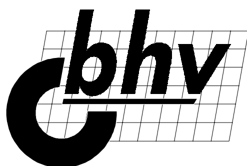
Ильдар Хабибуллин

САМОУЧИТЕЛЬ JAVA



Дюссельдорф ◆ Киев ◆ Москва ◆ Санкт-Петербург

УДК 681.3.06

Книга посвящена объектно-ориентированному языку программирования Java 2. Последовательно излагаются практические приемы работы с новейшими конструкциями языка, графической библиотекой классов, расширенной библиотекой Java 2D, со звуком, печатью, способами русификации программ. Около двухсот законченных программ иллюстрируют приведенные приемы программирования. Подробные схемы и описания классов и методов J2SDK позволят использовать книгу как настольный справочник по технологии Java.

Для широкого круга программистов

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зав. редакцией	*Наталья Таркова*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Ангелины Лужиной*
Зав. производством	*Николай Тверских*

Хабибуллин И. Ш.

Самоучитель Java. — СПб.: БХВ-Петербург, 2001. — 464 с.: ил.

ISBN 5-94157-041-4

© И. Ш. Хабибуллин, 2001

© Оформление, издательство "БХВ-Петербург", 2001

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 07.02.01.
Формат $70\times100^1/_{16}$. Печать офсетная. Усл. печ. л. 37,4.
Тираж 5000 экз. Заказ
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар, № 77.99.1.953.П.950.3.99
от 01.03.1999 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН.
199034, Санкт-Петербург, 9-я линия, 12.

Содержание

Введение ... 12

Что такое Java ... 13
Структура книги ... 14
Выполнение Java-программы ... 16
Что такое JDK ... 18
Что такое JRE ... 20
Как установить JDK ... 20
Как использовать JDK ... 21
Интегрированные среды Java ... 22
Особая позиция Microsoft ... 23
Java в Internet ... 23
Литература по Java ... 25
Благодарности ... 26

ЧАСТЬ I. БАЗОВЫЕ КОНСТРУКЦИИ ЯЗЫКА JAVA ... 27

Глава 1. Встроенные типы данных, операции над ними ... 28

Первая программа на Java ... 28
Комментарии ... 32
Константы ... 33
    Целые ... 33
    Действительные ... 34
    Символы ... 34
    Строки ... 35
Имена ... 36
Примитивные типы данных и операции ... 37
    Логический тип ... 38
    Логические операции ... 38
    Целые типы ... 40
    Операции над целыми типами ... 41
        Арифметические операции ... 41
        Приведение типов ... 42

Операции сравнения ... 44
Побитовые операции ... 44
Сдвиги ... 45
Вещественные типы ... 46
Операции присваивания ... 47
Условная операция ... 48
Выражения ... 48
Приоритет операций ... 50
Операторы ... 50
Блок ... 51
Операторы присваивания ... 52
Условный оператор ... 52
Операторы цикла ... 54
Оператор *continue* и метки ... 57
Оператор *break* ... 58
Оператор варианта ... 58
Массивы ... 60
Многомерные массивы ... 62
Заключение ... 64

Глава 2. Объектно-ориентированное программирование в Java ... 65

Парадигмы программирования ... 65
Принципы объектно-ориентированного программирования ... 68
Абстракция ... 68
Иерархия ... 70
Ответственность ... 72
Модульность ... 73
Принцип KISS ... 75
Как описать класс и подкласс ... 76
Абстрактные методы и классы ... 80
Окончательные члены и классы ... 81
Класс *Object* ... 81
Конструкторы класса ... 82
Операция *new* ... 83
Статические члены класса ... 84
Класс *Complex* ... 86
Метод *main()* ... 89
Где видны переменные ... 90
Вложенные классы ... 92
Отношения "быть частью" и "являться" ... 96
Заключение ... 97

Глава 3. Пакеты и интерфейсы ... 98

Пакет и подпакет ... 99
Права доступа к членам класса ... 100
Размещение пакетов по файлам ... 103
Импорт классов и пакетов ... 105

Java-файлы 106
Интерфейсы 106
Design patterns 111
Заключение 114

ЧАСТЬ II. ИСПОЛЬЗОВАНИЕ КЛАССОВ, ВХОДЯЩИХ В JAVA DEVELOPMENT KIT 115

Глава 4. Классы-оболочки 116

Числовые классы 117
Класс *Boolean* 119
Класс *Character* 119
Класс *BigInteger* 122
Класс *BigDecimal* 125
Класс *Class* 129

Глава 5. Работа со строками 132

Класс *String* 133
  Как создать строку 133
  Сцепление строк 138
Манипуляции строками 139
  Как узнать длину строки 139
  Как выбрать символы из строки 139
  Как выбрать подстроку 140
  Как сравнить строки 140
  Как найти символ в строке 142
  Как найти подстроку 143
  Как изменить регистр букв 144
  Как заменить отдельный символ 144
  Как убрать пробелы в начале и конце строки 144
  Как преобразовать данные другого типа в строку 144
Класс *StringBuffer* 145
  Конструкторы 146
  Как добавить подстроку 146
  Как вставить подстроку 146
  Как удалить подстроку 147
  Как удалить символ 147
  Как заменить подстроку 148
  Как перевернуть строку 148
Синтаксический разбор строки 148
  Класс *StringTokenizer* 148
Заключение 150

Глава 6. Классы-коллекции 151

Класс *Vector* 151
  Как создать вектор 152

Как добавить элемент в вектор ... 152
Как заменить элемент ... 152
Как узнать размер вектора ... 152
Как обратиться к элементу вектора ... 153
Как узнать, есть ли элемент в векторе ... 153
Как узнать индекс элемента ... 153
Как удалить элементы ... 153
Класс *Stack* ... 155
Класс *Hashtable* ... 156
Как создать таблицу ... 156
Как заполнить таблицу ... 157
Как получить значение по ключу ... 157
Как узнать наличие ключа или значения ... 157
Как получить все элементы таблицы ... 157
Как удалить элементы ... 158
Класс *Properties* ... 159
Интерфейс *Collection* ... 161
Интерфейс *List* ... 162
Интерфейс *Set* ... 162
Интерфейс *SortedSet* ... 163
Интерфейс *Map* ... 163
Вложенный интерфейс *Map.Entry* ... 164
Интерфейс *SortedMap* ... 164
Абстрактные классы-коллекции ... 165
Интерфейс *Iterator* ... 165
Интерфейс *ListIterator* ... 167
Классы, создающие списки ... 168
Двунаправленный список ... 168
Классы, создающие отображения ... 169
Упорядоченные отображения ... 169
Сравнение элементов коллекций ... 170
Классы, создающие множества ... 170
Упорядоченные множества ... 171
Действия с коллекциями ... 172
Методы класса *Collections* ... 172
Заключение ... 173

Глава 7. Классы-утилиты ... 174

Работа с массивами ... 174
Локальные установки ... 176
Работа с датами и временем ... 177
Часовой пояс и летнее время ... 178
Класс *Calendar* ... 178
Подкласс *GregorianCalendar* ... 178
Представление даты и времени ... 179
Получение случайных чисел ... 180
Копирование массивов ... 181
Взаимодействие с системой ... 181

ЧАСТЬ III. СОЗДАНИЕ ГРАФИЧЕСКОГО ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ И АППЛЕТОВ

ЧАСТЬ III. СОЗДАНИЕ ГРАФИЧЕСКОГО ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ И АППЛЕТОВ 183

Глава 8. Принципы построения графического интерфейса 184

Компонент и контейнер 186
Иерархия классов AWT 190
Заключение 191

Глава 9. Графические примитивы 192

Методы класса *Graphics* 192
Как задать цвет 193
Как нарисовать чертеж 194
Класс *Polygon* 195
Как вывести текст 197
Как установить шрифт 197
Как задать шрифт 197
Класс *FontMetrics* 203
Возможности Java 2D 207
Преобразование координат 208
Класс *AffineTransform* 208
Рисование фигур средствами Java 2D 211
Класс *BasicStroke* 212
Класс *GeneralPath* 215
Классы *GradientPaint* и *TexturePaint* 216
Вывод текста средствами Java 2D 218
Методы улучшения визуализации 223
Заключение 224

Глава 10. Основные компоненты 225

Класс *Component* 225
Класс *Cursor* 227
Как создать свой курсор 228
События 229
Класс *Container* 229
События 230
Компонент *Label* 230
События 231
Компонент *Button* 231
События 231
Компонент *Checkbox* 231
События 232
Класс *CheckboxGroup* 232
Как создать группу радиокнопок 232
Компонент *Choice* 234
События 235
Компонент *List* 235
События 236

Компоненты для ввода текста ... 237
Класс *TextComponent* ... 238
События ... 238
Компонент *TextField* ... 238
События ... 239
Компонент *TextArea* ... 239
События ... 240
Компонент *Scrollbar* ... 241
События ... 242
Контейнер *Panel* ... 244
Контейнер *ScrollPane* ... 245
Контейнер *Window* ... 246
События ... 247
Контейнер *Frame* ... 247
События ... 248
Контейнер *Dialog* ... 249
События ... 250
Контейнер *FileDialog* ... 251
События ... 252
Создание собственных компонентов ... 252
Компонент *Canvas* ... 253
Создание "легкого" компонента ... 255

Глава 11. Размещение компонентов ... 258

Менеджер *FlowLayout* ... 259
Менеджер *BorderLayout* ... 260
Менеджер *GridLayout* ... 263
Менеджер *CardLayout* ... 264
Менеджер *GridBagLayout* ... 266
Заключение ... 268

Глава 12. Обработка событий ... 269

Событие *ActionEvent* ... 276
Обработка действий мыши ... 277
Классы-адаптеры ... 280
Обработка действий клавиатуры ... 281
Событие *TextEvent* ... 282
Обработка действий с окном ... 282
Событие *ComponentEvent* ... 283
Событие *ContainerEvent* ... 284
Событие *FocusEvent* ... 284
Событие *ItemEvent* ... 284
Событие *AdjustmentEvent* ... 285
Несколько слушателей одного источника ... 287
Диспетчеризация событий ... 289
Создание собственного события ... 290

Глава 13. Создание меню ... 292

Всплывающее меню ... 297

Глава 14. Апплеты ... 301

Передача параметров ... 307
Параметры тега *<applet>* ... 310
Сведения об окружении апплета ... 311
Изображение и звук ... 312
Слежение за процессом загрузки ... 312
Класс *MediaTracker* ... 313
Защита от апплета ... 316
Заключение ... 317

Глава 15. Изображения и звук ... 318

Модель обработки "поставщик-потребитель" ... 319
Классы-фильтры ... 322
Как выделить фрагмент изображения ... 322
Как изменить цвет изображения ... 324
Как переставить пикселы изображения ... 325
Модель обработки прямым доступом ... 327
Преобразование изображения в Java 2D ... 329
Аффинное преобразование изображения ... 330
Изменение интенсивности изображения ... 332
Изменение составляющих цвета ... 334
Создание различных эффектов ... 335
Анимация ... 336
Улучшение изображения двойной буферизацией ... 339
Звук ... 343
Проигрывание звука в Java 2 ... 344
Синтез и запись звука в Java 2 ... 349

ЧАСТЬ IV. НЕОБХОДИМЫЕ КОНСТРУКЦИИ JAVA ... 353

Глава 16. Обработка исключительных ситуаций ... 354

Блоки перехвата исключения ... 356
Часть заголовка метода *throws* ... 359
Оператор *throw* ... 361
Иерархия классов-исключений ... 362
Порядок обработки исключений ... 364
Создание собственных исключений ... 364
Заключение ... 366

Глава 17. Подпроцессы ... 367

Класс *Thread* ... 369
Синхронизация подпроцессов ... 374

Согласование работы нескольких подпроцессов 377
Приоритеты подпроцессов 380
Подпроцессы-демоны 381
Группы подпроцессов 382
Заключение 383

Глава 18. Потоки ввода/вывода 384

Консольный ввод/вывод 389
Файловый ввод/вывод 392
Получение свойств файла 394
Буферизованный ввод/вывод 396
Поток простых типов Java 397
  Кодировка UTF-8 398
Прямой доступ к файлу 400
Каналы обмена информацией 400
Сериализация объектов 402
Печать в Java 405
Печать средствами Java 2D 408
  Печать файла 412
  Печать страниц с разными параметрами 414

Глава 19. Сетевые средства Java 417

Работа в WWW 420
Работа по протоколу TCP 425
Работа по протоколу UDP 429

Приложение. Развитие Java 433

Переход к Swing 433
Архиватор *jar* 434
  Создание архива 435
  Файл описания MANIFEST.MF 438
  Файл INDEX.LIST 439
Компоненты JavaBeans 439
Связь с базами данных через JDBC 441
Сервлеты 446
Java на сервере 451
Заключение 454

Список литературы 455

Предметный указатель 457

Введение

Книга, которую вы держите в руках, возникла из курса лекций, читаемых автором в течение последних лет для студентов младших курсов. Подобные книги рождаются после того, как студенты в сотый раз зададут один и тот же вопрос, который лектор уже несколько раз разъяснял в разных вариациях. Возникает желание отослать их к какой-нибудь литературе. Пересмотрев еще раз несколько десятков книг, использованных при подготовке лекций, порывшись в библиотеке и на прилавках книжных магазинов, лектор с удивлением обнаруживает, что не может предложить студентам ничего подходящего. Остается сесть за стол и написать книгу самому. Такое происхождение книги накладывает на нее определенные особенности. Она

- представляет собой сгусток практического опыта, накопленного автором и его студентами с 1996 г.;
- содержит ответы на часто задаваемые вопросы, последние "компьютерщики" называют *FAQ* (Frequency Asked Questions);
- написана кратко и сжато, как конспект лекций, в ней нет лишних слов (за исключением, может быть, тех, что вы только что прочитали);
- рассчитана на читателей, стремящихся быстро и всерьез ознакомиться с новинками компьютерных технологий;
- содержит много примеров применения конструкций Java, которые можно использовать как фрагменты больших производственных разработок в качестве "How to?";
- включает материал, являющийся обязательной частью подготовки специалиста по информационным технологиям;
- не предполагает знание какого-либо языка программирования, а для знатоков выделяются особенности языка Java среди других языков;
- предлагает обсуждение вопросов русификации Java.

Прочитав эту книгу, вы вступите в ряды программистов на Java — разработчиков технологии начала XXI века.

Если спустя несколько месяцев эта книга будет валяться на вашем столе с растрепанными страницами, залитыми кофе и засыпанными пеплом, с массой закладок и загнутых углов, а вы начнете сетовать на то, что книга недостаточно полна и слишком проста, и ее содержание тривиально и широко известно, тогда автор будет считать, что его скромный труд не пропал даром.

Ну что же, начнем!

Что такое Java

Это остров Ява в Малайском архипелаге, территория Индонезии. Это сорт кофе, который любят пить создатели Java (произносится "Джава", с ударением на первом слоге). А если серьезно, то ответить на этот вопрос трудно, потому что границы Java, и без того размытые, все время расширяются. Сначала Java (официальный день рождения технологии Java — 23 мая 1995 г.) предназначалась для программирования бытовых электронных устройств, таких как телефоны. Потом Java стала применяться для программирования браузеров — появились *апплеты*. Затем оказалось, что на Java можно создавать полноценные приложения. Их графические элементы стали оформлять в виде компонентов — появились *JavaBeans*, с которыми Java вошла в мир распределенных систем и промежуточного программного обеспечения, тесно связавшись с технологией CORBA. Остался один шаг до программирования серверов — этот шаг был сделан — появились *сервлеты* и *EJB* (Enterprise JavaBeans). Серверы должны взаимодействовать с базами данных — появились драйверы JDBC (Java DataBase Connection). Взаимодействие оказалось удачным, и многие системы управления базами данных и даже операционные системы включили Java в свое ядро, например Oracle, Linux, MacOS X, AIX. Что еще не охвачено? Назовите, и через полгода услышите, что Java уже вовсю применяется и там. Из-за этой размытости самого понятия его описывают таким же размытым словом — *технология*.

Такое быстрое и широкое распространение технологии Java не в последнюю очередь связано с тем, что она использует новый, специально созданный язык программирования, который так и называется — язык Java. Этот язык создан на базе языков Smalltalk, Pascal, C++ и др., вобрав их лучшие, по мнению создателей, черты и отбросив худшие. На этот счет есть разные мнения, но бесспорно, что язык получился удобным для изучения, написанные на нем программы легко читаются и отлаживаются: первую программу можно написать уже через час после начала изучения языка. Язык Java становится языком обучения объектно-ориентированному программированию, так же, как язык Pascal был языком обучения структурному программированию. Недаром на Java уже написано огромное количество программ, библиотек классов, а собственный апплет не написал только уж совсем ленивый.

Для полноты картины следует сказать, что создавать приложения для технологии Java можно не только на языке Java, уже появились и другие языки, есть даже компиляторы с языков Pascal и C++, но лучше все-таки использовать язык Java: на нем все аспекты технологии излагаются проще и удобнее.

По скромному мнению автора, язык Java будет использоваться для описания различных приемов объектно-ориентированного программирования так же, как для реализации алгоритмов применялся вначале язык Algol, а затем язык Pascal.

Ясно, что всю технологию Java нельзя изложить в одной книге, полное описание ее возможностей составит целую библиотеку. Эта книга посвящена только языку Java. Прочитав ее, вы сможете создавать Java-приложения любой сложности, свободно разбираться в литературе и листингах программ, продолжать изучение аспектов технологии Java по специальной литературе.

Язык Java тоже очень бурно развивается, некоторые его методы объявляются устаревшими (deprecated), появляются новые конструкции, увеличивается встроенная библиотека классов, но есть устоявшееся ядро языка, сохраняется его дух и стиль. Вот это-то устоявшееся и излагается в книге.

Структура книги

Книга состоит из четырех частей и приложения.

Первая часть содержит три главы, в которых рассматриваются базовые понятия языка. По прочтении ее вы сможете свободно разбираться в понятиях объектно-ориентированного программирования и их реализации на языке Java, создавать свои объектно-ориентированные программы, рассчитанные на консольный ввод/вывод.

В *главе 1* описываются типы исходных данных, операции с ними, выражения, массивы, операторы управления потоком информации, приводятся примеры записи часто встречающихся алгоритмов на Java. После знакомства с этой главой вы сможете писать программы на Java, реализующие любые вычислительные алгоритмы, встречающиеся в вашей практике.

В *главе 2* вводятся основные понятия объектно-ориентированного программирования: объект и метод, абстракция, инкапсуляция, наследование, полиморфизм, контракты методов и их поручения друг другу. Эта глава призвана привить вам "объектный" взгляд на реализацию сложных проектов, после ее прочтения вы научитесь описывать проект как совокупность взаимодействующих объектов. Здесь же предлагается реализация всех этих понятий на языке Java. Тут вы, наконец, поймете, что же такое эти объекты и как они взаимодействуют друг с другом.

В *главе 3* определяются пакеты классов и интерфейсы, ограничения доступа к классам и методам, на примерах подробно разбираются правила их ис-

пользования. Объясняется структура встроенной библиотеки классов Java API.

Во *второй части* рассматриваются пакеты основных классов, составляющих неотъемлемую часть Java, разбираются приемы работы с ними и приводятся примеры практического использования основных классов. Здесь вы увидите, как идеи объектно-ориентированного программирования реализуются на практике в сложных производственных библиотеках классов. После изучения этой части вы сможете реализовывать наиболее часто встречающиеся ситуации объектно-ориентированного программирования с помощью стандартных классов.

Глава 4 прослеживает иерархию стандартных классов и интерфейсов Java, на этом примере показано, как в профессиональных системах программирования реализуются концепции абстракции, инкапсуляции и наследования.

В *главе 5* подробно излагаются приемы работы со строками символов, которые, как и все в Java, являются объектами, приводятся примеры синтаксического анализа текстов.

В *главе 6* показано, как в языке Java реализованы контейнеры, позволяющие работать с совокупностями объектов и создавать сложные структуры данных.

Глава 7 описывает различные классы-утилиты, полезные во многих ситуациях при работе с датами, случайными числами, словарями и другими необходимыми элементами программ.

В *третьей части* объясняется создание графического интерфейса пользователя (ГИП) с помощью стандартной библиотеки классов AWT (Abstract Window Toolkit) и даны многочисленные примеры построения интерфейса. Подробно разбирается принятый в Java метод обработки событий, основанный на идее делегирования. Здесь же появляются апплеты как программы Java, работающие в окне браузера. Подробно обсуждается система безопасности выполнения апплетов. После прочтения третьей части вы сможете создавать полноценные приложения под графические платформы MS Windows, X Window System и др., а также программировать браузеры.

Глава 8 описывает иерархию классов библиотеки AWT, которую необходимо четко себе представлять для создания удобного интерфейса. Здесь же рассматривается библиотека графических классов Swing, постепенно становящаяся стандартной наряду с AWT.

В *главе 9* демонстрируются приемы рисования с помощью графических примитивов, способы задания цвета и использование шрифтов, а также решается вопрос русификации приложений Java.

В *главе 10* обсуждается понятие графической составляющей, рассматриваются готовые компоненты AWT и их применение, а также создание собственных компонентов.

В *главе 11* показано, какие способы размещения компонентов в графическом контейнере имеются в AWT, и как их применять в разных ситуациях.

В *главе 12* вводятся способы реагирования компонентов на сигналы от клавиатуры и мыши, а именно, модель делегирования, принятая в Java.

В *главе 13* описывается создание системы меню — необходимой составляющей графического интерфейса.

В *главе 14*, наконец-то, появляются апплеты — Java-программы, предназначенные для выполнения в окне браузера, и обсуждаются их особенности.

В *главе 15* рассматривается работа с изображениями и звуком средствами AWT.

В *четвертой части* изучаются конструкции языка Java, не связанные общей темой. Некоторые из них необходимы для создания надежных программ, учитывающих все нештатные ситуации, другие позволяют реализовывать сложное взаимодействие объектов. Здесь же рассматривается передача потоков данных от одной программы Java к другой. Внимательное изучение четвертой части позволит вам дополнить свои разработки гибкими средствами управления выполнением приложения, создавать сложные клиент-серверные системы.

Глава 16 описывает средства обработки исключительных ситуаций, возникающих во время выполнения готовой программы, встроенные в Java.

Глава 17 рассказывает об уникальном свойстве языка Java — способности создавать подпроцессы (threads) и управлять их взаимодействием прямо из программы.

В *главе 18* обсуждается концепция потока данных и ее реализация в Java для организации ввода/вывода на внешние устройства.

Глава 19, последняя по счету, но не по важности, рассматривает сетевые средства языка Java, позволяющие скрыть все сложности протоколов Internet и максимально облегчить написание клиент-серверных приложений.

В *приложении* описываются дополнительные аспекты технологии Java: компоненты JavaBeans, сервлеты, драйверы соединения с базами данных JDBC, и прослеживаются пути дальнейшего развития технологии Java. Ознакомившись с этим приложением, вы сможете ориентироваться в информации о современном состоянии технологии Java и выбрать себе материал для дальнейшего изучения.

Выполнение Java-программы

Как вы знаете, программа, написанная на одном из языков высокого уровня, к которым относится и язык Java, так называемый *исходный модуль* ("исходник" или "сырец" на жаргоне, от английского "source"), не может быть сразу же выполнена. Ее сначала надо откомпилировать, т. е. перевести в по-

следовательность машинных команд — *объектный модуль.* Но и он, как правило, не может быть сразу же выполнен: объектный модуль надо еще скомпоновать с библиотеками использованных в модуле функций и разрешить перекрестные ссылки между секциями объектного модуля, получив в результате *загрузочный модуль* — полностью готовую к выполнению программу.

Исходный модуль, написанный на Java, не может избежать этих процедур, но здесь проявляется главная особенность технологии Java — программа компилируется сразу в машинные команды, но не команды какого-то конкретного процессора, а в команды так называемой виртуальной машины Java (JVM, Java Virtual Machine). *Виртуальная машина Java* — это совокупность команд вместе с системой их выполнения. Для специалистов скажем, что виртуальная машина Java полностью стековая, так что не требуется сложная адресация ячеек памяти и большое количество регистров. Поэтому команды JVM короткие, большинство из них имеет длину 1 байт, отчего команды JVM называют *байт-кодами* (bytecodes), хотя имеются команды длиной 2 и 3 байта. Согласно статистическим исследованиям средняя длина команды составляет 1,8 байта. Полное описание команд и всей архитектуры JVM содержится в *спецификации виртуальной машины Java* (VMS, Virtual Machine Specification). Если вы хотите в точности узнать, как работает виртуальная машина Java, ознакомьтесь с этой спецификацией.

Другая особенность Java — все стандартные функции, вызываемые в программе, подключаются к ней только на этапе выполнения, а не включаются в байт-коды. Как говорят специалисты, происходит *динамическая компоновка* (dynamic binding). Это тоже сильно уменьшает объем откомпилированной программы.

Итак, на первом этапе программа, написанная на языке Java, переводится компилятором в байт-коды. Эта компиляция не зависит от типа какого-либо конкретного процессора и архитектуры некоего конкретного компьютера. Она может быть выполнена один раз сразу же после написания программы. Байт-коды записываются в одном или нескольких файлах, могут храниться во внешней памяти или передаваться по сети. Это особенно удобно благодаря небольшому размеру файлов с байт-кодами. Затем полученные в результате компиляции байт-коды можно выполнять на любом компьютере, имеющем систему, реализующую JVM. При этом не важен ни тип процессора, ни архитектура компьютера. Так реализуется принцип Java "Write once, run anywhere" — "Написано однажды, выполняется где угодно".

Интерпретация байт-кодов и динамическая компоновка значительно замедляют выполнение программ. Это не имеет значения в тех ситуациях, когда байт-коды передаются по сети, сеть все равно медленнее любой интерпретации, но в других ситуациях требуется мощный и быстрый компьютер. Поэтому постоянно идет усовершенствование интерпретаторов в сторону увеличения скорости интерпретации. Разработаны *JIT-компиляторы* (Just-In-Time), запоминающие уже интерпретированные участки кода в машинных

командах процессора и просто выполняющие эти участки при повторном обращении, например, в циклах. Это значительно увеличивает скорость повторяющихся вычислений. Фирма SUN разработала целую технологию HotSpot и включает ее в свою виртуальную машину Java. Но, конечно, наибольшую скорость может дать только специализированный процессор.

Фирма SUN Microsystems выпустила микропроцессоры PicoJava, работающие на системе команд JVM, и собирается выпускать целую линейку все более мощных Java-процессоров. Есть уже и Java-процессоры других фирм. Эти процессоры непосредственно выполняют байт-коды. Но при выполнении программ Java на других процессорах требуется еще интерпретация команд JVM в команды конкретного процессора, а значит, нужна программа-интерпретатор, причем для каждого типа процессоров и для каждой архитектуры компьютера следует написать свой интерпретатор.

Эта задача уже решена практически для всех компьютерных платформ. На них реализованы виртуальные машины Java, а для наиболее распространенных платформ имеется несколько реализаций JVM разных фирм. Все больше операционных систем и систем управления базами данных включают реализацию JVM в свое ядро. Создана и специальная операционная система JavaOS, применяемая в электронных устройствах. В большинство браузеров встроена виртуальная машина Java для выполнения апплетов.

Внимательный читатель уже заметил, что кроме реализации JVM для выполнения байт-кодов на компьютере еще нужно иметь набор функций, вызываемых из байт-кодов и динамически компонующихся с байт-кодами. Этот набор оформляется в виде библиотеки классов Java, состоящей из одного или нескольких *пакетов*. Каждая функция может быть записана байт-кодами, но, поскольку она будет храниться на конкретном компьютере, ее можно записать прямо в системе команд этого компьютера, избегнув тем самым интерпретации байт-кодов. Такие функции называют *"родными" методами* (native methods). Применение "родных" методов ускоряет выполнение программы.

Фирма SUN Microsystems — создатель технологии Java — бесплатно распространяет набор необходимых программных инструментов для полного цикла работы с этим языком программирования: компиляции, интерпретации, отладки, включающий и богатую библиотеку классов, под названием JDK (Java Development Kit). Есть наборы инструментальных программ и других фирм. Например, большой популярностью пользуется JDK фирмы IBM.

Что такое JDK

Набор программ и классов JDK содержит:

- компилятор `javac` из исходного текста в байт-коды;
- интерпретатор `java`, содержащий реализацию JVM;

- облегченный интерпретатор `jre` (в последних версиях отсутствует);
- программу просмотра апплетов `appletviewer`, заменяющую браузер;
- отладчик `jdb`;
- дизассемблер `javap`;
- программу архивации и сжатия `jar`;
- программу сбора документации `javadoc`;
- программу `javah` генерации заголовочных файлов языка C;
- программу `javakey` добавления электронной подписи;
- программу `native2ascii`, преобразующую бинарные файлы в текстовые;
- программы `rmic` и `rmiregistry` для работы с удаленными объектами;
- программу `serialver`, определяющую номер версии класса;
- библиотеки и заголовочные файлы "родных" методов;
- библиотеку классов Java API (Application Programming Interface).

В прежние версии JDK включались и отладочные варианты исполнимых программ: `javac_g`, `java_g` и т. д.

Компания SUN Microsystems постоянно развивает и обновляет JDK, каждый год появляются новые версии.

В 1996 г. была выпущена первая версия JDK 1.0, которая модифицировалась до версии с номером 1.0.2. В этой версии библиотека классов Java API содержала 8 пакетов. Весь набор JDK 1.0.2 поставлялся в упакованном виде в одном файле размером около 5 Мбайт, а после распаковки занимал около 8 Мбайт на диске.

В 1997 г. появилась версия JDK 1.1, последняя ее модификация, 1.1.8, выпущена в 1998 г. В этой версии было 23 пакета классов, занимала она 8,5 Мбайт в упакованном виде и около 30 Мбайт на диске.

В первых версиях JDK все пакеты библиотеки Java API были упакованы в один архивный файл classes.zip и вызывались непосредственно из этого архива, его не нужно распаковывать.

Затем набор инструментальных средств JDK был сильно переработан.

Версия JDK 1.2 вышла в декабре 1998 г. и содержала уже 57 пакетов классов. В архивном виде это файл размером почти 20 Мбайт и еще отдельный файл размером более 17 Мбайт с упакованной документацией. Полная версия располагается на 130 Мбайтах дискового пространства, из них около 80 Мбайт занимает документация.

Начиная с этой версии, все продукты технологии Java собственного производства компания SUN стала называть *Java 2 Platform, Standard Edition*, сокращенно J2SE, а JDK переименовала в *Java 2 SDK, Standard Edition* (Soft-

ware Development Kit), сокращенно J2SDK, поскольку выпускается еще *Java 2 SDK Enterprise Edition* и *Java 2 SDK Micro Edition*. Впрочем, сама компания SUN часто пользуется и старым названием, а в литературе утвердилось название Java 2. Кроме 57 пакетов классов, обязательных на любой платформе и получивших название *Core API*, в Java 2 SDK v1.2 входят еще дополнительные пакеты классов, называемые Standard Extension API.

В версии Java 2 SDK SE, v1.3, вышедшей в 2000 г., уже 76 пакетов классов, составляющих Core API. В упакованном виде это файл размером около 30 Мбайт, и еще файл с упакованной документацией размером 23 Мбайта. Все это распаковывается в 210 Мбайт дискового пространства. Эта версия требует процессор Pentium 166 и выше и не менее 32 Мбайт оперативной памяти.

В настоящее время версия JDK 1.0.2 уже не используется. Версия JDK 1.1.5 с графической библиотекой AWT встроена в популярные браузеры Internet Explorer 5.0 и Netscape Communicator 4.7, поэтому она применяется для создания апплетов. Технология Java 2 широко используется на серверах и в клиент-серверных системах.

Кроме JDK, компания SUN отдельно распространяет еще и набор JRE (Java Runtime Environment).

Что такое JRE

Набор программ и пакетов классов JRE содержит все необходимое для выполнения байт-кодов, в том числе интерпретатор `java` (в прежних версиях облегченнный интерпретатор `jre`) и библиотеку классов. Это часть JDK, не содержащая компиляторы, отладчики и другие средства разработки. Именно JRE или его аналог других фирм содержится в браузерах, умеющих выполнять программы на Java, операционных системах и системах управления базами данных.

Хотя JRE входит в состав JDK, фирма SUN распространяет этот набор и отдельным файлом.

Версия JRE 1.3.0 — это архивный файл размером около 8 Мбайт, разворачивающийся в 20 Мбайт на диске.

Как установить JDK

Набор JDK упаковывается в самораспаковывающийся архив. Раздобыв каким-либо образом этот архив: "выкачав" из Internet, с **http://java.sun.com /products/jdk/** или какого-то другого адреса, получив компакт-диск, вам остается только запустить файл с архивом на выполнение. Откроется окно установки, в котором среди всего прочего вам будет предложено выбрать ката-

лог (directory) установки, например, C:\jdk1.3. Если вы согласитесь с предлагаемым каталогом, то вам больше не о чем беспокоиться. Если вы указали собственный каталог, то проверьте после установки значение переменной `PATH`, набрав в командной строке окна **MS-DOS Prompt** (или окна **Command Prompt** в Windows NT/2000, а тот, кто работает в UNIX, сами знают, что делать) команду `set`. Переменная `PATH` должна содержать полный путь к подкаталогу bin этого каталога. Если нет, то добавьте этот путь, например, C:\jdk1.3\bin. Надо определить и специальную переменную `CLASSPATH`, содержащую пути к архивным файлам и каталогам с библиотеками классов. Системные библиотеки Java 2 подключаются автоматически, без переменной `CLASSPATH`.

Еще одно предупреждение: не следует распаковывать zip- и jar-архивы.

После установки вы получите каталог с названием, например, jdk1.3, а в нем подкаталоги:

- bin, содержащий исполнимые файлы;
- demo, содержащий примеры программ;
- docs, содержащий документацию, если вы ее установили;
- include, содержащий заголовочные файлы "родных" методов;
- jre, содержащий набор JRE;
- old-include, для совместимости со старыми версиями;
- lib, содержащий библиотеки классов и файлы свойств;
- src, с исходными текстами программ JDK. В новых версиях вместо каталога имеется упакованный файл src.jar.

Да-да! Набор JDK содержит исходные тексты большинства своих программ, написанные на Java. Это очень удобно. Вы всегда можете в точности узнать, как работает тот или иной метод обработки информации из JDK, посмотрев исходный код данного метода. Это очень полезно и для изучения Java на "живых" работающих примерах.

Как использовать JDK

Несмотря на то, что набор JDK предназначен для создания программ, работающих в графических средах, таких как MS Windows или X Window System, он ориентирован на выполнение из командной строки окна **MS-DOS Prompt** в Windows 95/98/ME или окна **Command Prompt** в Windows NT/2000. В системах UNIX можно работать и в текстовом режиме и в окне **Xterm**.

Написать программу на Java можно в любом текстовом редакторе, например, Notepad, WordPad в MS Windows, редакторах vi, emacs в UNIX. Надо только сохранить файл в текстовом формате и дать ему расширение java.

Пусть для примера, именем файла будет MyProgram.java, а сам файл сохранен в текущем каталоге.

После создания этого файла из командной строки вызывается компилятор `javac` и ему передается исходный файл как параметр:

```
javac MyProgram.java
```

Компилятор создает в том же каталоге по одному файлу на каждый класс, описанный в программе, называя каждый файл именем класса с расширением class. Допустим, в нашем примере имеется только один класс, названный `MyProgram`, тогда получаем файл с именем MyProgram.class, содержащий байт-коды.

Компилятор молчалив — если компиляция прошла успешно, он ничего не сообщит, на экране появится только приглашение операционной системы. Если же компилятор заметит ошибки, то он выведет на экран сообщения о них. Большое достоинство компилятора JDK в том, что он "отлавливает" много ошибок и выдает подробные и понятные сообщения о них.

Далее из командной строки вызывается интерпретатор байт-кодов `java`, которому передается файл с байт-кодами, причем его имя записывается без расширения (смысл этого вы узнаете позднее):

```
java MyProgram
```

На экране появляется вывод результатов работы программы или сообщения об ошибках времени выполнения.

Если работа из командной строки, столь милая сердцу "юниксоидов", кажется вам несколько устаревшей, используйте для разработки интегрированную среду.

Интегрированные среды Java

Сразу же после создания Java, уже в 1996 г., появились интегрированные среды разработки программ для Java, и их число все время возрастает. Некоторые из них являются просто интегрированными оболочками над JDK, вызывающими из одного окна текстовый редактор, компилятор и интерпретатор. Эти интегрированные среды требуют предварительной установки JDK. Другие содержат JDK в себе или имеют собственный компилятор, например, Java Workshop фирмы SUN Microsystems, JBuilder фирмы Inprise, Visual Age for Java фирмы IBM и множество других программных продуктов. Их можно устанавливать, не имея под руками JDK. Надо заметить, что перечисленные продукты написаны полностью на Java.

Большинство интегрированных сред являются средствами визуального программирования и позволяют быстро создавать пользовательский интерфейс, т е. относятся к классу средств RAD (Rapid Application Development).

Выбор какого-либо средства разработки диктуется, во-первых, возможностями вашего компьютера, ведь визуальные среды требуют больших ресурсов, во-вторых, личным вкусом, в-третьих, уже после некоторой практики, достоинствами компилятора, встроенного в программный продукт.

В России по традиции, идущей от TurboPascal к Delphi, большой популярностью пользуется JBuilder, позволяющий подключать сразу несколько JDK разных версий и использовать их компиляторы кроме собственного. Многие профессионалы предпочитают Visual Age for Java, в котором можно графически установить связи между объектами.

К технологии Java подключились и разработчики CASE-средств. Например, популярный во всем мире продукт Rational Rose может сгенерировать код на Java.

Особая позиция Microsoft

Вы уже, наверное, почувствовали смутное беспокойство, не встречая название этой фирмы. Дело в том, что, имея свою операционную систему, огромное число приложений к ней и богатейшую библиотеку классов, компания Microsoft не имела нужды в Java. Но и пройти мимо технологии, распространившейся всюду, компания Microsoft не могла и создала свой компилятор Java, а также визуальное средство разработки, включив его в Visual Studio. Этот компилятор включает в байт-коды вызовы объектов ActiveX. Следовательно, выполнять эти байт-коды можно только на компьютерах, имеющих доступ к ActiveX. Эта "нечистая" Java резко ограничивает круг применения байт-кодов, созданных компилятором фирмы Microsoft. В результате судебных разбирательств с SUN Microsystems компания Microsoft назвала свой продукт Visual J++. Виртуальная машина Java фирмы Microsoft умеет выполнять байт-коды, созданные "чистым" компилятором, но не всякий интерпретатор выполнит байт-коды, написанные с помощью Visual J++.

