети Internet (прод.)**

- клиент 549
- клиент-сервер 549
- масштабирование 554
- межсетевая защита
  - компьютер для (firewall) 967
  - пакеты данных 549
  - порт 550
  - протокол(ы)
    - file 559
    - ftp 559
    - gopher 559
    - HTTP (HyperText Transfer Protocol) 550, 559
    - IP (Internet Protocol) 549
    - TCP (Transmission Control Protocol) 549, 553
    - TCP/IP 549
    - UDP (User Datagram Protocol) 549, 553
    - whois 559
  - WWW (Web-протоколы) 559
    - действиаграммные 584
    - номера портов для разных протоколов 550
  - сервер(ы) 549
  - DNS 556
  - InterNIC 556
  - proxy 551
  - Web 549, 550
  - вычислительные 549
  - дисковые 549
  - кэшированные 555
  - кэширующий proxy HTTP 551, 563
  - однонаправленные (unicast) 792
  - печати 549
  - реплицированные (replicated) 792
  - сетевой посредник (proxy) 968
  - сокет(ы)
    - Berkeley 549
    - TCP/IP (для клиентов и серверов) 556
    - парадигма 548
    - сетевой 549
    - хост-компьютер 553, 554, 968

## Работа с изображениями

- Web-дизайн 744
- двойная буферизация 753
- изображение (как графический объект) 744
- интерфейсы
  - ImageObserver 748
  - ImageProducer 746, 759
- классы
  - Applet 746
  - Component 765
  - FilteredImageSource 764
  - Graphics 746
  - Image 744, 745
  - ImageFilter 764
  - ImageFilter.AreaAveragingScaleFilter 764
  - ImageFilter.CropImageFilter 764, 765
  - ImageFilter.ReplicateScaleFilter 764
  - ImageFilter.RGBImageFilter 764, 767
  - MediaTracker 756
  - MemoryImageSource 759
  - PixelGrabber 762
- анимация ячеек 779
- загрузка объекта изображения 746
- наблюдатель изображения (image observer) 747
- создание объекта изображения 745
- пакеты
  - java.awt 744
  - java.awt.image 744
  - javax.servlet 857—859
  - javax.servlet.http 858, 869
- производители изображений (image producers) 759
  - FilteredImageSource 759
  - MemoryImageSource 759
- форматы графических файлов
  - GIF 745
  - JPEG 745

## Разделители

таблица 49

## Расширение (повышение) типов 55, 71

Рекурсия 170

**C**

- Сборка "мусора" 153  
 Связывание вызовов  
     позднее 217  
     раннее 217  
 Сеанс (session) 884  
 Сервлет(ы)  
     HTTP-заголовок 854  
     HTTP-запрос 854  
     HTTP-запросы  
         GET 878  
         POST 878, 880  
         строка запроса (query string) 880  
 HTTP-ответ 854  
 MIME-типы  
     text/html 854  
     text/plain 854  
 Servlet API 859  
 инструменты JSDK 856  
 интерфейсы  
     HttpServlet, таблица методов 876  
     HttpServletRequest, таблица  
         методов 870  
     HttpServletResponse, таблица  
         методов 871  
     HttpSession, таблица методов 873  
     HttpSessionBindingListener 874  
     HttpSessionContext 874  
     Servlet, таблица методов 860, 864  
     ServletConfig, таблица методов  
         861, 864  
     ServletContext, таблица методов  
         861  
     ServletRequest, таблица методов  
         862  
     ServletResponse, таблица методов  
         863  
 классы  
     GenericServlet 855—864  
     HttpServletResponse 876  
     HttpSessionBindingEvent 877  
     HttpUtils, таблица методов 878  
     ServletException 865  
     ServletInputStream 864  
     ServletOutputStream 865  
     ServletRequest 865  
     UnavailableException 865  
 надежные 885  
 ненадежные 885  
 работа с сессиями  
     класс HttpSession 884  
     метод getSession() 884  
     метод getValue() 884  
     метод putValue() 884  
     метод removeValue() 884  
 сессии  
     создание 884  
     утилита servletrunner 856  
 сериализация 794  
 Сериализация (serialization) 539  
 Сигнатура  
     типов (type signature) 205  
 Синтаксический анализ  
     пробельные (whitespace) символы  
         477  
     разделители 477  
 Синтаксический анализ (parsing) 477  
 Сканер (лексический анализатор) 477  
 События  
     источники событий 614  
     таблица 625  
     классы событий  
         ActionEvent 617, 690, 701  
         AdjustmentEvent 618, 705  
         ComponentEvent 618, 619  
         ContainerEvent 619  
         EventObject (суперкласс) 615, 616  
         FocusEvent 620  
         InputEvent 620, 621  
         ItemEvent 621, 701, 702  
         KeyEvent 621, 622  
         MouseEvent 622, 623  
         TextEvent 624  
         WindowEvent 624  
     иерархия 615  
     таблица конструкторов и методов  
         616  
     определение 613  
 Сокет (socket)-сетевое соединение 302  
 Спецификаторы доступа 173, 224  
     private 173, 224  
     protected 173, 224  
     public 40, 173, 224

**Ссылки**

интерфейсные 233

мягкие 411

слабые 411

phantomные 411

**Статические члены класса**

блоки 177

методы 177

переменные 177

**Строки**

как объекты 185

массивы строк 187

**Структура коллекций (collections framework)** 413**Т**

Типы данных, простые 53

**У**

Управление доступом 173

Уровень доступа по умолчанию 173

**Ф****Файл**

CAB (file cabinet) 842

JAR (архив Java) 809

описания (manifest file) 810

**Фрейм (окно класса Frame)** 648

Функция удобств setDim 148

**Х****Хост-приложение** 647

Хэширование (рандомизация) 429

Хэш-код 361, 429

**Ц****Цветовые модели**

HSB 665

RGB 666

**Цифровые подписи** 810**Ч****Чистая виртуальная функция** 899

# Содержание

Об авторах .....	1
Благодарности .....	3
Предисловие .....	5
<b>ЧАСТЬ I. ЯЗЫК JAVA.....</b>	<b>9</b>
<b>Глава 1. Генезис Java .....</b>	<b>11</b>
Происхождение Java .....	11
Рождение современного программирования: С .....	12
Потребность в C++ .....	13
Этап становления Java .....	15
Создание Java .....	15
Почему Java важен для Internet .....	17
Java-апплеты и приложения .....	18
Защита .....	18
Мобильность .....	19
Волшебство Java: байт-код .....	19
Базовые термины Java .....	21
Простой .....	21
Объектно-ориентированный .....	22
Устойчивый .....	22
Многопоточный .....	23
Архитектурно-независимый .....	23
Интерпретируемость и высокая эффективность .....	23
Распределенный .....	24
Динамический .....	24
Продолжение революции .....	24
Свойства, добавленные версией 1.1 .....	25
Свойства, исключенные из версии 1.1 .....	25
Свойства, добавленные версией 2 .....	26
Свойства, исключенные из версии 2 .....	27
Java — не расширение HTML .....	27

<b>Глава 2. Обзор языка Java .....</b>	<b>29</b>
Объектно-ориентированное программирование .....	29
Две парадигмы программирования .....	29
Абстракция.....	30
Три принципа ООП.....	31
Первая простая программа .....	37
Ввод программы .....	37
Компиляция программы .....	38
Подробный взгляд на первую программу.....	39
Вторая короткая программа .....	41
Два оператора управления .....	43
Оператор <i>if</i> .....	43
Цикл <i>for</i> .....	45
Использование блоков кода.....	46
Лексические вопросы .....	48
Пробельные символы .....	48
Идентификаторы.....	48
Константы.....	49
Комментарии.....	49
Разделители.....	49
Ключевые слова языка Java.....	50
Библиотеки классов языка Java .....	51
<b>Глава 3. Типы данных, переменные и массивы .....</b>	<b>52</b>
Java — язык со строгой типизацией .....	52
Простые типы.....	53
Целые типы .....	53
Тип <i>byte</i> .....	54
Тип <i>short</i> .....	55
Тип <i>int</i> .....	55
Тип <i>long</i> .....	56
Типы с плавающей точкой .....	56
Тип <i>float</i> .....	57
Тип <i>double</i> .....	57
Символьный тип ( <i>char</i> ) .....	58
Булевский тип ( <i>boolean</i> ).....	59
Подробнее о литералах .....	60
Целочисленные литералы.....	60
Литералы с плавающей точкой.....	61
Булевые литералы .....	62
Символьные литералы .....	62
Строковые литералы .....	63
Переменные.....	63
Обявление переменной.....	63
Динамическая инициализация .....	64
Область действия и время жизни переменных.....	65
Преобразование и приведение типов .....	68
Автоматическое преобразование типов в Java .....	68
Приведение несовместимых типов .....	69

Автоматическое расширение типа в выражениях.....	70
Правила расширения типов.....	71
Массивы.....	72
Одномерные массивы.....	72
Многомерные массивы.....	75
Альтернативный синтаксис объявления массива .....	80
Несколько слов относительно строк.....	80
Замечание для программистов С/С++ по поводу указателей .....	81
<b>Глава 4. Операции.....</b>	<b>82</b>
Арифметические операции .....	82
Основные арифметические операции.....	83
Деление по модулю.....	84
Арифметические операции присваивания .....	85
Инкремент и декремент .....	86
Поразрядные операции .....	88
Поразрядные логические операции .....	90
Левый сдвиг .....	92
Правый сдвиг.....	94
Правый сдвиг без знака .....	96
Поразрядная операция присваивания.....	97
Операции отношений.....	98
Операции булевой логики .....	100
Короткие логические операции.....	102
Операция присваивания.....	102
Условная операция .....	103
Старшинство операций .....	104
Использование круглых скобок.....	105
<b>Глава 5. Управляющие операторы .....</b>	<b>106</b>
Операторы выбора Java .....	106
Оператор <i>if</i> .....	106
Оператор <i>switch</i> .....	110
Операторы цикла .....	115
Оператор цикла <i>while</i> .....	115
Оператор цикла <i>do while</i> .....	117
Оператор цикла <i>for</i> .....	120
Вложенные циклы.....	125
Операторы перехода .....	125
Использование оператора <i>break</i> .....	126
Использование оператора <i>continue</i> .....	130
Оператор <i>return</i> .....	132
<b>Глава 6. Введение в классы .....</b>	<b>133</b>
Основы классов.....	133
Общая форма класса.....	134
Простой класс .....	135