Чтобы прекратить появление несовместимых версий Java, фирма SUN разработала концепцию "чистой" Java, назвав ее *Pure Java*, и систему проверочных тестов на "чистоту" байт-кодов. Появились байт-коды, успешно прошедшие тесты, и средства разработки, выдающие "чистый" код и помеченные как *"100% Pure Java"*.

Кроме того, фирма SUN распространяет пакет программ Java Plug-in, который можно подключить к браузеру, заменив тем самым встроенный в браузер JRE на "родной".

Java в Internet

Разработанная для применения в сетях, Java просто не могла не найти отражения на сайтах Internet. Действительно, масса сайтов полностью посвяще-

на или содержит информацию о технологии Java. Одна только фирма SUN содержит несколько сайтов с информацией о Java:

- **http://www.sun.com/** — здесь все ссылки, отсюда можно скопировать JDK;
- **http://java.sun.com/** — основной сайт Java, отсюда тоже можно скопировать JDK;
- **http://developer.java.sun.com/** — масса полезных вещей для разработчика;
- **http://industry.java.sun.com/** — новости технологии Java;
- **http://www.javasoft.com/** — сайт фирмы Javasoft, подразделения SUN;
- **http://www.gamelan.com/**.

На сайте фирмы IBM есть большой раздел **http://www.ibm.com/developer/java/**, где можно найти очень много полезного для программиста.

Компания Microsoft содержит информацию о Java на своем сайте: **http://www.microsoft.com/java/**.

Большой вклад в развитие технологии Java вносит корпорация Oracle: **http://www.oracle.com/**.

Существует множество специализированных сайтов:

- **http://java.iba.com.by/** — Java team IBA (Белоруссия);
- **http://www.artima.com/**;
- **http://www.freewarejava.com/**;
- **http://www.jars.com/** — Java Review Service;
- **http://www.javable.com** — русскоязычный сайт;
- **http://www.javaboutique.com/**;
- **http://www.javalobby.com/**;
- **http://www.javalogy.com/**;
- **http://www.javaranch.com/**;
- **http://www.javareport.com/** — независимый источник информации для разработчиков;
- **http://www.javaworld.com** — электронный журнал;
- **http://www.jfind.com/** — сборник программ и статей;
- **http://www.jguru.com/** — советы специалистов;
- **http://www.novocode.com/**;
- **http://www.sigs.com/jro/** — Java Report Online;
- **http://www.sys-con.com/java/**;
- **http://theserverside.com/** — вопросы создания серверных Java-приложений;

- **http://servlets.chat.ru/**;
- **http://javapower.da.ru/** — собрание FAQ на русском языке;
- **http://www.purejava.ru/**;
- **http:/java7.da.ru/**;
- **http://codeguru.earthweb.com/java/** — большой сборник апплетов и других программ;
- **http://securingjava.com/** — обсуждаются вопросы безопасности;
- **http://www.servlets.com/** — вопросы по написанию апплетов;
- **http://www.servletsource.com/**;
- **http://coolservlets.com/**;
- **http://www.servletforum.com/**;
- **http://www.javacats.com/**.

Персональные сайты:

- **http://www.bruceeckel.com/** — сайт Bruce Eckel;
- **http://www.davidreilly.com/java/**;
- **http://www.comita.spb.ru/users/sergeya/java/** — хозяин, Сергей Астахов, собрал здесь буквально все, касающееся русификации Java.

К сожалению, адреса сайтов часто меняются. Возможно, вы и не найдете некоторые из перечисленных сайтов, зато возникнет много других.

Литература по Java

Перечислим здесь только основные, официальные и почти официальные издания, более полное описание чрезвычайно многочисленной литературы дано в списке литературы в конце книги.

Полное и строгое описание языка изложено в книге *The Java Language Specification, Second Edition. James Gosling, Bill Joy, Guy Steele, Gilad Bracha.* Эта книга в электронном виде находится по адресу **http://java.sun.com/docs /books/jls/second_edition/html/j.title.doc.html** и занимает в упакованном виде около 400 Кбайт.

Столь же полное и строгое описание виртуальной машины Java изложено в книге *The Java Virtual Machine Specification, Second Edition. Tim Lindholm, Frank Yellin.* В электронном виде она находится по адресу **http://java.sun.com /docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html**.

Здесь же необходимо отметить книгу "отца" технологии Java Джеймса Гослинга, написанную вместе с Кеном Арнольдом. Имеется русский перевод Гослинг Дж., Арнольд К. Язык программирования Java: Пер. с англ. — СПб.: Питер, 1997. — 304 с.: ил.

Компания SUN Microsystems содержит на своем сайте постоянно обновляемый электронный учебник Java Tutorial, размером уже более 14 Мбайт: **http://java.sun.com/docs/books/tutorial/**. Время от времени появляется его печатное издание *The Java Tutorial, Second Edition: Object-Oriented Programming for the Internet. Mary Campione, Kathy Walrath.*

Полное описание Java API содержится в документации, но есть печатное издание *The Java Application Programming Interface. James Gosling, Frank Yellin and the Java Team, Volume 1: Core Packages*; *Volume 2: Window Toolkit and Applets.*

Благодарности

Автор рад воспользоваться представившейся возможностью, чтобы поблагодарить всех, принявших участие в выпуске этой книги.

Отдельная благодарность Евгению Рыбакову, предложившему ее издать и так быстро оформившему договор, что автор не успел передумать; моим студентам с их бесконечными вопросами; всем "фидошникам" эхо-конференции RU.JAVA и, особенно, Вячеславу Педаку и Сергею Астахову — настоящим мастерам Java; своим друзьям — "сишникам", убежденным в том, что "Жаба — это отстой", и сыну, Камилю, для которого эта книга, собственно, и писалась.

Часть I

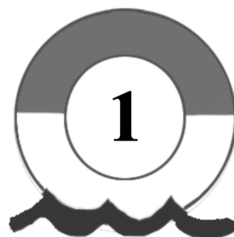
Базовые конструкции языка Java

Глава 1. Встроенные типы данных, операции над ними

Глава 2. Объектно-ориентированное программирование в Java

Глава 3. Пакеты и интерфейсы

ГЛАВА 1



Встроенные типы данных, операции над ними

Приступая к изучению нового языка, полезно поинтересоваться, какие исходные данные могут обрабатываться средствами этого языка, в каком виде их можно задавать, и какие стандартные средства обработки этих данных заложены в язык. Это довольно скучное занятие, поскольку в каждом развитом языке программирования множество типов данных и еще больше правил их использования. Однако несоблюдение этих правил приводит к появлению скрытых ошибок, обнаружить которые иногда бывает очень трудно. Ну что же, в каждом ремесле приходится сначала "играть гаммы", и мы не можем от этого уйти.

Все правила языка Java исчерпывающе изложены в его спецификации, сокращенно называемой JLS. Иногда, чтобы понять, как выполняется та или иная конструкция языка Java, приходится обращаться к спецификации, но, к счастью, это бывает редко, правила языка Java достаточно просты и естественны.

В этой главе перечислены примитивные типы данных, операции над ними, операторы управления, и показаны "подводные камни", которых следует избегать при их использовании. Но начнем, по традиции, с простейшей программы.

Первая программа на Java

По давней традиции, восходящей к языку C, учебники по языкам программирования начинаются с программы "Hello, World!". Не будем нарушать эту традицию. В листинге 1.1 эта программа в самом простом виде, записанная на языке Java.

Листинг 1.1. Первая программа на языке Java

```
class HelloWorld{
public static void main(String[] args){
 System.out.println("Hello, XXI Century World!");
}
}
```

Вот и все, всего пять строчек! Но даже на этом простом примере можно заметить целый ряд существенных особенностей языка Java.

- Всякая программа представляет собой один или несколько классов, в этом простейшем примере только один *класс* (class).
- Начало класса отмечается служебным словом `class`, за которым следует имя класса, выбираемое произвольно, в данном случае `HelloWorld`. Все, что содержится в классе, записывается в фигурных скобках и составляет *тело класса* (class body).
- Все действия производятся с помощью методов обработки информации, коротко говорят просто *метод* (method). Это название употребляется в языке Java вместо названия "функция", применяемого в других языках.
- Методы различаются по именам. Один из методов обязательно должен называться `main`, с него начинается выполнение программы. В нашей простейшей программе только один метод, а значит, имя ему `main`.
- Как и положено функции, метод всегда выдает в результате (чаще говорят, *возвращает* (returns)) только одно значение, тип которого обязательно указывается перед именем метода. Метод может и не возвращать никакого значения, играя роль процедуры, как в нашем случае. Тогда вместо типа возвращаемого значения записывается слово `void`, как это и сделано в примере.
- После имени метода в скобках, через запятую, перечисляются *аргументы* (arguments) или *параметры* метода. Для каждого аргумента указывается его тип и, через пробел, имя. В примере только один аргумент, его тип — массив, состоящий из строк символов. Строка символов — это встроенный в Java API тип `String`, а квадратные скобки — признак массива. Имя массива может быть произвольным, в примере выбрано имя `args`.
- Перед типом возвращаемого методом значения могут быть записаны *модификаторы* (modifiers). В примере их два: слово `public` означает, что этот метод доступен отовсюду; слово `static` обеспечивает возможность вызова метода `main()` в самом начале выполнения программы. Модификаторы вообще необязательны, но для метода `main()` они необходимы.

Замечание

В тексте этой книги после имени метода ставятся скобки, чтобы подчеркнуть, что это имя именно метода, а не простой переменной.

- Все, что содержит метод, *тело метода* (method body), записывается в фигурных скобках.

Единственное действие, которое выполняет метод `main()` в примере, заключается в вызове другого метода со сложным именем `System.out.println` и передаче ему на обработку одного аргумента, текстовой константы `"Hello, 21th Century World!"`. Текстовые константы записываются в кавычках, которые являются только ограничителями и не входят в состав текста.

Составное имя `System.out.println` означает, что в классе `System`, входящем в Java API, определяется переменная с именем `out`, содержащая экземпляры одного из классов Java API, класса `PrintStream`, в котором есть метод `println()`. Все это станет ясно позднее, а пока просто будем писать это длинное имя.

Действие метода `println()` заключается в выводе своего аргумента в выходной поток, связанный обычно с выводом на экран текстового терминала, в окно **MS-DOS Prompt** или **Command Prompt** или **Xterm**, в зависимости от вашей системы. После вывода курсор переходит на начало следующей строки экрана, на что указывает окончание `ln`, слово println — сокращение слов print line. В составе Java API есть и метод `print()`, оставляющий курсор в конце выведенной строки. Разумеется, это прямое влияние языка Pascal.

Сделаем сразу важное замечание. Язык Java различает строчные и прописные буквы, имена `main`, `Main`, `MAIN` различны с "точки зрения" компилятора Java. В примере важно писать `String`, `System` с заглавной буквы, а `main` с маленькой. Но внутри текстовой константы неважно, писать `Century` или `century`, компилятор вообще не "смотрит" на нее, разница будет видна только на экране.

Замечание

Язык Java различает прописные и строчные буквы.

Свои имена можно записывать как угодно, можно было бы дать классу имя `helloworld` или `helloWorld`, но между Java-программистами заключено соглашение, называемое "Code Conventions for the Java Programming Language", хранящееся по адресу **http://java.sun.com/docs/codeconv/index.html**. Вот несколько пунктов этого соглашения:

- имена классов начинаются с прописной буквы; если имя содержит несколько слов, то каждое слово начинается с прописной буквы;
- имена методов и переменных начинаются со строчной буквы; если имя содержит несколько слов, то каждое следующее слово начинается со строчной буквы;

- ❒ имена констант записываются полностью прописными буквами; если имя состоит из нескольких слов, то между ними ставится знак подчеркивания.

Конечно, эти правила необязательны, хотя они и входят в JLS, п. 6.8, но сильно облегчают понимание кода и придают программе характерный для Java стиль.

Стиль определяют не только имена, но и размещение текста программы по строкам, например, расположение фигурных скобок: оставлять ли открывающую фигурную скобку в конце строки с заголовком класса или метода или переносить на следующую строку? Почему-то этот пустяшный вопрос вызывает ожесточенные споры, некоторые средства разработки, например JBuilder, даже предлагают выбрать определенный стиль расстановки фигурных скобок. Многие фирмы устанавливают свой внутрифирменный стиль. В книге мы постараемся следовать стилю "Code Conventions" и в том, что касается разбиения текста программы на строки (компилятор же рассматривает всю программу как одну длинную строку, для него программа — это просто последовательность символов), и в том, что касается отступов (indent) в тексте.

Итак, программа написана в каком-либо текстовом редакторе, например, Notepad. Теперь ее надо сохранить в файле, имя которого совпадает с именем класса, содержащего метод `main()`, и дать имени файла расширение java. Это правило очень желательно выполнять. При этом система исполнения Java будет быстро находить метод `main()` для начала работы, просто отыскивая класс, совпадающий с именем файла.

Совет

Называйте файл с программой именем класса, содержащего метод `main()`, соблюдая регистр букв.

В нашем примере, сохраним программу в файле с именем HelloWorld.java в текущем каталоге. Затем вызовем компилятор, передавая ему имя файла в качестве аргумента:

```
javac HelloWorld.java
```

Компилятор создаст файл с байт-кодами, даст ему имя `HelloWorld.class` и запишет этот файл в текущий каталог.

Осталось вызвать интерпретатор, передав ему в качестве аргумента имя класса (а не файла):

```
java HelloWorld
```

На экране появится:

```
Hello, 21st Century World!
```

Замечание

Не указывайте расширение class при вызове интерпретатора.

На рис. 1.1 показано, как все это выглядит в окне **Command Prompt** операционной системы MS Windows 2000.

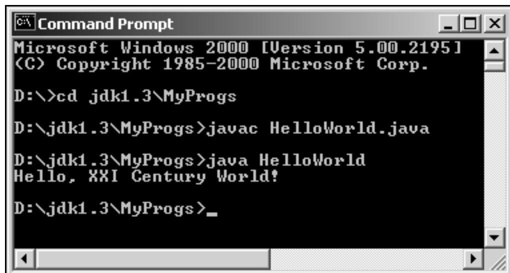


Рис. 1.1. Окно **Command Prompt**

При работе в интегрированной среде все эти действия вызываются выбором соответствующих пунктов меню или "горячими" клавишами — единых правил здесь нет.

Комментарии

В текст программы можно вставить комментарии, которые компилятор не будет учитывать. Они очень полезны для пояснений по ходу программы. В период отладки можно выключать из действий один или несколько операторов, пометив их символами комментария, как говорят программисты, "закомментарив" их. Комментарии вводятся таким образом:

- за двумя наклонными чертами подряд `//`, без пробела между ними, начинается комментарий, продолжающийся до конца строки;
- за наклонной чертой и звездочкой `/*` начинается комментарий, который может занимать несколько строк, до звездочки и наклонной черты `*/` (без пробелов между этими знаками).

Комментарии очень удобны для чтения и понимания кода, они превращают программу в документ, описывающий ее действия. Программу с хорошими комментариями называют *самодокументированной.* Поэтому в Java введены комментарии третьего типа, а в состав JDK — программа `javadoc`, извлекающая эти комментарии в отдельные файлы формата HTML и создающая гиперссылки между ними: за наклонной чертой и двумя звездочками подряд, без пробелов, `/**` начинается комментарий, который может занимать несколько строк до звездочки (одной) и наклонной черты `*/` и обрабатываться программой `javadoc`. В такой комментарий можно вставить указания программе `javadoc`, которые начинаются с символа `@`.

Именно так создается документация к JDK.

Добавим комментарии к нашему примеру (листинг 1.2).

Листинг 1.2. Первая программа с комментариями

```
/**
* Разъяснение содержания и особенностей программы...
* @author Имя Фамилия (автора)
* @version 1.0 (это версия программы)
*/
class HelloWorld{                            // HelloWorld — это только имя
  // Следующий метод начинает выполнение программы
  public static void main(String[] args){  // args не используются
    /* Следующий метод просто выводит свой аргумент
     *  на экран дисплея */
    System.out.println("Hello, 21st Century World!");
    // Следующий вызов закомментирован,
    // метод не будет выполняться
    // System.out.println("Farewell, 20th Century!");
  }
}
```

Звездочки в начале строк не имеют никакого значения, они написаны просто для выделения комментария. Пример, конечно, перегружен пояснениями (это плохой стиль), здесь просто показаны разные формы комментариев.

Константы

В языке Java можно записывать константы разных типов в разных видах. Перечислим их.

Целые

Целые константы можно записывать в трех системах счисления:

- в десятичной форме: `+5`, `-7`, `12345678`;
- в восьмеричной форме, начиная с нуля: `027`, `-0326`, `0777`; в записи таких констант недопустимы цифры 8 и 9;

Замечание

Число, начинающееся с нуля, записано в восьмеричной форме, а не в десятичной.

- в шестнадцатеричной форме, начиная с нуля и латинской буквы `x` или `X`: `0xff0a`, `0xFC2D`, `0X45a8`, `0X77FF`; здесь строчные и прописные буквы не различаются.

Целые константы хранятся в формате типа int (см. ниже).

В конце целой константы можно записать букву прописную L или строчную l, тогда константа будет сохраняться в длинном формате типа long (см. ниже): +25L, -0371, 0xffL, 0XDFDFl.

Совет

Не используйте при записи длинных целых констант строчную латинскую букву l, ее легко спутать с единицей.

Действительные

Действительные константы записываются только в десятичной системе счисления в двух формах:

- с фиксированной точкой: 37.25, -128.678967, +27.035;
- с плавающей точкой: 2.5e34, -0.345e-25, 37.2E+4; можно писать строчную или прописную латинскую букву E; пробелы и скобки недопустимы.

В конце действительной константы можно поставить букву F или f, тогда константа будет сохраняться в формате типа float (см. ниже): 3.5f, -45.67F, 4.7e-5f. Можно приписать и букву D (или d): 0.045D, -456.77889d, означающую тип double, но это излишне, поскольку действительные константы и так хранятся в формате типа double.

Символы

Для записи одиночных символов используются следующие формы.

- Печатные символы можно записать в апострофах: 'a', 'N', '?'.
- Управляющие символы записываются в апострофах с обратной наклонной чертой:
  - '\n' — символ перевода строки newline с кодом ASCII 10;
  - '\r' — символ возврата каретки CR с кодом 13;
  - '\f' — символ перевода страницы FF с кодом 12;
  - '\b' — символ возврата на шаг BS с кодом 8;
  - '\t' — символ горизонтальной табуляции HT с кодом 9;
  - '\\' — обратная наклонная черта;
  - '\"' — кавычка;
  - '\'' — апостроф.
- Код любого символа с десятичной кодировкой от 0 до 255 можно задать, записав его не более чем тремя цифрами в восьмеричной системе счис-

ления в апострофах после обратной наклонной черты: `'\123'` — буква `S`, `'\346'` — буква `ж` в кодировке CP1251. Не рекомендуется использовать эту форму записи для печатных и управляющих символов, перечисленных в предыдущем пункте, поскольку компилятор сразу же переведет восьмеричную запись в указанную выше форму. Наибольший код `'\377'` — десятичное число 255.

- Код любого символа в кодировке Unicode набирается в апострофах после обратной наклонной черты и латинской буквы `u` ровно четырьмя шестнадцатеричными цифрами: `'\u0053'` — буква `S`, `'\u0416'` — буква `Ж`.

Символы хранятся в формате типа `char` (см. ниже).

Примечание

Прописные русские буквы в кодировке Unicode занимают диапазон от `'\u0410'` — заглавная буква `А`, до `'\u042F'` — заглавная `Я`, строчные буквы от `'\u0430'` — `а`, до `'\044F'` — `я`.

В какой бы форме ни записывались символы, компилятор переводит их в Unicode, включая и исходный текст программы.

Замечание

Компилятор и исполняющая система Java работают только с кодировкой Unicode.

Строки

Строки символов заключаются в кавычки. Управляющие символы и коды записываются в строках точно так же, с обратной наклонной чертой, но, разумеется, без апострофов, и оказывают то же действие. Строки могут располагаться только на одной строке исходного кода, нельзя открывающую кавычку поставить на одной строке, а закрывающую — на следующей.

Вот некоторые примеры:

```
"Это строка\nс переносом"
"\"Спартак\" — Чемпион!"
```

Замечание

Строки символов нельзя начинать на одной строке исходного кода, а заканчивать на другой.

Для строковых констант определена операция сцепления, обозначаемая плюсом.

`"Сцепление " + "строк"` дает в результате строку `"Сцепление строк"`.

Чтобы записать длинную строку в виде одной строковой константы, надо после закрывающей кавычки на первой и следующих строках поставить

плюс +; тогда компилятор соберет две (или более) строки в одну строковую константу, например:

```
"Одна строковая константа, записанная "+
"на двух строках исходного текста"
```

Тот, кто попытается выводить символы в кодировке Unicode, например, слово "Россия":

```
System.out.println("\u0429\u043e\u0441\u0441\u0438\u044f");
```

должен знать, что Windows 95/98/ME вообще не работает с Unicode, а Windows NT/2000 использует для вывода в окно **Command Prompt** шрифт Terminal, в котором русские буквы расположены в начальных кодах Unicode, почему-то в кодировке CP866, и разбросаны по другим сегментам Unicode.

Не все шрифты Unicode содержат начертания (glyphs) всех символов, поэтому будьте осторожны при выводе строк в кодировке Unicode.

Совет

Используйте Unicode напрямую только в крайних случаях.

Имена

Имена (names) переменных, классов, методов и других объектов могут быть простыми (общее название — *идентификаторы* (idenifiers)) и *составными* (qualified names). Идентификаторы в Java составляются из так называемых *букв Java* (Java letters) и арабских цифр 0—9, причем первым символом идентификатора не может быть цифра. (Действительно, как понять запись 2e3: как число 2000,0 или как имя переменной?) В число букв Java обязательно входят прописные и строчные латинские буквы, знак доллара $ и знак подчеркивания _, а так же символы национальных алфавитов.

Замечание

Не указывайте в именах знак доллара. Компилятор Java использует его для записи имен вложенных классов.

Вот примеры правильных идентификаторов:

```
a1       my_var     var3_5    _var    veryLongVarName
aName    theName    a2Vh36kBnMt456dX
```

В именах лучше не использовать строчную букву l, которую легко спутать с единицей, и букву o, которую легко принять за нуль.

Не забывайте о рекомендациях "Code Conventions".

В классе `Character`, входящем в состав Java API, есть два метода, проверяющие, пригоден ли данный символ для использования в идентификаторе: `isJavaIdentifierStart()`, проверяющий, является ли символ буквой Java, и `isJavaIdentifierPart()`, выясняющий, является ли символ буквой или цифрой.

Служебные слова Java, такие как `class`, `void`, `static`, зарезервированы, их нельзя использовать в качестве идентификаторов своих объектов.

Составное имя (qualified name) — это несколько идентификаторов, разделенных точками, без пробелов, например, уже встречавшееся нам имя `System.out.println`.

Примитивные типы данных и операции

Все типы исходных данных, встроенные в язык Java, делятся на две группы: *примитивные типы* (primitive types) и *ссылочные типы* (reference types).

Ссылочные типы делятся на *массивы* (arrays), *классы* (classes) и *интерфейсы* (interfaces).

Примитивных типов всего восемь. Их можно разделить на *логический* (иногда говорят *булев*) тип `boolean` и *числовые* (numeric).

К числовым типам относятся *целые* (integral[1]) и *вещественные* (floating-point) типы.

Целых типов пять: `byte`, `short`, `int`, `long`, `char`.

Символы можно использовать везде, где используется тип `int`, поэтому JLS причисляет их к целым типам. Например, их можно использовать в арифметических вычислениях, скажем, можно написать `2 + 'Ж'`, к двойке будет прибавляться кодировка Unicode `'\u0416'` буквы `'Ж'`. В десятичной форме это число 1046 и в результате сложения получим 1048.

Напомним, что в записи `2 + "Ж"` плюс понимается как сцепление строк, двойка будет преобразована в строку, в результате получится строка `"2Ж"`.

Вещественных типов два: `float` и `double`.

На рис. 1.2 показана иерархия типов данных Java.

Поскольку по имени переменной невозможно определить ее тип, все переменные обязательно должны быть описаны перед их использованием. Описание заключается в том, что записывается имя типа, затем, через пробел, список имен переменных, разделенных запятой. Для всех или некоторых переменных можно указать начальные значения после знака равенства, ко-

---

[1] Название "integral" не является устоявшимся термином. Так названа категория целых типов данных в книге *The Java Language Specification, Second Edition. James Gosling, Bill Joy, Guy Steele, Gilad Bracha (см. введение). — Ред.*

торыми могут служить любые константные выражения того же типа. Описание каждого типа завершается точкой с запятой. В программе может быть сколько угодно описаний каждого типа.

Замечание для специалистов

Java — язык со строгой типизацией (strongly typed language).

Разберем каждый тип подробнее.

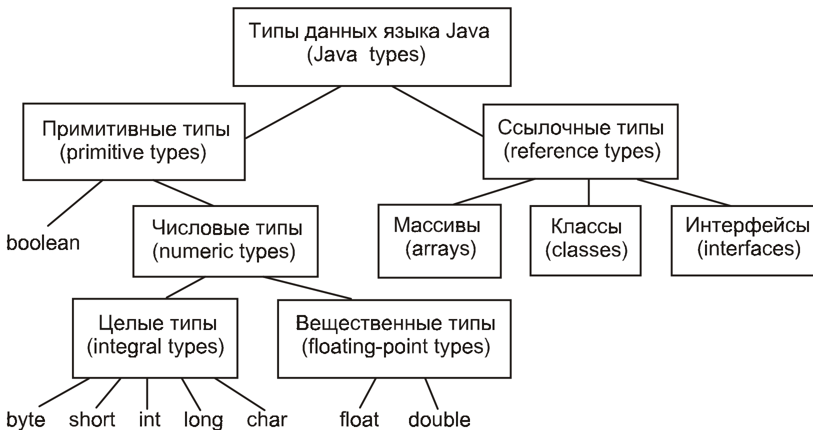


Рис. 1.2. Типы данных языка Java

Логический тип

Значения логического типа `boolean` возникают в результате различных сравнений, вроде `2 > 3`, и используются, главным образом, в условных операторах и операторах циклов. Логических значений всего два: `true` (истина) и `false` (ложь). Это служебные слова Java. Описание переменных этого типа выглядит так:

```
boolean b = true, bb = false, bool2;
```

Над логическими данными можно выполнять операции присваивания, например, `bool2 = true`, в том числе и составные с логическими операциями; сравнение на равенство `b == bb` и на неравенство `b != bb`, а также логические операции.

Логические операции

Логические операции:

- отрицание (NOT) `!` (обозначается восклицательным знаком);
- конъюнкция (AND) `&` (амперсанд);

- дизъюнкция (OR) | (вертикальная черта);
- исключающее ИЛИ (XOR) ^ (каре).

Они выполняются над логическими данными, их результатом будет тоже логическое значение true или false. Про них можно ничего не знать, кроме того, что представлено в табл. 1.1.

Таблица 1.1. *Логические операции*

b1	**b2**	**!b1**	**b1 & b2**	**b1 | b2**	**b1 ^ b2**
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

Словами эти правила можно выразить так:

- отрицание меняет значение истинности;
- конъюнкция истинна, только если оба операнда истинны;
- дизъюнкция ложна, только если оба операнда ложны;
- исключающее ИЛИ истинно, только если значения операндов различны.

Замечание

Если бы Шекспир был программистом, фразу "To be or not to be" он написал бы так: 2b | ! 2b.

Кроме перечисленных четырех логических операций есть еще две логические операции сокращенного вычисления:

- сокращенная конъюнкция (conditional-AND) &&;
- сокращенная дизъюнкция (conditional-OR) ||.

Удвоенные знаки амперсанда и вертикальной черты следует записывать без пробелов.

Правый операнд сокращенных операций вычисляется только в том случае, если от него зависит результат операции, т. е. если левый операнд конъюнкции имеет значение true, или левый операнд дизъюнкции имеет значение false.

Это правило очень удобно и ловко используется, например, можно записывать выражения (n != 0) && (m/n > 0.001) или (n == 0) || (m/n > 0.001) не опасаясь деления на нуль.

Замечание

Практически всегда в Java используются именно сокращенные логические операции.

Целые типы

Спецификация языка Java, JLS, определяет разрядность (количество байтов, выделяемых для хранения значений типа в оперативной памяти) и диапазон значений каждого типа. Для целых типов они приведены в табл. 1.2.

Таблица 1.2. *Целые типы*

Тип	Разрядность (байт)	Диапазон
`byte`	1	от –128 до 127
`short`	2	от –32768 до 32767
`int`	4	от –2147483648 до 2147483647
`long`	8	от –9223372036854775808 до 9223372036854775807
`char`	2	от `'\u0000'` до `'\uFFFF'`, в десятичной форме от 0 до 65535

Впрочем, для Java разрядность не столь важна, на некоторых компьютерах она может отличаться от указанной в таблице, а вот диапазон значений должен выдерживаться неукоснительно.

Хотя тип `char` занимает два байта, в арифметических вычислениях он участвует как тип `int`, ему выделяется 4 байта, два старших байта заполняются нулями.

Примеры определения переменных целых типов:

```
byte b1 = 50, b2 = -99, b3;
short det = 0, ind = 1;
int i = -100, j = 100, k = 9999;
long big = 50, veryBig = 2147483648L;
char c1 = 'A', c2 = '?', newLine = '\n';
```

Целые типы хранятся в двоичном виде с дополнительным кодом. Последнее означает, что для отрицательных чисел хранится не их двоичное представление, а *дополнительный код* этого двоичного представления.

Дополнительный же код получается так: в двоичном представлении все нули меняются на единицы, а единицы на нули, после чего к результату прибавляется единица, разумеется, в двоичной арифметике.

Например, значение 50 переменной b1, определенной выше, будет храниться в одном байте с содержимым 00110010, а значение -99 переменной b2 — в байте с содержимым, которое вычисляем так: число 99 переводим в двоичную форму, получая 01100011, меняем единицы и нули, получая 10011100, и прибавляем единицу, получив окончательно байт с содержимым 10011101.

Смысл всех этих сложностей в том, что сложение числа с его дополнительным кодом в двоичной арифметике даст в результате нуль, старший бит просто теряется. Это означает, что в такой странной арифметике дополнительный код числа является противоположным к нему числом, числом с обратным знаком. А это, в свою очередь, означает, что вместо того, чтобы вычесть из числа A число B, можно к A прибавить дополнительный код числа B. Таким образом, операция вычитания исключается из набора машинных операций.

Над целыми типами можно производить массу операций. Их набор восходит к языку C, он оказался удобным и кочует из языка в язык почти без изменений. Особенности применения этих операций в языке Java показаны на примерах.

Операции над целыми типами

Все операции, которые производятся над целыми числами, можно разделить на следующие группы.

Арифметические операции

К арифметическим операциям относятся:

- сложение + (плюс);
- вычитание - (дефис);
- умножение * (звездочка);
- деление / (наклонная черта — слэш);
- взятие остатка от деления (деление по модулю) % (процент);
- инкремент (увеличение на единицу) ++;
- декремент (уменьшение на единицу) --.

Между сдвоенными плюсами и минусами нельзя оставлять пробелы.

Сложение, вычитание и умножение целых значений выполняются как обычно, а вот деление целых значений в результате дает опять целое (так называемое *"целое деление"*), например, 5/2 даст в результате 2, а не 2.5, а 5/(-3) даст -1. Дробная часть попросту отбрасывается, происходит усечение частного. Это поначалу обескураживает, но потом оказывается удобным для усечения чисел.

Замечание

В Java принято целочисленное деление.

Это странное для математики правило естественно для программирования: если оба операнда имеют один и тот же тип, то и результат имеет тот же тип. Достаточно написать `5/2.0` или `5.0/2` или `5.0/2.0` и получим `2.5` как результат деления вещественных чисел.

Операция *деление по модулю* определяется так: `a % b = a — (a / b) * b`; например, `5%2` даст в результате `1`, а `5%(-3)` даст `2`, т. к. `5 = (-3) * (-1) + 2`, но `(-5)%3` даст `-2`, поскольку `-5 = 3 * (-1) — 2`.

Операции *инкремент* и *декремент* означают увеличение или уменьшение значения переменной на единицу и применяются только к переменным, но не к константам или выражениям, нельзя написать `5++` или `(a + b)++`.

Например, после приведенных выше описаний `i++` даст `-99`, а `j--` даст `99`.

Интересно, что эти операции можно записать и перед переменной: `++i`, `--j`. Разница проявится только в выражениях: при первой форме записи *(постфиксной)* в выражении участвует старое значение переменной и только потом происходит увеличение или уменьшение ее значения. При второй форме записи *(префиксной)* сначала изменится переменная и ее новое значение будет участвовать в выражении.

Например, после приведенных выше описаний, `(k++) + 5` даст в результате `10004`, а переменная `k` примет значение `10000`. Но в той же исходной ситуации `(++k) + 5` даст `10005`, а переменная `k` станет равной `10000`.

Приведение типов

Результат арифметической операции имеет тип `int`, кроме того случая, когда один из операндов типа `long`. В этом случае результат будет типа `long`.

Перед выполнением арифметической операции всегда происходит *повышение* (promotion) типов `byte`, `short`, `char`. Они преобразуются в тип `int`, а может быть, и в тип `long`, если другой операнд типа `long`. Операнд типа `int` повышается до типа `long`, если другой операнд типа `long`. Конечно, числовое значение операнда при этом не меняется.

Это правило приводит иногда к неожиданным результатам. Попытка откомпилировать простую программу, представленную в листинге 1.3, приведет к сообщениям компилятора, показанным на рис. 1.3.

Листинг 1.3. Неверное определение переменной

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

```
    byte b1 = 50, b2 = -99;
    short k = b1 + b2;        // Неверно!
    System.out.println("k=" + k);
  }
}
```

Эти сообщения означают, что в файле InvalidDef.java, в строке 4, обнаружена возможная потеря точности (possible loss of precision). Затем приводятся обнаруженный (found) и нужный (required) типы, выводится строка, в которой обнаружена (а не сделана) ошибка, и отмечается символ, при разборе которого найдена ошибка. Затем указано общее количество обнаруженных (а не сделанных) ошибок (1 error).

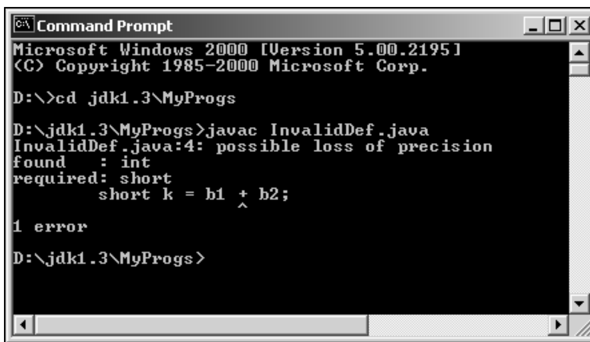


Рис. 1.3. Сообщения компилятора об ошибке

В таких случаях следует выполнить явное приведение типа. В данном случае это будет *сужение* (narrowing) типа `int` до типа `short`. Оно осуществляется операцией явного приведения, которая записывается перед приводимым значением в виде имени типа в скобках. Определение

```
short k = (short)(b1 + b2);
```

будет верным.

Сужение осуществляется просто отбрасыванием старших битов, что необходимо учитывать для больших значений. Например, определение

```
byte b = (byte)300;
```

даст переменной `b` значение `44`. Действительно, в двоичном представлении числа `300`, равном `100101100`, отбрасывается старший бит и получается `00101100`.

Таким же образом можно произвести и явное *расширение* (widening) типа, если в этом есть необходимость.

Если результат целой операции выходит за диапазон своего типа `int` или `long`, то автоматически происходит приведение по модулю, равному длине

этого диапазона, и вычисления продолжаются, переполнение никак не отмечается.

Замечание

В языке Java нет целочисленного переполнения.

Операции сравнения

В языке Java шесть обычных операций сравнения целых чисел по величине:

- больше >;
- меньше <;
- больше или равно >=;
- меньше или равно <=;
- равно ==;
- не равно `!=`.

Сдвоенные символы записываются без пробелов, их нельзя переставлять местами, запись => будет неверной.

Результат сравнения — логическое значение: `true`, в результате, например, сравнения `3 != 5`; или `false`, например, в результате сравнения `3 == 5`.

Для записи сложных сравнений следует привлекать логические операции. Например, в вычислениях часто приходится делать проверки вида `a < x < b`. Подобная запись на языке Java приведет к сообщению об ошибке, поскольку первое сравнение, `a < x`, даст `true` или `false`, а Java не знает, больше это, чем `b`, или меньше. В данном случае следует написать выражение `(a < x) && (x < b)`, причем здесь скобки можно опустить, написать просто `a < x && x < b`, но об этом немного позднее.

Побитовые операции

Иногда приходится изменять значения отдельных битов в целых данных. Это выполняется с помощью побитовых (bitwise) операций путем наложения маски. В языке Java есть четыре побитовые операции:

- дополнение (complement) `~` (тильда);
- побитовая конъюнкция (bitwise AND) `&`;
- побитовая дизъюнкция (bitwise OR) `|`;
- побитовое исключающее ИЛИ (bitwise XOR) `^`.

Они выполняются поразрядно, после того как оба операнда будут приведены к одному типу `int` или `long`, так же как и для арифметических операций, а значит, и к одной разрядности. Операции над каждой парой битов выполняются согласно табл. 1.3.

Таблица 1.3. Побитовые операции

n1	n2	~n1	n1 & n2	n1 | n2	n1 ^ n2
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

В нашем примере `b1 == 50`, двоичное представление `00110010`, `b2 == -99`, двоичное представление `10011101`. Перед операцией происходит повышение до типа `int`. Получаем представления из 32-х разрядов для `b1` — `0...00110010`, для `b2` — `1...10011101`. В результате побитовых операций получаем:

- `~b2 == 98`, двоичное представление `0...01100010`;
- `b1 & b2 == 16`, двоичное представление `0...00010000`;
- `b1 | b2 == -65`, двоичное представление `1...10111111`;
- `b1 ^ b2 == -81`, двоичное представление `1...10101111`.

Двоичное представление каждого результата занимает 32 бита.

Заметьте, что дополнение `~x` всегда эквивалентно `(-x)-1`.

Сдвиги

В языке Java есть три операции сдвига двоичных разрядов:

- сдвиг влево <<;
- сдвиг вправо >>;
- беззнаковый сдвиг вправо >>>.

Эти операции своеобразны тем, что левый и правый операнды в них имеют разный смысл. Слева стоит значение целого типа, а правая часть показывает, на сколько двоичных разрядов сдвигается значение, стоящее в левой части.

Например, операция `b1 << 2` сдвинет влево на `2` разряда предварительно повышенное значение `0...00110010` переменной `b1`, что даст в результате `0...011001000`, десятичное 200. Освободившиеся справа разряды заполняются нулями, левые разряды, находящиеся за 32-м битом, теряются.

Операция `b2 << 2` сдвинет повышенное значение `1...10011101` на два разряда влево. В результате получим `1...1001110100`, десятичное значение −396.

Заметьте, что сдвиг влево на `n` разрядов эквивалентен умножению числа на `2` в степени `n`.

Операция `b1 >> 2` даст в результате `0...00001100`, десятичное 12, а `b2 >> 2` — результат `1...11100111`, десятичное −25, т. е. слева распространяется старший бит, правые биты теряются. Это так называемый *арифметический сдвиг*.

Операция беззнакового сдвига во всех случаях ставит слева на освободившиеся места нули, осуществляя *логический сдвиг*. Но вследствие предварительного повышения это имеет эффект только для нескольких старших разрядов отрицательных чисел. Так, `b2 >>> 2` имеет результатом `001...100111`, десятичное число 1 073 741 799.

Если же мы хотим получить логический сдвиг исходного значения `10011101` переменной `b2`, т. е., `0...00100111`, надо предварительно наложить на `b2` *маску*, обнулив старшие биты: `(b2 & 0xFF) >>> 2`.

Замечание

Будьте осторожны при использовании сдвигов вправо.

Вещественные типы

Вещественных типов в Java два: `float` и `double`. Они характеризуются разрядностью, диапазоном значений и точностью представления, отвечающим стандарту IEEE 754-1985 с некоторыми изменениями. К обычным вещественным числам добавляются еще три значения.

1. Положительная бесконечность, выражаемая константой `POSITIVE_INFINITY` и возникающая при переполнении положительного значения, например, в результате операции умножения `3.0*6e307`.
2. Отрицательная бесконечность `NEGATIVE_INFINITY`.
3. "Не число", записываемое константой `NaN` (Not a Number) и возникающее при делении вещественного числа на нуль или умножении нуля на бесконечность.

В *главе 4* мы поговорим о них подробнее.

Кроме того, стандарт различает положительный и отрицательный нуль, возникающий при делении на бесконечность соответствующего знака, хотя сравнение `0.0 == -0.0` дает `true`.

Операции с бесконечностями выполняются по обычным математическим правилам.

Во всем остальном вещественные типы — это обычные вещественные значения, к которым применимы все арифметические операции и сравнения, перечисленные для целых типов. Характеристики вещественных типов приведены в табл. 1.4.

Знатокам C/C++

В языке Java взятие остатка от деления `%`, инкремент `++` и декремент `--` применяются и к вещественным типам.

Таблица 1.4. *Вещественные типы*

Тип	Разрядность	Диапазон	Точность
`float`	4	3,4e–38 < $\lvert x \rvert$ < 3,4e38	7—8 цифр
`double`	8	1,7e–308 < $\lvert x \rvert$ < 1,7e308	17 цифр

Примеры определения вещественных типов:

```
float x = 0.001, y = -34.789;
double z1 = -16.2305, z2;
```

Поскольку к вещественным типам применимы все арифметические операции и сравнения, целые и вещественные значения можно смешивать в операциях. При этом правило приведения типов дополняется такими условиями:

- если в операции один операнд имеет тип `double`, то и другой приводится к типу `double`;
- если один операнд имеет тип `float`, то и другой приводится к типу `float`;
- в противном случае действует правило приведения целых значений.

Операции присваивания

Простая операция присваивания (simple assignment operator) записывается знаком равенства `=`, слева от которого стоит переменная, а справа выражение, совместимое с типом переменной: `x = 3.5`, `y = 2 * (x — 0.567) / (x + 2)`, `b = x < y`, `bb = x >= y && b`.

Операция присваивания действует так: выражение, стоящее после знака равенства, вычисляется и приводится к типу переменной, стоящей слева от знака равенства. Результатом операции будет приведенное значение правой части.

Операция присваивания имеет еще одно, побочное, действие: переменная, стоящая слева, получает приведенное значение правой части, старое ее значение теряется.

В операции присваивания левая и правая части неравноправны, нельзя написать `3.5 = x`. После операции `x = y` изменится переменная `x`, став равной `y`, а после `y = x` изменится `y`.

Кроме простой операции присваивания есть еще 11 *составных* операций присваивания (compound assignment operators): `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `>>>=`. Символы записываются без пробелов, нельзя переставлять их местами.

Все составные операции присваивания действуют по одной схеме:

`x` *`op`*`= a` эквивалентно `x = (`*`тип`*` x)`, т. е. `(x `*`op`*` a)`.

Напомним, что переменная `ind` типа `short` определена у нас со значением `1`. Присваивание `ind += 7.8` даст в результате число `8`, то же значение получит и переменная `ind`. Эта операция эквивалентна простой операции присваивания `ind = (short)(ind + 7.8)`.

Перед присваиванием, при необходимости, автоматически производится приведение типа. Поэтому:

```
byte b = 1;
b = b + 10;      // Ошибка!
b += 10;         // Правильно!
```

Перед сложением `b + 50` происходит повышение `b` до типа `int`, результат сложения тоже будет типа `int` и, в первом случае, не может быть присвоен переменной `b` без явного приведения типа. Во втором случае перед присваиванием произойдет сужение результата сложения до типа `byte`.

Условная операция

Эта своеобразная операция имеет три операнда. Вначале записывается произвольное логическое выражение, т. е. имеющее в результате `true` или `false`, затем знак вопроса, потом два произвольных выражения, разделенных двоеточием, например,

```
x < 0 ? 0 : x
x > y ? x — y : x + y
```

Условная операция выполняется так. Сначала вычисляется логическое выражение. Если получилось значение `true`, то вычисляется первое выражение после вопросительного знака `?` и его значение будет результатом всей операции. Последнее выражение при этом не вычисляется. Если же получилось значение `false`, то вычисляется только последнее выражение, его значение будет результатом операции.

Это позволяет написать `n == 0 ? m : m / n` не опасаясь деления на нуль. Условная операция поначалу кажется странной, но она очень удобна для записи небольших разветвлений.

Выражения

Из констант и переменных, операций над ними, вызовов методов и скобок составляются *выражения* (expressions). Разумеется, все элементы выражения должны быть совместимы, нельзя написать, например, `2 + true`. При вычислении выражения выполняются четыре правила:

1. Операции одного приоритета вычисляются слева направо: x + y + z вычисляется как (x + y) + z. Исключение: операции присваивания вычисляются справа налево: x = y = z вычисляется как x = (y = z).
2. Левый операнд вычисляется раньше правого.
3. Операнды полностью вычисляются перед выполнением операции.
4. Перед выполнением составной операции присваивания значение левой части сохраняется для использования в правой части.

Следующие примеры показывают особенности применения первых трех правил. Пусть

```
int a = 3, b = 5;
```

Тогда результатом выражения `b + (b = 3)` будет число `8`; но результатом выражения `(b = 3) + b` будет число `6`. Выражение `b += (b = 3)` даст в результате `8`, потому что вычисляется как первое из приведенных выше выражений.

Знатокам C/C++

Большинство компиляторов языка C++ во всех этих случаях вычислят значение 8.

Четвертое правило можно продемонстрировать так. При тех же определениях `a` и `b` в результате вычисления выражения `b += a += b += 7` получим `20`. Хотя операции присваивания выполняются справа налево и после первой, правой, операции значение `b` становится равным `12`, но в последнем, левом, присваивании участвует старое значение `b`, равное `5`. А в результате двух последовательных вычислений `a += b += 7; b += a;` получим `27`, поскольку во втором выражении участвует уже новое значение переменной `b`, равное `12`.

Знатокам C/C++

Большинство компиляторов C++ в обоих случаях вычислят 27.

Выражения могут иметь сложный и запутанный вид. В таких случаях возникает вопрос о приоритете операций, о том, какие операции будут выполнены в первую очередь. Естественно, умножение и деление производится раньше сложения и вычитания. Остальные правила перечислены в следующем разделе.

Порядок вычисления выражения всегда можно отрегулировать скобками, их можно ставить сколько угодно. Но здесь важно соблюдать "золотую середину". При большом количестве скобок снижается наглядность выражения и легко ошибиться в расстановке скобок. Если выражение со скобками корректно, то компилятор может отследить только парность скобок, но не правильность их расстановки.

Приоритет операций

Операции перечислены в порядке убывания приоритета. Операции на одной строке имеют одинаковый приоритет.

1. Постфиксные операции ++ и --.
2. Префиксные операции ++ и --, дополнение ~ и отрицание `!`.
3. Приведение типа `(тип)`.
4. Умножение `*`, деление `/` и взятие остатка `%`.
5. Сложение + и вычитание -.
6. Сдвиги <<, >>, >>>.
7. Сравнения >, <, >=, <=.
8. Сравнения ==, `!=`.
9. Побитовая конъюнкция `&`.
10. Побитовое исключающее ИЛИ ^.
11. Побитовая дизъюнкция `|`.
12. Конъюнкция `&&`.
13. Дизъюнкция `||`.
14. Условная операция `?:`.
15. Присваивания =, +=, -=, `*=`, /=, `%=`, &=, ^=, `|=`, <<, >>, >>>.

Здесь перечислены не все операции языка Java, список будет дополняться по мере изучения новых операций.

Знатокам C/C++

В Java нет операции "запятая", но *список выражений* используется в операторе цикла `for`.

Операторы

Как вы знаете, любой алгоритм, предназначенный для выполнения на компьютере, можно разработать, используя только линейные вычисления, разветвления и циклы.

Записать его можно в разных формах: в виде блок-схемы, на псевдокоде, на обычном языке, как мы записываем кулинарные рецепты, или как-нибудь еще "алгоритмы".

Всякий язык программирования должен иметь средства записи алгоритмов. Они называются *операторами* (statements) языка. Минимальный набор опе-

раторов должен содержать оператор для записи линейных вычислений, условный оператор для записи разветвлений и оператор цикла.

Обычно состав операторов языка программирования шире: для удобства записи алгоритмов в язык включаются несколько операторов цикла, оператор варианта, операторы перехода, операторы описания объектов.

Набор операторов языка Java включает:

- операторы описания переменных и других объектов (они были рассмотрены выше);
- операторы-выражения;
- операторы присваивания;
- условный оператор `if`;
- три оператора цикла `while`, `do-while`, `for`;
- оператор варианта `switch`;
- операторы перехода `break`, `continue` и `return`;
- блок `{}`;
- пустой оператор — просто точка с запятой.

Здесь приведен не весь набор операторов Java, он будет дополняться по мере изучения языка.

Замечание

В языке Java нет оператора `goto`.

Всякий оператор завершается точкой с запятой.

Можно поставить точку с запятой в конце любого выражения, и оно станет оператором (expression statement). Но смысл это имеет только для операций присваивания, инкремента и декремента и вызовов методов. В остальных случаях это бесполезно, потому что вычисленное значение выражения потеряется.

Знатокам Pascal

Точка с запятой в Java не разделяет операторы, а является частью оператора.

Линейное выполнение алгоритма обеспечивается последовательной записью операторов. Переход со строки на строку в исходном тексте не имеет никакого значения для компилятора, он осуществляется только для наглядности и читаемости текста.

Блок

Блок заключает в себе нуль или несколько операторов с целью использовать их как один оператор в тех местах, где по правилам языка можно записать

только один оператор. Например, `{x = 5; y = 7;}`. Можно записать и пустой блок, просто пару фигурных скобок `{}`.

Блоки операторов часто используются для ограничения области действия переменных и просто для улучшения читаемости текста программы.

Операторы присваивания

Точка с запятой в конце любой операции присваивания превращает ее в оператор присваивания. Побочное действие операции — присваивание — становится в операторе основным.

Разница между операцией и оператором присваивания носит лишь теоретический характер. Присваивание чаще используется как оператор, а не операция.

Условный оператор

Условный оператор (if-then-else statement) в языке Java записывается так:

```
if (логВыр) оператор1 else оператор2
```

и действует следующим образом. Сначала вычисляется логическое выражение *логВыр*. Если результат `true`, то действует *оператор1* и на этом действие условного оператора завершается, *оператор2* не действует, далее будет выполняться следующий за `if` оператор. Если результат `false`, то действует *оператор2*, при этом *оператор1* вообще не выполняется.

Условный оператор может быть сокращенным (if-then statement):

```
if (логВыр) оператор1
```

и в случае `false` не выполняется ничего.

Синтаксис языка не позволяет записывать несколько операторов ни в ветви `then`, ни в ветви `else`. При необходимости составляется блок операторов в фигурных скобках. Соглашения "Code Conventions" рекомендуют всегда использовать фигурные скобки и размещать оператор на нескольких строках с отступами, как в следующем примере:

```
if (a < x){
    x = a + b;
} else {
    x = a — b;
}
```

Это облегчает добавление операторов в каждую ветвь при изменении алгоритма. Мы не будем строго следовать этому правилу, чтобы не увеличивать объем книги.

Очень часто одним из операторов является снова условный оператор, например:

```
if (n == 0){
    sign = 0;
} else if (n < 0){
    sign = -1;
} else {
    sign = 1;
}
```

При этом может возникнуть такая ситуация ("dangling else"):

```
int ind = 5, x = 100;
if (ind >= 10) if (ind <= 20) x = 0; else x = 1;
```

Сохранит переменная `x` значение `100` или станет равной `1`? Здесь необходимо волевое решение, и общее для большинства языков, в том числе и Java, правило таково: ветвь `else` относится к ближайшему слева условию `if`, не имеющему своей ветви `else`. Поэтому в нашем примере переменная `x` останется равной `100`.

Изменить этот порядок можно с помощью блока:

```
if (ind > 10) {if (ind < 20) x = 0; else x = 1;}
```

Вообще не стоит увлекаться сложными вложенными условными операторами. Проверки условий занимают много времени. По возможности лучше использовать логические операции, например, в нашем примере можно написать

```
if (ind >= 10 && ind <= 20) x = 0;
else x = 1;
```

В листинге 1.4 вычисляются корни квадратного уравнения $ax^2 + bx + c = 0$ для любых коэффициентов, в том числе и нулевых.

Листинг 1.4. Вычисление корней квадратного уравнения

```
class QuadraticEquation{

  public static void main(String[] args){
    double a = 0.5, b = -2.7, c = 3.5, d, eps=1e-8;
    if (Math.abs(a) < eps)
      if (Math.abs(b) < eps)
        if (Math.abs(c) < eps)         // Все коэффициенты равны нулю
          System.out.println("Решение — любое число");
        else
          System.out.println("Решений нет");
```

```
      else
        System.out.println("x1 = x2 = " +(-c / b));
    else {                              // Коэффициенты не равны нулю
      if((d = b*b — 4*a*c)< 0.0){      // Комплексные корни
         d = 0.5 * Math.sqrt(-d) / a;
         a = -0.5 * b/ a;
         System.out.println("x1 = " +a+ " +i " +d+
             ", x2 = " +a+ " -i " +d);
      } else {                          // Вещественные корни
         d = 0.5 * Math.sqrt(d) / a;
         a = -0.5 * b / a;
         System.out.println("x1 = " +(a + d)+ ", x2 = " +(a — d));
      }
    }
  }
}
```

В этой программе использованы методы вычисления модуля `abs()` и квадратного корня `sqrt()`вещественного числа из встроенного в Java API класса `Math`. Поскольку все вычисления с вещественными числами производятся приближенно, мы считаем, что коэффициент уравнения равен нулю, если его модуль меньше 0,00000001. Обратите внимание на то, как в методе `println()` используется сцепление строк, и на то, как операция присваивания при вычислении дискриминанта вложена в логическое выражение.

"Продвинутым" пользователям

Вам уже хочется вводить коэффициенты `a`, `b` и `c` прямо с клавиатуры? Пожалуйста, используйте метод `System.in.read(byte[] bt)`, но учтите, что этот метод записывает вводимые цифры в массив байтов `bt` в кодировке ASCII, в каждый байт по одной цифре. Массив байтов затем надо преобразовать в вещественное число, например, методом `Double(new String(bt)).doubleValue()`. Непонятно? Но это еще не все, нужно обработать исключительные ситуации, которые могут возникнуть при вводе *(см. главу 18)*.

Операторы цикла

Основной оператор цикла — оператор `while` — выглядит так:

`while (`*логВыр*`)` *оператор*

Вначале вычисляется логическое выражение *логВыр*; если его значение `true`, то выполняется `оператор`, образующий цикл. Затем снова вычисляется *логВыр* и действует `оператор`, и так до тех пор, пока не получится значение `false`. Если *логВыр* изначально равняется `false`, то *оператор* не будет выполнен ни разу. Предварительная проверка обеспечивает безопасность выполнения цикла, позволяет избежать переполнения, деления на нуль и других неприятностей. Поэтому оператор `while` является основным, а в некоторых языках и единственным оператором цикла.

Оператор в цикле может быть и пустым, например, следующий фрагмент кода:

```
int i = 0;
double s = 0.0;
while ((s += 1.0 / ++i) < 10);
```

вычисляет количество `i` сложений, которые необходимо сделать, чтобы гармоническая сумма `s` достигла значения 10. Такой стиль характерен для языка С. Не стоит им увлекаться, чтобы не превратить текст программы в шифровку, на которую вы сами через пару недель будете смотреть с недоумением.

Можно организовать и бесконечный цикл:

```
while (true) оператор
```

Конечно, из такого цикла следует предусмотреть какой-то выход, например, оператором `break`, как в листинге 1.5. В противном случае программа зациклится, и вам придется прекращать ее выполнение "комбинацией из трех пальцев" <Ctrl>+<Alt>+<Del> в MS Windows 95/98/ME, комбинацией <Ctrl>+<C> в UNIX или через **Task Manager** в Windows NT/2000.

Если в цикл надо включить несколько операторов, то следует образовать блок операторов `{}`.

Второй оператор цикла — оператор `do-while` — имеет вид

```
do оператор while (логВыр)
```

Здесь сначала выполняется оператор, а потом происходит вычисление логического выражения *логВыр*. Цикл выполняется, пока *логВыр* остается равным `true`.

Знатокам Pascal

В цикле `do-while` проверяется условие продолжения, а не окончания цикла.

Существенное различие между этими двумя операторами цикла только в том, что в цикле `do-while` *оператор* обязательно выполнится хотя бы один раз.

Например, пусть задана какая-то функция f(x), имеющая на отрезке [a; b] ровно один корень. В листинге 1.5 приведена программа, вычисляющая этот корень приближенно методом деления пополам (бисекции, дихотомии).

Листинг 1.5. Нахождение корня нелинейного уравнения методом бисекции

```
class Bisection{

  static double f(double x){
    return x*x*x — 3*x*x + 3;          // Или что-то другое
  }
```

```
  public static void main(String[] args){
    double a = 0.0, b = 1.5, c, y, eps = 1e-8;
    do{
      c = 0.5 *(a + b); y = f(c);
      if (Math.abs(y) < eps) break;
      // Корень найден. Выходим из цикла

      // Если на концах отрезка [a; c]
      // функция имеет разные знаки:
      if (f(a) * y < 0.0) b = c;
        // Значит, корень здесь. Переносим точку b в точку c

        // В противном случае:
      else a = c;
        // Переносим точку a в точку c

        // Продолжаем, пока отрезок [a; b] не станет мал
    } while(Math.abs(b-a) >= eps);
    System.out.println("x = " +c+ ", f(" +c+ ") = " +y);
  }
}
```

Класс `Bisection` сложнее предыдущих примеров: в нем кроме метода `main()` есть еще метод вычисления функции f(x). Здесь метод `f()` очень прост: он вычисляет значение многочлена и возвращает его в качестве значения функции, причем все это выполняется одним оператором:

`return` *`выражение`*

В методе `main()` появился еще один новый оператор `break`, который просто прекращает выполнение цикла, если мы по счастливой случайности наткнулись на приближенное значение корня. Внимательный читатель заметил и появление модификатора `static` в объявлении метода `f()`. Он необходим потому, что метод `f()` вызывается из статического метода `main()`.

Третий оператор цикла — оператор `for` — выглядит так:

`for (`*`списокВыр1`*`;` *`логВыр`*`;` *`списокВыр2`*`)` *`оператор`*

Перед выполнением цикла вычисляется список выражений *`списокВыр1`*. Это нуль или несколько выражений, перечисленных через запятую. Они вычисляются слева направо, и в следующем выражении уже можно использовать результат предыдущего выражения. Как правило, здесь задаются начальные значения переменным цикла.

Затем вычисляется логическое выражение *`логВыр`*. Если оно истинно, `true`, то действует `оператор`, потом вычисляются слева направо выражения из списка выражений *`списокВыр2`*. Далее снова проверяется *`логВыр`*. Если оно ис-

тинно, то выполняется `оператор` и *списокВыр2* и т. д. Как только *логВыр* станет равным `false`, выполнение цикла заканчивается.

Короче говоря, выполняется последовательность операторов

```
списокВыр1;
while (логВыр){
    оператор
    списокВыр2;
}
```

с тем исключением, что, если оператором в цикле является оператор `continue`, то *списокВыр2* все-таки выполняется.

Вместо *списокВыр1* может стоять одно определение переменных обязательно с начальным значением. Такие переменные известны только в пределах этого цикла.

Любая часть оператора `for` может отсутствовать: цикл может быть пустым, выражения в заголовке тоже, при этом точки с запятой сохраняются. Можно задать бесконечный цикл:

```
for (;;) оператор
```

В этом случае в теле цикла следует предусмотреть какой-нибудь выход.

Хотя в операторе `for` заложены большие возможности, используется он, главным образом, для перечислений, когда их число известно, например, фрагмент кода

```
int s = 0;
for (int k = 1; k <= N; k++) s += k * k;
// Здесь переменная k уже неизвестна
```

вычисляет сумму квадратов первых `N` натуральных чисел.

Оператор *continue* и метки

Оператор `continue` используется только в операторах цикла. Он имеет две формы. Первая форма состоит только из слова `continue` и осуществляет немедленный переход к следующей итерации цикла. В очередном фрагменте кода оператор `continue` позволяет обойти деление на нуль:

```
for (int i = 0; i < N; i++){
   if (i == j) continue;
   s += 1.0 / (i - j);
}
```

Вторая форма содержит метку:

```
continue метка
```

Метка записывается, как все идентификаторы, из букв Java, цифр и знака подчеркивания, но не требует никакого описания. Метка ставится перед оператором или открывающей фигурной скобкой и отделяется от них двоеточием. Так получается *помеченный оператор* или *помеченный блок.*

Знатокам Pascal

Метка не требует описания и не может начинаться с цифры.

Вторая форма используется только в случае нескольких вложенных циклов для немедленного перехода к очередной итерации одного из объемлющих циклов, а именно, помеченного цикла.

Оператор *break*

Оператор `break` используется в операторах цикла и операторе варианта для немедленного выхода из этих конструкций.

Оператор

```
break метка
```

применяется внутри помеченных операторов цикла, оператора варианта или помеченного блока для немедленного выхода за эти операторы. Следующая схема поясняет эту конструкцию.

```
M1: { // Внешний блок
  M2: { // Вложенный блок — второй уровень
    M3: { // Третий уровень вложенности...
          if (что-то случилось) break M2;
          // Если true, то здесь ничего не выполняется
        }
        // Здесь тоже ничего не выполняется
      }
      // Сюда передается управление
    }
```

Поначалу сбивает с толку то обстоятельство, что метка ставится перед блоком или оператором, а управление передается за этот блок или оператор. Поэтому не стоит увлекаться оператором `break` с меткой.

Оператор варианта

Оператор варианта `switch` организует разветвление по нескольким направлениям. Каждая ветвь отмечается константой или константным выражением какого-либо целого типа (кроме `long`) и выбирается, если значение определенного выражения совпадет с этой константой. Вся конструкция выглядит так.

```
switch (целВыр){
 case констВыр1: оператор1
 case констВыр2: оператор2
 ...
 case констВырN: операторN
 default: операторDef
}
```

Стоящее в скобках выражение *целВыр* может быть типа `byte`, `short`, `int`, `char`, но не `long`. Целые числа или целочисленные выражения, составленные из констант, *констВыр* тоже не должны иметь тип `long`.

Оператор варианта выполняется так. Все константные выражения вычисляются заранее, на этапе компиляции, и должны иметь отличные друг от друга значения. Сначала вычисляется целочисленное выражение *целВыр*. Если оно совпадает с одной из констант, то выполняется оператор, отмеченный этой константой. Затем выполняются ("fall through labels") все следующие операторы, включая и *операторDef*, и работа оператора варианта заканчивается.

Если же ни одна константа не равна значению выражения, то выполняется *операторDef* и все следующие за ним операторы. Поэтому ветвь `default` должна записываться последней. Ветвь `default` может отсутствовать, тогда в этой ситуации оператор варианта вообще ничего не делает.

Таким образом, константы в вариантах `case` играют роль только меток, точек входа в оператор варианта, а далее выполняются все оставшиеся операторы в порядке их записи.

Знатокам Pascal

После выполнения одного варианта оператор `switch` продолжает выполнять все оставшиеся варианты.

Чаще всего необходимо "пройти" только одну ветвь операторов. В таком случае используется оператор `break`, сразу же прекращающий выполнение оператора `switch`. Может понадобиться выполнить один и тот же оператор в разных ветвях `case`. В этом случае ставим несколько меток `case` подряд. Вот простой пример.

```
switch(dayOfWeek){
  case 1: case 2: case 3: case 4:case 5:
      System.out.println("Week-day"); break;
  case 6: case 7:
      System.out.println("Week-end"); break;
  default:
      System.out.println("Unknown day");
}
```

Замечание

Не забывайте завершать варианты оператором `break`.

Массивы

Как всегда в программировании *массив* — это совокупность переменных одного типа, хранящихся в смежных ячейках оперативной памяти.

Массивы в языке Java относятся к ссылочным типам и описываются своеобразно, но характерно для ссылочных типов. Описание производится в три этапа.

Первый этап — *объявление* (declaration). На этом этапе определяется только переменная типа *ссылка* (reference) *на массив*, содержащая тип массива. Для этого записывается имя типа элементов массива, квадратными скобками указывается, что объявляется ссылка на массив, а не простая переменная, и перечисляются имена переменных типа ссылка, например,

```
double[] a, b;
```

Здесь определены две переменные — ссылки `a` и `b` на массивы типа `double`. Можно поставить квадратные скобки и непосредственно после имени. Это удобно делать среди определений обычных переменных:

```
int I = 0, ar[], k = -1;
```

Здесь определены две переменные целого типа `I` и `k`, и объявлена ссылка на целочисленный массив `ar`.

Второй этап — *определение* (instantation). На этом этапе указывается количество элементов массива, называемое его *длиной*, выделяется место для массива в оперативной памяти, переменная-ссылка получает адрес массива. Все эти действия производятся еще одной операцией языка Java — операцией `new` *тип*, выделяющей участок в оперативной памяти для объекта указанного в операции типа и возвращающей в качестве результата адрес этого участка. Например,

```
a = new double[5];
b = new double[100];
ar = new int[50];
```

Индексы массивов всегда начинаются с `0`. Массив `a` состоит из пяти переменных `a[0]`, `a[1]`, , `a[4]`. Элемента `a[5]` в массиве нет. Индексы можно задавать любыми целочисленными выражениями, кроме типа `long`, например, `a[i+j]`, `a[i%5]`, `a[++i]`. Исполняющая система Java следит за тем, чтобы значения этих выражений не выходили за границы длины массива.

Третий этап — *инициализация* (initialization). На этом этапе элементы массива получают начальные значения. Например,

```
a[0] = 0.01; a[1] = -3.4; a[2] = 2.89; a[3] = 4.5; a[4] = -6.7;
for (int i = 0; i < 100; i++) b[i] = 1.0 / i;
for (int i = 0; i < 50; i++) ar[i] = 2 * i + 1;
```

Первые два этапа можно совместить:

```
double[] a = new double[5], b = new double[100];
int i = 0, ar[] = new int[50], k = -1;
```

Можно сразу задать и начальные значения, записав их в фигурных скобках через запятую в виде констант или константных выражений. При этом даже необязательно указывать количество элементов массива, оно будет равно количеству начальных значений:

```
double[] a = {0.01, -3.4, 2.89, 4.5, -6.7};
```

Можно совместить второй и третий этап:

```
a = new double[] {0.1, 0.2, -0.3, 0.45, -0.02};
```

Можно даже создать безымянный массив, сразу же используя результат операции `new`, например, так:

```
System.out.println(new char[] {'H', 'e', 'l', 'l', 'o'});
```

Ссылка на массив не является частью описанного массива, ее можно перебросить на другой массив того же типа операцией присваивания. Например, после присваивания `a = b` обе ссылки `a` и `b` указывают на один и тот же массив из 100 вещественных переменных типа `double` и содержат один и тот же адрес.

Ссылка может присвоить "пустое" значение `null`, не указывающее ни на какой адрес оперативной памяти:

```
ar = null;
```

После этого массив, на который указывала данная ссылка, теряется, если на него не было других ссылок.

Кроме простой операции присваивания, со ссылками можно производить еще только сравнения на равенство, например, `a == b`, и неравенство, `a != b`. При этом сопоставляются адреса, содержащиеся в ссылках, мы можем узнать, не ссылаются ли они на один и тот же массив.

Замечание для специалистов

Массивы в Java всегда определяются динамически, хотя ссылки на них задаются статически.

Кроме ссылки на массив, для каждого массива автоматически определяется целая константа с одним и тем же именем `length`. Она равна длине массива. Для каждого массива имя этой константы уточняется именем массива через точку. Так, после наших определений, константа `a.length` равна `5`, константа `b.length` равна `100`, а `ar.length` равна `50`.

Последний элемент массива `a` можно записать так: `a[a.length — 1]`, предпоследний — `a[a.length — 2]` и т. д. Элементы массива обычно перебираются в цикле вида:

```
double aMin = a[0], aMax = aMin;
for (int i = 1; i < a.length; i++){
   if (a[i] < aMin) aMin = a[i];
   if (a[i] > aMax) aMax = a[i];
}
double range = aMax — aMin;
```

Здесь вычисляется диапазон значений массива.

Элементы массива — это обыкновенные переменные своего типа, с ними можно производить все операции, допустимые для этого типа: `(a[2] + a[4]) / a[0]` и т. д.

Знатокам C/C++

Массив символов в Java не является строкой, даже если он заканчивается нуль-символом `'\u0000'`.

Многомерные массивы

Элементами массивов в Java могут быть снова массивы. Можно объявить:

```
char[][] c;
```

что эквивалентно

```
char[] c[];
```

или

```
char c[][];
```

Затем определяем внешний массив:

```
c = new char[3][];
```

Становится ясно, что `c` — массив, состоящий из трех элементов-массивов. Теперь определяем его элементы-массивы:

```
c[0] = new char[2];
c[1] = new char[4];
c[2] = new char[3];
```

После этих определений переменная `c.length` равна `3`, `c[0].length` равна `2`, `c[1].length` равна `4` и `c[2].length` равна `3`.

Наконец, задаем начальные значения `c[0][0] = 'a'`, `c[0][1] = 'r'`, `c[1][0] = 'r'`, `c[1][1] = 'a'`, `c[1][2] = 'y'` и т. д.

Замечание

Двумерный массив в Java не обязан быть прямоугольным.

Описания можно сократить:

```
int[][] d = new int[3][4];
```

А начальные значения задать так:

```
int[][] inds = {{1, 2, 3}, {4, 5, 6}};
```

В листинге 1.6 приведен пример программы, вычисляющей первые 10 строк треугольника Паскаля, заносящей их в треугольный массив и выводящей его элементы на экран. Рис. 1.4 показывает вывод этой программы.

Листинг 1.6. Треугольник Паскаля

```
class PascalTriangle{
  public static final int LINES = 10;    // Так определяются константы

  public static void main(String[] args){
    int[][] p = new int[LINES][];
    p[0] = new int[1];
    System.out.println(p[0][0] = 1);
    p[1] = new int[2];
    p[1][0] = p[1][1] = 1;
    System.out.println(p[1][0] + " " + p[1][1]);
    for (int i = 2; i < LINES; i++){
      p[i] = new int[i+1];
      System.out.print((p[i][0] = 1) + " ");
      for (int j = 1; j < i; j++)
        System.out.print((p[i][j] = p[i-1][j-1] + p[i-1][j]) + " ");
      System.out.println(p[i][i] = 1);
    }
  }
}
```

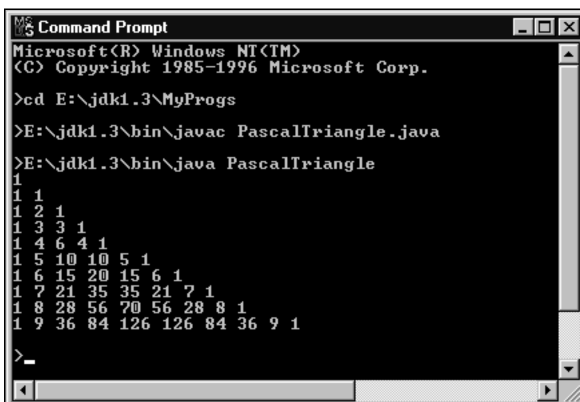


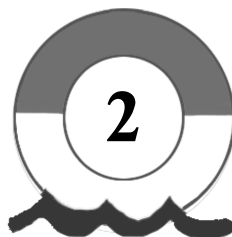
Рис. 1.4. Вывод треугольника Паскаля в окно **Command Prompt**

Заключение

Уф-ф-ф!! Вот вы и одолели базовые конструкции языка. Раз вы добрались до этого места, значит, умеете уже очень много. Вы можете написать программу на Java, отладить ее, устранив ошибки, и выполнить. Вы способны запрограммировать любой не слишком сложный вычислительный алгоритм, обрабатывающий числовые данные.

Теперь можно перейти к вопросам создания сложных производственных программ. Такие программы требуют тщательного планирования. Сделать это помогает объектно-ориентированное программирование, к которому мы теперь переходим.

ГЛАВА 2



Объектно-ориентированное программирование в Java

Вся полувековая история программирования компьютеров, а может быть, и история всей науки — это попытка совладать со сложностью окружающего мира. Задачи, встающие перед программистами, становятся все более громоздкими, информация, которую надо обработать, растет как снежный ком. Еще недавно обычными единицами измерения информации были килобайты и мегабайты, а сейчас уже говорят только о гигабайтах и терабайтах. Как только программисты предлагают более-менее удовлетворительное решение предложенных задач, тут же возникают новые, еще более сложные задачи. Программисты придумывают новые методы, создают новые языки. За полвека появилось несколько сотен языков, предложено множество методов и стилей. Некоторые методы и стили становятся общепринятыми и образуют на некоторое время так называемую *парадигму программирования*.

Парадигмы программирования

Первые, даже самые простые программы, написанные в машинных кодах, составляли сотни строк совершенно непонятного текста. Для упрощения и ускорения программирования придумали языки высокого уровня: FORTRAN, Algol и сотни других, возложив рутинные операции по созданию машинного кода на компилятор. Те же программы, переписанные на языках высокого уровня, стали гораздо понятнее и короче. Но жизнь потребовала решения более сложных задач, и программы снова увеличились в размерах, стали необозримыми.

Возникла идея: оформить программу в виде нескольких, по возможности простых, процедур или функций, каждая из которых решает свою определенную задачу. Написать, откомпилировать и отладить небольшую процедуру можно легко и быстро. Затем остается только собрать все процедуры в нужном порядке в одну программу. Кроме того, один раз написанные про-

цедуры можно затем использовать в других программах как строительные кирпичики. *Процедурное программирование* быстро стало парадигмой. Во все языки высокого уровня включили средства написания процедур и функций. Появилось множество библиотек процедур и функций на все случаи жизни.

Встал вопрос о том, как выявить структуру программы, разбить программу на процедуры, какую часть кода выделить в отдельную процедуру, как сделать алгоритм решения задачи простым и наглядным, как удобнее связать процедуры между собой. Опытные программисты предложили свои рекомендации, названные *структурным программированием*. Структурное программирование оказалось удобным и стало парадигмой. Появились языки программирования, например Pascal, на которых удобно писать структурные программы. Более того, на них очень трудно написать неструктурные программы.

Сложность стоящих перед программистами задач проявилась и тут: программы стали содержать сотни процедур, и опять оказались необозримыми. "Кирпичики" стали слишком маленькими. Потребовался новый стиль программирования.

В это же время обнаружилось, что удачная или неудачная структура исходных данных может сильно облегчить или усложнить их обработку. Одни исходные данные удобнее объединить в массив, для других больше подходит структура дерева или стека. Никлаус Вирт даже назвал свою книгу "Алгоритмы + структуры данных = программы".

Возникла идея объединить исходные данные и все процедуры их обработки в один модуль. Эта идея *модульного программирования* быстро завоевала умы и на некоторое время стала парадигмой. Программы составлялись из отдельных модулей, содержащих десяток-другой процедур и функций. Эффективность таких программ тем выше, чем меньше модули зависят друг от друга. Автономность модулей позволяет создавать и библиотеки модулей, чтобы потом использовать их в качестве строительных блоков для программы.

Для того чтобы обеспечить максимальную независимость модулей друг от друга, надо четко отделить процедуры, которые будут вызываться другими модулями, — *открытые* (public) процедуры, от вспомогательных, которые обрабатывают данные, заключенные в этот модуль, — *закрытых* (private) процедур. Первые перечисляются в отдельной части модуля — *интерфейсе* (interface), вторые участвуют только в *реализации* (implementation) модуля. Данные, занесенные в модуль, тоже делятся на открытые, указанные в интерфейсе и доступные для других модулей, и закрытые, доступные только для процедур того же модуля. В разных языках программирования это деление производится по-разному. В языке Turbo Pascal модуль специально делится на интерфейс и реализацию, в языке С интерфейс выносится в отдельные "головные" (header) файлы. В языке С++, кроме того, для описания интерфейса можно воспользоваться абстрактными классами. В языке Java

есть специальная конструкция для описания интерфейсов, которая так и называется — interface, но можно написать и абстрактные классы.