Объявление объектов.....	138
Операция <i>new</i> .....	139
Назначение ссылочных переменных объекта .....	140
Представление методов .....	141
Добавление метода к классу <i>Box</i> .....	142
Возврат значений .....	144
Добавление метода с параметрами.....	146
Конструкторы.....	148
Параметризованные конструкторы .....	150
Ключевое слово <i>this</i> .....	152
Скрытие переменной экземпляра .....	152
Сборка "мусора" .....	153
Метод <i>finalize()</i> .....	153
Класс <i>Stack</i> .....	154
<b>Глава 7. Методы и классы.....</b>	<b>158</b>
Перегрузка методов.....	158
Перегрузка конструкторов.....	162
Использование объектов в качестве параметров .....	164
Передача аргументов .....	167
Возврат объектов.....	169
Рекурсия.....	170
Управление доступом .....	172
Статические элементы.....	177
Спецификатор <i>final</i> .....	179
Ревизия массивов.....	179
Вложенные и внутренние классы .....	181
Класс <i>String</i> .....	185
Использование аргументов командной строки .....	188
<b>Глава 8. Наследование .....</b>	<b>189</b>
Основы наследования.....	189
Доступ к элементам и наследование.....	191
Практический пример .....	192
Переменная суперкласса может ссылаться на объект подкласса .....	195
Использование ключевого слова <i>super</i> .....	196
Вызов конструктора суперкласса с помощью первой формы <i>super</i> .....	196
Использование второй формы <i>super</i> .....	200
Создание многоуровневой иерархии .....	201
Когда вызываются конструкторы.....	204
Переопределение методов.....	205
Динамическая диспетчеризация методов .....	208
Зачем нужны переопределенные методы?.....	210
Применение переопределения методов.....	211
Использование абстрактных классов.....	212
Использование ключевого слова <i>final</i> с наследованием .....	216
Использование <i>final</i> для отказа от переопределения.....	216
Использование <i>final</i> для отмены наследования .....	217
Класс <i>Object</i> .....	217

<b>Глава 9. Пакеты и интерфейсы.....</b>	<b>219</b>
Пакеты.....	220
Определение пакета .....	220
Использование CLASSPATH .....	221
Короткий пример пакета.....	222
Защита доступа.....	223
Пример управления доступом .....	224
Импорт пакетов.....	228
Интерфейсы.....	230
Определение интерфейса .....	231
Реализация интерфейсов .....	232
Применения интерфейсов.....	235
Переменные в интерфейсах .....	239
Расширение интерфейсов .....	241
<b>Глава 10. Обработка исключений .....</b>	<b>243</b>
Основные принципы обработки исключений .....	243
Типы исключений.....	244
Неотловленные исключения.....	245
Использование операторов <i>try</i> и <i>catch</i> .....	246
Отображение описания исключения .....	248
Множественные операторы <i>catch</i> .....	249
Вложенные операторы <i>try</i> .....	251
Оператор <i>throw</i> .....	253
Методы с ключевым словом <i>throws</i> .....	255
Блок <i>finally</i> .....	256
Встроенные исключения Java .....	258
Создание собственных подклассов исключений .....	260
Использование исключений .....	262
<b>Глава 11. Многопоточное программирование .....</b>	<b>263</b>
Поточная модель Java .....	264
Приоритеты потоков .....	265
Синхронизация.....	266
Передача сообщений .....	266
Класс <i>Thread</i> и интерфейс <i>Runnable</i> .....	267
Главный поток.....	268
Создание потока .....	270
Реализация интерфейса <i>Runnable</i> .....	270
Расширение <i>Thread</i> .....	272
Выбор подхода .....	274
Создание множественных потоков .....	274
Использование методов <i>isAlive()</i> и <i>join()</i> .....	276
Приоритеты потоков .....	278
Синхронизация .....	281
Использование синхронизированных методов .....	282
Оператор <i>synchronized</i> .....	284