Так возникла идея о скрытии, *инкапсуляции* (incapsulation) данных и методов их обработки. Подобные идеи периодически возникают в дизайне бытовой техники. То телевизоры испещряются кнопками и топорщатся ручками и движками на радость любознательному телезрителю, господствует "приборный" стиль, то вдруг все куда-то пропадает, а на панели остаются только кнопка включения и ручка громкости. Любознательный телезритель берется за отвертку.

Инкапсуляция, конечно, производится не для того, чтобы спрятать от другого модуля что-то любопытное. Здесь преследуются две основные цели. Первая — обеспечить безопасность использования модуля, вынести в интерфейс, сделать общедоступными только те методы обработки информации, которые не могут испортить или удалить исходные данные. Вторая цель — уменьшить сложность, скрыв от внешнего мира ненужные детали реализации.

Опять возник вопрос, каким образом разбить программу на модули? Тут кстати оказались методы решения старой задачи программирования — моделирования действий искусственных и природных объектов: роботов, станков с программным управлением, беспилотных самолетов, людей, животных, растений, систем обеспечения жизнедеятельности, систем управления технологическими процессами.

В самом деле, каждый объект — робот, автомобиль, человек — обладает определенными характеристиками. Ими могут служить: вес, рост, максимальная скорость, угол поворота, грузоподъемность, фамилия, возраст. Объект может производить какие-то действия: перемещаться в пространстве, поворачиваться, поднимать, копать, расти или уменьшаться, есть, пить, рождаться и умирать, изменяя свои первоначальные характеристики. Удобно смоделировать объект в виде модуля. Его характеристики будут данными, постоянными или переменными, а действия — процедурами.

Оказалось удобным сделать и обратное — разбить программу на модули так, чтобы она превратилась в совокупность взаимодействующих объектов. Так возникло *объектно-ориентированное программирование* (object-oriented programming), сокращенно ООП (OOP) — современная парадигма программирования.

В виде объектов можно представить совсем неожиданные понятия. Например, окно на экране дисплея — это объект, имеющий ширину `width` и высоту `height`, расположение на экране, описываемое обычно координатами `(x, y)` левого верхнего угла окна, а также шрифт, которым в окно выводится текст, скажем, Times New Roman, цвет фона `color`, несколько кнопок, линейки прокрутки и другие характеристики. Окно может перемещаться по экрану методом `move()`, увеличиваться или уменьшаться в размерах методом

`size()`, сворачиваться в ярлык методом `iconify()`, как-то реагировать на действия мыши и нажатия клавиш. Это полноценный объект! Кнопки, полосы прокрутки и прочие элементы окна — это тоже объекты со своими размерами, шрифтами, перемещениями.

Разумеется, считать, что окно само "умеет" выполнять действия, а мы только даем ему поручения: "Свернись, развернись, передвинься", — это несколько неожиданный взгляд на вещи, но ведь сейчас можно подавать команды не только мышью и клавишами, но и голосом!

Идея объектно-ориентированного программирования оказалась очень плодотворной и стала активно развиваться. Выяснилось, что удобно ставить задачу сразу в виде совокупности действующих объектов — возник *объектно-ориентированный анализ*, ООА. Решили проектировать сложные системы в виде объектов — *появилось объектно-ориентированное проектирование*, ООП (OOD, object-oriented design).

Рассмотрим подробнее принципы объектно-ориентированного программирования.

Принципы объектно-ориентированного программирования

Объектно-ориентированное программирование развивается уже более двадцати лет. Имеется несколько школ, каждая из которых предлагает свой набор принципов работы с объектами и по-своему излагает эти принципы. Но есть несколько общепринятых понятий. Перечислим их.

Абстракция

Описывая поведение какого-либо объекта, например автомобиля, мы строим его модель. Модель, как правило, не может описать объект полностью, реальные объекты слишком сложны. Приходится отбирать только те характеристики объекта, которые важны для решения поставленной перед нами задачи. Для описания грузоперевозок важной характеристикой будет грузоподъемность автомобиля, а для описания автомобильных гонок она не существенна. Но для моделирования гонок обязательно надо описать метод набора скорости данным автомобилем, а для грузоперевозок это не столь важно.

Мы должны *абстрагироваться* от некоторых конкретных деталей объекта. Очень важно выбрать правильную степень абстракции. Слишком высокая степень даст только приблизительное описание объекта, не позволит правильно моделировать его поведение. Слишком низкая степень абстракции сделает модель очень сложной, перегруженной деталями, и потому непригодной.

Например, можно совершенно точно предсказать погоду на завтра в определенном месте, но расчеты по такой модели продлятся трое суток даже на самом мощном компьютере. Зачем нужна модель, опаздывающая на два дня? Ну а точность модели, используемой синоптиками, мы все знаем сами. Зато расчеты по этой модели занимают всего несколько часов.

Описание каждой модели производится в виде одного или нескольких *классов* (classes). Класс можно считать проектом, слепком, чертежом, по которому затем будут создаваться конкретные объекты. Класс содержит описание переменных и констант, характеризующих объект. Они называются ***полями класса*** (class fields). Процедуры, описывающие поведение объекта, называются ***методами класса*** (class methods). Внутри класса можно описать и ***вложенные классы*** (nested classes) и ***вложенные интерфейсы***. Поля, методы и вложенные классы первого уровня являются ***членами класса*** (class members). Разные школы объектно-ориентированного программирования предлагают разные термины, мы используем терминологию, принятую в технологии Java.

Вот набросок описания автомобиля:

```
class Automobile{
  int maxVelocity;   // Поле, содержащее наибольшую скорость автомобиля
  int speed;         // Поле, содержащее текущую скорость автомобиля
  int weight;        // Поле, содержащее вес автомобиля
// Прочие поля...
  void moveTo(int x, int y){    // Метод, моделирующий перемещение
                                // автомобиля. Параметры x и y — не поля
    int a = 1;                  // Локальная переменная — не поле
    // Тело метода. Здесь описывается закон
    // перемещения автомобиля в точку (x, y)
  }
// Прочие методы...
}
```

Знатокам Pascal

В Java нет вложенных процедур и функций, в теле метода нельзя описать другой метод.

После того как описание класса закончено, можно создавать конкретные объекты, *экземпляры* (instances) описанного класса. Создание экземпляров производится в три этапа, подобно описанию массивов. Сначала объявляются ссылки на объекты: записывается имя класса, и через пробел перечисляются экземпляры класса, точнее, ссылки на них.

```
Automobile lada2110, fordScorpio, oka;
```

Затем операцией `new` определяются сами объекты, под них выделяется оперативная память, ссылка получает адрес этого участка в качестве своего значения.

```
lada2110 = new Automobile();
fordScorpio = new Automobile();
oka = new Automobile();
```

На третьем этапе происходит инициализация объектов, задаются начальные значения. Этот этап, как правило, совмещается со вторым, именно для этого в операции `new` повторяется имя класса со скобками `Automobile()`. Это так называемый ***конструктор*** (constructor) класса, но о нем поговорим попозже.

Поскольку имена полей, методов и вложенных классов у всех объектов одинаковы, они заданы в описании класса, их надо уточнять именем ссылки на объект:

```
lada2110.maxVelocity = 150;
fordScorpio.maxVelocity = 180;
oka.maxVelocity = 350;      // Почему бы и нет?
oka.moveTo(35, 120);
```

Напомним, что текстовая строка в кавычках понимается в Java как объект класса `String`. Поэтому можно написать

```
int strlen = "Это объект класса String".length();
```

Объект "строка" выполняет метод `length()`, один из методов своего класса `String`, подсчитывающий число символов в строке. В результате получаем значение `strlen`, равное `24`. Подобная странная запись встречается в программах на Java на каждом шагу.

Во многих ситуациях строят несколько моделей с разной степенью детализации. Скажем, для конструирования пальто и шубы нужна менее точная модель контуров человеческого тела и его движений, а для конструирования фрака или вечернего платья — уже гораздо более точная. При этом более точная модель, с меньшей степенью абстракции, будет использовать уже имеющиеся методы менее точной модели.

Не кажется ли вам, что класс `Automobile` сильно перегружен? Действительно, в мире выпущены миллионы автомобилей разных марок и видов. Что между ними общего, кроме четырех колес? Да и колес может быть больше или меньше. Не лучше ли написать отдельные классы для легковых и грузовых автомобилей, для гоночных автомобилей и вездеходов? Как организовать все это множество классов? На этот вопрос объектно-ориентированное программирование отвечает так: надо организовать иерархию классов.

Иерархия

Иерархия объектов давно используется для их классификации. Особенно детально она проработана в биологии. Все знакомы с семействами, родами

и видами. Мы можем сделать описание своих домашних животных (pets): кошек (cats), собак (dogs), коров (cows) и прочих следующим образом:

```
class Pet{      // Здесь описываем общие свойства всех домашних любимцев
  Master person;                       // Хозяин животного
  int weight, age, eatTime[];          // Вес, возраст, время кормления
  int eat(int food, int drink, int time){      // Процесс кормления
    // Начальные действия...
    if (time == eatTime[i]) person.getFood(food, drink);
    // Метод потребления пищи
  }
  void voice();                        // Звуки, издаваемые животным
  // Прочее…
}
```

Затем создаем классы, описывающие более конкретные объекты, связывая их с общим классом:

```
class Cat extends Pet{   // Описываются свойства, присущие только кошкам:
  int mouseCatched;      // число пойманных мышей
  void toMouse();        // процесс ловли мышей
  // Прочие свойства
}
class Dog extends Pet{   // Свойства собак:
  void preserve();       // охранять
}
```

Заметьте, что мы не повторяем общие свойства, описанные в классе `Pet`. Они наследуются автоматически. Мы можем определить объект класса `Dog` и использовать в нем все свойства класса `Pet` так, как будто они описаны в классе `Dog`:

```
Dog tuzik = new Dog(), sharik = new Dog();
```

После этого определения можно будет написать

```
tuzik.age = 3;
int p = sharik.eat(30, 10, 12);
```

А классификацию продолжить так:

```
class Pointer extends Dog{ … }     // Свойства породы Пойнтер
class Setter extends Dog{ … }      // Свойства сеттеров
```

Заметьте, что на каждом следующем уровне иерархии в класс добавляются новые свойства, но ни одно свойство не пропадает. Поэтому и употребляется слово `extends` — "расширяет" и говорят, что класс `Dog` — *расширение* (extension) класса `Pet`. С другой стороны, количество объектов при этом уменьшается: собак меньше, чем всех домашних животных. Поэтому часто

говорят, что класс Dog — *подкласс* (subclass) класса Pet, а класс Pet — *суперкласс* (superclass) или надкласс класса Dog.

Часто используют генеалогическую терминологию: родительский класс, дочерний класс, класс-потомок, класс-предок, возникают племянники и внуки, вся беспокойная семейка вступает в отношения, достойные мексиканского сериала.

В этой терминологии говорят о *наследовании* (inheritance) классов, в нашем примере класс Dog наследует класс Pet.

Мы еще не определили счастливого владельца нашего домашнего зоопарка. Опишем его в классе Master. Делаем набросок:

```
class Master{        // Хозяин животного
  String name;       // Фамилия, имя
  // Другие сведения
  void getFood(int food, int drink);   // Кормление
  // Прочее
}
```

Хозяин и его домашние животные постоянно соприкасаются в жизни. Их взаимодействие выражается глаголами "гулять", "кормить", "охранять", "чистить", "ласкаться", "проситься" и прочими. Для описания взаимодействия объектов применяется третий принцип объектно-ориентированного программирования — обязанность или ответственность.

Ответственность

В нашем примере рассматривается только взаимодействие в процессе кормления, описываемое методом eat(). В этом методе животное обращается к хозяину, умоляя его применить метод getFood().

В англоязычной литературе подобное обращение описывается словом message. Это понятие неудачно переведено на русский язык ни к чему не обязывающим словом *"сообщение"*. Лучше было бы использовать слово "послание", "поручение" или даже "распоряжение". Но термин "сообщение" устоялся и нам придется его применять. Почему же не используется словосочетание "вызов метода", ведь говорят: "Вызов процедуры"? Потому что между этими понятиями есть, по крайней мере, три отличия.

- Сообщение идет к конкретному объекту, знающему метод решения задачи, в примере этот объект — текущее значение переменной person. У каждого объекта свое текущее состояние, свои значения полей класса, и это может повлиять на выполнение метода.
- Способ выполнения поручения, содержащегося в сообщении, зависит от объекта, которому оно послано. Один хозяин поставит миску с "Chappi", другой бросит кость, третий выгонит собаку на улицу. Это интересное

свойство называется *полиморфизмом* (polymorphism) и будет обсуждаться ниже.

- Обращение к методу произойдет только на этапе выполнения программы, компилятор ничего не знает про метод. Это называется *"поздним связыванием"* в противовес *"раннему связыванию"*, при котором процедура присоединяется к программе на этапе компоновки.

Итак, объект `sharik`, выполняя свой метод `eat()`, посылает сообщение объекту, ссылка на который содержится в переменной `person`, с просьбой выдать ему определенное количество еды и питья. Сообщение записано в строке `person.getFood(food, drink)`.

Этим сообщением заключается *контракт* (contract) между объектами, суть которого в том, что объект `sharik` берет на себя *ответственность* (responsibility) задать правильные параметры в сообщении, а объект — текущее значение `person` — возлагает на себя *ответственность* применить метод кормления `getFood()`, каким бы он ни был.

Для того чтобы правильно реализовать принцип ответственности, применяется четвертый принцип объектно-ориентированного программирования — *модульность* (modularity).

Модульность

Этот принцип утверждает — каждый класс должен составлять отдельный модуль. Члены класса, к которым не планируется обращение извне, должны быть инкапсулированы.

В языке Java инкапсуляция достигается добавлением модификатора `private` к описанию члена класса. Например:

```
private int mouseCatched;
private String name;
private void preserve();
```

Эти члены классов становятся *закрытыми*, ими могут пользоваться только экземпляры того же самого класса, например, `tuzik` может дать поручение `sharik.preserve()`.

А если в классе `Master` мы напишем

```
private void getFood(int food, int drink);
```

то метод `getFood()` не будет найден, и несчастный `sharik` не сможет получить пищу.

В противоположность закрытости мы можем объявить некоторые члены класса *открытыми*, записав вместо слова `private` модификатор `public`, например:

```
public void getFood(int food, int drink);
```

К таким членам может обратиться любой объект любого класса.

Знатокам С++

> В языке Java словами `private`, `public` и `protected` отмечается каждый член класса в отдельности.

Принцип модульности предписывает открывать члены класса только в случае необходимости. Вспомните надпись: "Нормальное положение шлагбаума — закрытое".

Если же надо обратиться к полю класса, то рекомендуется включить в класс специальные *методы доступа* (access methods), отдельно для чтения этого поля (get method) и для записи в это поле (set method). Имена методов доступа рекомендуется начинать со слов `get` и `set`, добавляя к этим словам имя поля. Для JavaBeans эти рекомендации возведены в ранг закона.

В нашем примере класса `Master` методы доступа к полю `Name` в самом простом виде могут выглядеть так:

```
public String getName(){
  return name;
}
public void setName(String newName){
  name = newName;
}
```

В реальных ситуациях доступ ограничивается разными проверками, особенно в *set-методах*, меняющих значения полей. Можно проверять тип вводимого значения, задавать диапазон значений, сравнивать со списком допустимых значений.

Кроме методов доступа рекомендуется создавать проверочные *is-методы*, возвращающие логическое значение `true` или `false`. Например, в класс `Master` можно включить метод, проверяющий, задано ли имя хозяина:

```
public boolean isEmpty(){
  return name == null ? true : false;
}
```

и использовать этот метод для проверки при доступе к полю `Name`, например:

```
if (master01.isEmpty()) master01.setName("Иванов");
```

Итак, мы оставляем открытыми только методы, необходимые для взаимодействия объектов. При этом удобно спланировать классы так, чтобы зависимость между ними была наименьшей, как принято говорить в теории ООП, было наименьшее *зацепление* (low coupling) между классами. Тогда структура программы сильно упрощается. Кроме того, такие классы удобно использовать как строительные блоки для построения других программ.

Напротив, члены класса должны активно взаимодействовать друг с другом, как говорят, иметь тесную функциональную *связность* (high cohestion). Для этого в класс следует включать все методы, описывающие поведение моделируемого объекта, и только такие методы, ничего лишнего. Одно из правил достижения сильной функциональной связности, введенное Карлом Либерхером (Karl J. Lieberherr), получило название *закон Деметра*. Закон гласит: "в методе `m()` класса `A` следует использовать только методы класса `A`, методы классов, к которым принадлежат аргументы метода `m()`, и методы классов, экземпляры которых создаются внутри метода `m()`".

Объекты, построенные по этим правилам, подобны кораблям, снабженным всем необходимым. Они уходят в автономное плавание, готовые выполнить любое поручение, на которое рассчитана их конструкция.

Будут ли закрытые члены класса доступны его наследникам? Если в классе `Pet` написано

```
private Master person;
```

то можно ли использовать `sharik.person`? Разумеется, нет. Ведь в противном случае каждый, интересующийся закрытыми полями класса `A`, может расширить его классом `B`, и просмотреть закрытые поля класса `A` через экземпляры класса `B`.

Когда надо разрешить доступ наследникам класса, но нежелательно открывать его всему миру, тогда в Java используется *защищенный* (protected) доступ, отмечаемый модификатором `protected`, например, объект `sharik` может обратиться к полю `person` родительского класса `Pet`, если в классе `Pet` это поле описано так:

```
protected Master person;
```

Следует сразу сказать, что на доступ к члену класса влияет еще и пакет, в котором находится класс, но об этом поговорим в следующей главе.

Из этого общего схематического описания принципов объектно-ориентированного программирования видно, что язык Java позволяет легко воплощать все эти принципы. Вы уже поняли, как записать класс, его поля и методы, как инкапсулировать члены класса, как сделать расширение класса и какими принципами следует при этом пользоваться. Разберем теперь подробнее правила записи классов и рассмотрим дополнительные их возможности.

Но, говоря о принципах ООП, я не могу удержаться от того, чтобы не напомнить основной принцип всякого программирования.

Принцип KISS

Самый основной, базовый и самый великий принцип программирования — принцип KISS — не нуждается в разъяснении и переводе: "Keep It Simple, Stupid!"

Как описать класс и подкласс

Итак, описание класса начинается со слова `class`, после которого записывается имя класса. Соглашения "Code Conventions" рекомендуют начинать имя класса с заглавной буквы.

Перед словом `class` можно записать модификаторы класса (class modifiers). Это одно из слов `public`, `abstract`, `final`, `strictfp`. Перед именем вложенного класса можно поставить, кроме того, модификаторы `protected`, `private`, `static`. Модификаторы мы будем вводить по мере изучения языка.

Тело класса, в котором в любом порядке перечисляются поля, методы, вложенные классы и интерфейсы, заключается в фигурные скобки.

При описании поля указывается его тип, затем, через пробел, имя и, может быть, начальное значение после знака равенства, которое можно записать константным выражением. Все это уже описано в *главе 1.*

Описание поля может начинаться с одного или нескольких необязательных модификаторов `public`, `protected`, `private`, `static`, `final`, `transient`, `volatile`. Если надо поставить несколько модификаторов, то перечислять их JLS рекомендует в указанном порядке, поскольку некоторые компиляторы требуют определенного порядка записи модификаторов. С модификаторами мы будем знакомиться по мере необходимости.

При описании метода указывается тип возвращаемого им значения или слово `void`, затем, через пробел, имя метода, потом, в скобках, список параметров. После этого в фигурных скобках расписывается выполняемый метод.

Описание метода может начинаться с модификаторов `public`, `protected`, `private`, `abstract`, `static`, `final`, `synchronized`, `native`, `strictfp`. Мы будем вводить их по необходимости.

В списке параметров через запятую перечисляются тип и имя каждого параметра. Перед типом какого-либо параметра может стоять модификатор `final`. Такой параметр нельзя изменять внутри метода. Список параметров может отсутствовать, но скобки сохраняются.

Перед началом работы метода для каждого параметра выделяется ячейка оперативной памяти, в которую копируется значение параметра, заданное при обращении к методу. Такой способ называется передачей параметров *по значению.*

В листинге 2.1 показано, как можно оформить метод деления пополам для нахождения корня нелинейного уравнения из листинга 1.5.

Листинг 2.1. Нахождение корня нелинейного уравнения методом бисекции

```
class Bisection2{
  private static double final EPS = 1e-8;   // Константа
  private double a = 0.0, b = 1.5, root;    // Закрытые поля
```

```
  public  double getRoot(){return root;}    // Метод доступа

  private double f(double x){
    return x*x*x — 3*x*x + 3;               // Или что-то другое
  }

  private void bisect(){      // Параметров нет —
                              // метод работает с полями экземпляра
    double y = 0.0;           // Локальная переменная — не поле
    do{
      root = 0.5 *(a + b); y = f(root);
      if (Math.abs(y) < EPS) break;
      // Корень найден. Выходим из цикла

      // Если на концах отрезка [a; root]
      // функция имеет разные знаки:
      if (f(a) * y < 0.0) b = root;
        // значит, корень здесь
        // Переносим точку b в точку root

        // В противном случае:
      else a = root;
        // переносим точку a в точку root

        // Продолжаем, пока [a; b] не станет мал
    } while(Math.abs(b-a) >= EPS);
  }

  public static void main(String[] args){
    Bisection2 b2 = new Bisection2();
    b2.bisect();
    System.out.println("x = " +
      b2.getRoot() +            // Обращаемся к корню через метод доступа
      ", f() = " +b2.f(b2.getRoot()));
   }
}
```

В описании метода `f()` сохранен старый, процедурный стиль: метод получает аргумент, обрабатывает его и возвращает результат. Описание метода `bisect()` выполнено в духе ООП: метод активен, он сам обращается к полям экземпляра `b2` и сам заносит результат в нужное поле. Метод `bisect()` — это внутренний механизм класса `Bisection2`, поэтому он закрыт (private).

Имя метода, число и типы параметров образуют *сигнатуру* (signature) метода. Компилятор различает методы не по их именам, а по сигнатурам. Это позволяет записывать разные методы с одинаковыми именами, различающиеся числом и/или типами параметров.

Замечание

Тип возвращаемого значения не входит в сигнатуру метода, значит, методы не могут различаться только типом результата их работы.

Например, в классе `Automobile` мы записали метод `moveTo(int x, int y)`, обозначив пункт назначения его географическими координатами. Можно определить еще метод `moveTo(String destination)` для указания географического названия пункта назначения и обращаться к нему так:

```
oka.moveTo("Москва");
```

Такое дублирование методов называется ***перегрузкой*** (overloading). Перегрузка методов очень удобна в использовании. Вспомните, в главе 1 мы выводили данные любого типа на экран методом `println()` не заботясь о том, данные какого именно типа мы выводим. На самом деле мы использовали разные методы с одним и тем же именем `println`, даже не задумываясь об этом. Конечно, все эти методы надо тщательно спланировать и заранее описать в классе. Это и сделано в классе `PrintStream`, где представлено около двадцати методов `print()` и `println()`.

Если же записать метод с тем же именем в подклассе, например:

```
class Truck extends Automobile{
  void moveTo(int x, int y){
     // Какие-то действия
  }
     // Что-то еще
}
```

то он перекроет метод суперкласса. Определив экземпляр класса `Truck`, например:

```
Truck gazel = new Truck();
```

и записав `gazel.moveTo(25, 150)`, мы обратимся к методу класса `Truck`. Произойдет ***переопределение*** (overriding) метода.

При переопределении права доступа к методу можно только расширить. Открытый метод `public` должен остаться открытым, защищенный `protected` может стать открытым.

Можно ли внутри подкласса обратиться к методу суперкласса? Да, можно, если уточнить имя метода словом `super`, например, `super.moveTo(30, 40)`. Можно уточнить и имя метода, записанного в этом же классе, словом `this`, например, `this.moveTo(50, 70)`, но в данном случае это уже излишне. Таким же образом можно уточнять и совпадающие имена полей, а не только методов.

Данные уточнения подобны тому, как мы говорим про себя "я", а не "Иван Петрович", и говорим "отец", а не "Петр Сидорович".

Переопределение методов приводит к интересным результатам. В классе `Pet` мы описали метод `voice()`. Переопределим его в подклассах и используем в классе `Chorus`, как показано в листинге 2.2.

Листинг 2.2. Пример полиморфного метода

```
abstract class Pet{
  abstract void voice();
}
class Dog extends Pet{
  int k = 10;
  void voice(){
    System.out.println("Gav-gav!");
  }
}
class Cat extends Pet{
  void voice(){
    System.out.println("Miaou!");
  }
}
class Cow extends Pet{
  void voice(){
    System.out.println("Mu-u-u!");
  }
}
public class Chorus{
  public static void main(String[] args){
     Pet[] singer = new Pet[3];
     singer[0] = new Dog();
     singer[1] = new Cat();
     singer[2] = new Cow();
     for (int i = 0; i < singer.length; i++)
       singer[i].voice();
  }
}
```

На рис. 2.1 показан вывод этой программы. Животные поют своими голосами!

Все дело здесь в определении поля `singer[]`. Хотя массив ссылок `singer[]` имеет тип `Pet`, каждый его элемент ссылается на объект своего типа `Dog`, `Cat`, `Cow`. При выполнении программы вызывается метод конкретного объекта, а не метод класса, которым определялось имя ссылки. Так в Java реализуется полиморфизм.

Знатокам C++

В языке Java все методы являются виртуальными функциями.

Внимательный читатель заметил в описании класса `Pet` новое слово `abstract`. Класс `Pet` и метод `voice()` являются абстрактными.

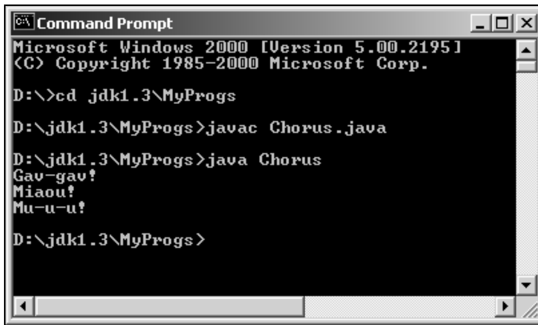


Рис. 2.1. Результат выполнения программы `Chorus`

Абстрактные методы и классы

При описании класса `Pet` мы не можем задать в методе `voice()` никакой полезный алгоритм, поскольку у всех животных совершенно разные голоса.

В таких случаях мы записываем только заголовок метода и ставим после закрывающей список параметров скобки точку с запятой. Этот метод будет *абстрактным* (abstract), что необходимо указать компилятору модификатором `abstract`.

Если класс содержит хоть один абстрактный метод, то создать его экземпляры, а тем более использовать их, не удастся. Такой класс становится *абстрактным*, что обязательно надо указать модификатором `abstract`.

Как же использовать абстрактные классы? Только порождая от них подклассы, в которых переопределены абстрактные методы.

Зачем же нужны абстрактные классы? Не лучше ли сразу написать нужные классы с полностью определенными методами, а не наследовать их от абстрактного класса? Для ответа снова обратимся к листингу 2.2.

Хотя элементы массива `singer[]` ссылаются на подклассы `Dog`, `Cat`, `Cow`, но все-таки это переменные типа `Pet` и ссылаться они могут только на поля и методы, описанные в суперклассе `Pet`. Дополнительные поля подкласса для них недоступны. Попробуйте обратиться, например, к полю `k` класса `Dog`, написав `singer[0].k`. Компилятор "скажет", что он не может реализовать такую ссылку. Поэтому метод, который реализуется в нескольких подклассах, приходится выносить в суперкласс, а если там его нельзя реализовать, то объявить абстрактным. Таким образом, абстрактные классы группируются на вершине иерархии классов.

Кстати, можно задать пустую реализацию метода, просто поставив пару фигурных скобок, ничего не написав между ними, например:

```
void voice(){}
```

Получится полноценный метод. Но это искусственное решение, запутывающее структуру класса.

Замкнуть же иерархию можно окончательными классами.

Окончательные члены и классы

Пометив метод модификатором `final`, можно запретить его переопределение в подклассах. Это удобно в целях безопасности. Вы можете быть уверены, что метод выполняет те действия, которые вы задали. Именно так определены математические функции `sin()`, `cos()` и прочие в классе `Math`. Мы уверены, что метод `Math.cos(x)` вычисляет именно косинус числа `x`. Разумеется, такой метод не может быть абстрактным.

Для полной безопасности, поля, обрабатываемые окончательными методами, следует сделать закрытыми (private).

Если же пометить модификатором `final` весь класс, то его вообще нельзя будет расширить. Так определен, например, класс `Math`:

```
public final class Math{ . . . }
```

Для переменных модификатор `final` имеет совершенно другой смысл. Если пометить модификатором `final` описание переменной, то ее значение (а оно должно быть обязательно задано или здесь же, или в блоке инициализации или в конструкторе) нельзя изменить ни в подклассах, ни в самом классе. Переменная превращается в константу. Именно так в языке Java определяются константы:

```
public final int MIN_VALUE = -1, MAX_VALUE = 9999;
```

По соглашению "Code Conventions" константы записываются прописными буквами, слова в них разделяются знаком подчеркивания.

На самой вершине иерархии классов Java стоит класс `Object`.

Класс *Object*

Если при описании класса мы не указываем никакое расширение, т. е. не пишем слово `extends` и имя класса за ним, как при описании класса `Pet`, то Java считает этот класс расширением класса `Object`, и компилятор дописывает это за нас:

```
class Pet extends Object{ . . . }
```

Можно записать это расширение и явно.

Сам же класс `Object` не является ничьим наследником, от него начинается иерархия любых классов Java. В частности, все массивы — прямые наследники класса `Object`.

Поскольку такой класс может содержать только общие свойства всех классов, в него включено лишь несколько самых общих методов, например, метод `equals()`, сравнивающий данный объект на равенство с объектом, заданным в аргументе, и возвращающий логическое значение. Его можно использовать так:

```
Object obj1 = new Dog(), obj2 = new Cat();
if (obj1.equals(obj2)) ...
```

Оцените объектно-ориентированный дух этой записи: объект `obj1` активен, он сам сравнивает себя с другим объектом. Можно, конечно, записать и `obj2.equals(obj1)`, сделав активным объект `obj2`, с тем же результатом.

Как указывалось в *главе 1*, ссылки можно сравнивать на равенство и неравенство:

```
obj1 == obj2; obj1 != obj2;
```

В этом случае сопоставляются адреса объектов, мы можем узнать, не указывают ли обе ссылки на один и тот же объект.

Метод `equals()` же сравнивает содержимое объектов в их текущем состоянии, фактически он реализован в классе `Object` как тождество: объект равен только самому себе. Поэтому его часто переопределяют в подклассах, более того, правильно спроектированные, "хорошо воспитанные", классы должны переопределить методы класса `Object`, если их не устраивает стандартная реализация.

Второй метод класса `Object`, который следует переопределять в подклассах, — метод `toString()`. Это метод без параметров, который пытается содержимое объекта преобразовать в строку символов и возвращает объект класса `String`.

К этому методу исполняющая система Java обращается каждый раз, когда требуется представить объект в виде строки, например, в методе `println()`.

Конструкторы класса

Вы уже обратили внимание на то, что в операции `new`, определяющей экземпляры класса, повторяется имя класса со скобками. Это похоже на обращение к методу, но что за "метод", имя которого полностью совпадает с именем класса?

Такой "метод" называется *конструктором класса* (class constructor). Его своеобразие заключается не только в имени. Перечислим особенности конструктора.

- Конструктор имеется в любом классе. Даже если вы его не написали, компилятор Java сам создаст *конструктор по умолчанию* (default constructor), который, впрочем, пуст, он не делает ничего, кроме вызова конструктора суперкласса.
- Конструктор выполняется автоматически при создании экземпляра класса, после распределения памяти и обнуления полей, но до начала использования создаваемого объекта.
- Конструктор не возвращает никакого значения. Поэтому в его описании не пишется даже слово `void`, но можно задать один из трех модификаторов `public`, `protected` или `private`.
- Конструктор не является методом, он даже не считается членом класса. Поэтому его нельзя наследовать или переопределить в подклассе.
- Тело конструктора может начинаться:
  - с вызова одного из конструкторов суперкласса, для этого записывается слово `super()` с параметрами в скобках, если они нужны;
  - с вызова другого конструктора того же класса, для этого записывается слово `this()` с параметрами в скобках, если они нужны.

  Если же `super()` в начале конструктора не указан, то вначале выполняется конструктор суперкласса без аргументов, затем происходит инициализация полей значениями, указанными при их объявлении, а уж потом то, что записано в конструкторе.

Во всем остальном конструктор можно считать обычным методом, в нем разрешается записывать любые операторы, даже оператор `return`, но только пустой, без всякого возвращаемого значения.

В классе может быть несколько конструкторов. Поскольку у них одно и то же имя, совпадающее с именем класса, то они должны отличаться типом и/или количеством параметров.

В наших примерах мы ни разу не рассматривали конструкторы классов, поэтому при создании экземпляров наших классов вызывался конструктор класса `Object`.

Операция *new*

Пора подробнее описать операцию с одним операндом, обозначаемую словом `new`. Она применяется для выделения памяти массивам и объектам.

В первом случае в качестве операнда указывается тип элементов массива и количество его элементов в квадратных скобках, например:

```
double a[] = new double[100];
```

Во втором случае операндом служит конструктор класса. Если конструктора в классе нет, то вызывается конструктор по умолчанию.

Числовые поля класса получают нулевые значения, логические поля — значение `false`, ссылки — значение `null`.

Результатом операции `new` будет ссылка на созданный объект. Эта ссылка может быть присвоена переменной типа ссылка на данный тип:

```
Dog k9 = new Dog();
```

но может использоваться и непосредственно

```
new Dog().voice();
```

Здесь после создания безымянного объекта сразу выполняется его метод `voice()`. Такая странная запись встречается в программах, написанных на Java, на каждом шагу.

Статические члены класса

Разные экземпляры одного класса имеют совершенно независимые друг от друга поля, принимающие разные значения. Изменение поля в одном экземпляре никак не влияет на то же поле в другом экземпляре. В каждом экземпляре для таких полей выделяется своя ячейка памяти. Поэтому такие поля называются переменными *экземпляра класса* (instance variables) или переменными *объекта*.

Иногда надо определить поле, общее для всего класса, изменение которого в одном экземпляре повлечет изменение того же поля во всех экземплярах. Например, мы хотим в классе `Automobile` отмечать порядковый заводской номер автомобиля. Такие поля называются *переменными класса* (class variables). Для переменных класса выделяется только одна ячейка памяти, общая для всех экземпляров. Переменные класса образуются в Java модификатором `static`. В листинге 2.3 мы записываем этот модификатор при определении переменной `number`.

Листинг 2.3. Статическая переменная

```
class Automobile{
  private static int number;
  Automobile(){
    number++;
    System.out.println("From Automobile constructor:"+
                       " number = "+number);
  }
}
public class AutomobileTest{
  public static void main(String[] args){
```

```
    Automobile lada2105    = new Automobile(),
               fordScorpio = new Automobile(),
               oka         = new Automobile();
  }
}
```

Получаем результат, показанный на рис. 2.2.

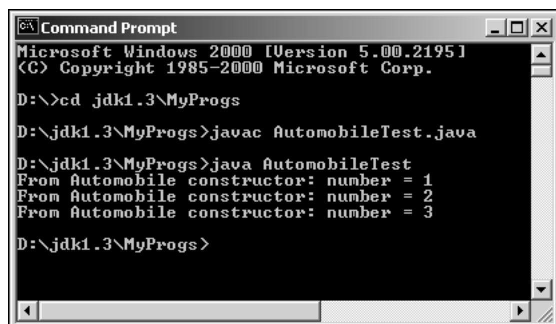


Рис. 2.2. Изменение статической переменной

Интересно, что к статическим переменным можно обращаться с именем класса, `Automobile.number`, а не только с именем экземпляра, `lada2105.number`, причем это можно делать, даже если не создан ни один экземпляр класса.

Для работы с такими *статическими переменными* обычно создаются *статические методы*, помеченные модификатором `static`. Для методов слово `static` имеет совсем другой смысл. Исполняющая система Java всегда создает в памяти только одну копию машинного кода метода, разделяемую всеми экземплярами, независимо от того, статический это метод или нет.

Основная особенность статических методов — они выполняются сразу во всех экземплярах класса. Более того, они могут выполняться, даже если не создан ни один экземпляр класса. Достаточно уточнить имя метода именем класса (а не именем объекта), чтобы метод мог работать. Именно так мы пользовались методами класса `Math`, не создавая его экземпляры, а просто записывая `Math.abs(x)`, `Math.sqrt(x)`. Точно так же мы использовали метод `System.out.println()`. Да и методом `main()` мы пользуемся, вообще не создавая никаких объектов.

Поэтому статические методы называются *методами класса* (class methods), в отличие от нестатических методов, называемых *методами экземпляра* (instance methods).

Отсюда вытекают другие особенности статических методов:

- в статическом методе нельзя использовать ссылки `this` и `super`;
- в статическом методе нельзя прямо, не создавая экземпляров, ссылаться на нестатические поля и методы;

- статические методы не могут быть абстрактными;
- статические методы переопределяются в подклассах только как статические.

Именно поэтому в листинге 1.5 мы пометили метод `f()` модификатором `static`. Но в листинге 2.1 мы работали с экземпляром `b2` класса `Bisection2`, и нам не потребовалось объявлять метод `f()` статическим.

Статические переменные инициализируются еще до начала работы конструктора, но при инициализации можно использовать только константные выражения. Если же инициализация требует сложных вычислений, например, циклов для задания значений элементам статических массивов или обращений к методам, то эти вычисления заключают в блок, помеченный словом `static`, который тоже будет выполнен до запуска конструктора:

```
static int[] a = new a[10];
static{
for(int k = 0; k < a.length; k++)
 a[k] = k * k;
}
```

Операторы, заключенные в такой блок, выполняются только один раз, при первой загрузке класса, а не при создании каждого экземпляра.