Межпоточные связи .....	286
Блокировка .....	291
Приостановка, возобновление и остановка потоков .....	294
Приостановка, возобновление и остановка потоков в Java 1.1 и более ранних версиях .....	294
Приостановка, возобновление и остановка потока в Java 2 .....	297
Использование многопоточности .....	299
<b>Глава 12. Ввод/вывод, апплеты и другие темы .....</b>	<b>301</b>
Основы ввода/вывода .....	301
Потоки.....	302
Байтовые и символьные потоки.....	302
Предопределенные потоки.....	305
Чтение консольного ввода .....	306
Чтение символов.....	307
Чтение строк.....	308
Запись консольного вывода.....	309
Класс <i>PrintWriter</i> .....	310
Чтение и запись файлов .....	312
Апплеты. Основы программирования .....	315
Модификаторы <i>transient</i> и <i>volatile</i> .....	319
Использование <i>instanceof</i> .....	319
Ключевое слово <i>strictfp</i> .....	322
Native-методы.....	323
Проблемы native-методов .....	327
<b>ЧАСТЬ II. БИБЛИОТЕКА JAVA .....</b>	<b>329</b>
<b>Глава 13. Обработка строк .....</b>	<b>331</b>
<i>String</i> -конструкторы .....	332
Длина строки .....	334
Специальные строковые операции .....	334
Строковые литералы .....	335
Конкатенация строк .....	335
Конкатенация других типов данных.....	336
Преобразование строк и метод <i>toString()</i> .....	337
Извлечение символов .....	338
Метод <i>charAt()</i> .....	338
Метод <i>getChars()</i> .....	339
Метод <i>getBytes()</i> .....	339
Метод <i>toCharArray()</i> .....	340
Сравнение строк .....	340
Методы <i>equals()</i> и <i>equalsIgnoreCase()</i> .....	340
Метод <i>regionMatches()</i> .....	341
Методы <i>startsWith()</i> и <i>endsWith()</i> .....	341
Сравнение <i>equals()</i> и операции <i>==</i> .....	342
Метод <i>compareTo()</i> .....	343

Поиск строк.....	344
Изменение строки.....	346
Метод <i>substring()</i> .....	346
Метод <i>concat()</i> .....	347
Метод <i>replace()</i> .....	348
Метод <i>trim()</i> .....	348
Преобразование данных, использующее метод <i>valueOf()</i> .....	349
Изменение регистра символов в строке .....	350
Класс <i>StringBuffer</i> .....	350
Конструкторы <i>StringBuffer</i> .....	351
Методы <i>length()</i> и <i>capacity()</i> .....	351
Метод <i>ensureCapacity()</i> .....	352
Метод <i>setLength()</i> .....	352
Методы <i>charAt()</i> и <i>setCharAt()</i> .....	353
Метод <i>getChars()</i> .....	353
Метод <i>append()</i> .....	354
Метод <i>insert()</i> .....	355
Метод <i>reverse()</i> .....	355
Методы <i>delete()</i> и <i>deleteCharAt()</i> .....	356
Метод <i>replace()</i> .....	357
Метод <i>substring()</i> .....	357
<b>Глава 14. Пакет <i>java.lang</i>.....</b>	<b>358</b>
Оболочки простых типов .....	359
Класс <i>Number</i> .....	359
Оболочки <i>Double</i> и <i>Float</i> .....	360
Оболочки <i>Byte</i> , <i>Short</i> , <i>Integer</i> и <i>Long</i> .....	365
Оболочка <i>Character</i> .....	374
Оболочка <i>Boolean</i> .....	378
Класс <i>Void</i> .....	378
Класс <i>Process</i> .....	379
Класс <i>Runtime</i> .....	379
Управление памятью .....	381
Выполнение других программ .....	382
Класс <i>System</i> .....	384
Использование метода <i>currentTimeMillis()</i> .....	386
Использование метода <i>arraycopy()</i> .....	387
Свойства среды.....	387
Класс <i>Object</i> .....	388
Использование метода <i>clone()</i> и интерфейса <i>Cloneable</i> .....	389
Класс <i>Class</i> .....	391
Класс <i>ClassLoader</i> .....	394
Класс <i>Math</i> .....	395
Трансцендентные функции .....	395
Экспоненциальные функции .....	396
Округление функций .....	397
Разные методы класса <i>Math</i> .....	398
Компилятор .....	399