Здесь внимательный читатель, наверное, поймал меня: "А говорил, что все действия выполняются только с помощью методов!" Каюсь: блоки статической инициализации, и блоки инициализации экземпляра записываются вне всяких методов и выполняются до начала выполнения не то что метода, но даже конструктора.

Класс *Complex*

Комплексные числа широко используются не только в математике. Они часто применяются в графических преобразованиях, в построении фракталов, не говоря уже о физике и технических дисциплинах. Но класс, описывающий комплексные числа, почему-то не включен в стандартную библиотеку Java. Восполним этот пробел.

Листинг 2.4 длинный, но просмотрите его внимательно, при обучении языку программирования очень полезно чтение программ на этом языке. Более того, только программы и стоит читать, пояснения автора лишь мешают вникнуть в смысл действий (шутка).

Листинг 2.4. Класс `Complex`

```
class Complex{
  private static final double EPS = 1e-12; // Точность вычислений
  private double re, im;           // Действительная и мнимая часть
```

```
  // Четыре конструктора
Complex(double re, double im){
  this.re = re; this.im = im;
}
Complex(double re){this(re, 0.0);}
Complex(){this(0.0, 0.0);}
Complex(Complex z){this(z.re, z.im);}

  // Методы доступа
public double getRe(){return re;}
public double getIm(){return im;}
public Complex getZ(){return new Complex(re, im);}
public void setRe(double re){this.re = re;}
public void setIm(double im){this.im = im;}
public void setZ(Complex z){re = z.re; im = z.im;}

  // Модуль и аргумент комплексного числа
public double mod(){return Math.sqrt(re * re + im * im);}
public double arg(){return Math.atan2(re, im);}

  // Проверка: действительное число?
public boolean isReal(){return Math.abs(im) < EPS;}

public void pr(){              // Вывод на экран
  System.out.println(re + (im < 0.0 ? "" : "+") + im + "i");
}

  // Переопределение методов класса Object
public boolean equals(Complex z){
  return Math.abs(re — z.re) < EPS &&
         Math.abs(im — z.im) < EPS;
}
public String toString(){
  return "Complex: " + re + " " + im;
}
  // Методы, реализующие операции +=, -=, *=, /=
public void add(Complex z){re += z.re; im += z.im;}
public void sub(Complex z){re -= z.re; im -= z.im;}
public void mul(Complex z){
  double t = re * z.re — im * z.im;
        im = re * z.im + im * z.re;
        re = t;
}
public void div(Complex z){
  double m = z.mod();
  double t = re * z.re — im * z.im;
  im = (im * z.re — re * z.im) / m;
```

```
    re = t / m;
  }
    // Методы, реализующие операции +, -, *, /
  public Complex plus(Complex z){
    return new Complex(re + z.re, im + z.im);
  }
  public Complex minus(Complex z){
    return new Complex(re — z.re, im — z.im);
  }
  public Complex asterisk(Complex z){
    return new Complex(
      re * z.re — im * z.im, re * z.im + im * z.re);
  }
public Complex slash(Complex z){
    double m = z.mod();
    return new Complex(
      (re * z.re — im * z.im) / m, (im * z.re — re * z.im) / m);
  }
}

          // Проверим работу класса Complex
public class ComplexTest{
  public static void main(String[] args){
    Complex z1 = new Complex(),
            z2 = new Complex(1.5),
            z3 = new Complex(3.6, -2.2),
            z4 = new Complex(z3);
    System.out.println();        // Оставляем пустую строку
    System.out.print("z1 = "); z1.pr();
    System.out.print("z2 = "); z2.pr();
    System.out.print("z3 = "); z3.pr();
    System.out.print("z4 = "); z4.pr();
    System.out.println(z4);      // Работает метод toString()

    z2.add(z3);
    System.out.print("z2 + z3 = "); z2.pr();
    z2.div(z3);
    System.out.print("z2 / z3 = "); z2.pr();
    z2 = z2.plus(z2);
    System.out.print("z2 + z2 = "); z2.pr();
    z3 = z2.slash(z1);
    System.out.print("z2 / z1 = "); z3.pr();
  }
}
```

На рис. 2.3 показан вывод этой программы.

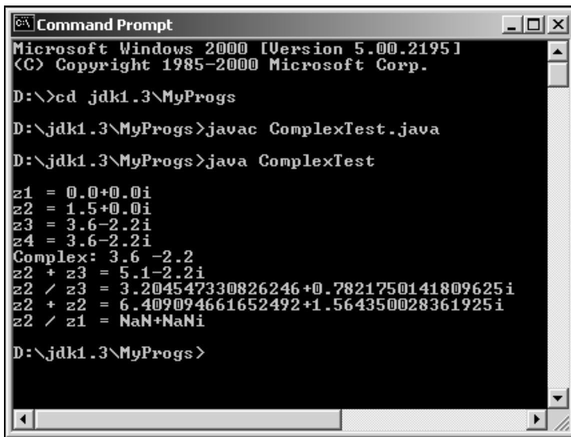


Рис. 2.3. Вывод программы ComplexTest

Метод *main()*

Всякая программа, оформленная как *приложение* (application), должна содержать метод с именем main. Он может быть один на все приложение или содержаться в некоторых классах этого приложения, а может находиться и в каждом классе.

Метод main() записывается как обычный метод, может содержать любые описания и действия, но он обязательно должен быть открытым (public), статическим (static), не иметь возвращаемого значения (void). Его аргументом обязательно должен быть массив строк (String[]). По традиции этот массив называют args, хотя имя может быть любым.

Эти особенности возникают из-за того, что метод main() вызывается автоматически исполняющей системой Java в самом начале выполнения приложения. При вызове интерпретатора java указывается класс, где записан метод main(), с которого надо начать выполнение. Поскольку классов с методом main() может быть несколько, можно построить приложение с дополнительными точками входа, начиная выполнение приложения в разных ситуациях из различных классов.

Часто метод main() заносят в каждый класс с целью отладки. В этом случае в метод main() включают тесты для проверки работы всех методов класса.

При вызове интерпретатора java можно передать в метод main() несколько параметров, которые интерпретатор заносит в массив строк. Эти параметры перечисляются в строке вызова java через пробел сразу после имени класса. Если же параметр содержит пробелы, надо заключить его в кавычки. Кавычки не будут включены в параметр, это только ограничители.

Все это легко понять на примере листинга 2.5, в котором записана программа, просто выводящая параметры, передаваемые в метод main() при запуске.

Листинг 2.5. Передача параметров в метод main()

```
class Echo{
  public static void main(String[] args){
     for (int i = 0; i < args.length; i++)
        System.out.println("args[" + i + "]= " + args[i]);
  }
}
```

На рис. 2.4 показаны результаты работы этой программы с разными вариантами задания параметров.

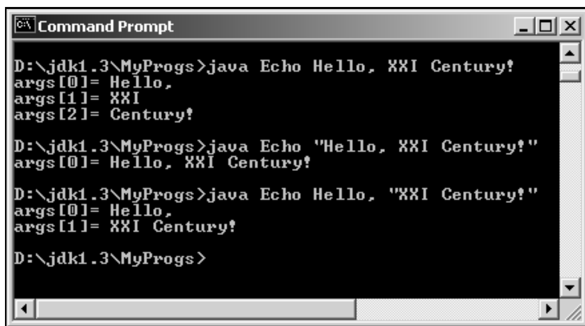


Рис. 2.4. Вывод параметров командной строки

Как видите, имя класса не входит в число параметров. Оно и так известно в методе `main()`.

Знатокам C/C++

Поскольку в Java имя файла всегда совпадает с именем класса, содержащего метод `main()`, оно не заносится в `args[0]`. Вместо `argc` используется `args.length`. Доступ к переменным среды разрешен не всегда и осуществляется другим способом. Некоторые значения можно просмотреть так:

`System.getProperties().list(System.out);`.

Где видны переменные

В языке Java нестатические переменные можно объявлять в любом месте кода между операторами. Статические переменные могут быть только полями класса, а значит, не могут объявляться внутри методов и блоков. Какова же *область видимости* (scope) переменных? Из каких методов мы можем обратиться к той или иной переменной? В каких операторах использовать? Рассмотрим на примере листинга 2.6 разные случаи объявления переменных.

Листинг 2.6. Видимость и инициализация переменных

```
class ManyVariables{

  static int x = 9, y;   // Статические переменные — поля класса
                         // Они известны во всех методах и блоках класса
                         // Переменная y получает значение 0

  static{       // Блок инициализации статических переменных
                // Выполняется один раз при первой загрузке класса после
                // инициализаций в объявлениях переменных
    x = 99;     // Оператор выполняется вне всякого метода!
  }

  int a = 1, p;            // Нестатические переменные — поля экземпляра
    // Известны во всех методах и блоках класса, в которых они
    // не перекрыты другими переменными с тем же именем
    // Переменная p получает значение 0

  {             // Блок инициализации экземпляра
                // Выполняется при создании каждого экземпляра после
                // инициализаций при объявлениях переменных
    p = 999;    // Оператор выполняется вне всякого метода!
  }

  static void f(int b){    // Параметр метода b — локальная
                           // переменная, известна только внутри метода
    int a = 2;       // Это вторая переменная с тем же именем "a"
                     // Она известна только внутри метода f() и
                     // здесь перекрывает первую "a"
    int c;           // Локальная переменная, известна только в методе f()
                     // Не получает никакого начального значения
                     // и должна быть определена перед применением

    { int c = 555;   // Ошибка! Попытка повторного объявления
      int x = 333;   // Локальная переменная, известна только в этом блоке
    }
    // Здесь переменная x уже неизвестна

    for (int d = 0; d < 10; d++){
      // Переменная цикла d известна только в цикле
      int a = 4;     // Ошибка!
      int e = 5;     // Локальная переменная, известна только в цикле for
      e++;           // Инициализируется при каждом выполнении цикла
      System.out.println("e = " + e);   // Выводится всегда "e = 6"
    }
    // Здесь переменные d и e неизвестны
  }
```

```
  public static void main(String[] args){
    int a = 9999;    // Локальная переменная, известна
                     // только внутри метода main()
    f(a);
  }
}
```

Обратите внимание на то, что переменным класса и экземпляра неявно присваиваются нулевые значения. Символы неявно получают значение `'\u0000'`, логические переменные — значение `false`, ссылки получают неявно значение `null`.

Локальные же переменные неявно не инициализируются. Им должны либо явно присваиваться значения, либо они обязаны определяться до первого использования. К счастью, компилятор замечает неопределенные локальные переменные и сообщает о них.

Внимание

Поля класса при объявлении обнуляются, локальные переменные автоматически не инициализируются.

В листинге 2.6 появилась еще одна новая конструкция: *блок инициализации экземпляра* (instance initialization). Это просто блок операторов в фигурных скобках, но записывается он вне всякого метода, прямо в теле класса. Этот блок выполняется при создании каждого экземпляра, после инициализации при объявлении переменных, но до выполнения конструктора. Он играет такую же роль, как и static-блок для статических переменных. Зачем же он нужен, ведь все его содержимое можно написать в начале конструктора? В тех случаях, когда конструктор написать нельзя, а именно, в безымянных внутренних классах.

Вложенные классы

В этой главе уже несколько раз упоминалось, что в теле класса можно сделать описание другого, *вложенного* (nested) класса. А во вложенном классе можно снова описать вложенный, *внутренний* (inner) класс и т. д. Эта матрешка кажется вполне естественной, но вы уже поднаторели в написании классов, и у вас возникает масса вопросов.

- Можем ли мы из вложенного класса обратиться к членам внешнего класса? Можем, для того это все и задумывалось.
- А можем ли мы в таком случае определить экземпляр вложенного класса, не определяя экземпляры внешнего класса? Нет, не можем, сначала надо определить хоть один экземпляр внешнего класса, матрешка ведь!

- А если экземпляров внешнего класса несколько, как узнать, с каким экземпляром внешнего класса работает данный экземпляр вложенного класса? Имя экземпляра вложенного класса уточняется именем связанного с ним экземпляра внешнего класса. Более того, при создании вложенного экземпляра операция `new` тоже уточняется именем внешнего экземпляра.
- А...?

Хватит вопросов, давайте разберем все по порядку.

Все вложенные классы можно разделить на вложенные *классы-члены* класса (member classes), описанные вне методов, и вложенные *локальные классы* (local classes), описанные внутри методов и/или блоков. Локальные классы, как и все локальные переменные, не являются членами класса.

Классы-члены могут быть объявлены статическим модификатором `static`. Поведение статических классов-членов ничем не отличается от поведения обычных классов, отличается только обращение к таким классам. Поэтому они называются *вложенными классами верхнего уровня* (nested top-level classes), хотя статические классы-члены можно вкладывать друг в друга. В них можно объявлять статические члены. Используются они обычно для того, чтобы сгруппировать вспомогательные классы вместе с основным классом.

Все нестатические вложенные классы называются *внутренними* (inner). В них нельзя объявлять статические члены.

Локальные классы, как и все локальные переменные, известны только в блоке, в котором они определены. Они могут быть *безымянными* (anonymous classes).

В листинге 2.7 рассмотрены все эти случаи.

Листинг 2.7. Вложенные классы

```
class Nested{
  static private int pr;   // Переменная pr объявлена статической,
           // чтобы к ней был доступ из статических классов A и AB
  String s = "Member of Nested";
  // Вкладываем статический класс.
  static class A{          // Полное имя этого класса — Nested.A
    private int a=pr;
    String s = "Member of A";
     // Во вложенный класс A вкладываем еще один статический класс
    static class AB{       // Полное имя класса — Nested.A.AB
      private int ab=pr;
      String s = "Member of AB";
    }
  }
```

```
  // В класс Nested вкладываем нестатический класс
  class B{                 // Полное имя этого класса — Nested.B
    private int b=pr;
    String s = "Member of B";
    // В класс B вкладываем еще один класс
    class BC{              // Полное имя класса — Nested.B.BC
        private int bc=pr;
        String s = "Member of BC";
    }
    void f(final int i){   // Без слова final переменные i и j
      final int j = 99;    // нельзя использовать в локальном классе D
      class D{         // Локальный класс D известен только внутри f()
         private int d=pr;
         String s = "Member of D";
         void pr(){
               // Обратите внимание на то, как различаются
               // переменные с одним и тем же именем "s"
           System.out.println(s + (i+j));   // "s" эквивалентно "this.s"
           System.out.println(B.this.s);
           System.out.println(Nested.this.s);
//         System.out.println(AB.this.s);   // Нет доступа
//         System.out.println(A.this.s);    // Нет доступа
         }
      }
      D d = new D();       // Объект определяется тут же, в методе f()
      d.pr();              // Объект известен только в методе f()
    }
  }
  void m(){
     new Object(){         // Создается объект безымянного класса,
                           // указывается конструктор его суперкласса
        private int e = pr;
        void g(){
          System.out.println("From g());
        }
       }.g();  // Тут же выполняется метод только что созданного объекта
  }
}

public class NestedClasses{
  public static void main(String[] args){
    Nested nest = new Nested();     // Последовательно раскрываются
                                    // три матрешки
    Nested.A theA = nest.new A();   // Полное имя класса и уточненная
          // операция new. Но конструктор только вложенного класса
```

```
    Nested.A.AB theAB = theA.new AB();    // Те же правила. Операция
                                        // new уточняется только одним именем
    Nested.B theB = nest.new B();         // Еще одна матрешка
    Nested.B.BC theBC = theB.new BC();

    theB.f(999);        // Методы вызываются обычным образом
    nest.m();
  }
}
```

Ну как? Поняли что-нибудь? Если вы все поняли и готовы применять эти конструкции в своих программах, значит вы — выдающийся талант и можете перейти к следующему пункту. Если вы ничего не поняли, значит вы — нормальный человек. Помните принцип KISS и используйте вложенные классы как можно реже.

Для остальных дадим пояснения.

- Как видите, доступ к полям внешнего класса `Nested` возможен отовсюду, даже к закрытому полю `pr`. Именно для этого в Java и введены вложенные классы. Остальные конструкции введены вынужденно, для того чтобы увязать концы с концами.
- Язык Java позволяет использовать одни и те же имена в разных областях видимости — пришлось уточнять константу `this` именем класса: `Nested.this`, `B.this`.
- В безымянном классе не может быть конструктора, ведь имя конструктора должно совпадать с именем класса, — пришлось использовать имя суперкласса, в примере это класс `Object`. Вместо конструктора в безымянном классе используется блок инициализации экземпляра.
- Нельзя создать экземпляр вложенного класса, не создав предварительно экземпляр внешнего класса, — пришлось подстраховать это правило уточнением операции `new` именем экземпляра внешнего класса — `nest.new`, `theA.new`, `theB.new`.
- При определении экземпляра указывается полное имя вложенного класса, но в операции `new` записывается просто конструктор класса.

Введение вложенных классов сильно усложнило синтаксис и поставило много задач разработчикам языка. Это еще не все. Дотошный читатель уже зарядил новую обойму вопросов.

- Можно ли наследовать вложенные классы? Можно.
- Как из подкласса обратиться к методу суперкласса? Константа `super` уточняется именем соответствующего суперкласса, подобно константе `this`.

- А могут ли вложенные классы быть расширениями других классов? Могут.
- А как? KISS!!!

Механизм вложенных классов станет понятнее, если посмотреть, какие файлы с байт-кодами создал компилятор:

- Nested1D.class — локальный класс `D`, вложенный в класс `Nested`;
- Nested$1.class — безымянный класс;
- Nested$A$AB.class — класс `Nested.A.AB`;
- Nested$A.class — класс `Nested.A`;
- Nested$B$BC.class — класс `Nested.B.BC`;
- Nested$B.class — класс `Nested.B`;
- Nested.class — внешний класс `Nested`;
- NestedClasses.class — класс с методом `main()`.

Компилятор разложил матрешки и, как всегда, создал отдельные файлы для каждого класса. При этом, поскольку в идентификаторах недопустимы точки, компилятор заменил их знаками доллара. Для безымянного класса компилятор придумал имя. Локальный класс компилятор пометил номером.

Оказывается, вложенные классы существуют только на уровне исходного кода. Виртуальная машина Java ничего не знает о вложенных классах. Она работает с обычными внешними классами. Для взаимодействия объектов вложенных классов компилятор вставляет в них специальные закрытые поля. Поэтому в локальных классах можно использовать только константы объемлющего метода, т. е. переменные, помеченные словом `final`. Виртуальная машина просто не догадается передавать изменяющиеся значения переменных в локальный класс. Таким образом не имеет смысла помечать вложенные классы `private`, все равно они выходят на самый внешний уровень.

Все эти вопросы можно не брать в голову. Вложенные классы — это прямое нарушение принципа KISS, и в Java используются только в самом простом виде, главным образом, при обработке событий, возникающих при действиях с мышью и клавиатурой.

В каких же случаях создавать вложенные классы? В теории ООП вопрос о создании вложенных классов решается при рассмотрении отношений "быть частью" и "являться".

Отношения "быть частью" и "являться"

Теперь у нас появились две различные иерархии классов. Одну иерархию образует наследование классов, другую — вложенность классов.

Определив, какие классы будут написаны в вашей программе, и сколько их будет, подумайте, как спроектировать взаимодействие классов? Вырастить пышное генеалогическое дерево классов-наследников или расписать матрешку вложенных классов?

Теория ООП советует прежде всего выяснить, в каком отношении находятся ваши классы `P` и `Q` — в отношении "класс `Q` является экземпляром класса `P`" ("a class `Q` is a class `P`") или в отношении "класс `Q` — часть класса `P`" ("a class `Q` has a class `P`").

Например: "Собака является животным" или "Собака — часть животного"? Ясно, что верно первое отношение "is-a", поэтому мы и определили класс `Dog` как расширение класса `Pet`.

Отношение "is-a" — это отношение "обобщение-детализация", отношение большей или меньшей абстракции, и ему соответствует наследование классов.

Отношение "has-a" — это отношение "целое-часть", ему соответствует вложение.

Заключение

После прочтения этой главы вы получили представление о современной парадигме программирования — объектно-ориентированном программировании и реализации этой парадигмы в языке Java. Если вас заинтересовало ООП, обратитесь к специальной литературе [3, 4, 5, 6].

Не беда, если вы не усвоили сразу принципы ООП. Для выработки "объектного" взгляда на программирование нужны время и практика. Вторая и третья части книги как раз и дадут вам эту практику. Но сначала необходимо ознакомиться с важными понятиями языка Java — пакетами и интерфейсами.

ГЛАВА 3

Пакеты и интерфейсы

В стандартную библиотеку Java API входят сотни классов. Каждый программист в ходе работы добавляет к ним десятки своих. Множество классов становится необозримым. Уже давно принято классы объединять в библиотеки. Но библиотеки классов, кроме стандартной, не являются частью языка.

Разработчики Java включили в язык дополнительную конструкцию — *пакеты* (packages). Все классы Java распределяются по пакетам. Кроме классов пакеты могут включать в себя интерфейсы и вложенные *подпакеты* (subpackages). Образуется древовидная структура пакетов и подпакетов.

Эта структура в точности отображается на структуру файловой системы. Все файлы с расширением class (содержащие байт-коды), образующие пакет, хранятся в одном каталоге файловой системы. Подпакеты собраны в подкаталоги этого каталога.

Каждый пакет образует одно *пространство имен* (namespace). Это означает, что все имена классов, интерфейсов и подпакетов в пакете должны быть уникальны. Имена в разных пакетах могут совпадать, но это будут разные программные единицы. Таким образом, ни один класс, интерфейс или подпакет не может оказаться сразу в двух пакетах. Если надо использовать два класса с одинаковыми именами из разных пакетов, то имя класса уточняется именем пакета: `пакет.Класс`. Такое уточненное имя называется *полным именем класса* (fully qualified name).

Все эти правила, опять-таки, совпадают с правилами хранения файлов и подкаталогов в каталогах.

Пакетами пользуются еще и для того, чтобы добавить к уже имеющимся правам доступа к членам класса `private`, `protected` и `public` еще один, "пакетный" уровень доступа.

Если член класса не отмечен ни одним из модификаторов `private`, `protected`, `public`, то, по умолчанию, к нему осуществляется *пакетный доступ* (default

access), а именно, к такому члену может обратиться любой метод любого класса из того же пакета. Пакеты ограничивают и доступ к классу целиком — если класс не помечен модификатором `public`, то все его члены, даже открытые, `public`, не будут видны из других пакетов.

Как же создать пакет и разместить в нем классы и подпакеты?

Пакет и подпакет

Чтобы создать пакет надо просто в первой строке java-файла с исходным кодом записать строку `package` *`имя;`*, например:

```
package mypack;
```

Тем самым создается пакет с указанным именем `mypack` и все классы, записанные в этом файле, попадут в пакет `mypack`. Повторяя эту строку в начале каждого исходного файла, включаем в пакет новые классы.

Имя подпакета уточняется именем пакета. Чтобы создать подпакет с именем, например, `subpack`, следует в первой строке исходного файла написать:

```
package mypack.subpack;
```

и все классы этого файла и всех файлов с такой же первой строкой попадут в подпакет `subpack` пакета `mypack`.

Можно создать и подпакет подпакета, написав что-нибудь вроде

```
package mypack.subpack.sub;
```

и т. д. сколько угодно раз.

Поскольку строка `package` *`имя;`* только одна и это обязательно первая строка файла, каждый класс попадает только в один пакет или подпакет.

Компилятор Java может сам создать каталог с тем же именем mypack, а в нем подкаталог subpack, и разместить в них class-файлы с байт-кодами.

Полные имена классов `A`, `B` будут выглядеть так: `mypack.A`, `mypack.subpack.B`.

Фирма SUN рекомендует записывать имена пакетов строчными буквами, тогда они не будут совпадать с именами классов, которые, по соглашению, начинаются с прописной. Кроме того, фирма SUN советует использовать в качестве имени пакета или подпакета доменное имя своего сайта, записанное в обратном порядке, например:

```
com.sun.developer
```

До сих пор мы ни разу не создавали пакет. Куда же попадали наши файлы с откомпилированными классами?

Компилятор всегда создает для таких классов *безымянный пакет* (unnamed package), которому соответствует текущий каталог (current working directory)

файловой системы. Вот поэтому у нас class-файл всегда оказывался в том же каталоге, что и соответствующий java-файл.

Безымянный пакет служит обычно хранилищем небольших пробных или промежуточных классов. Большие проекты лучше хранить в пакетах. Например, библиотека классов Java 2 API хранится в пакетах `java`, `javax`, `org.omg`. Пакет `java` содержит только подпакеты `applet`, `awt`, `beans`, `io`, `lang`, `math`, `net`, `rmi`, `security`, `sql`, `text`, `util` и ни одного класса. Эти пакеты имеют свои подпакеты, например, пакет создания ГИП и графики `java.awt` содержит подпакеты `color`, `datatransfer`, `dnd`, `event`, `font`, `geometry`, `im`, `image`, `print`.

Конечно, состав пакетов меняется от версии к версии.

Права доступа к членам класса

Пришло время подробно разобрать различные ограничения доступа к полям и методам класса.

Рассмотрим большой пример. Пусть имеется пять классов, размещенных в двух пакетах, как показано на рис. 3.1.

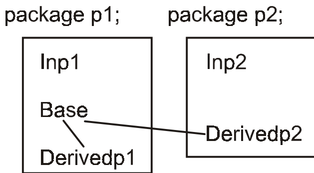


Рис. 3.1. Размещение наших классов по пакетам

В файле Base.java описаны три класса: `Inp1`, `Base` и класс `Derivedp1`, расширяющий класс `Base`. Эти классы размещены в пакете `p1`. В классе `Base` определены переменные всех четырех типов доступа, а в методах `f()` классов `Inp1` и `Derivedp1` сделана попытка доступа ко всем полям класса `Base`. Неудачные попытки отмечены комментариями. В комментариях помещены сообщения компилятора. Листинг 3.1 показывает содержимое этого файла.

Листинг 3.1. Файл Base.java с описанием пакета `p1`

```
package p1;

class Inp1{
  public void f(){
    Base b = new Base();
//    b.priv = 1; // "priv has private access in p1.Base"
    b.pack = 1;
    b.prot = 1;
```

```
    b.publ = 1;
  }
}

public class Base{
  private   int priv = 0;
            int pack = 0;
  protected int prot = 0;
  public    int publ = 0;
}

class Derivedp1 extends Base{
  public void f(Base a){
//    a.priv = 1;        // "priv has private access in p1.Base"
    a.pack = 1;
    a.prot = 1;
    a.publ = 1;
//    priv = 1;          // "priv has private access in p1.Base"
    pack = 1;
    prot = 1;
    publ = 1;
  }
}
```

Как видно из листинга 3.1, в пакете недоступны только закрытые, `private`, поля другого класса.

В файле Inp2.java описаны два класса: `Inp2` и класс `Derivedp2`, расширяющий класс `Base`. Эти классы находятся в другом пакете `p2`. В этих классах тоже сделана попытка обращения к полям класса `Base`. Неудачные попытки прокомментированы сообщениями компилятора. Листинг 3.2 показывает содержимое этого файла.

Напомним, что класс `Base` должен быть помечен при своем описании в пакете `p1` модификатором `public`, иначе из пакета `p2` не будет видно ни одного его члена.

Листинг 3.2. Файл Inp2.java с описанием пакета `p2`

```
package p2;
import p1.Base;

class Inp2{
  public static void main(String[] args){
    Base b = new Base();
//    b.priv = 1;      // "priv has private access in p1.Base"
//    b.pack = 1;      // "pack is not public in p1.Base;
                       // cannot be accessed from outside package"
```

```
//    b.prot = 1;       // "prot has protected access in p1.Base"
    b.publ = 1;
  }
}

class Derivedp2 extends Base{
  public void f(Base a){
//    a.priv = 1;       // "priv has private access in p1.Base"
//    a.pack = 1;       // "pack is not public in p1.Base; cannot
                        // be accessed from outside package"
//    a.prot = 1;       // "prot has protected access in p1.Base"
    a.publ = 1;
//    priv = 1;         // "priv has private access in p1.Base"
//    pack = 1;         // "pack is not public in p1.Base; cannot
                        // be accessed from outside package"
    prot = 1;
    publ = 1;
    super.prot = 1;
  }
}
```

Здесь, в другом пакете, доступ ограничен в большей степени.

Из независимого класса можно обратиться только к открытым, `public`, полям класса другого пакета. Из подкласса можно обратиться еще и к защищенным, `protected`, полям, но только унаследованным непосредственно, а не через экземпляр суперкласса.

Все указанное относится не только к полям, но и к методам.

Подытожим все сказанное в табл. 3.1.

Таблица 3.1. *Права доступа к полям и методам класса*

	Класс	**Пакет**	**Пакет и подклассы**	**Все классы**
`private`	+			
`"package"`	+	+		
`protected`		+	+	*
`public`	+	+	+	+

Особенность доступа к `protected`-полям и методам из чужого пакета отмечена звездочкой.

Размещение пакетов по файлам

То обстоятельство, что class-файлы, содержащие байт-коды классов, должны быть размещены по соответствующим каталогам, накладывает свои особенности на процесс компиляции и выполнения программы.

Обратимся к тому же примеру. Пусть в каталоге D:\jdk1.3\MyProgs\ch3 есть пустой подкаталог classes и два файла — Base.java и Inp2.java, — содержимое которых показано в листингах 3.1 и 3.2. Рис. 3.2 демонстрирует структуру каталогов уже после компиляции.

Мы можем проделать всю работу вручную.

1. В каталоге classes создаем подкаталоги p1 и p2.
2. Переносим файл Base.java в каталог p1 и делаем p1 текущим каталогом.
3. Компилируем Base.java, получая в каталоге p1 три файла: Base.class, Inp1.class, Derivedp1.class.
4. Переносим файл Inp2.java в каталог p2.
5. Снова делаем текущим каталог classes.
6. Компилируем второй файл, указывая путь p2\Inp2.java.
7. Запускаем программу `java p2.Inp2`.

Вместо шагов 2 и 3 можно просто создать три class-файла в любом месте, а потом перенести их в каталог p1. В class-файлах не хранится никакая информация о путях к файлам.

Смысл действий 5 и 6 в том, что при компиляции файла Inp2.java компилятор уже должен знать класс `p1.Base`, а отыскивает он файл с этим классом по пути p1\Base.class, начиная от текущего каталога.

Обратите внимание на то, что в последнем действии 7 надо указывать полное имя класса.

Если использовать ключи (options) командной строки компилятора, то можно выполнить всю работу быстрее.

1. Вызываем компилятор с ключом `-d` *`путь`*, указывая параметром `путь` начальный каталог для пакета:

   ```
   javac -d classes Base.java
   ```

   Компилятор создаст в каталоге classes подкаталог p1 и поместит туда три class-файла.

2. Вызываем компилятор с еще одним ключом `-classpath` *`путь`*, указывая параметром `путь` каталог classes, в котором находится подкаталог с уже откомпилированным пакетом `p1`:

   ```
   javac -classpath classes -d classes Inp2.java
   ```

Компилятор, руководствуясь ключом `-d`, создаст в каталоге classes подкаталог p2 и поместит туда два class-файла, при создании которых он "заглядывал" в каталог p1, руководствуясь ключом `-classpath`.

3. Делаем текущим каталог classes.
4. Запускаем программу `java p2.Inp2`.

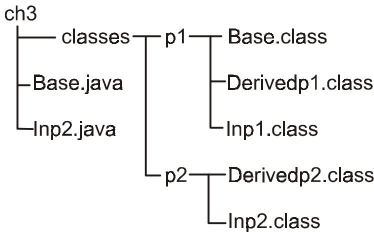


Рис. 3.2. Структура каталогов

```
Command Prompt
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

D:\>cd jdk1.3\MyProgs\ch3

D:\jdk1.3\MyProgs\ch3>javac -d classes Base.java

D:\jdk1.3\MyProgs\ch3>javac -classpath classes -d classes Inp2.java

D:\jdk1.3\MyProgs\ch3>dir /s
 Volume in drive D has no label.
 Volume Serial Number is AC95-B8D2

 Directory of D:\jdk1.3\MyProgs\ch3

18.10.2000  18:05       <DIR>          .
18.10.2000  18:05       <DIR>          ..
18.10.2000  17:44                  604 Base.java
18.10.2000  18:08       <DIR>          classes
18.10.2000  17:48                  911 Inp2.java
               2 File(s)          1 515 bytes

 Directory of D:\jdk1.3\MyProgs\ch3\classes

18.10.2000  18:08       <DIR>          .
18.10.2000  18:08       <DIR>          ..
18.10.2000  18:07       <DIR>          p1
18.10.2000  18:08       <DIR>          p2
               0 File(s)              0 bytes

 Directory of D:\jdk1.3\MyProgs\ch3\classes\p1

18.10.2000  18:07       <DIR>          .
18.10.2000  18:07       <DIR>          ..
18.10.2000  18:11                  329 Base.class
18.10.2000  18:11                  348 Derivedp1.class
18.10.2000  18:11                  340 Inp1.class
               3 File(s)          1 017 bytes

 Directory of D:\jdk1.3\MyProgs\ch3\classes\p2

18.10.2000  18:08       <DIR>          .
18.10.2000  18:08       <DIR>          ..
18.10.2000  18:12                  313 Derivedp2.class
18.10.2000  18:12                  316 Inp2.class
               2 File(s)            629 bytes

     Total Files Listed:
               7 File(s)          3 161 bytes
              11 Dir(s)   3 744 546 816 bytes free

D:\jdk1.3\MyProgs\ch3>cd classes

D:\jdk1.3\MyProgs\ch3\classes>java p2.Inp2
```

Рис. 3.3. Протокол компиляции и запуска программы

Для "юниксоидов" все это звучит, как музыка, ну а прочим придется вспомнить MS DOS.

Конечно, если вы используете для работы не компилятор командной строки, а какое-нибудь IDE, то все эти действия будут сделаны без вашего участия.

На рис. 3.2 отображена структура каталогов после компиляции.

На рис. 3.3 показан вывод этих действий в окно **Command Prompt** и содержимое каталогов после компиляции.

Импорт классов и пакетов

Внимательный читатель заметил во второй строке листинга 3.2 новый оператор `import`. Для чего он нужен?

Дело в том, что компилятор будет искать классы только в одном пакете, именно, в том, что указан в первой строке файла. Для классов из другого пакета надо указывать полные имена. В нашем примере они короткие, и мы могли бы писать в листинге 3.2 вместо `Base` полное имя `p1.Base`.

Но если полные имена длинные, а используются классы часто, то стучать по клавишам, набирая полные имена, становится утомительно. Вот тут-то мы и пишем операторы `import`, указывая компилятору полные имена классов.

Правила использования оператора `import` очень просты: пишется слово `import` и, через пробел, полное имя класса, завершенное точкой с запятой. Сколько классов надо указать, столько операторов `import` и пишется.

Это тоже может стать утомительным и тогда используется вторая форма оператора `import` — указывается имя пакета или подпакета, а вместо короткого имени класса ставится звездочка `*`. Этой записью компилятору предписывается просмотреть весь пакет. В нашем примере можно было написать

```
import p1.*;
```

Напомним, что импортировать можно только открытые классы, помеченные модификатором `public`.

Внимательный читатель и тут настороже. Мы ведь пользовались методами классов стандартной библиотеки, не указывая ее пакетов? Да, правильно.

Пакет `java.lang` просматривается всегда, его необязательно импортировать. Остальные пакеты стандартной библиотеки надо указывать в операторах `import`, либо записывать полные имена классов.

Подчеркнем, что оператор `import` вводится только для удобства программистов и слово "импортировать" не означает никаких перемещений классов.

Знатокам C/C++

Оператор `import` не эквивалентен директиве препроцессора `include` — он не подключает никакие файлы.

Java-файлы

Теперь можно описать структуру исходного файла с текстом программы на языке Java.

- В первой строке файла может быть необязательный оператор `package`.
- В следующих строках могут быть необязательные операторы `import`.
- Далее идут описания классов и интерфейсов.

Еще два правила.

- Среди классов файла может быть только один открытый `public`-класс.
- Имя файла должно совпадать с именем открытого класса, если последний существует.

Отсюда следует, что, если в проекте есть несколько открытых классов, то они должны находиться в разных файлах.

Соглашение "Code Conventions" рекомендует открытый класс, который, если он имеется в файле, нужно описывать первым.

Интерфейсы

Вы уже заметили, что получить расширение можно только от одного класса, каждый класс `B` или `C` происходит из неполной семьи, как показано на рис. 3.4, а. Все классы происходят только от "Адама", от класса `Object`. Но часто возникает необходимость породить класс `D` от двух классов `B` и `C`, как показано на рис. 3.4, б. Это называется *множественным наследованием* (multiple inheritance). В множественном наследовании нет ничего плохого. Трудности возникают, если классы `B` и `C` сами порождены от одного класса `A`, как показано на рис. 3.4, в. Это так называемое "ромбовидное" наследование.

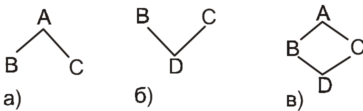


Рис. 3.4. Разные варианты наследования

В самом деле, пусть в классе `A` определен метод `f()`, к которому мы обращаемся из некоего метода класса `D`. Можем мы быть уверены, что метод `f()` выполняет то, что написано в классе `A`, т. е. это метод `A.f()`? Может, он переопределен в классах `B` и `C`? Если так, то каким вариантом мы пользуемся: `B.f()` или `C.f()`? Конечно, можно определить экземпляры классов и обращаться к методам этих экземпляров, но это совсем другой разговор.

В разных языках программирования этот вопрос решается по-разному, главным образом, уточнением имени метода `f()`. Но при этом всегда нару-

шается принцип KISS. Вокруг множественного наследования всегда много споров, есть его ярые приверженцы и столь же ярые противники. Не будем встревать в эти споры, наше дело — наилучшим образом использовать средства языка для решения своих задач.

Создатели языка Java после долгих споров и размышлений поступили радикально — запретили множественное наследование вообще. При расширении класса после слова `extends` можно написать только одно имя суперкласса. С помощью уточнения `super` можно обратиться только к членам непосредственного суперкласса.

Но что делать, если все-таки при порождении надо использовать несколько предков? Например, у нас есть общий класс автомобилей `Automobile`, от которого можно породить класс грузовиков `Truck` и класс легковых автомобилей `Car`. Но вот надо описать пикап `Pickup`. Этот класс должен наследовать свойства и грузовых, и легковых автомобилей.