Классы <i>Thread</i> , <i>ThreadGroup</i> и интерфейс <i>Runnable</i> .....	399
Интерфейс <i>Runnable</i> .....	399
Класс <i>Thread</i> .....	399
Класс <i>ThreadGroup</i> .....	402
Классы <i>ThreadLocal</i> и <i>InheritableThreadLocal</i> .....	407
Класс <i>Package</i> .....	408
Класс <i>RuntimePermission</i> .....	409
Класс <i>Throwable</i> .....	409
Класс <i>SecurityManager</i> .....	410
Интерфейс <i>Comparable</i> .....	410
Пакеты <i>java.lang.ref</i> и <i>java.lang.reflect</i> .....	410
Пакет <i>java.lang.ref</i> .....	410
Пакет <i>java.lang.reflect</i> .....	411
<b>Глава 15. Пакет <i>java.util</i>: структура коллекций .....</b>	<b>412</b>
Краткий обзор коллекций.....	413
Интерфейсы коллекций .....	415
Интерфейс <i>Collection</i> .....	416
Интерфейс <i>List</i> .....	419
Интерфейс <i>Set</i> .....	421
Интерфейс <i>SortedSet</i> .....	421
Классы <i>Collection</i> .....	422
Класс <i>ArrayList</i> .....	423
Получение массива из <i>ArrayList</i> -объекта .....	425
Класс <i>LinkedList</i> .....	427
Класс <i>HashSet</i> .....	428
Класс <i>TreeSet</i> .....	430
Доступ к коллекции через итератор.....	431
Использование итератора .....	432
Коллекции пользовательских классов .....	434
Работа с картами отображений.....	436
Интерфейсы карт .....	436
Классы карт отображений .....	440
Компараторы .....	444
Использование компаратора .....	445
Алгоритмы коллекций .....	448
Массивы.....	452
Наследованные классы и интерфейсы .....	456
Интерфейс <i>Enumeration</i> .....	457
Класс <i>Vector</i> .....	457
Класс <i>Stack</i> .....	463
Класс <i>Dictionary</i> .....	465
Класс <i>Hashtable</i> .....	466
Класс <i>Properties</i> .....	470
Использование методов <i>store()</i> и <i>load()</i> .....	474
Резюме.....	476
<b>Глава 16. Пакет <i>java.util</i>: сервисные классы .....</b>	<b>477</b>
Класс <i> StringTokenizer</i> .....	477
Класс <i> BitSet</i> .....	479

Класс <i>Date</i> .....	482
Сравнение дат.....	484
Класс <i>Calendar</i> .....	484
Класс <i>GregorianCalendar</i> .....	488
Класс <i>TimeZone</i> .....	490
Класс <i>SimpleTimeZone</i> .....	491
Класс <i>Locale</i> .....	492
Класс <i>Random</i> .....	493
Класс <i>Observable</i> .....	496
Интерфейс <i>Observer</i> .....	497
Пример наблюдателя .....	498
Пакет <i>java.util.zip</i> .....	500
Пакет <i>java.util.jar</i> .....	500
<b>Глава 17. Ввод/вывод: обзор пакета <i>java.io</i>.....</b>	<b>501</b>
Классы и интерфейсы ввода/вывода Java .....	501
Класс <i>File</i> .....	503
Каталоги .....	506
Использование интерфейса <i>FilenameFilter</i> .....	507
Альтернативный метод <i>listFiles()</i> .....	508
Создание каталогов .....	509
Поточные классы .....	509
Байтовые потоки .....	509
Класс <i>InputStream</i> .....	510
Класс <i>OutputStream</i> .....	510
Класс <i>FileInputStream</i> .....	511
Класс <i> FileOutputStream</i> .....	513
Класс <i>ByteArrayInputStream</i> .....	515
Класс <i>ByteArrayOutputStream</i> .....	516
Фильтрованные байтовые потоки .....	517
Буферизированные байтовые потоки .....	518
Класс <i>SequenceInputStream</i> .....	522
Класс <i>PrintStream</i> .....	523
Класс <i>RandomAccessFile</i> .....	524
Символьные потоки.....	525
Класс <i>Reader</i> .....	525
Класс <i>Writer</i> .....	526
Класс <i>FileReader</i> .....	527
Класс <i>FileWriter</i> .....	528
Класс <i>CharArrayReader</i> .....	529
Класс <i>CharArrayWriter</i> .....	530
Класс <i>BufferedReader</i> .....	531
Класс <i> BufferedWriter</i> .....	533
Класс <i>PushbackReader</i> .....	533
Класс <i> PrintWriter</i> .....	534
Использование поточного ввода/вывода.....	535
Улучшение метода <i>wc()</i> с помощью класса <i>StreamTokenizer</i> .....	537

Сериализация .....	539
Интерфейс <i>Serializable</i> .....	540
Интерфейс <i>Externalizable</i> .....	540
Интерфейс <i>ObjectOutput</i> .....	540
Класс <i> ObjectOutputStream</i> .....	541
Интерфейс <i> ObjectInput</i> .....	542
Класс <i> ObjectInputStream</i> .....	543
Пример с сериализацией .....	545
Предимущества потоков.....	547
<b>Глава 18. Работа в сети .....</b>	<b>548</b>
Основы работы в сети .....	548
Обзор сокетов .....	549
Клиент-сервер .....	549
Зарезервированные сокеты .....	550
Роху-серверы .....	551
Адресация Internet.....	552
Java и сеть.....	552
Сетевые классы и интерфейсы .....	553
Класс <i>InetAddress</i> .....	553
Производственные методы.....	554
Методы экземпляра .....	555
Сокеты TCP/IP клиентов.....	556
Пример работы с сокет-соединением (программа Whois).....	558
Использование URL.....	559
Формат .....	559
Класс <i>URLConnection</i> .....	561
Сокеты TCP/IP серверов.....	562
Кэширующий роху HTTP-сервер .....	563
Исходный код.....	564
Дейтаграммы.....	584
Класс <i>DatagramPacket</i> .....	584
Дейтаграммный сервер и клиент.....	585
Достоинства сети .....	587
<b>Глава 19. Класс <i>Applet</i>.....</b>	<b>588</b>
Основы аплетов.....	588
Класс <i>Applet</i> .....	589
Архитектура аплета .....	592
Скелетная схема аплета .....	593
Инициализация и завершение аплета .....	594
Переопределение метода <i>update()</i> .....	595
Простые методы отображения аплетов .....	596
Требование перерисовки .....	598
Аплет с бегущим заголовком .....	599
Использование окна состояния.....	602
Тег < <i>applet</i> >.....	603
Пересылка параметров в аплеты .....	605
Усовершенствованный аплет заголовка .....	606