В таких случаях используется еще одна конструкция языка Java — интерфейс. Внимательно проанализировав ромбовидное наследование, теоретики ООП выяснили, что проблему создает только реализация методов, а не их описание.

Интерфейс (interface), в отличие от класса, содержит только константы и заголовки методов, без их реализации.

Интерфейсы размещаются в тех же пакетах и подпакетах, что и классы, и компилируются тоже в class-файлы.

Описание интерфейса начинается со слова `interface`, перед которым может стоять модификатор `public`, означающий, как и для класса, что интерфейс доступен всюду. Если же модификатора `public` нет, интерфейс будет виден только в своем пакете.

После слова `interface` записывается имя интерфейса, потом может стоять слово `extends` и список интерфейсов-предков через запятую. Таким образом, интерфейсы могут порождаться от интерфейсов, образуя свою, независимую от классов, иерархию, причем в ней допускается множественное наследование интерфейсов. В этой иерархии нет корня, общего предка.

Затем, в фигурных скобках, записываются в любом порядке константы и заголовки методов. Можно сказать, что в интерфейсе все методы абстрактные, но слово `abstract` писать не надо. Константы всегда статические, но слова `static` и `final` указывать не нужно.

Все константы и методы в интерфейсах всегда открыты, не надо даже указывать модификатор `public`.

Вот какую схему можно предложить для иерархии автомобилей:

```
interface Automobile{ . . . }
interface Car extends Automobile{ . . . }
```

```
interface Truck extends Automobile{ . . . }
interface Pickup extends Car, Truck{ . . . }
```

Таким образом, интерфейс — это только набросок, эскиз. В нем указано, что делать, но не указано, как это делать.

Как же использовать интерфейс, если он полностью абстрактен, в нем нет ни одного полного метода?

Использовать нужно не интерфейс, а его *реализацию* (implementation). Реализация интерфейса — это класс, в котором расписываются методы одного или нескольких интерфейсов. В заголовке класса после его имени или после имени его суперкласса, если он есть, записывается слово `implements` и, через запятую, перечисляются имена интерфейсов.

Вот как можно реализовать иерархию автомобилей:

```
interface Automobile{ . . . }
interface Car extends Automobile{ . . . }
class Truck implements Automobile{ . . . }
class Pickup extends Truck implements Car{ . . . }
```

или так:

```
interface Automobile{ . . . }
interface Car extends Automobile{ . . . }
interface Truck extends Automobile{ . . . }
class Pickup implements Car, Truck{ . . . }
```

Реализация интерфейса может быть неполной, некоторые методы интерфейса расписаны, а другие — нет. Такая реализация — абстрактный класс, его обязательно надо пометить модификатором `abstract`.

Как реализовать в классе `Pickup` метод `f()`, описанный и в интерфейсе `Car`, и в интерфейсе `Truck` с одинаковой сигнатурой? Ответ простой — никак. Такую ситуацию нельзя реализовать в классе `Pickup`. Программу надо спроектировать по-другому.

Итак, интерфейсы позволяют реализовать средствами Java чистое объектно-ориентированное проектирование, не отвлекаясь на вопросы реализации проекта.

Мы можем, приступая к разработке проекта, записать его в виде иерархии интерфейсов, не думая о реализации, а затем построить по этому проекту иерархию классов, учитывая ограничения одиночного наследования и видимости членов классов.

Интересно то, что мы можем создавать ссылки на интерфейсы. Конечно, указывать такая ссылка может только на какую-нибудь реализацию интерфейса. Тем самым мы получаем еще один способ организации полиморфизма.

Листинг 3.3 показывает, как можно собрать с помощью интерфейса хор домашних животных из листинга 2.2.

Листинг 3.3. Использование интерфейса для организации полиморфизма

```
interface Voice{
  void voice();
}

class Dog implements Voice{
  public void voice(){
    System.out.println("Gav-gav!");
  }
}

class Cat implements Voice{
  public void voice(){
    System.out.println("Miaou!");
  }
}

class Cow implements Voice{
  public void voice(){
    System.out.println("Mu-u-u!");
  }
}

public class Chorus{
  public static void main(String[] args){
     Voice[] singer = new Voice[3];
     singer[0] = new Dog();
     singer[1] = new Cat();
     singer[2] = new Cow();
     for(int i = 0; i < singer.length; i++)
        singer[i].voice();
  }
}
```

Здесь используется интерфейс `Voice` вместо абстрактного класса `Pet`, описанного в листинге 2.2.

Что же лучше использовать: абстрактный класс или интерфейс? На этот вопрос нет однозначного ответа.

Создавая абстрактный класс, вы волей-неволей погружаете его в иерархию классов, связанную условиями одиночного наследования и единым предком — классом `Object`. Пользуясь интерфейсами, вы можете свободно проектировать систему, не задумываясь об этих ограничениях.

С другой стороны, в абстрактных классах можно сразу реализовать часть методов. Реализуя же интерфейсы, вы обречены на скучное переопределение всех методов.

Вы, наверное, заметили и еще одно ограничение: все реализации методов интерфейсов должны быть открытыми, `public`, поскольку при переопределении можно лишь расширять доступ, а методы интерфейсов всегда открыты.

Вообще же наличие и классов, и интерфейсов дает разработчику богатые возможности проектирования. В нашем примере, вы можете включить в хор любой класс, просто реализовав в нем интерфейс `Voice`.

Наконец, можно использовать интерфейсы просто для определения констант, как показано в листинге 3.4.

Листинг 3.4. Система управления светофором

```
interface Lights{
  int RED    =  0;
  int YELLOW =  1;
  int GREEN  =  2;
  int ERROR  = -1;
}
class Timer implements Lights{
  private int delay;
  private static int light = RED;

  Timer(int sec){delay = 1000 * sec;}

  public int shift(){
    int count = (light++) % 3;
    try{
      switch(count){
        case RED:    Thread.sleep(delay);   break;
        case YELLOW: Thread.sleep(delay/3); break;
        case GREEN:  Thread.sleep(delay/2); break;
      }
    }catch(Exception e){return ERROR;}
    return count;
  }
}

class TrafficRegulator{
  private static Timer t = new Timer(1);

  public static void main(String[] args){
    for(int k = 0; k < 10; k++)
      switch(t.shift()){
```

```
        case Lights.RED:    System.out.println("Stop!"); break;
        case Lights.YELLOW: System.out.println("Wait!"); break;
        case Lights.GREEN:  System.out.println("Go!");   break;
        case Lights.ERROR:  System.err.println("Time Error"); break;
        default: System.err.println("Unknown light."); return;
      }
  }
}
```

Здесь, в интерфейсе `Lights`, определены константы, общие для всего проекта.

Класс `Timer` реализует этот интерфейс и использует константы напрямую как свои собственные. Метод `shift()` этого класса подает сигналы переключения светофору с разной задержкой в зависимости от цвета. Задержку осуществляет метод `sleep()` класса `Thread` из стандартной библиотеки, которому передается время задержки в миллисекундах. Этот метод нуждается в обработке исключений `try{}catch(){}`, о которой мы будем говорить в *главе 16*.

Класс `TrafficRegulator` не реализует интерфейс `Lights` и пользуется полными именами `Lights.RED` и т. д. Это возможно потому, что константы `RED`, `YELLOW` и `GREEN` по умолчанию являются статическими.

Теперь нам известны все средства языка Java, позволяющие проектировать решение поставленной задачи. Заканчивая разговор о проектировании, нельзя не упомянуть о постоянно пополняемой коллекции образцов проектирования (design patterns).

Design patterns

В математике давно выработаны общие методы решения типовых задач. Доказательство теоремы начинается со слов: "Проведем доказательство от противного" или: "Докажем это методом математической индукции", и вы сразу представляете себе схему доказательства, его путь становится вам понятен.

Нет ли подобных общих методов в программировании? Есть.

Допустим, вам поручили автоматизировать метеорологическую станцию. Информация от различных датчиков или, другими словами, *контроллеров* температуры, давления, влажности, скорости ветра поступает в цифровом виде в компьютер. Там она обрабатывается: вычисляются усредненные значения по регионам, на основе многодневных наблюдений делается прогноз на завтра, т. е. создается *модель* метеорологической картины местности. Затем прогноз выводится по разным каналам: на экран монитора, самописец, передается по сети. Он представляется в разных *видах*: колонках чисел, графиках, диаграммах.

Естественно спроектировать такую автоматизированную систему из трех частей.

- Первая часть, назовем ее *Контроллером* (controller), принимает сведения от датчиков и преобразует их в какую-то единообразную форму, пригодную для дальнейшей обработки, например, приводит к одному масштабу. При этом для каждого датчика надо написать свой модуль, на вход которого поступают сигналы конкретного устройства, а на выходе образуется унифицированная информация.
- Вторая часть, назовем ее *Моделью* (model), принимает эту унифицированную информацию от Контроллера, ничего не зная о датчике и не интересуясь тем, от какого именно датчика она поступила, и преобразует ее по своим алгоритмам опять-таки к какому-то однообразному виду, например, к последовательности чисел.
- Третья часть системы, *Вид* (view), непосредственно связана с устройствами вывода и преобразует поступившую от Модели последовательность чисел в график, диаграмму или пакет для отправки по сети. Для каждого устройства придется написать свой модуль, учитывающий особенности именно этого устройства.

В чем удобство такой трехзвенной схемы? Она очень гибка. Замена одного датчика приведет к замене только одного модуля в Контроллере, ни Модель, ни Вид этого даже не заметят. Надо представить прогноз в каком-то новом виде, например, для телевидения? Пожалуйста, достаточно написать один модуль и вставить его в Вид. Изменился алгоритм обработки данных? Меняем Модель.

Эта схема разработана еще в 80-х годах прошлого столетия[1] в языке Smalltalk и получила название MVC (Model-View-Controller). Оказалось, что она применима во многих областях, далеких от метеорологии, всюду, где удобно отделить обработку от ввода и вывода информации.

Сбор информации часто организуется так. На экране дисплея открывается поле ввода, в которое вы набиваете сведения, допустим, фамилии в произвольном порядке, а в соседнем поле вывода отображается обработанная информация, например, список фамилий по алфавиту. Будьте уверены, что эта программа организована по схеме MVC. Контроллером служит поле ввода, Видом — поле вывода, а Моделью — метод сортировки фамилий. В третьей части книги мы рассмотрим примеры реализации этой схемы.

К середине 90-х годов накопилось много подобных схем. В них сконцентрирован многолетний опыт тысяч программистов, выражены наилучшие решения типовых задач.

---

[1] То есть XX века. — *Ред.*

Вот, пожалуй, самая простая из этих схем. Надо написать класс, у которого можно создать только один экземпляр, но этим экземпляром должны пользоваться объекты других классов. Для решения этой задачи предложена схема Singleton, представленная в листинге 3.5.

Листинг 3.5. Схема Singleton

```
final class Singleton{
  private static Singleton s = new Singleton(0);
  private int k;
  private Singleton(int i){k = i;}
  public static Singleton getReference(){return s;}
  public int getValue(){return k;}
  public void setValue(int i){k = i;}
}
public class SingletonTest{
  public static void main(String[] args){
    Singleton ref = Singleton.getReference();
    System.out.println(ref.getValue());
    ref.setValue(ref.getValue() + 5);
    System.out.println(ref.getValue());
  }
}
```

Класс `Singleton` окончательный — его нельзя расширить. Его конструктор закрытый — никакой метод не может создать экземпляр этого класса. Единственный экземпляр `s` класса `Singleton` — статический, он создается внутри класса. Зато любой объект может получить ссылку на экземпляр методом `getReference()`, изменить состояние экземпляра `s` методом `setValue()` или просмотреть его текущее состояние методом `getValue()`.

Это только схема — класс `Singleton` надо еще наполнить полезным содержимым, но идея выражена ясно и полностью.

Схемы проектирования были систематизированы и изложены в книге [7]. Четыре автора этой книги были прозваны "бандой четырех" (Gang of Four), а книга, коротко, "GoF". Схемы обработки информации получили название "Design Patterns". Русский термин еще не устоялся. Говорят о "шаблонах", "схемах разработки", "шаблонах проектирования".

В книге GoF описаны 23 шаблона, разбитые на три группы:

1. Шаблоны создания объектов: Factory, Abstract Factory, Singleton, Builder, Prototype.
2. Шаблоны структуры объектов: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.

3. Шаблоны поведения объектов: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template, Visitor.

Описания даны, в основном, на языке C++. В книге [8] те же шаблоны представлены на языке Java. Той же теме посвящено электронное издание [9]. В книге [10] подробно обсуждаются вопросы разработки систем на основе design patterns.

Мы, к сожалению, не можем разобрать подробно design patterns в этой книге. Но каждый программист начала XXI века должен их знать. Описание многих разработок начинается словами: "Проект решен на основе шаблона", и структура проекта сразу становится ясна для всякого, знакомого с design patterns.

По ходу книги мы будем указывать, на основе какого шаблона сделана та или иная разработка.

Заключение

Вот мы и закончили первую часть книги. Теперь вы знаете все основные конструкции языка Java, позволяющие спроектировать и реализовать проект любой сложности на основе ООП. Оставшиеся конструкции языка, не менее важные, но реже используемые, отложим до четвертой части. Вторую и третью часть книги посвятим изучению классов и методов, входящих в Core API. Это будет для вас хорошей тренировкой.

Язык Java, как и все современные языки программирования, — это не только синтаксические конструкции, но и богатая библиотека классов. Знание этих классов и умение пользоваться ими как раз и определяет программиста-практика.

Часть II

Использование классов, входящих в Java Development Kit

Глава 4. Классы-оболочки

Глава 5. Работа со строками

Глава 6. Классы-коллекции

Глава 7. Классы-утилиты

ГЛАВА 4

Классы-оболочки

Java — полностью объектно-ориентированный язык. Это означает, что все, что только можно, в Java представлено объектами.

Восемь примитивных типов нарушают это правило. Они оставлены в Java из-за многолетней привычки к числам и символам. Да и арифметические действия удобнее и быстрее производить с обычными числами, а не с объектами классов.

Но и для этих типов в языке Java есть соответствующие классы — *классы-оболочки* (wrapper) примитивных типов. Конечно, они предназначены не для вычислений, а для действий, типичных при работе с классами — создания объектов, преобразования объектов, получения численных значений объектов в разных формах и передачи объектов в методы по ссылке.

На рис. 4.1 показана одна из ветвей иерархии классов Java. Для каждого примитивного типа есть соответствующий класс. Числовые классы имеют общего предка — абстрактный класс `Number`, в котором описаны шесть методов, возвращающих числовое значение, содержащееся в классе, приведенное к соответствующему примитивному типу: `byteValue()`, `doubleValue()`, `floatValue()`, `intValue()`, `longValue()`, `shortValue()`. Эти методы переопределены в каждом из шести числовых классов-оболочек.

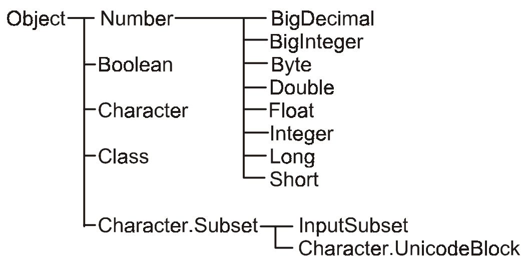


Рис. 4.1. Классы примитивных типов

Помимо метода сравнения объектов `equals()`, переопределенного из класса `Object`, все описанные в этой главе классы, кроме `Boolean` и `Class`, имеют метод `compareTo()`, сравнивающий числовое значение, содержащееся в данном объекте, с числовым значением объекта — аргумента метода `compareTo()`. В результате работы метода получается целое значение:

- 0, если значения равны;
- отрицательное число (−1), если числовое значение в данном объекте меньше, чем в объекте-аргументе;
- положительное число (+1), если числовое значение в данном объекте больше числового значения, содержащегося в аргументе.

Что полезного в классах-оболочках?

Числовые классы

В каждом из шести числовых классов-оболочек есть статические методы преобразования строки символов типа `String`, представляющей число, в соответствующий примитивный тип: `Byte.parseByte()`, `Double.parseDouble()`, `Float.parseFloat()`, `Integer.parseInt()`, `Long.parseLong()`, `Short.parseShort()`. Исходная строка типа `String`, как всегда в статических методах, задается как аргумент метода. Эти методы полезны при вводе данных в поля ввода, обработке параметров командной строки, т. е. всюду, где числа представляются строками цифр со знаками плюс или минус и десятичной точкой.

В каждом из этих классов есть статические константы `MAX_VALUE` и `MIN_VALUE`, показывающие диапазон числовых значений соответствующих примитивных типов. В классах `Double` и `Float` есть еще константы `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, `NaN`, о которых шла речь в *главе 1*, и логические методы проверки `isNaN()`, `isInfinite()`.

Если вы хорошо знаете двоичное представление вещественных чисел, то можете воспользоваться статическими методами `floatToIntBits()` и `doubleToLongBits()`, преобразующими вещественное значение в целое. Вещественное число задается как аргумент метода. Затем вы можете изменить отдельные биты побитными операциями и преобразовать измененное целое число обратно в вещественное значение методами `intBitsToFloat()` и `longBitsToDouble()`.

Статическими методами `toBinaryString()`, `toHexString()` и `toOctalString()` классов `Integer` и `Long` можно преобразовать целые значения типов `int` и `long`, заданные как аргумент метода, в строку символов, показывающую двоичное, шестнадцатеричное или восьмеричное представление числа.

В листинге 4.1 показано применение этих методов, а рис. 4.2 демонстрирует вывод результатов.

```
Command Prompt
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

D:\>cd jdk1.3\MyProgs

D:\jdk1.3\MyProgs>javac NumberTest.java

D:\jdk1.3\MyProgs>java NumberTest -20456 45.758
i = -20456
sh = -20456
d = 45.758
k1.intValue() = -20456
d1.intValue() = 45
k1 > k2? -1
x = Infinity
x isNaN? false
x isInfinite? true
x == Infinity? true
d = 4631636683301662491
i = 11111111111111111011000000011000
i = ffffb018
i = 37777730030

D:\jdk1.3\MyProgs>
```

Рис. 4.2. Методы числовых классов

Листинг 4.1. Методы числовых классов

```
class NumberTest{
  public static void main(String[] args){
    int i = 0;
    short sh = 0;
    double d = 0;
    Integer k1 = new Integer(55);
    Integer k2 = new Integer(100);
    Double d1  = new Double(3.14);
    try{
       i = Integer.parseInt(args[0]);
      sh = Short.parseShort(args[0]);
       d = Double.parseDouble(args[1]);
      d1 = new Double(args[1]);
      k1 = new Integer(args[0]);
    }catch(Exception e){}
    double x = 1.0 / 0.0;
    System.out.println("i = " + i);
    System.out.println("sh = " + sh);
    System.out.println("d = " + d);
    System.out.println("k1.intValue() = " + k1.intValue());
    System.out.println("d1.intValue() = " + d1.intValue());
    System.out.println("k1 > k2? " + k1.compareTo(k2));
    System.out.println("x = " + x);
    System.out.println("x isNaN? " + Double.isNaN(x));
    System.out.println("x isInfinite? " + Double.isInfinite(x));
    System.out.println("x == Infinity? " +
          (x == Double.POSITIVE_INFINITY));
```

```
    System.out.println("d = " + Double.doubleToLongBits(d));
    System.out.println("i = " + Integer.toBinaryString(i));
    System.out.println("i = " + Integer.toHexString(i));
    System.out.println("i = " + Integer.toOctalString(i));
  }
}
```

Методы `parseInt()` и конструкторы классов требуют обработки исключений, поэтому в листинг 4.1 вставлен блок `try{}catch(){}`. Обработку исключительных ситуаций мы разберем в *главе 16*.

Класс *Boolean*

Это очень небольшой класс, предназначенный главным образом для того, чтобы передавать логические значения в методы по ссылке.

Конструктор `Boolean(String s)` создает объект, содержащий значение `true`, если строка `s` равна `"true"` в любом сочетании регистров букв, и значение `false` — для любой другой строки.

Логический метод `booleanValue()` возвращает логическое значение, хранящееся в объекте.

Класс *Character*

В этом классе собраны статические константы и методы для работы с отдельными символами.

Статический метод

```
digit(char ch, in radix)
```

переводит цифру `ch` системы счисления с основанием `radix` в ее числовое значение типа `int`.

Статический метод

```
forDigit(int digit, int radix)
```

производит обратное преобразование целого числа `digit` в соответствующую цифру (тип `char`) в системе счисления с основанием `radix`.

Основание системы счисления должно находиться в диапазоне от `Character.MIN_RADIX` до `Character.MAX_RADIX`.

Метод `toString()` переводит символ, содержащийся в классе, в строку с тем же символом.

Статические методы `toLowerCase()`, `toUpperCase()`, `toTitleCase()` возвращают символ, содержащийся в классе, в указанном регистре. Последний из

этих методов предназначен для правильного перевода в верхний регистр четырех кодов Unicode, не выражающихся одним символом.

Множество статических логических методов проверяют различные характеристики символа, переданного в качестве аргумента метода:

- `isDefined()` — выясняет, определен ли символ в кодировке Unicode;
- `isDigit()` — проверяет, является ли символ цифрой Unicode;
- `isIdentifierIgnorable()` — выясняет, нельзя ли использовать символ в идентификаторах;
- `isISOControl()` — определяет, является ли символ управляющим;
- `isJavaIdentifierPart()` — выясняет, можно ли использовать символ в идентификаторах;
- `isJavaIdentifierStart()` — определяет, может ли символ начинать идентификатор;
- `isLetter()` — проверяет, является ли символ буквой Java;
- `isLetterOrDigit()` — проверяет, является ли символ буквой или цифрой Unicode;
- `isLowerCase()` — определяет, записан ли символ в нижнем регистре;
- `isSpaceChar()` — выясняет, является ли символ пробелом в смысле Unicode;
- `isTitleCase()` — проверяет, является ли символ титульным;
- `isUnicodeIdentifierPart()` — выясняет, можно ли использовать символ в именах Unicode;
- `isUnicodeIdentifierStart()` — проверяет, является ли символ буквой Unicode;
- `isUpperCase()` — проверяет, записан ли символ в верхнем регистре;
- `isWhitespace()` — выясняет, является ли символ пробельным.

Точные диапазоны управляющих символов, понятия верхнего и нижнего регистра, титульного символа, пробельных символов, лучше всего посмотреть по документации Java API.

Листинг 4.2 демонстрирует использование этих методов, а на рис. 4.3 показан вывод этой программы.

Листинг 4.2. Методы класса `Character` в программе `CharacterTest`

```
class CharacterTest{
  public static void main(String[] args){
    char ch = '9';
    Character c1 = new Character(ch);
    System.out.println("ch = " + ch);
```

```
    System.out.println("c1.charValue() = " +
           c1.charValue());
    System.out.println("number of 'A' = " +
           Character.digit('A', 16));
    System.out.println("digit for 12 = " +
           Character.forDigit(12, 16));
    System.out.println("c1 = " + c1.toString());
    System.out.println("ch isDefined? " +
           Character.isDefined(ch));
    System.out.println("ch isDigit? " +
           Character.isDigit(ch));
    System.out.println("ch isIdentifierIgnorable? " +
           Character.isIdentifierIgnorable(ch));
    System.out.println("ch isISOControl? " +
           Character.isISOControl(ch));
    System.out.println("ch isJavaIdentifierPart? " +
           Character.isJavaIdentifierPart(ch));
    System.out.println("ch isJavaIdentifierStart? " +
           Character.isJavaIdentifierStart(ch));
    System.out.println("ch isLetter? " +
           Character.isLetter(ch));
    System.out.println("ch isLetterOrDigit? " +
           Character.isLetterOrDigit(ch));
    System.out.println("ch isLowerCase? " +
           Character.isLowerCase(ch));
    System.out.println("ch isSpaceChar? " +
           Character.isSpaceChar(ch));
    System.out.println("ch isTitleCase? " +
           Character.isTitleCase(ch));
    System.out.println("ch isUnicodeIdentifierPart? " +
           Character.isUnicodeIdentifierPart(ch));
    System.out.println("ch isUnicodeIdentifierStart? " +
           Character.isUnicodeIdentifierStart(ch));
    System.out.println("ch isUpperCase? " +
           Character.isUpperCase(ch));
    System.out.println("ch isWhitespace? " +
           Character.isWhitespace(ch));
  }
}
```

В класс `Character` вложены классы `Subset` и `UnicodeBlock`, причем класс `Unicode` и еще один класс, `InputSubset`, являются расширениями класса `Subset`, как это видно на рис. 4.1. Объекты этого класса содержат подмножества Unicode.

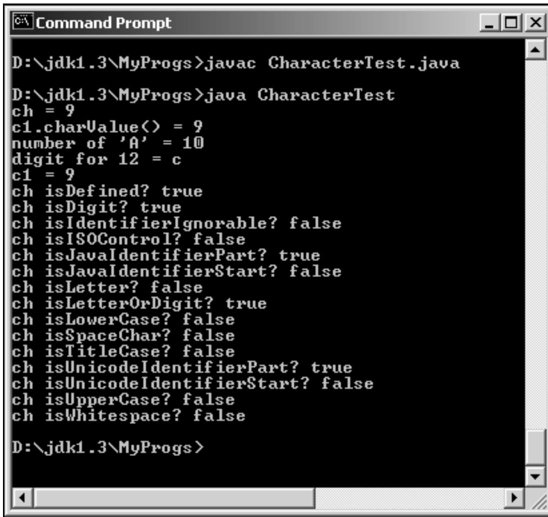


Рис. 4.3. Методы класса `Character` в программе `CharacterTest`

Вместе с классами-оболочками удобно рассмотреть два класса для работы со сколь угодно большими числами.

Класс *BigInteger*

Все примитивные целые типы имеют ограниченный диапазон значений. В целочисленной арифметике Java нет переполнения, целые числа приводятся по модулю, равному диапазону значений.

Для того чтобы было можно производить целочисленные вычисления с любой разрядностью, в состав Java API введен класс `BigInteger`, хранящийся в пакете `java.math`. Этот класс расширяет класс `Number`, следовательно, в нем переопределены методы `doubleValue()`, `floatValue()`, `intValue()`, `longValue()`. Методы `byteValue()` и `shortValue()` не переопределены, а прямо наследуются от класса `Number`.

Действия с объектами класса `BigInteger` не приводят ни к переполнению, ни к приведению по модулю. Если результат операции велик, то число разрядов просто увеличивается. Числа хранятся в двоичной форме с дополнительным кодом.

Перед выполнением операции числа выравниваются по длине распространением знакового разряда.

Шесть конструкторов класса создают объект класса `BigDecimal` из строки символов (знака числа и цифр) или из массива байтов.

Две константы — `ZERO` и `ONE` — моделируют нуль и единицу в операциях с объектами класса `BigInteger`.

Метод `toByteArray()` преобразует объект в массив байтов.

Большинство методов класса `BigInteger` моделируют целочисленные операции и функции, возвращая объект класса `BigInteger`:

- `abs()` — возвращает объект, содержащий абсолютное значение числа, хранящегося в данном объекте `this`;
- `add(x)` — операция `this + x`;
- `and(x)` — операция `this & x`;
- `andNot(x)` — операция `this & (~x)`;
- `divide(x)` — операция `this / x`;
- `divideAndRemainder(x)` — возвращает массив из двух объектов класса `BigInteger`, содержащих частное и остаток от деления `this` на `x`;
- `gcd(x)` — наибольший общий делитель абсолютных значений объекта `this` и аргумента `x`;
- `max(x)` — наибольшее из значений объекта `this` и аргумента `x`;
- `min(x)` — наименьшее из значений объекта `this` и аргумента `x`;
- `mod(x)` — остаток от деления объекта `this` на аргумент метода `x`;
- `modInverse(x)` — остаток от деления числа, обратного объекту `this`, на аргумент `x`;
- `modPow(n, m)` — остаток от деления объекта `this`, возведенного в степень `n`, на `m`;
- `multiply(x)` — операция `this * x`;
- `negate()` — перемена знака числа, хранящегося в объекте;
- `not()` — операция `~this`;
- `or(x)` — операция `this | x`;
- `pow(n)` — операция возведения числа, хранящегося в объекте, в степень `n`;
- `remainder(x)` — операция `this % x`;
- `shiftLeft(n)` — операция `this << n`;
- `shiftRight(n)` — операция `this >> n`;
- `signum()` — функция `sign(x)`;
- `subtract(x)` — операция `this - x`;
- `xor(x)` — операция `this ^ x`.

В листинге 4.3 приведены примеры использования данных методов, а рис. 4.4 показывает результаты выполнения этого листинга.

```
Command Prompt

D:\jdk1.3\MyProgs>javac BigIntegerTest.java

D:\jdk1.3\MyProgs>java BigIntegerTest
bits in a  = 57
bits in b  = 67
a + b = 88988888888888888887
a & b = 72133975954525752
a & ~b = 27866024045474247
a / b = 0
a / b: q = 0, r = 99999999999999999
gcd(a, b) = 1
max(a, b) = 88888888888888888888
min(a, b) = 99999999999999999
a mod b = 99999999999999999
1/a mod b = 87543098654209765319
a^n mod b = 36948590717373350119
a * b = 8888888888888888799911111111111111112
-a = -99999999999999999
~a = -100000000000000000
a | b = 88916754912934363135
a ^ 3 = 999999999999999970000000000000000299999999999999999
a % b = 99999999999999999
a << 3 = 799999999999999992
a >> 3 = 12499999999999999
sign(a) = 1
a - b = -88788888888888888889
a ^ b = 88844620936979837383

D:\jdk1.3\MyProgs>_
```

Рис. 4.4. Методы класса `BigInteger` в программе `BigIntegerTest`

Листинг 4.3. Методы класса `BigInteger` в программе `BigIntegerTest`

```
import java.math.BigInteger;

class BigIntegerTest{
  public static void main(String[] args){
    BigInteger a = new BigInteger("99999999999999999");
    BigInteger b = new BigInteger("88888888888888888888");
    System.out.println("bits in a  = " + a.bitLength());
    System.out.println("bits in b  = " + b.bitLength());
    System.out.println("a + b = " + a.add(b));
    System.out.println("a & b = " + a.and(b));
    System.out.println("a & ~b = " + a.andNot(b));
    System.out.println("a / b = " + a.divide(b));
    BigInteger[] r = a.divideAndRemainder(b);
    System.out.println("a / b: q = " + r[0] + ", r = " + r[1]);
    System.out.println("gcd(a, b) = " + a.gcd(b));
    System.out.println("max(a, b) = " + a.max(b));
    System.out.println("min(a, b) = " + a.min(b));
    System.out.println("a mod b = " + a.mod(b));
    System.out.println("1/a mod b = " + a.modInverse(b));
    System.out.println("a^n mod b = " + a.modPow(a, b));
    System.out.println("a * b = " + a.multiply(b));
    System.out.println("-a = " + a.negate());
    System.out.println("~a = " + a.not());
    System.out.println("a | b = " + a.or(b));
```

```
    System.out.println("a ^ 3 = " + a.pow(3));
    System.out.println("a % b = " + a.remainder(b));
    System.out.println("a << 3 = " + a.shiftLeft(3));
    System.out.println("a >> 3 = " + a.shiftRight(3));
    System.out.println("sign(a) = " + a.signum());
    System.out.println("a — b = " + a.subtract(b));
    System.out.println("a ^ b = " + a.xor(b));
  }
}
```

Обратите внимание на то, что в программу листинга 4.3 надо импортировать пакет `java.math`.

Класс *BigDecimal*

Класс `BigDecimal` расположен в пакете `java.math`.

Каждый объект этого класса хранит два целочисленных значения: мантиссу вещественного числа в виде объекта класса `BigInteger`, и неотрицательный десятичный порядок числа типа `int`.

Например, для числа 76.34862 будет храниться мантисса 7 634 862 в объекте класса `BigInteger`, и порядок 5 как целое число типа `int`.

Таким образом, мантисса может содержать любое количество цифр, а порядок ограничен значением константы `Integer.MAX_VALUE`.

Результат операции над объектами класса `BigDecimal` округляется по одному из восьми правил, определяемых следующими статическими целыми константами:

- `ROUND_CEILING` — округление в сторону большего целого;
- `ROUND_DOWN` — округление к нулю, к меньшему по модулю целому значению;
- `ROUND_FLOOR` — округление к меньшему целому;
- `ROUND_HALF_DOWN` — округление к ближайшему целому, среднее значение округляется к меньшему целому;
- `ROUND_HALF_EVEN` — округление к ближайшему целому, среднее значение округляется к четному числу;
- `ROUND_HALF_UP` — округление к ближайшему целому, среднее значение округляется к большему целому;
- `ROUND_UNNECESSARY` — предполагается, что результат будет целым, и округление не понадобится;
- `ROUND_UP` — округление от нуля, к большему по модулю целому значению.

В классе `BigDecimal` четыре конструктора:

- `BigDecimal(BigInteger bi)` — объект будет хранить большое целое `bi`, порядок равен нулю;
- `BigDecimal(BigInteger mantissa, int scale)` — задается мантисса `mantissa` и неотрицательный порядок `scale` объекта; если порядок `scale` отрицателен, возникает исключительная ситуация;
- `BigDecimal(double d)` — объект будет содержать вещественное число удвоенной точности `d`; если значение `d` бесконечно или `NaN`, то возникает исключительная ситуация;
- `BigDecimal(String val)` — число задается строкой символов `val`, которая должна содержать запись числа по правилам языка Java.

При использовании третьего из перечисленных конструкторов возникает неприятная особенность, отмеченная в документации. Поскольку вещественное число при переводе в двоичную форму представляется, как правило, бесконечной двоичной дробью, то при создании объекта, например, `BigDecimal(0.1)`, мантисса, хранящаяся в объекте, окажется очень большой. Она показана на рис. 4.5. Но при создании такого же объекта четвертым конструктором, `BigDecimal("0.1")`, мантисса будет равна просто `1`.

В классе переопределены методы `doubleValue()`, `floatValue()`, `intValue()`, `longValue()`.

Большинство методов этого класса моделируют операции с вещественными числами. Они возвращают объект класса `BigDecimal`. Здесь буква `x` обозначает объект класса `BigDecimal`, буква `n` — целое значение типа `int`, буква `r` — способ округления, одну из восьми перечисленных выше констант:

- `abs()` — абсолютное значение объекта `this`;
- `add(x)` — операция `this + x`;
- `divide(x, r)` — операция `this / x` с округлением по способу `r`;
- `divide(x, n, r)` — операция `this / x` с изменением порядка и округлением по способу `r`;
- `max(x)` — наибольшее из `this` и `x`;
- `min(x)` — наименьшее из `this` и `x`;
- `movePointLeft(n)` — сдвиг влево на `n` разрядов;
- `movePointRight(n)` — сдвиг вправо на `n` разрядов;
- `multiply(x)` — операция `this * x`;
- `negate()` — возвращает объект с обратным знаком;
- `scale()` — возвращает порядок числа;
- `setScale(n)` — устанавливает новый порядок `n`;

- `setScale(n, r)` — устанавливает новый порядок `n` и округляет число при необходимости по способу `r`;
- `signum()` — знак числа, хранящегося в объекте;
- `subtract(x)` — операция `this - x`;
- `toBigInteger()` — округление числа, хранящегося в объекте;
- `unscaledValue()` — возвращает мантиссу числа.

Листинг 4.4 показывает примеры использования этих методов, а рис. 4.5 — вывод результатов.

```
D:\jdk1.3\MyProgs>javac BigDecimalTest.java

D:\jdk1.3\MyProgs>java BigDecimalTest
|x| = 12345.67890123456789
x + y = -12345.64432227456789
x / y = -357028.63536770822170
x / y = -357028.635368
max(x, y) = 0.03457896
min(x, y) = -12345.67890123456789
x << 3 = -12.34567890123456789
x >> 3 = -12345678.90123456789
x * y = -426.9007368986340736855944
-x = 12345.67890123456789
scale of x = 14
increase scale of x to 20 = -12345.67890123456789000000
decrease scale of x to 10 = -12345.6789012346
sign(x) = -1
x - y = -12345.71348019456789
round x = -12345
mantissa of x = -1234567890123456789
mantissa of 0.1 =
= 1000000000000000055511151231257827021181583404541015625

D:\jdk1.3\MyProgs>_
```

Рис. 4.5. Методы класса `BigDecimal` в программе `BigDecimalTest`

Листинг 4.4. Методы класса `BigDecimal` в программе `BigDecimalTest`

```
import java.math.*;

class BigDecimalTest{
  public static void main(String[] args){
    BigDecimal x = new BigDecimal("-12345.67890123456789");
    BigDecimal y = new BigDecimal("345.7896e-4");
    BigDecimal z = new BigDecimal(new BigInteger("123456789"),8);
    System.out.println("|x| = " + x.abs());
    System.out.println("x + y = " + x.add(y));
    System.out.println("x / y = " + x.divide(y, BigDecimal.ROUND_DOWN));
    System.out.println("x / y = " +
           x.divide(y, 6, BigDecimal.ROUND_HALF_EVEN));
    System.out.println("max(x, y) = " + x.max(y));
    System.out.println("min(x, y) = " + x.min(y));
    System.out.println("x << 3 = " + x.movePointLeft(3));
    System.out.println("x >> 3 = " + x.movePointRight(3));
```

```
    System.out.println("x * y = " + x.multiply(y));
    System.out.println("-x = " + x.negate());
    System.out.println("scale of x = " + x.scale());
    System.out.println("increase scale of x to 20 = " + x.setScale(20));
    System.out.println("decrease scale of x to 10 = " +
           x.setScale(10, BigDecimal.ROUND_HALF_UP));
    System.out.println("sign(x) = " + x.signum());
    System.out.println("x — y = " + x.subtract(y));
    System.out.println("round x = " + x.toBigInteger());
    System.out.println("mantissa of x = " + x.unscaledValue());
    System.out.println("mantissa of 0.1 =\n= " +
            new BigDecimal(0.1).unscaledValue());
  }
}
```

Приведем еще один пример. Напишем простенький калькулятор, выполняющий четыре арифметических действия с числами любой величины. Он работает из командной строки. Программа представлена в листинге 4.5, а примеры использования калькулятора — на рис. 4.6.