Методы <i>getDocumentBase()</i> и <i>getCodeBase()</i> .....	608
Интерфейс <i>AppletContext</i> и метод <i>showDocument()</i> .....	609
Интерфейс <i>AudioClip</i> .....	611
Интерфейс <i>AppletStub</i> .....	611
Вывод на консоль .....	611
<b>Глава 20. Обработка событий .....</b>	<b>612</b>
Два механизма обработки событий.....	612
Модель делегирования событий.....	613
События .....	613
Источники событий.....	614
Блок прослушивания событий.....	615
Классы событий .....	615
Класс <i>ActionEvent</i> .....	617
Класс <i>AdjustmentEvent</i> .....	618
Класс <i>ComponentEvent</i> .....	618
Класс <i>ContainerEvent</i> .....	619
Класс <i>FocusEvent</i> .....	620
Класс <i>InputEvent</i> .....	620
Класс <i>ItemEvent</i> .....	621
Класс <i>KeyEvent</i> .....	621
Класс <i>MouseEvent</i> .....	622
Класс <i>TextEvent</i> .....	624
Класс <i>WindowEvent</i> .....	624
Элементы-источники событий .....	625
Интерфейсы прослушивания событий .....	625
Интерфейс <i>ActionListener</i> .....	627
Интерфейс <i>AdjustmentListener</i> .....	627
Интерфейс <i>ComponentListener</i> .....	627
Интерфейс <i>ContainerListener</i> .....	627
Интерфейс <i>FocusListener</i> .....	627
Интерфейс <i>ItemListener</i> .....	628
Интерфейс <i>KeyListener</i> .....	628
Интерфейс <i>MouseListener</i> .....	628
Интерфейс <i>MouseMotionListener</i> .....	628
Интерфейс <i>TextListener</i> .....	629
Интерфейс <i>WindowListener</i> .....	629
Использование модели делегирования событий .....	629
Обработка событий мыши .....	630
Обработка событий клавиатуры.....	633
Классы-адаптеры.....	636
Внутренние классы .....	638
Анонимные внутренние классы .....	640
<b>Глава 21. Введение в AWT: работа с окнами, графикой и текстом .....</b>	<b>642</b>
Классы AWT .....	643
Основы оконной графики.....	645
Класс <i>Component</i> .....	646

Класс <i>Container</i> .....	646
Класс <i>Panel</i> .....	647
Класс <i>Window</i> .....	647
Класс <i>Frame</i> .....	647
Класс <i>Canvas</i> .....	648
Работа с фреймовыми окнами.....	648
Установка размеров окна .....	648
Скрытие и показ окна .....	649
Установка заголовка окна .....	649
Закрытие фрейм-окна.....	649
Создание фрейм-окна в апплете .....	649
Обработка событий фрейм-окна .....	651
Создание оконной программы .....	656
Отображение информации в окне .....	658
Работа с графикой .....	658
Рисование линий.....	659
Рисование прямоугольников .....	660
Рисование эллипсов и кругов.....	661
Рисование дуг .....	662
Рисование многоугольников.....	663
Установка размеров графики .....	664
Работа с цветом .....	665
Цветовые методы .....	665
Установка текущего цвета графики .....	666
Апплет с демонстрацией цветов .....	667
Установка режима рисования .....	668
Работа со шрифтами .....	670
Определение доступных шрифтов .....	671
Создание и выбор шрифта .....	673
Получение информации о шрифте .....	673
Управление текстовым выводом с помощью класса <i>FontMetrics</i> .....	676
Отображение многострочного текста .....	678
Выравнивание текста по центру .....	680
Выравнивание многострочного текста .....	681
Исследование текста и графики .....	685
<b>Глава 22. Использование элементов управления, менеджеров компоновки и меню AWT .....</b>	<b>686</b>
Элементы управления. Основные понятия .....	687
Добавление и удаление элементов управления .....	687
Реагирование на элементы управления .....	688
Текстовые метки .....	688
Использование кнопок .....	689
Обработка кнопок .....	690
Применение флагков .....	693
Обработка флагков .....	694
Класс <i>CheckboxGroup</i> .....	695
Элемент управления <i>Choice</i> .....	697
Обработка списков типа <i>Choice</i> .....	698