Листинг 4.5. Простейший калькулятор

```
import java.math.*;
class Calc{
  public static void main(String[] args){
    if (args.length < 3){
      System.err.println("Usage: java Calc operand operator operand");
      return;
    }
    BigDecimal a = new BigDecimal(args[0]);
    BigDecimal b = new BigDecimal(args[2]);
    switch (args[1].charAt(0)){
      case '+': System.out.println(a.add(b));      break;
      case '-': System.out.println(a.subtract(b)); break;
      case '*': System.out.println(a.multiply(b)); break;
      case '/': System.out.println(a.divide(b,
                     BigDecimal.ROUND_HALF_EVEN)); break;
      default : System.out.println("Invalid operator");
    }
  }
}
```

Почему символ умножения — звездочка — заключен на рис. 4.6 в кавычки? "Юниксоидам" это понятно, а для других дадим краткое пояснение.

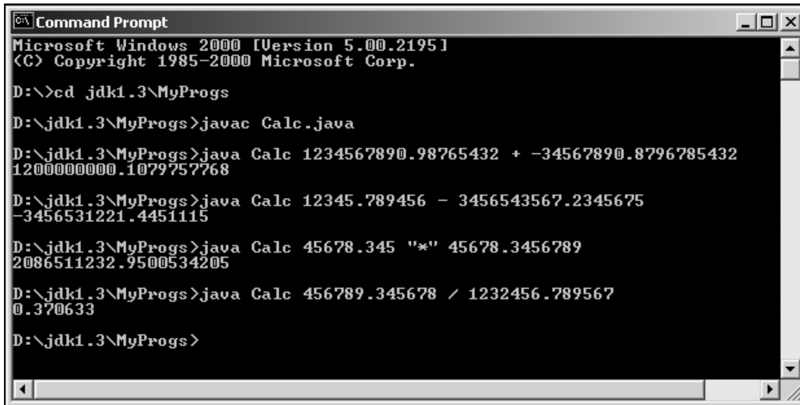


Рис. 4.6. Результаты работы калькулятора

Это особенность операционной системы, а не языка Java. Введенную с клавиатуры строку вначале просматривает командная оболочка (shell) операционной системы, а звездочка для нее — указание подставить на это место все имена файлов из текущего каталога. Оболочка сделает это, и интерпретатор Java получит от нее длинную строку, в которой вместо звездочки стоят имена файлов через пробел.

Звездочка в кавычках понимается командной оболочкой как обычный символ. Командная оболочка снимает кавычки и передает интерпретатору Java то, что надо.

Класс *Class*

Класс `Object`, стоящий во главе иерархии классов Java, представляет все объекты, действующие в системе, является их общей оболочкой. Всякий объект можно считать экземпляром класса `Object`.

Класс с именем `Class` представляет характеристики класса, экземпляром которого является объект. Он хранит информацию о том, не является ли объект на самом деле интерфейсом, массивом или примитивным типом, каков суперкласс объекта, каково имя класса, какие в нем конструкторы, поля, методы и вложенные классы.

В классе `Class` нет конструкторов, экземпляр этого класса создается исполняющей системой Java во время загрузки класса и предоставляется методом `getClass()` класса `Object`, например:

```
String s = "Это строка";
Class c = s.getClass();
```

Статический метод `forName(String class)` возвращает объект класса `Class` для класса, указанного в аргументе, например:

```
Class c1 = Class.forName("java.lang.String");
```

Но этот способ создания объекта класса `Class` считается устаревшим (deprecated). В новых версиях JDK для этой цели используется специальная конструкция — к имени класса через точку добавляется слово `class`:

```
Class c2 = java.lang.String.class;
```

Логические методы `isArray()`, `isInterface()`, `isPrimitive()` позволяют уточнить, не является ли объект массивом, интерфейсом или примитивным типом.

Если объект ссылочного типа, то можно извлечь сведения о вложенных классах, конструкторах, методах и полях методами `getDeclaredClasses()`, `getDeclaredConstructors()`, `getDeclaredMethods()`, `getDeclaredFields()`, в виде массива классов, соответственно, `Class`, `Constructor`, `Method`, `Field`. Последние три класса расположены в пакете `java.lang.reflect` и содержат сведения о конструкторах, полях и методах аналогично тому, как класс `Class` хранит сведения о классах.

Методы `getClasses()`, `getConstructors()`, `getInterfaces()`, `getMethods()`, `getFields()` возвращают такие же массивы, но не всех, а только открытых членов класса.

Метод `getSuperclass()` возвращает суперкласс объекта ссылочного типа, `getPackage()` — пакет, `getModifiers()` — модификаторы класса в битовой форме. Модификаторы можно затем расшифровать методами класса `Modifier` из пакета `java.lang.reflect`.

Листинг 4.6 показывает применение этих методов, а рис. 4.7 — вывод результатов.

Листинг 4.6. Методы класса `Class` в программе `ClassTest`

```
import java.lang.reflect.*;

class ClassTest{
  public static void main(String[] args){
   Class c = null, c1 = null, c2 = null;
   Field[] fld = null;
   String s = "Some string";
   c = s.getClass();
   try{
     c1 = Class.forName("java.lang.String");   // Старый стиль
     c2 = java.lang.String.class;              // Новый стиль
     if (!c1.isPrimitive())
```

```
        fld = c1.getDeclaredFields();           // Все поля класса String
    }catch(Exception e){}
    System.out.println("Class      c: " + c);
    System.out.println("Class     c1: " + c1);
    System.out.println("Class     c2: " + c2);
    System.out.println("Superclass c: " + c.getSuperclass());
    System.out.println("Package    c: " + c.getPackage());
    System.out.println("Modifiers  c: " + c.getModifiers());
    for(int i = 0; i < fld.length; i++)
        System.out.println(fld[i]);
  }
}
```

Методы, возвращающие свойства классов, вызывают исключительные ситуации, требующие обработки. Поэтому в программу введен блок `try{}catch(){}`. Рассмотрение обработки исключительных ситуаций мы откладываем до *главы 16*.

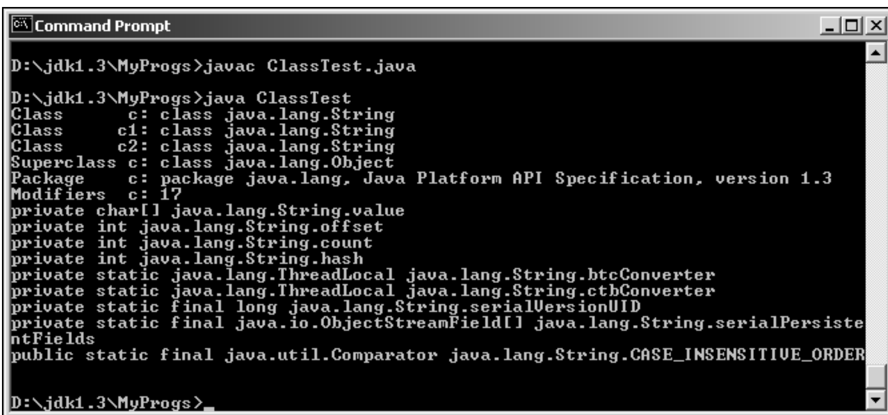


Рис. 4.7. Методы класса `Class` в программе `ClassTest`

ГЛАВА 5

Работа со строками

Очень большое место в обработке информации занимает работа с текстами. Как и многое другое, текстовые строки в языке Java являются объектами. Они представляются экземплярами класса `String` или класса `StringBuffer`.

Поначалу это необычно и кажется слишком громоздким, но, привыкнув, вы оцените удобство работы с классами, а не с массивами символов.

Конечно, возможно занести текст в массив символов типа `char` или даже в массив байтов типа `byte`, но тогда вы не сможете использовать готовые методы работы с текстовыми строками.

Зачем в язык введены два класса для хранения строк? В объектах класса `String` хранятся строки-константы неизменной длины и содержания, так сказать, отлитые в бронзе. Это значительно ускоряет обработку строк и позволяет экономить память, разделяя строку между объектами, использующими ее. Длину строк, хранящихся в объектах класса `StringBuffer`, можно менять, вставляя и добавляя строки и символы, удаляя подстроки или сцепляя несколько строк в одну строку. Во многих случаях, когда надо изменить длину строки типа `String`, компилятор Java неявно преобразует ее к типу `StringBuffer`, меняет длину, потом преобразует обратно в тип `String`. Например, следующее действие

```
String s = "Это" + " одна " + "строка";
```

компилятор выполнит так:

```
String s = new StringBuffer().append("Это").append(" одна ")
      .append("строка").toString();
```

Будет создан объект класса `StringBuffer`, в него последовательно добавлены строки `"Это"`, `" одна "`, `"строка"`, и получившийся объект класса `StringBuffer` будет приведен к типу `String` методом `toString()`.

Напомним, что символы в строках хранятся в кодировке Unicode, в которой каждый символ занимает два байта. Тип каждого символа `char`.

Класс *String*

Перед работой со строкой ее следует создать. Это можно сделать разными способами.

Как создать строку

Самый простой способ создать строку — это организовать ссылку типа `String` на строку-константу:

```
String s1 = "Это строка.";
```

Если константа длинная, можно записать ее в нескольких строках текстового редактора, связывая их операцией сцепления:

```
String s2 = "Это длинная строка, " +
    "записанная в двух строках исходного текста";
```

Замечание

Не забывайте разницу между пустой строкой `String s = ""`, не содержащей ни одного символа, и пустой ссылкой `String s = null`, не указывающей ни на какую строку и не являющейся объектом.

Самый правильный способ создать объект с точки зрения ООП — это вызвать его конструктор в операции `new`. Класс `String` предоставляет вам девять конструкторов:

- `String()` — создается объект с пустой строкой;
- `String(String str)` — из одного объекта создается другой, поэтому этот конструктор используется редко;
- `String(StringBuffer str)` — преобразованная копия объекта класса `BufferString`;
- `String(byte[] byteArray)` — объект создается из массива байтов `byteArray`;
- `String(char[] charArray)` — объект создается из массива `charArray` символов Unicode;
- `String(byte[] byteArray, int offset, int count)` — объект создается из части массива байтов `byteArray`, начинающейся с индекса `offset` и содержащей `count` байтов;
- `String(char[] charArray, int offset, int count)` — то же, но массив состоит из символов Unicode;

- `String(byte[] byteArray, String encoding)` — символы, записанные в массиве байтов, задаются в Unicode-строке, с учетом кодировки `encoding`;
- `String(byte[] byteArray, int offset, int count, String encoding)` — то же самое, но только для части массива.

При неправильном задании индексов `offset`, `count` или кодировки `encoding` возникает исключительная ситуация.

Конструкторы, использующие массив байтов `byteArray`, предназначены для создания Unicode-строки из массива байтовых ASCII-кодировок символов. Такая ситуация возникает при чтении ASCII-файлов, извлечении информации из базы данных или при передаче информации по сети.

В самом простом случае компилятор для получения двухбайтовых символов Unicode добавит к каждому байту старший нулевой байт. Получится диапазон `'\u0000'`—`'\u00FF'` кодировки Unicode, соответствующий кодам Latin1. Тексты на кириллице будут выведены неправильно.

Если же на компьютере сделаны местные установки, как говорят на жаргоне "установлена локаль" (locale) (в MS Windows это выполняется утилитой Regional Options в окне **Control Panel**), то компилятор, прочитав эти установки, создаст символы Unicode, соответствующие местной кодовой странице. В русифицированном варианте MS Windows это обычно кодовая страница CP1251.

Если исходный массив с кириллическим ASCII-текстом был в кодировке CP1251, то строка Java будет создана правильно. Кириллица попадет в свой диапазон `'\u0400'`—`'\u04FF'` кодировки Unicode.

Но у кириллицы есть еще, по меньшей мере, четыре кодировки.

- В MS-DOS применяется кодировка CP866.
- В UNIX обычно применяется кодировка KOI8-R.
- На компьютерах Apple Macintosh используется кодировка MacCyrillic.
- Есть еще и международная кодировка кириллицы ISO8859-5.

Например, байт `11100011` (`0xE3` в шестнадцатеричной форме) в кодировке CP1251 представляет кириллическую букву `г`, в кодировке CP866 — букву `у`, в кодировке KOI8-R — букву `ц`, в ISO8859-5 — букву `у`, в MacCyrillic — букву `г`.

Если исходный кириллический ASCII-текст был в одной из этих кодировок, а местная кодировка CP1251, то Unicode-символы строки Java не будут соответствовать кириллице.

В этих случаях используются последние два конструктора, в которых параметром `encoding` указывается, какую кодовую таблицу использовать конструктору при создании строки.

Листинг 5.1 показывает различные случаи записи кириллического текста. В нем создаются три массива байтов, содержащих слово "Россия" в трех кодировках.

- Массив `byteCP1251` содержит слово "Россия" в кодировке CP1251.
- Массив `byteCP866` содержит слово "Россия" в кодировке CP866.
- Массив `byteKOI8R` содержит слово "Россия" в кодировке KOI8-R.

Из каждого массива создаются по три строки с использованием трех кодовых таблиц.

Кроме того, из массива символов `c[]` создается строка `s1`, из массива байтов, записанного в кодировке CP866, создается строка `s2`. Наконец, создается ссылка `s3` на строку-константу.

Листинг 5.1. Создание кириллических строк

```
class StringTest{
  public static void main(String[] args){
    String winLikeWin  = null, winLikeDOS  = null, winLikeUNIX  = null;
    String dosLikeWin  = null, dosLikeDOS  = null, dosLikeUNIX  = null;
    String unixLikeWin = null, unixLikeDOS = null, unixLikeUNIX = null;
    String msg = null;

    byte[] byteCp1251 = {
      (byte)0xD0, (byte)0xEE, (byte)0xF1,
      (byte)0xF1, (byte)0xE8, (byte)0xFF
    };
    byte[] byteCp866 = {
      (byte)0x90, (byte)0xAE, (byte)0xE1,
      (byte)0xE1, (byte)0xA8, (byte)0xEF
    };
    byte[] byteKOI8R = {
      (byte)0xF2, (byte)0xCF, (byte)0xD3,
      (byte)0xD3, (byte)0xC9, (byte)0xD1
    };
    char[] c = {'Р', 'о', 'с', 'с', 'и', 'я'};
    String s1 = new String(c);
    String s2 = new String(byteCp866);         // Для консоли MS Windows
    String s3 = "Россия";
    System.out.println();

    try{
          // Сообщение в Cp866 для вывода на консоль MS Windows.
      msg = new String("\"Россия\" в ".getBytes("Cp866"), "Cp1251");

      winLikeWin   = new String(byteCp1251, "Cp1251");     // Правильно
      winLikeDOS   = new String(byteCp1251, "Cp866");
      winLikeUNIX  = new String(byteCp1251, "KOI8-R");
      dosLikeWin   = new String(byteCp866, "Cp1251");      // Для консоли
```

```
      dosLikeDOS   = new String(byteCp866, "Cp866");       // Правильно
      dosLikeUNIX  = new String(byteCp866, "KOI8-R");
      unixLikeWin  = new String(byteKOI8R, "Cp1251");
      unixLikeDOS  = new String(byteKOI8R, "Cp866");
      unixLikeUNIX = new String(byteKOI8R, "KOI8-R");      // Правильно

      System.out.print(msg + "Cp1251: ");
      System.out.write(byteCp1251);
      System.out.println();
      System.out.print(msg + "Cp866 : ");
      System.out.write(byteCp866);
      System.out.println();
      System.out.print(msg + "KOI8-R: ");
      System.out.write(byteKOI8R);
    }catch(Exception e){
      e.printStackTrace();
    }
    System.out.println();
    System.out.println();
    System.out.println(msg + "char array      : " + s1);
    System.out.println(msg + "default encoding: " + s2);
    System.out.println(msg + "string constant : " + s3);
    System.out.println();
    System.out.println(msg + "Cp1251 -> Cp1251: " + winLikeWin);
    System.out.println(msg + "Cp1251 -> Cp866 : " + winLikeDOS);
    System.out.println(msg + "Cp1251 -> KOI8-R: " + winLikeUNIX);
    System.out.println(msg + "Cp866  -> Cp1251: " + dosLikeWin);
    System.out.println(msg + "Cp866  -> Cp866 : " + dosLikeDOS);
    System.out.println(msg + "Cp866  -> KOI8-R: " + dosLikeUNIX);
    System.out.println(msg + "KOI8-R -> Cp1251: " + unixLikeWin);
    System.out.println(msg + "KOI8-R -> Cp866 : " + unixLikeDOS);
    System.out.println(msg + "KOI8-R -> KOI8-R: " + unixLikeUNIX);
  }
}
```

Все эти данные выводятся на консоль MS Windows 2000, как показано на рис. 5.1.

В первые три строки консоли выводятся массивы байтов `byteCP1251`, `byteCP866` и `byteKOI8R` без преобразования в Unicode. Это выполняется методом `write()` класса `FilterOutputStream` из пакета `java.io`.

В следующие три строки консоли выведены строки Java, полученные из массива символов `c[]`, массива `byteCP866` и строки-константы.

Следующие строки консоли содержат преобразованные массивы.

Вы видите, что на консоль правильно выводится только массив в кодировке CP866, записанный в строку с использованием кодовой таблицы CP1251.

В чем дело? Здесь свой вклад в проблему русификации вносит вывод потока символов на консоль или в файл.

```
Command Prompt

D:\jdk1.3\MyProgs>javac StringTest.java

D:\jdk1.3\MyProgs>java StringTest

"Россия" в Cp1251: ╚юёё╕ш
"Россия" в Cp866 : Россия
"Россия" в KOI8-R: ЄтЁЁшЁ

"Россия" в char array      : ╚юёёш
"Россия" в default encoding: Россия
"Россия" в string constant : ╚юёёш

"Россия" в Cp1251 -> Cp1251: ╚юёёш
"Россия" в Cp1251 -> Cp866 : ?▌╕╕°а
"Россия" в Cp1251 -> KOI8-R: я═▀▀Fг
"Россия" в Cp866  -> Cp1251: Россия
"Россия" в Cp866  -> Cp866 : ╚юёёш
"Россия" в Cp866  -> KOI8-R: ??└└?╬
"Россия" в KOI8-R -> Cp1251: ЄтЁЁшЁ
"Россия" в KOI8-R -> Cp866 : к?????
"Россия" в KOI8-R -> KOI8-R: ╚юёёш

D:\jdk1.3\MyProgs>java StringTest>codes.txt

D:\jdk1.3\MyProgs>
```

Рис. 5.1. Вывод кириллической строки на консоль MS Windows 2000

Как уже упоминалось в *главе 1*, в консольное окно **Command Prompt** операционной системы MS Windows текст выводится в кодировке CP866.

Для того чтобы учесть это, слова "\"Россия\" в" преобразованы в массив байтов, содержащий символы в кодировке CP866, а затем переведены в строку `msg`.

В предпоследней строке рис. 5.1 сделано перенаправление вывода программы в файл codes.txt. В MS Windows 2000 вывод текста в файл происходит в кодировке CP1251. На рис. 5.2 показано содержимое файла codes.txt в окне программы Notepad.

```
codes.txt - Notepad
File Edit Format Help

"ђ®ббЁп" ў Cp1251: Россия
"ђ®ббЁп" ў Cp866 : ђ®ббЁп
"ђ®ббЁп" ў KOI8-R: тПУУЙС

"ђ®ббЁп" ў char array      : Россия
"ђ®ббЁп" ў default encoding: ђ®ббЁп
"ђ®ббЁп" ў string constant : Россия

"ђ®ббЁп" ў Cp1251 -> Cp1251: Россия
"ђ®ббЁп" ў Cp1251 -> Cp866 : ?юёёш
"ђ®ббЁп" ў Cp1251 -> KOI8-R: пНЯЯХЪ
"ђ®ббЁп" ў Cp866  -> Cp1251: ђ®ббЁп
"ђ®ббЁп" ў Cp866  -> Cp866 : Россия
"ђ®ббЁп" ў Cp866  -> KOI8-R: ??AA?O
"ђ®ббЁп" ў KOI8-R -> Cp1251: тПУУЙС
"ђ®ббЁп" ў KOI8-R -> Cp866 : Є?????
"ђ®ббЁп" ў KOI8-R -> KOI8-R: Россия
```

Рис. 5.2. Вывод кириллической строки в файл

Как видите, кириллица выглядит совсем по-другому. Правильные символы Unicode кириллицы получаются, если использовать ту же кодовую таблицу, в которой записан исходный массив байтов.

Вопросы русификации мы еще будем обсуждать в *главах 9* и *18*, а пока заметьте, что при создании строки из массива байтов лучше указывать ту же самую кириллическую кодировку, в которой записан массив. Тогда вы получите строку Java с правильными символами Unicode.

При выводе же строки на консоль, в окно, в файл или при передаче по сети лучше преобразовать строку Java с символами Unicode по правилам вывода в нужное место.

Еще один способ создать строку — это использовать два статических метода `copyValueOf(char[] charArray)` и `copyValueOf(char[] charArray, int offset, int length)`.

Они создают строку по заданному массиву символов и возвращают ее в качестве результата своей работы. Например, после выполнения следующего фрагмента программы

```
char[] c = {'С', 'и', 'м', 'в', 'о', 'л', 'ь', 'н', 'ы', 'й'};
String s1 = String.copyValueOf(c);
String s2 = String.copyValueOf(c, 3, 7);
```

получим в объекте `s1` строку `"Символьный"`, а в объекте `s2` — строку `"вольный"`.

Сцепление строк

Со строками можно производить операцию *сцепления строк* (concatenation), обозначаемую знаком плюс `+`. Эта операция создает новую строку, просто составленную из состыкованных первой и второй строк, как показано в начале данной главы. Ее можно применять и к константам, и к переменным. Например:

```
String attention = "Внимание: ";
String s = attention + "неизвестный символ";
```

Вторая операция — присваивание `+=` — применяется к переменным в левой части:

```
attention += s;
```

Поскольку операция `+` перегружена со сложения чисел на сцепление строк, встает вопрос о приоритете этих операций. У сцепления строк приоритет выше, чем у сложения, поэтому, записав `"2" + 2 + 2`, получим строку `"222"`. Но, записав `2 + 2 + "2"`, получим строку `"42"`, поскольку действия выполняются слева направо. Если же запишем `"2" + (2 + 2)`, то получим `"24"`.

Манипуляции строками

В классе `String` есть множество методов для работы со строками. Посмотрим, что они позволяют делать.

Как узнать длину строки

Для того чтобы узнать длину строки, т. е. количество символов в ней, надо обратиться к методу `length()`:

```
String s = "Write once, run anywhere.";
int len = s.length();
```

или еще проще

```
int len = "Write once, run anywhere.".length();
```

поскольку строка-константа — полноценный объект класса `String`.

Заметьте, что строка — это не массив, у нее нет поля `length`.

Внимательный читатель, изучивший рис. 4.7, готов со мной не согласиться. Ну, что же, действительно, символы хранятся в массиве, но он закрыт, как и все поля класса `String`.

Как выбрать символы из строки

Выбрать символ с индексом `ind` (индекс первого символа равен нулю) можно методом `charAt(int ind)`. Если индекс `ind` отрицателен или не меньше чем длина строки, возникает исключительная ситуация. Например, после определения

```
char ch = s.charAt(3);
```

переменная `ch` будет иметь значение `'t'`.

Все символы строки в виде массива символов можно получить методом `toCharArray()`, возвращающим массив символов.

Если же надо включить в массив символов `dst`, начиная с индекса `ind` массива подстроку от индекса `begin` включительно до индекса `end` исключительно, то используйте метод `getChars(int begin, int end, char[] dst, int ind)` типа `void`.

В массив будет записано `end — begin` символов, которые займут элементы массива, начиная с индекса `ind` до индекса `ind + (end — begin) — 1`.

Этот метод создает исключительную ситуацию в следующих случаях:

- ссылка `dst == null`;
- индекс `begin` отрицателен;
- индекс `begin` больше индекса `end`;

- индекс `end` больше длины строки;
- индекс `ind` отрицателен;
- `ind + (end — begin) > dst.length`.

Например, после выполнения

```
char[] ch = {'К', 'о', 'н', 'е', 'ц', ' ', 'с', 'в', 'е', 'т', 'а'};
"Пароль легко найти".getChars(2, 8, ch, 2);
```

результат будет таков:

```
ch = {'К', 'о', 'р', 'о', 'л', 'ь', ' ', 'л', 'е', 'т', 'а'};
```

Если надо получить массив байтов, содержащий все символы строки в байтовой кодировке ASCII, то используйте метод `getBytes()`.

Этот метод при переводе символов из Unicode в ASCII использует локальную кодовую таблицу.

Если же надо получить массив байтов не в локальной кодировке, а в какой-то другой, используйте метод `getBytes(String encoding)`.

Так сделано в листинге 5.1 при создании объекта `msg`. Строка "\"Россия в\"" перекодировалась в массив CP866-байтов для правильного вывода кириллицы в консольное окно **Command Prompt** операционной системы Windows 2000.

Как выбрать подстроку

Метод `substring(int begin, int end)` выделяет подстроку от символа с индексом `begin` включительно до символа с индексом `end` исключительно. Длина подстроки будет равна `end — begin`.

Метод `substring(int begin)` выделяет подстроку от индекса `begin` включительно до конца строки.

Если индексы отрицательны, индекс `end` больше длины строки или `begin` больше чем `end`, то возникает исключительная ситуация.

Например, после выполнения

```
String s = "Write once, run anywhere.";
String sub1 = s.substring(6, 10);
String sub2 = s.substring(16);
```

получим в строке `sub1` значение `"once"`, а в `sub2` — значение `"anywhere"`.

Как сравнить строки

Операция сравнения == сопоставляет только ссылки на строки. Она выясняет, указывают ли ссылки на одну и ту же строку. Например, для строк

```
String s1 = "Какая-то строка";
String s2 = "Другая строка";
```

сравнение `s1 == s2` дает в результате `false`.

Значение `true` получится, только если обе ссылки указывают на одну и ту же строку, например, после присваивания `s1 = s2`.

Интересно, что если мы определим `s2` так:

```
String s2 = "Какая-то строка";
```

то сравнение `s1 == s2` даст в результате `true`, потому что компилятор создаст только один экземпляр константы `"Какая-то строка"` и направит на него все ссылки.

Вы, разумеется, хотите сравнивать не ссылки, а содержимое строк. Для этого есть несколько методов.

Логический метод `equals(Object obj)`, переопределенный из класса `Object`, возвращает `true`, если аргумент `obj` не равен `null`, является объектом класса `String`, и строка, содержащаяся в нем, полностью идентична данной строке вплоть до совпадения регистра букв. В остальных случаях возвращается значение `false`.

Логический метод `equalsIgnoreCase(Object obj)` работает так же, но одинаковые буквы, записанные в разных регистрах, считаются совпадающими.

Например, `s2.equals("другая строка")` даст в результате `false`, а `s2.equalsIgnoreCase("другая строка")` возвратит `true`.

Метод `compareTo(String str)` возвращает целое число типа `int`, вычисленное по следующим правилам:

1. Сравниваются символы данной строки `this` и строки `str` с одинаковым индексом, пока не встретятся различные символы с индексом, допустим `k`, или пока одна из строк не закончится.
2. В первом случае возвращается значение `this.charAt(k) — str.charAt(k)`, т. е. разность кодировок Unicode первых несовпадающих символов.
3. Во втором случае возвращается значение `this.length() — str.length()`, т. е. разность длин строк.
4. Если строки совпадают, возвращается `0`.

Если значение `str` равно `null`, возникает исключительная ситуация.

Нуль возвращается в той же ситуации, в которой метод `equals()` возвращает `true`.

Метод `compareToIgnoreCase(String str)` производит сравнение без учета регистра букв, точнее говоря, выполняется метод

```
this.toUpperCase().toLowerCase().compareTo(
       str.toUpperCase().toLowerCase());
```

Еще один метод — `compareTo(Object obj)` создает исключительную ситуацию, если `obj` не является строкой. В остальном он работает как метод `compareTo(String str)`.

Эти методы не учитывают алфавитное расположение символов в локальной кодировке.

Русские буквы расположены в Unicode по алфавиту, за исключением одной буквы. Заглавная буква `Ё` расположена перед всеми кириллическими буквами, ее код `'\u0401'`, а строчная буква `ё` — после всех русских букв, ее код `'\u0451'`.

Если вас такое расположение не устраивает, задайте свое размещение букв с помощью класса `RuleBasedCollator` из пакета `java.text`.

Сравнить подстроку данной строки `this` с подстрокой той же длины `len` другой строки `str` можно логическим методом

```
regionMatches(int ind1, String str, int ind2, int len)
```

Здесь `ind1` — индекс начала подстроки данной строки `this`, `ind2` — индекс начала подстроки другой строки `str`. Результат `false` получается в следующих случаях:

- хотя бы один из индексов `ind1` или `ind2` отрицателен;
- хотя бы одно из `ind1 + len` или `ind2 + len` больше длины соответствующей строки;
- хотя бы одна пара символов не совпадает.

Этот метод различает символы, записанные в разных регистрах. Если надо сравнивать подстроки без учета регистров букв, то используйте логический метод:

```
regionMatches(boolean flag, int ind1, String str, int ind2, int len)
```

Если первый параметр `flag` равен `true`, то регистр букв при сравнении подстрок не учитывается, если `false` — учитывается.

Как найти символ в строке

Поиск всегда ведется с учетом регистра букв.

Первое появление символа `ch` в данной строке `this` можно отследить методом `indexOf(int ch)`, возвращающим индекс этого символа в строке или `-1`, если символа `ch` в строке `this` нет.

Например, `"Молоко".indexOf('о')` выдаст в результате `1`.

Конечно, этот метод выполняет в цикле последовательные сравнения `this.charAt(k++) == ch`, пока не получит значение `true`.

Второе и следующие появления символа `ch` в данной строке `this` можно отследить методом `indexOf(int ch, int ind)`.

Этот метод начинает поиск символа `ch` с индекса `ind`. Если `ind < 0`, то поиск идет с начала строки, если `ind` больше длины строки, то символ не ищется, т. е. возвращается `-1`.

Например, `"Молоко".indexOf('о', indexOf('о') + 1)` даст в результате `3`.

Последнее появление символа `ch` в данной строке `this` отслеживает метод `lastIndexOf(int ch)`. Он просматривает строку в обратном порядке. Если символ `ch` не найден, возвращается `-1`.

Например, `"Молоко".lastIndexOf('о')` даст в результате `5`.

Предпоследнее и предыдущие появления символа `ch` в данной строке `this` можно отследить методом `lastIndexOf(int ch, int ind)`, который просматривает строку в обратном порядке, начиная с индекса `ind`.

Если `ind` больше длины строки, то поиск идет от конца строки, если `ind < 0`, то возвращается `-1`.

Как найти подстроку

Поиск всегда ведется с учетом регистра букв.

Первое вхождение подстроки `sub` в данную строку `this` отыскивает метод `indexOf(String sub)`. Он возвращает индекс первого символа первого вхождения подстроки `sub` в строку или `-1`, если подстрока `sub` не входит в строку `this`. Например, `"Раскраска".indexOf("рас")` даст в результате `4`.

Если вы хотите начать поиск не с начала строки, а с какого-то индекса `ind`, используйте метод `indexOf(String sub, int ind)`. Если `ind < 0`, то поиск идет с начала строки, если `ind` больше длины строки, то символ не ищется, т. е. возвращается `-1`.

Последнее вхождение подстроки `sub` в данную строку `this` можно отыскать методом `lastIndexOf(String sub)`, возвращающим индекс первого символа последнего вхождения подстроки `sub` в строку `this` или `(-1)`, если подстрока `sub` не входит в строку `this`.

Последнее вхождение подстроки `sub` не во всю строку `this`, а только в ее начало до индекса `ind` можно отыскать методом `lastIndexOf(String str, int ind)`. Если `ind` больше длины строки, то поиск идет от конца строки, если `ind < 0`, то возвращается `-1`.

Для того чтобы проверить, не начинается ли данная строка `this` с подстроки `sub`, используйте логический метод `startsWith(String sub)`, возвращающий `true`, если данная строка `this` начинается с подстроки `sub`, или совпадает с ней, или подстрока `sub` пуста.

Можно проверить и появление подстроки `sub` в данной строке `this`, начиная с некоторого индекса `ind` логическим методом `startsWith(String sub, int ind)`. Если индекс `ind` отрицателен или больше длины строки, возвращается `false`.

Для того чтобы проверить, не заканчивается ли данная строка this подстрокой sub, используйте логический метод endsWith(String sub). Учтите, что он возвращает true, если подстрока sub совпадает со всей строкой или подстрока sub пуста.

Например, if (fileName.endsWith(".java")) отследит имена файлов с исходными текстами Java.

Перечисленные выше методы создают исключительную ситуацию, если sub == null.

Если вы хотите осуществить поиск, не учитывающий регистр букв, измените предварительно регистр всех символов строки.

Как изменить регистр букв

Метод toLowerCase() возвращает новую строку, в которой все буквы переведены в нижний регистр, т. е. сделаны строчными.

Метод toUpperCase() возвращает новую строку, в которой все буквы переведены в верхний регистр, т. е. сделаны прописными.

При этом используется локальная кодовая таблица по умолчанию. Если нужна другая локаль, то применяются методы toLowerCase(Locale loc) и toUpperCase(Locale loc).

Как заменить отдельный символ

Метод replace(int old, int new) возвращает новую строку, в которой все вхождения символа old заменены символом new. Если символа old в строке нет, то возвращается ссылка на исходную строку.

Например, после выполнения "Рука в руку сует хлеб".replace('у', 'е') получим строку "Река в реке сеет хлеб".

Регистр букв при замене учитывается.

Как убрать пробелы в начале и конце строки

Метод trim() возвращает новую строку, в которой удалены начальные и конечные символы с кодами, не превышающими '\u0020'.

Как преобразовать данные другого типа в строку

В языке Java принято соглашение — каждый класс отвечает за преобразование других типов в тип этого класса и должен содержать нужные для этого методы.

Класс `String` содержит восемь статических методов `valueOf(type elem)` преобразования в строку примитивных типов `boolean`, `char`, `int`, `long`, `float`, `double`, массива `char[]`, и просто объекта типа `Object`.

Девятый метод `valueOf(char[] ch, int offset, int len)` преобразует в строку подмассив массива `ch`, начинающийся с индекса `offset` и имеющий `len` элементов.

Кроме того, в каждом классе есть метод `toString()`, переопределенный или просто унаследованный от класса `Object`. Он преобразует объекты класса в строку. Фактически, метод `valueOf()` вызывает метод `toString()` соответствующего класса. Поэтому результат преобразования зависит от того, как реализован метод `toString()`.

Еще один простой способ — сцепить значение `elem` какого-либо типа с пустой строкой: `"" + elem`. При этом неявно вызывается метод `elem.toString()`.

Класс *StringBuffer*

Объекты класса `StringBuffer` — это строки переменной длины. Только что созданный объект имеет буфер определенной ***емкости*** (capacity), по умолчанию достаточной для хранения 16 символов. Емкость можно задать в конструкторе объекта.

Как только буфер начинает переполняться, его емкость автоматически увеличивается, чтобы вместить новые символы.

В любое время емкость буфера можно увеличить, обратившись к методу

```
ensureCapacity(int minCapacity)
```

Этот метод изменит емкость, только если `minCapacity` будет больше длины хранящейся в объекте строки. Емкость будет увеличена по следующему правилу. Пусть емкость буфера равна N. Тогда новая емкость будет равна

```
Max(2 * N + 2, minCapacity)
```

Таким образом, емкость буфера нельзя увеличить менее чем вдвое.

Методом `setLength(int newLength)` можно установить любую длину строки. Если она окажется больше текущей длины, то дополнительные символы будут равны `'\u0000'`. Если она будет меньше текущей длины, то строка будет обрезана, последние символы потеряются, точнее, будут заменены символом `'\u0000'`. Емкость при этом не изменится.

Если число `newLength` окажется отрицательным, возникнет исключительная ситуация.

Совет

Будьте осторожны, устанавливая новую длину объекта.

Количество символов в строке можно узнать, как и для объекта класса `String`, методом `length()`, а емкость — методом `capacity()`.

Создать объект класса `StringBuffer` можно только конструкторами.

Конструкторы

В классе `StringBuffer` три конструктора:

- `StringBuffer()` — создает пустой объект с емкостью 16 символов;
- `StringBuffer(int capacity)` — создает пустой объект заданной емкости `capacity`;
- `StringBuffer(String str)` — создает объект емкостью `str.length() + 16`, содержащий строку `str`.

Как добавить подстроку

В классе `StringBuffer` есть десять методов `append()`, добавляющих подстроку в конец строки. Они не создают новый экземпляр строки, а возвращают ссылку на ту же самую, но измененную строку.

Основной метод `append(String str)` присоединяет строку `str` в конец данной строки. Если ссылка `str == null`, то добавляется строка `"null"`.

Шесть методов `append(type elem)` добавляют примитивные типы `boolean`, `char`, `int`, `long`, `float`, `double`, преобразованные в строку.

Два метода присоединяют к строке массив `str` и подмассив `sub` символов, преобразованные в строку: `append(char[] str)` и `append(char[] sub, int offset, int len)`.

Десятый метод добавляет просто объект `append(Object obj)`. Перед этим объект `obj` преобразуется в строку своим методом `toString()`.

Как вставить подстроку

Десять методов `insert()` предназначены для вставки строки, указанной параметром метода, в данную строку. Место вставки задается первым параметром метода `ind`. Это индекс элемента строки, перед которым будет сделана вставка. Он должен быть неотрицательным и меньше длины строки, иначе возникнет исключительная ситуация. Строка раздвигается, емкость буфера при необходимости увеличивается. Методы возвращают ссылку на ту же преобразованную строку.

Основной метод `insert(int ind, String str)` вставляет строку `str` в данную строку перед ее символом с индексом `ind`. Если ссылка `str == null`, вставляется строка `"null"`.

Например, после выполнения

```
String s = new StringBuffer("Это большая строка").
     insert(4, "не").toString();
```

получим `s == "Это небольшая строка"`.

Метод `sb.insert(sb.length(), "xxx")` будет работать так же, как метод `sb.append("xxx")`.

Шесть методов `insert(int ind, type elem)` вставляют примитивные типы `boolean`, `char`, `int`, `long`, `float`, `double`, преобразованные в строку.

Два метода вставляют массив `str` и подмассив `sub` символов, преобразованные в строку:

```
insert(int ind, char[] str)
insert(int ind, char[] sub, int offset, int len)
```

Десятый метод вставляет просто объект:

```
insert(int ind, Object obj)
```

Объект `obj` перед добавлением преобразуется в строку своим методом `toString()`.