Использование списков .....	700
Обработка списков .....	701
Управление полосами прокрутки .....	703
Обработка полос прокрутки .....	705
Использование класса <i>TextField</i> .....	707
Обработка <i>TextField</i> .....	708
Использование <i>TextArea</i> .....	709
Понятие менеджера компоновки .....	711
Менеджер <i>FlowLayout</i> .....	713
Класс <i>BorderLayout</i> .....	714
Использование вставок .....	716
Менеджер <i>GridLayout</i> .....	718
Класс <i>CardLayout</i> .....	719
Панели меню и меню .....	723
Диалоговые окна .....	729
Класс <i>FileDialog</i> .....	734
Обработка событий путем расширения AWT-компонентов .....	736
Расширение класса <i>Button</i> .....	737
Расширение класса <i>Checkbox</i> .....	738
Расширение группы флагков .....	739
Расширение класса <i>Choice</i> .....	740
Расширение класса <i>List</i> .....	741
Расширение класса <i>Scrollbar</i> .....	742
Исследование элементов управления, меню и менеджеров компоновки .....	743
<b>Глава 23. Работа с изображениями.....</b>	<b>744</b>
Форматы графических файлов .....	745
Создание, загрузка и просмотр изображений .....	745
Создание объекта изображения .....	745
Загрузка изображения .....	746
Просмотр изображения .....	746
Интерфейс <i>ImageObserver</i> .....	748
Пример с <i>ImageObserver</i> .....	750
Двойная буферизация .....	752
Класс <i>MediaTracker</i> .....	755
Интерфейс <i>ImageProducer</i> .....	759
Производитель изображений <i>MemoryImageSource</i> .....	759
Интерфейс <i>ImageConsumer</i> .....	761
Класс <i>PixelGrabber</i> .....	761
Класс <i>ImageFilter</i> .....	764
Фильтр <i>CropImageFilter</i> .....	765
Фильтр <i>RGBImageFilter</i> .....	767
Анимация ячеек .....	779
Дополнительные классы изображений Java 2 .....	782
<b>Глава 24. Дополнительные пакеты.....</b>	<b>783</b>
Пакеты ядра Java API .....	783
Отражение .....	786

Вызов удаленных методов (RMI) .....	791
Простое RMI-приложение клиент-сервер.....	791
Текстовое форматирование .....	796
Класс <i>DateFormat</i> .....	796
Класс <i>SimpleDateFormat</i> .....	798
<b>ЧАСТЬ III. РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ .....</b>	<b>801</b>
<b>Глава 25. Компоненты Java Beans .....</b>	<b>803</b>
Что такое Java Bean-компонент?.....	804
Преимущества технологии Java Beans.....	804
Инструментарий построения приложений .....	805
Комплект разработчика Bean-компонентов.....	806
Установка BDK .....	806
Запуск BDK .....	806
Использование BDK .....	807
JAR-файлы .....	809
Файлы описания .....	810
Утилита <i>jar</i> .....	810
Интропсекция .....	812
Проектные шаблоны для свойств.....	812
Проектные шаблоны для событий .....	814
Методы .....	815
Разработка простого Bean-компонента .....	815
Создание нового Bean-компонента.....	816
Использование связанных свойств .....	819
Алгоритм .....	820
Использование интерфейса <i>BeanInfo</i> .....	821
Ограниченные свойства .....	823
Сохраняемость.....	823
Конфигураторы .....	824
Java Beans API.....	824
Будущее Bean-технологии .....	827
<b>Глава 26. Система Swing .....</b>	<b>829</b>
Класс <i>JApplet</i> .....	830
Значки и метки .....	831
Текстовые поля .....	833
Кнопки .....	834
Класс <i>JButton</i> .....	834
Флажки.....	836
Переключатели .....	838
Поля со списком .....	840
Панели со вкладками .....	842
Панели прокрутки.....	844
Деревья .....	846
Таблицы .....	850
Другие возможности и будущее Swing-технологии .....	852

<b>Глава 27. Сервлеты .....</b>	<b>854</b>
Предпосылки .....	854
Жизненный цикл сервлета .....	855
Java Servlet Development Kit .....	856
Простой сервле <sup>t</sup> .....	857
Создание и компиляция исходного кода сервлета .....	857
Запуск утилиты <i>Servletrunner</i> .....	858
Запуск Web-браузера и запрос сервлета .....	858
Servlet API .....	858
Пакет <i>javax.servlet</i> .....	859
Интерфейс <i>Servlet</i> .....	860
Интерфейс <i>ServletConfig</i> .....	861
Интерфейс <i>ServletContext</i> .....	861
Интерфейс <i>ServletRequest</i> .....	862
Интерфейс <i>ServletResponse</i> .....	863
Интерфейс <i>SingleThreadModel</i> .....	864
Класс <i>GenericServlet</i> .....	864
Класс <i>ServletInputStream</i> .....	864
Класс <i>ServletOutputStream</i> .....	865
Класс <i>ServletException</i> .....	865
Класс <i>UnavailableException</i> .....	865
Чтение параметров сервлета .....	865
Чтение параметров инициализации .....	867
Пакет <i>javax.servlet.http</i> .....	869
Интерфейс <i>HttpServletRequest</i> .....	870
Интерфейс <i>HttpServletResponse</i> .....	871
Интерфейс <i>HttpSession</i> .....	873
Интерфейс <i>HttpSessionBindingListener</i> .....	874
Интерфейс <i>HttpSessionContext</i> .....	874
Класс <i>Cookie</i> .....	874
Класс <i>HttpServlet</i> .....	876
Класс <i>HttpSessionBindingEvent</i> .....	877
Класс <i>HttpUtils</i> .....	878
Обработка запросов и ответов HTTP .....	878
Обработка GET-запросов HTTP .....	878
Обработка POST-запросов HTTP .....	880
Использование cookie-данных .....	881
Прослеживание сеанса .....	884
Проблемы защиты .....	885
Исследование сервлетов .....	886
<b>Глава 28. Миграция из C++ в Java .....</b>	<b>887</b>
Различия между C++ и Java .....	887
Что Java исключил из C++ .....	887
Новые свойства, добавленные в Java .....	889
Отличающиеся свойства .....	890

Исключение указателей в C++ .....	891
Преобразование параметров типа указателя .....	892
Преобразование указателей, работающих на массивах .....	894
Ссылочные параметры C++ в сравнении со ссылочными параметрами Java .....	896
Преобразование абстрактных классов C++ в Java-интерфейсы .....	899
Преобразование умалчиваемых аргументов .....	903
Преобразование иерархий множественного наследования C++ .....	905
Деструкторы в сравнении с методом <i>finalize()</i> .....	907
<b>ЧАСТЬ IV. ПРИМЕНЕНИЕ JAVA .....</b>	<b>911</b>
<b>Глава 29. Апплет <i>DynamicBillboard</i>.....</b>	<b>913</b>
Тег < <i>applet</i> >.....	913
Обзор исходного кода.....	915
<i>DynamicBillboard.java</i> .....	915
<i>BillData.java</i> .....	923
<i>BillTransition.java</i> .....	925
<i>ColumnTransition.java</i> .....	927
<i>FadeTransition.java</i> .....	930
<i>SmashTransition.java</i> .....	933
<i>TearTransition.java</i> .....	937
<i>UnrollTransition.java</i> .....	941
Динамический код.....	945
<b>Глава 30. <i>ImageMenu</i>: Web-меню изображений .....</b>	<b>947</b>
Исходное изображение.....	949
Тег < <i>applet</i> >.....	950
Методы.....	951
Метод <i>init()</i> .....	951
Метод <i>update()</i> .....	951
Метод <i>lateInit()</i> .....	951
Метод <i>paint()</i> .....	951
Метод <i>mouseExited()</i> .....	952
Метод <i>mouseDragged()</i> .....	952
Метод <i>mouseMoved()</i> .....	952
Метод <i>mouseReleased()</i> .....	953
Код.....	953
Резюме.....	955
<b>Глава 31. Апплет <i>Lavatron</i>: дисплей для спортивной арены .....</b>	<b>956</b>
Как работает <i>Lavatron</i> .....	956
Исходный код.....	958
Тег < <i>applet</i> >.....	958
<i>Lavatron.java</i> .....	959
Класс <i>IntHash</i> .....	963
Апплет <i>Hot Lava</i> .....	966

<b>Глава 32. <i>Scrabblet</i>: многопользовательская игра в слова .....</b>	<b>967</b>
Вопросы сетевой безопасности .....	967
Игра .....	968
Подсчет очков .....	971
Исходный код .....	975
Ter <applet> .....	975
Scrabblet.java .....	976
IntroCanvas.java .....	987
Board.java .....	989
Bag.java .....	1005
Letter.java .....	1007
ServerConnection.java .....	1012
Код сервера .....	1017
Server.java .....	1017
ClientConnection.java .....	1020
Совершенствование <i>Scrabblet</i> .....	1025
<b>Приложение. Использование документационных комментариев Java.....</b>	<b>1027</b>
Теги <i>javadoc</i> .....	1027
Ter @author .....	1028
Ter @deprecated .....	1029
Ter @exception .....	1029
Ter {@link} .....	1029
Ter @param .....	1029
Ter @return .....	1029
Ter @see .....	1030
Ter @serial .....	1030
Ter @serialData .....	1030
Ter @serialField .....	1030
Ter @since .....	1031
Ter @throws .....	1031
Ter @version .....	1031
Общая форма документационного комментария .....	1031
Что выводит <i>javadoc</i> .....	1032
Пример использования документационных комментариев .....	1032
<b>Предметный указатель.....</b>	<b>1034</b>