Как удалить подстроку

Метод `delete(int begin, int end)` удаляет из строки символы, начиная с индекса `begin` включительно до индекса `end` исключительно, если `end` больше длины строки, то до конца строки.

Например, после выполнения

```
String s = new StringBuffer("Это небольшая строка").
     delete(4, 6).toString();
```

получим `s == "Это большая строка"`.

Если `begin` отрицательно, больше длины строки или больше `end`, возникает исключительная ситуация.

Если `begin == end`, удаление не происходит.

Как удалить символ

Метод `deleteCharAt(int ind)` удаляет символ с указанным индексом `ind`. Длина строки уменьшается на единицу.

Если индекс `ind` отрицателен или больше длины строки, возникает исключительная ситуация.

Как заменить подстроку

Метод `replace(int begin, int end, String str)` удаляет символы из строки, начиная с индекса `begin` включительно до индекса `end` исключительно, если `end` больше длины строки, то до конца строки, и вставляет вместо них строку `str`.

Если `begin` отрицательно, больше длины строки или больше `end`, возникает исключительная ситуация.

Разумеется, метод `replace()` — это последовательное выполнение методов `delete()` и `insert()`.

Как перевернуть строку

Метод `reverse()` меняет порядок расположения символов в строке на обратный порядок.

Например, после выполнения

```
String s = new StringBuffer("Это небольшая строка").
     reverse().toString();
```

получим `s == "акортс яашьлобен отЭ"`.

Синтаксический разбор строки

Задача разбора введенного текста — *парсинг* (parsing) — вечная задача программирования, наряду с сортировкой и поиском. Написана масса программ-парсеров (parser), разбирающих текст по различным признакам. Есть даже программы, генерирующие парсеры по заданным правилам разбора: YACC, LEX и др.

Но задача остается. И вот очередной программист, отчаявшись найти что-нибудь подходящее, берется за разработку собственной программы разбора.

В пакет `java.util` входит простой класс `StringTokenizer`, облегчающий разбор строк.

Класс *StringTokenizer*

Класс `StringTokenizer` из пакета `java.util` небольшой, в нем три конструктора и шесть методов.

Первый конструктор `StringTokenizer(String str)` создает объект, готовый разбить строку `str` на слова, разделенные пробелами, символами табуляции `'\t'`, перевода строки `'\n'` и возврата каретки `'\r'`. Разделители не включаются в число слов.

Второй конструктор `StringTokenizer(String str, String delimeters)` задает разделители вторым параметром `delimeters`, например:

```
StringTokenizer("Казнить,нельзя:пробелов-нет", " \t\n\r,:-");
```

Здесь первый разделитель — пробел. Потом идут символ табуляции, символ перевода строки, символ возврата каретки, запятая, двоеточие, дефис. Порядок расположения разделителей в строке `delimeters` не имеет значения. Разделители не включаются в число слов.

Третий конструктор позволяет включить разделители в число слов:

```
StringTokenizer(String str, String delimeters, boolean flag);
```

Если параметр `flag` равен `true`, то разделители включаются в число слов, если `false` — нет. Например:

```
StringTokenizer("a — (b + c) / b * c", " \t\n\r+*-/(), true);
```

В разборе строки на слова активно участвуют два метода:

- метод `nextToken()` возвращает в виде строки следующее слово;
- логический метод `hasMoreTokens()` возвращает `true`, если в строке еще есть слова, и `false`, если слов больше нет.

Третий метод `countTokens()` возвращает число оставшихся слов.

Четвертый метод `nextToken(String newDelimeters)` позволяет "на ходу" менять разделители. Следующее слово будет выделено по новым разделителям `newDelimeters`; новые разделители действуют далее вместо старых разделителей, определенных в конструкторе или предыдущем методе `nextToken()`.

Оставшиеся два метода `nextElement()` и `hasMoreElements()` реализуют интерфейс `Enumeration`. Они просто обращаются к методам `nextToken()` и `hasMoreTokens()`.

Схема очень проста (листинг 5.2).

Листинг 5.2. Разбиение строки на слова

```
String s = "Строка, которую мы хотим разобрать на слова";
StringTokenizer st = new StringTokenizer(s, " \t\n\r,.");
while(st.hasMoreTokens()){
   // Получаем слово и что-нибудь делаем с ним, например,
   // просто выводим на экран
   System.out.println(st.nextToken());
}
```

Полученные слова обычно заносятся в какой-нибудь класс-коллекцию: `Vector`, `Stack` или другой, наиболее подходящий для дальнейшей обработки текста контейнер. Классы-коллекции мы рассмотрим в следующей главе.

Заключение

Все методы представленных в этой главе классов написаны на языке Java. Их исходные тексты можно посмотреть, они входят в состав JDK. Это очень полезное занятие. Просмотрев исходный текст, вы получаете полное представление о том, как работает метод.

В последних версиях JDK исходные тексты хранятся в упакованном архиватором `jar` файле src.jar, лежащем в корневом каталоге JDK, например, в каталоге D:\jdk1.3.

Чтобы распаковать их, перейдите в каталог jdk1.3:

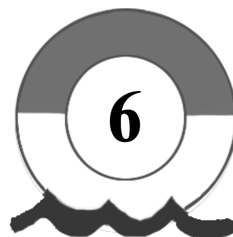
```
D: > cd jdk1.3
```

и вызовите архиватор `jar` следующим образом:

```
D:\jdk1.3 > jar -xf src.jar
```

В каталоге jdk1.3 появится подкаталог src, а в нем подкаталоги, соответствующие пакетам и подпакетам JDK, с исходными файлами.

ГЛАВА 6



Классы-коллекции

В листинге 5.2 мы разобрали строку на слова. Как их сохранить для дальнейшей обработки?

До сих пор мы пользовались массивами. Они удобны, если необходимо быстро обработать однотипные элементы, например, просуммировать числа, найти наибольшее и наименьшее значение, отсортировать элементы. Но уже для поиска нужных сведений в большом объеме информации массивы неудобны. Для этого лучше использовать бинарные деревья поиска.

Кроме того, массивы всегда имеют постоянную, предварительно заданную, длину, в массивы неудобно добавлять элементы. При удалении элемента из массива оставшиеся элементы следует перенумеровывать.

При решении задач, в которых количество элементов заранее неизвестно, элементы надо часто удалять и добавлять, надо искать другие способы хранения.

В языке Java с самых первых версий есть класс `Vector`, предназначенный для хранения переменного числа элементов самого общего типа `Object`.

Класс *Vector*

В классе `Vector` из пакета `java.util` хранятся элементы типа `Object`, а значит, любого типа. Количество элементов может быть любым и наперед не определяться. Элементы получают индексы 0, 1, 2, ... К каждому элементу вектора можно обратиться по индексу, как и к элементу массива.

Кроме количества элементов, называемого *размером* (size) вектора, есть еще размер буфера — *емкость* (capacity) вектора. Обычно емкость совпадает с размером вектора, но можно ее увеличить методом `ensureCapacity(int minCapacity)` или сравнять с размером вектора методом `trimToSize()`.

В Java 2 класс `Vector` переработан, чтобы включить его в иерархию классов-коллекций. Поэтому многие действия можно совершать старыми и новыми методами. Рекомендуется использовать новые методы, поскольку старые могут быть исключены из следующих версий Java.

Как создать вектор

В классе четыре конструктора:

- `Vector()` — создает пустой объект нулевой длины;
- `Vector(int capacity)` — создает пустой объект указанной емкости `capacity`;
- `Vector(int capacity, int increment)` — создает пустой объект указанной емкости `capacity` и задает число `increment`, на которое увеличивается емкость при необходимости;
- `Vector(Collection c)` — вектор создается по указанной коллекции.

Если `capacity` отрицательно, создается исключительная ситуация.

После создания вектора его можно заполнять элементами.

Как добавить элемент в вектор

Метод `add(Object element)` позволяет добавить элемент в конец вектора (то же делает старый метод `addElement(Object element)`.

Методом `add(int index, Object element)` или старым методом `insertElementAt(Object element, int index)` можно вставить элемент в указанное место `index`. Элемент, находившийся на этом месте, и все последующие элементы сдвигаются, их индексы увеличиваются на единицу.

Метод `addAll(Collection coll)` позволяет добавить в конец вектора все элементы коллекции `coll`.

Методом `addAll(int index, Collection coll)` возможно вставить в позицию `index` все элементы коллекции `coll`.

Как заменить элемент

Метод `set(int index, Object element)` заменяет элемент, стоявший в векторе в позиции `index`, на элемент `element` (то же позволяет выполнить старый метод `setElementAt(Object element, int index)`).

Как узнать размер вектора

Количество элементов в векторе всегда можно узнать методом `size()`.

Метод `capacity()` возвращает емкость вектора.

Логический метод `isEmpty()` возвращает `true`, если в векторе нет ни одного элемента.

Как обратиться к элементу вектора

Обратиться к первому элементу вектора можно методом `firstElement()`, к последнему — методом `lastElement()`, к любому элементу — методом `get(int index)` или старым методом `elementAt(int index)`.

Эти методы возвращают объект класса `Object`. Перед использованием его следует привести к нужному типу.

Получить все элементы вектора в виде массива типа `Object[]` можно методами `toArray()` и `toArray(Object[] a)`. Второй метод заносит все элементы вектора в массив `a`, если в нем достаточно места.

Как узнать, есть ли элемент в векторе

Логический метод `contains(Object element)` возвращает `true`, если элемент `element` находится в векторе.

Логический метод `containsAll(Collection c)` возвращает `true`, если вектор содержит все элементы указанной коллекции.

Как узнать индекс элемента

Четыре метода позволяют отыскать позицию указанного элемента `element`:

- `indexOf(Object element)` — возвращает индекс первого появления элемента в векторе;
- `indexOf(Object element, int begin)` — ведет поиск, начиная с индекса `begin` включительно;
- `lastIndexOf(Object element)` — возвращает индекс последнего появления элемента в векторе;
- `lastIndexOf(Object element, int start)` — ведет поиск от индекса `start` включительно к началу вектора.

Если элемент не найден, возвращается −1.

Как удалить элементы

Логический метод `remove(Object element)` удаляет из вектора первое вхождение указанного элемента `element`. Метод возвращает `true`, если элемент найден и удаление произведено.

Метод `remove(int index)` удаляет элемент из позиции `index` и возвращает его в качестве своего результата типа `Object`.

Аналогичные действия позволяют выполнить старые методы типа `void`: `removeElement(Object element)` и `removeElementAt(int index)`, не возвращающие результата.

Удалить диапазон элементов можно методом `removeRange(int begin, int end)`, не возвращающим результата. Удаляются элементы от позиции `begin` включительно до позиции `end` исключительно.

Удалить из данного вектора все элементы коллекции `coll` возможно логическим методом `removeAll(Collection coll)`.

Удалить последние элементы можно, просто урезав вектор методом `setSize(int newSize)`.

Удалить все элементы, кроме входящих в указанную коллекцию `coll`, разрешает логический метод `retainAll(Collection coll)`.

Удалить все элементы вектора можно методом `clear()` или старым методом `removeAllElements()` или обнулив размер вектора методом `setSize(0)`.

Листинг 6.1 расширяет листинг 5.2, обрабатывая выделенные из строки слова с помощью вектора.

Листинг 6.1. Работа с вектором

```
Vector v = new Vector();
String s = "Строка, которую мы хотим разобрать на слова.";
StringTokenizer st = new StringTokenizer(s, " \t\n\r,.");
while (st.hasMoreTokens()){
   // Получаем слово и заносим в вектор
   v.add(st.nextToken());                  // Добавляем в конец вектора
}
System.out.println(v.firstElement());      // Первый элемент
System.out.println(v.lastElement());       // Последний элемент
v.setSize(4);                              // Уменьшаем число элементов
v.add("собрать.");                         // Добавляем в конец
                                           // укороченного вектора
v.set(3, "опять");                         // Ставим в позицию 3
for (int i = 0; i < v.size(); i++)         // Перебираем весь вектор
    System.out.print(v.get(i) + " ");
System.out.println();
```

Класс `Vector` является примером того, как можно объекты класса `Object`, а значит, любые объекты, объединить в коллекцию. Этот тип коллекции упорядочивает и даже нумерует элементы. В векторе есть первый элемент, есть последний элемент. К каждому элементу обращаются непосредственно по

индексу. При добавлении и удалении элементов оставшиеся элементы автоматически перенумеровываются.

Второй пример коллекции — класс `Stack` — расширяет класс `Vector`.

Класс *Stack*

Класс `Stack` из пакета `java.util` объединяет элементы в стек.

Стек (stack) реализует порядок работы с элементами подобно магазину винтовки — первым выстрелит патрон, положенный в магазин последним, — или подобно железнодорожному тупику — первым из тупика выйдет вагон, загнанный туда последним. Такой порядок обработки называется LIFO (Last In — First Out).

Перед работой создается пустой стек конструктором `Stack()`.

Затем на стек кладутся и снимаются элементы, причем доступен только "верхний" элемент, тот, что положен на стек последним.

Дополнительно к методам класса `Vector` класс `Stack` содержит пять методов, позволяющих работать с коллекцией как со стеком:

- `push(Object item)` — помещает элемент `item` в стек;
- `pop()` — извлекает верхний элемент из стека;
- `peek()` — читает верхний элемент, не извлекая его из стека;
- `empty()` — проверяет, не пуст ли стек;
- `search(Object item)` — находит позицию элемента `item` в стеке. Верхний элемент имеет позицию 1, под ним элемент 2 и т. д. Если элемент не найден, возвращается −1.

Листинг 6.2 показывает, как можно использовать стек для проверки парности символов.

Листинг 6.2. Проверка парности скобок

```
import java.util.*;
class StackTest{
  static boolean checkParity(String expression,
                     String open, String close){
    Stack stack = new Stack();
    StringTokenizer st = new StringTokenizer(expression,
                          " \t\n\r+*/-(){}", true);
    while (st.hasMoreTokens()){
      String tmp = st.nextToken();
      if (tmp.equals(open))  stack.push(open);
```

```
      if (tmp.equals(close)) stack.pop();
    }
    if (stack.isEmpty()) return true;
    return false;
  }
  public static void main(String[] args){
    System.out.println(
       checkParity("a — (b — (c — a) / (b + c) — 2), "(",  ")"));
  }
}
```

Как видите, коллекции значительно облегчают обработку наборов данных.

Еще один пример коллекции совсем другого рода — таблицы — предоставляет класс `Hashtable`.

Класс *Hashtable*

Класс `Hashtable` расширяет абстрактный класс `Dictionary`. В объектах этого класса хранятся пары "ключ — значение".

Из таких пар "Фамилия И. О. — номер" состоит, например, телефонный справочник.

Еще один пример — анкета. Ее можно представить как совокупность пар "Фамилия — Иванов", "Имя — Петр", "Отчество — Сидорович", "Год рождения — 1975" и т. д.

Подобных примеров можно привести множество.

Каждый объект класса `Hashtable` кроме *размера* (size) — количества пар, имеет еще две характеристики: *емкость* (capacity) — размер буфера, и *показатель загруженности* (load factor) — процент заполненности буфера, по достижении которого увеличивается его размер.

Как создать таблицу

Для создания объектов класс `Hashtable` предоставляет четыре конструктора:

- `Hashtable()` — создает пустой объект с начальной емкостью в 101 элемент и показателем загруженности 0,75;
- `Hashtable(int capacity)` — создает пустой объект с начальной емкостью `capacity` и показателем загруженности 0,75;
- `Hashtable(int capacity, float loadFactor)` — создает пустой объект с начальной емкостью `capacity` и показателем загруженности `loadFactor`;
- `Hashtable(Map f)` — создает объект класса `Hashtable`, содержащий все элементы отображения `f`, с емкостью, равной удвоенному числу элементов отображения `f`, но не менее 11, и показателем загруженности 0,75.

Как заполнить таблицу

Для заполнения объекта класса `Hashtable` используются два метода:

- `Object put(Object key, Object value)` — добавляет пару "`key` — `value`", если ключа `key` не было в таблице, и меняет значение `value` ключа `key`, если он уже есть в таблице. Возвращает старое значение ключа или `null`, если его не было. Если хотя бы один параметр равен `null`, возникает исключительная ситуация;
- `void putAll(Map f)` — добавляет все элементы отображения `f`.

В объектах-ключах `key` должны быть реализованы методы `hashCode()` и `equals()`.

Как получить значение по ключу

Метод `get(Object key)` возвращает значение элемента с ключом `key` в виде объекта класса `Object`. Для дальнейшей работы его следует преобразовать к конкретному типу.

Как узнать наличие ключа или значения

Логический метод `containsKey(Object key)` возвращает `true`, если в таблице есть ключ `key`.

Логический метод `containsValue(Object value)` или старый метод `contains(Object value)` возвращают `true`, если в таблице есть ключи со значением `value`.

Логический метод `isEmpty()` возвращает `true`, если в таблице нет элементов.

Как получить все элементы таблицы

Метод `values()` представляет все значения `value` таблицы в виде интерфейса `Collection`. Все модификации в объекте `Collection` изменяют таблицу, и наоборот.

Метод `keySet()` предоставляет все ключи `key` таблицы в виде интерфейса `Set`. Все изменения в объекте `Set` корректируют таблицу, и наоборот.

Метод `entrySet()` представляет все пары "`key` — `value`" таблицы в виде интерфейса `Set`. Все модификации в объекте `Set` изменяют таблицу, и наоборот.

Метод `toString()` возвращает строку, содержащую все пары.

Старые методы `elements()` и `keys()` возвращают значения и ключи в виде интерфейса `Enumeration`.

Как удалить элементы

Метод `remove(Object key)` удаляет пару с ключом `key`, возвращая значение этого ключа, если оно есть, и `null`, если пара с ключом `key` не найдена.

Метод `clear()` удаляет все элементы, очищая таблицу.

В листинге 6.3 показано, как можно использовать класс `Hashtable` для создания телефонного справочника, а на рис. 6.1 — вывод этой программы.

Листинг 6.3. Телефонный справочник

```
import java.util.*;

class PhoneBook{
  public static void main(String[] args){
    Hashtable yp = new Hashtable();
    String name = null;
    yp.put("John",   "123-45-67");
    yp.put("Lennon", "567-34-12");
    yp.put("Bill",   "342-65-87");
    yp.put("Gates",  "423-83-49");
    yp.put("Batman", "532-25-08");
    try{
        name = args[0];
    }catch(Exception e){
      System.out.println("Usage: java PhoneBook Name");
      return;
    }
    if (yp.containsKey(name))
       System.out.println(name + "'s phone = " + yp.get(name));
    else
       System.out.println("Sorry, no such name");
  }
}
```

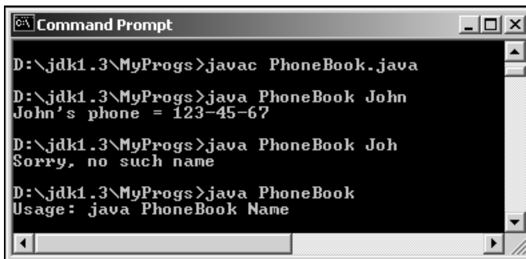


Рис. 6.1. Работа с телефонной книгой

Класс *Properties*

Класс `Properties` расширяет класс `Hashtable`. Он предназначен в основном для ввода и вывода пар свойств системы и их значений. Пары хранятся в виде строк типа `String`.

В классе `Properties` два конструктора:

- `Properties()` — создает пустой объект;
- `Properties(Properties default)` — создает объект с заданными парами свойств `default`.

Кроме унаследованных от класса `Hashtable` методов в классе `Properties` есть еще следующие методы.

- Два метода, возвращающих значение ключа-строки в виде строки:
  - `String getProperty(String key)` — возвращает значение по ключу `key`;
  - `String getProperty(String key, String defaultValue)` — возвращает значение по ключу `key`; если такого ключа нет, возвращается `defaultValue`.
- Метод `setProperty(String key, String value)` добавляет новую пару, если ключа `key` нет, и меняет значение, если ключ `key` есть.
- Метод `load(InputStream in)` загружает свойства из входного потока `in`.
- Методы `list(PrintStream out)` и `list(PrintWriter out)` выводят свойства в выходной поток `out`.
- Метод `store(OutputStream out, String header)` выводит свойства в выходной поток `out` с заголовком `header`.

Очень простой листинг 6.4 и рис. 6.2 демонстрируют вывод всех системных свойств Java.

Листинг 6.4. Вывод системных свойств

```
class Prop{
  public static void main(String[] args){
    System.getProperties().list(System.out);
  }
}
```

Примеры классов `Vector`, `Stack`, `Hashtable`, `Properties` показывают удобство классов-коллекций. Поэтому в Java 2 разработана целая иерархия коллекций. Она показана на рис. 6.3. Курсивом записаны имена интерфейсов. Пунктирные линии указывают классы, реализующие эти интерфейсы.

Все коллекции разбиты на три группы, описанные в интерфейсах `List`, `Set` и `Map`.

```
Command Prompt
D:\jdk1.3\MyProgs>javac Prop.java

D:\jdk1.3\MyProgs>java Prop
-- listing properties --
java.runtime.name=Java(TM) 2 Runtime Environment, Stand...
sun.boot.library.path=D:\JBuilder4\jdk1.3\jre\bin
java.vm.version=1.3.0-C
java.vm.vendor=Sun Microsystems Inc.
java.vendor.url=http://java.sun.com/
path.separator=;
java.vm.name=Java HotSpot(TM) Client VM
file.encoding.pkg=sun.io
java.vm.specification.name=Java Virtual Machine Specification
user.dir=D:\jdk1.3\MyProgs
java.runtime.version=1.3.0-C
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
os.arch=x86
java.io.tmpdir=D:\DOCUME~1\Habib\LOCALS~1\Temp\
line.separator=

java.vm.specification.vendor=Sun Microsystems Inc.
java.awt.fonts=
os.name=Windows 2000
java.library.path=D:\WINNT\system32;.;D:\WINNT\System32...
java.specification.name=Java Platform API Specification
java.class.version=47.0
os.version=5.0
user.home=D:\Documents and Settings\Habib
user.timezone=
java.awt.printerjob=sun.awt.windows.WPrinterJob
file.encoding=Cp1251
java.specification.version=1.3
user.name=Habib
java.class.path=.
java.vm.specification.version=1.0
java.home=D:\JBuilder4\jdk1.3\jre
user.language=ru
java.specification.vendor=Sun Microsystems Inc.
awt.toolkit=sun.awt.windows.WToolkit
java.vm.info=mixed mode
java.version=1.3.0
java.ext.dirs=D:\JBuilder4\jdk1.3\jre\lib\ext
sun.boot.class.path=D:\JBuilder4\jdk1.3\jre\lib\rt.jar;D:...
java.vendor=Sun Microsystems Inc.
file.separator=\
java.vendor.url.bug=http://java.sun.com/cgi-bin/bugreport...
sun.cpu.endian=little
sun.io.unicode.encoding=UnicodeLittle
user.region=RU
sun.cpu.isalist=pentium i486 i386

D:\jdk1.3\MyProgs>
```

Рис. 6.2. Системные свойства

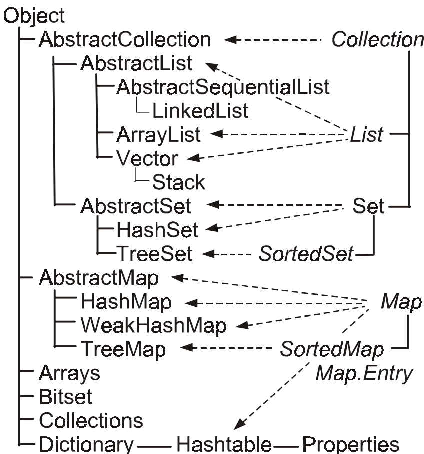


Рис. 6.3. Иерархия классов и интерфейсов-коллекций

Примером реализации интерфейса List может служить класс Vector, примером реализации интерфейса Map — класс Hashtable.

Коллекции List и Set имеют много общего, поэтому их общие методы объединены и вынесены в суперинтерфейс Collection.

Посмотрим, что, по мнению разработчиков Java API, должно содержаться в этих коллекциях.

Интерфейс *Collection*

Интерфейс Collection из пакета java.util описывает общие свойства коллекций List и Set. Он содержит методы добавления и удаления элементов, проверки и преобразования элементов:

- boolean add(Object obj) — добавляет элемент obj в конец коллекции; возвращает false, если такой элемент в коллекции уже есть, а коллекция не допускает повторяющиеся элементы; возвращает true, если добавление прошло удачно;
- boolean addAll(Collection coll) — добавляет все элементы коллекции coll в конец данной коллекции;
- void clear() — удаляет все элементы коллекции;
- boolean contains(Object obj) — проверяет наличие элемента obj в коллекции;
- boolean containsAll(Collection coll) — проверяет наличие всех элементов коллекции coll в данной коллекции;
- boolean isEmpty() — проверяет, пуста ли коллекция;
- Iterator iterator() — возвращает итератор данной коллекции;
- boolean remove(Object obj) — удаляет указанный элемент из коллекции; возвращает false, если элемент не найден, true, если удаление прошло успешно;
- boolean removeAll(Collection coll) — удаляет элементы указанной коллекции, лежащие в данной коллекции;
- boolean retainAll(Collection coll) — удаляет все элементы данной коллекции, кроме элементов коллекции coll;
- int size() — возвращает количество элементов в коллекции;
- Object[] toArray() — возвращает все элементы коллекции в виде массива;
- Object[] toArray(Object[] a) — записывает все элементы коллекции в массив a, если в нем достаточно места.

Интерфейс *List*

Интерфейс `List` из пакета `java.util`, расширяющий интерфейс `Collection`, описывает методы работы с упорядоченными коллекциями. Иногда их называют *последовательностями* (sequence). Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу. В отличие от коллекции `Set` элементы коллекции `List` могут повторяться.

Класс `Vector` — одна из реализаций интерфейса `List`.

Интерфейс `List` добавляет к методам интерфейса `Collection` методы, использующие индекс `index` элемента:

- `void add(int index, Object obj)` — вставляет элемент `obj` в позицию `index`; старые элементы, начиная с позиции `index`, сдвигаются, их индексы увеличиваются на единицу;
- `boolean addAll(int index, Collection coll)` — вставляет все элементы коллекции `coll`;
- `Object get(int index)` — возвращает элемент, находящийся в позиции `index`;
- `int indexOf(Object obj)` — возвращает индекс первого появления элемента `obj` в коллекции;
- `int lastIndexOf(Object obj)` — возвращает индекс последнего появления элемента `obj` в коллекции;
- `ListIterator listIterator()` — возвращает итератор коллекции;
- `ListIterator listIterator(int index)` — возвращает итератор конца коллекции от позиции `index`;
- `Object set(int index, Object obj)` — заменяет элемент, находящийся в позиции `index`, элементом `obj`;
- `List subList(int from, int to)` — возвращает часть коллекции от позиции `from` включительно до позиции `to` исключительно.

Интерфейс *Set*

Интерфейс `Set` из пакета `java.util`, расширяющий интерфейс `Collection`, описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов. Это соответствует математическому понятию *множества* (set). Такие коллекции удобны для проверки наличия или отсутствия у элемента свойства, определяющего множество. Новые методы в интерфейс `Set` не добавлены, просто метод `add()` не станет добавлять еще одну копию элемента, если такой элемент уже есть в множестве.

Этот интерфейс расширен интерфейсом `SortedSet`.

Интерфейс *SortedSet*

Интерфейс `SortedSet` из пакета `java.util`, расширяющий интерфейс `Set`, описывает упорядоченное множество, отсортированное по естественному порядку возрастания его элементов или по порядку, заданному реализацией интерфейса `Comparator`.

Элементы не нумеруются, но есть понятие первого, последнего, большего и меньшего элемента.

Дополнительные методы интерфейса отражают эти понятия:

- `Comparator comparator()` — возвращает способ упорядочения коллекции;
- `Object first()` — возвращает первый, меньший элемент коллекции;
- `SortedSet headSet(Object toElement)` — возвращает начальные, меньшие элементы до элемента `toElement` исключительно;
- `Object last()` — возвращает последний, больший элемент коллекции;
- `SortedSet subSet(Object fromElement, Object toElement)` — возвращает подмножество коллекции от элемента `fromElement` включительно до элемента `toElement` исключительно;
- `SortedSet tailSet(Object fromElement)` — возвращает последние, большие элементы коллекции от элемента `fromElement` включительно.

Интерфейс *Map*

Интерфейс `Map` из пакета `java.util` описывает коллекцию, состоящую из пар "ключ — значение". У каждого ключа только одно значение, что соответствует математическому понятию однозначной функции или *отображения* (map).

Такую коллекцию часто называют еще *словарем* (dictionary) или *ассоциативным массивом* (associative array).

Обычный массив — простейший пример словаря с заранее заданным числом элементов. Это отображение множества первых неотрицательных целых чисел на множество элементов массива, множество пар "индекс массива — элемент массива".

Класс `HashTable` — одна из реализаций интерфейса `Map`.

Интерфейс `Map` содержит методы, работающие с ключами и значениями:

- `boolean containsKey(Object key)` — проверяет наличие ключа `key`;
- `boolean containsValue(Object value)` — проверяет наличие значения `value`;
- `Set entrySet()` — представляет коллекцию в виде множества, каждый элемент которого — пара из данного отображения, с которой можно работать методами вложенного интерфейса `Map.Entry`;

- Object get(Object key) — возвращает значение, отвечающее ключу key;
- Set keySet() — представляет ключи коллекции в виде множества;
- Object put(Object key, Object value) — добавляет пару "key — value", если такой пары не было, и заменяет значение ключа key, если такой ключ уже есть в коллекции;
- void putAll(Map m) — добавляет к коллекции все пары из отображения m;
- Collection values() — представляет все значения в виде коллекции.

В интерфейс Map вложен интерфейс Map.Entry, содержащий методы работы с отдельной парой.

Вложенный интерфейс *Map.Entry*

Этот интерфейс описывает методы работы с парами, полученными методом entrySet():

- методы getKey() и getValue() позволяют получить ключ и значение пары;
- метод setValue(Object value) меняет значение в данной паре.

Интерфейс *SortedMap*

Интерфейс SortedMap, расширяющий интерфейс Map, описывает упорядоченную по ключам коллекцию Map. Сортировка производится либо в естественном порядке возрастания ключей, либо в порядке, описываемом в интерфейсе Comparator.

Элементы не нумеруются, но есть понятия большего и меньшего из двух элементов, первого, самого маленького, и последнего, самого большого элемента коллекции. Эти понятия описываются следующими методами:

- Comparator comparator() — возвращает способ упорядочения коллекции;
- Object firstKey() — возвращает первый, меньший элемент коллекции;
- SortedMap headMap(Object toKey) — возвращает начало коллекции до элемента с ключом toKey исключительно;
- Object lastKey() — возвращает последний, больший ключ коллекции;
- SortedMap subMap(Object fromKey, Object toKey) — возвращает часть коллекции от элемента с ключом fromKey включительно до элемента с ключом toKey исключительно;
- SortedMap tailMap(Object fromKey) — возвращает остаток коллекции от элемента fromKey включительно.

Вы можете создать свои коллекции, реализовав рассмотренные интерфейсы. Это дело трудное, поскольку в интерфейсах много методов. Чтобы облегчить

эту задачу, в Java API введены частичные реализации интерфейсов — абстрактные классы-коллекции.

Абстрактные классы-коллекции

Эти классы лежат в пакете `java.util`.

Абстрактный класс `AbstractCollection` реализует интерфейс `Collection`, но оставляет нереализованными методы `iterator()`, `size()`.

Абстрактный класс `AbstractList` реализует интерфейс `List`, но оставляет нереализованным метод `get(int)` и унаследованный метод `size()`. Этот класс позволяет реализовать коллекцию с прямым доступом к элементам, подобно массиву.

Абстрактный класс `AbstractSequantialList` реализует интерфейс `List`, но оставляет нереализованным метод `listIterator(int index)` и унаследованный метод `size()`. Данный класс позволяет реализовать коллекции с последовательным доступом к элементам с помощью итератора `ListIterator`.

Абстрактный класс `AbstractSet` реализует интерфейс `Set`, но оставляет нереализованными методы, унаследованные от `AbstractCollection`.

Абстрактный класс `AbstractMap` реализует интерфейс `Map`, но оставляет нереализованным метод `entrySet()`.

Наконец, в составе Java API есть полностью реализованные классы-коллекции. Помимо уже рассмотренных классов `Vector`, `Stack`, `Hashtable` и `Properties`, это классы `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, `HashMap`, `TreeMap`, `WeakHashMap`.

Для работы с этими классами разработаны интерфейсы `Iterator`, `ListIterator`, `Comparator` и классы `Arrays` и `Collections`.

Перед тем как рассмотреть использование данных классов, обсудим понятие итератора.

Интерфейс *Iterator*

В 70—80-х годах прошлого столетия, после того как была осознана важность правильной организации данных в определенную структуру, большое внимание уделялось изучению и построению различных структур данных: связанных списков, очередей, деков, стеков, деревьев, сетей.

Вместе с развитием структур данных развивались и алгоритмы работы с ними: сортировка, поиск, обход, хэширование.

Этим вопросам посвящена обширная литература, посмотрите, например, книгу [11].

В 90-х годах было решено заносить данные в определенную коллекцию, скрыв ее внутреннюю структуру, а для работы с данными использовать методы этой коллекции.

В частности, задачу обхода возложили на саму коллекцию. В Java API введен интерфейс `Iterator`, описывающий способ обхода всех элементов коллекции. В каждой коллекции есть метод `iterator()`, возвращающий реализацию интерфейса `Iterator` для указанной коллекции. Получив эту реализацию, можно обходить коллекцию в некотором порядке, определенном данным итератором, с помощью методов, описанных в интерфейсе `Iterator` и реализованных в этом итераторе. Подобная техника использована в классе `StringTokenizer`.

В интерфейсе `Iterator` описаны всего три метода:

- логический метод `hasNext()` возвращает `true`, если обход еще не завершен;
- метод `next()`делает текущим следующий элемент коллекции и возвращает его в виде объекта класса `Object`;
- метод `remove()` удаляет текущий элемент коллекции.

Можно представить себе дело так, что итератор — это указатель на элемент коллекции. При создании итератора указатель устанавливается перед первым элементом, метод `next()` перемещает указатель на первый элемент и показывает его. Следующее применение метода `next()` перемещает указатель на второй элемент коллекции и показывает его. Последнее применение метода `next()` выводит указатель за последний элемент коллекции.

Метод `remove()`, пожалуй, излишен, он уже не относится к задаче обхода коллекции, но позволяет при просмотре коллекции удалять из нее ненужные элементы.

В листинге 6.5 к тексту листинга 6.1 добавлена работа с итератором.

Листинг 6.5. Использование итератора вектора

```
Vector v = new Vector();
String s = "Строка, которую мы хотим разобрать на слова.";
StringTokenizer st = new StringTokenizer(s, " \t\n\r,.");
while (st.hasMoreTokens()){
   // Получаем слово и заносим в вектор.
   v.add(st.nextToken());                 // Добавляем в конец вектора
}
System.out.println(v.firstElement());     // Первый элемент
System.out.println(v.lastElement());      // Последний элемент
v.setSize(4);                             // Уменьшаем число элементов
v.add("собрать.");              // Добавляем в конец укороченного вектора
v.set(3, "опять");                        // Ставим в позицию 3
```

```
for (int i = 0; i < v.size(); i++)        // Перебираем весь вектор
    System.out.print(v.get(i) + " ");
System.out.println();
Iterator it = v.iterator();               // Получаем итератор вектора
try{
  while(it.hasNext())                     // Пока в векторе есть элементы,
     System.out.println(it.next());       // выводим текущий элемент
}catch(Exception e){}
```

Интерфейс *ListIterator*

Интерфейс `ListIterator` расширяет интерфейс `Iterator`, обеспечивая перемещение по коллекции как в прямом, так и в обратном направлении. Он может быть реализован только в тех коллекциях, в которых есть понятия следующего и предыдущего элемента и где элементы пронумерованы.

В интерфейс `ListIterator` добавлены следующие методы:

- `void add(Object element)` — добавляет элемент `element` перед текущим элементом;
- `boolean hasPrevious()` — возвращает `true`, если в коллекции есть элементы, стоящие перед текущим элементом;
- `int nextIndex()` — возвращает индекс текущего элемента; если текущим является последний элемент коллекции, возвращает размер коллекции;
- `Object previous()` — возвращает предыдущий элемент и делает его текущим;
- `int previousIndex()` — возвращает индекс предыдущего элемента;
- `void set(Object element)` — заменяет текущий элемент элементом `element`; выполняется сразу после `next()` или `previous()`.

Как видите, итераторы могут изменять коллекцию, в которой они работают, добавляя, удаляя и заменяя элементы. Чтобы это не приводило к конфликтам, предусмотрена исключительная ситуация, возникающая при попытке использования итераторов параллельно "родным" методам коллекции. Именно поэтому в листинге 6.5 действия с итератором заключены в блок `try{}catch(){}`.

Изменим окончание листинга 6.5 с использованием итератора `ListIterator`.

```
// Текст листинга 6.1...
// ...
ListIterator lit = v.listIterator();       // Получаем итератор вектора
                      // Указатель сейчас находится перед началом вектора
try{
  while(lit.hasNext())                     // Пока в векторе есть элементы
```

```
    System.out.println(lit.next());       // Переходим к следующему
                                          // элементу и выводим его
               // Теперь указатель за концом вектора. Пройдем к началу
  while(lit.hasPrevious())
    System.out.println(lit.previous());
}catch(Exception e){}
```

Интересно, что повторное применение методов `next()` и `previous()` друг за другом будет выдавать один и тот же текущий элемент.

Посмотрим теперь, какие возможности предоставляют классы-коллекции Java 2.

Классы, создающие списки

Класс `ArrayList` полностью реализует интерфейс `List` и итератор типа `Iterator`. Класс `ArrayList` очень похож на класс `Vector`, имеет тот же набор методов и может использоваться в тех же ситуациях.

В классе `ArrayList` три конструктора:

- `ArrayList()` — создает пустой объект;
- `ArrayList(Collection coll)` — создает объект, содержащий все элементы коллекции `coll`;
- `ArrayList(int initCapacity)` — создает пустой объект емкости `initCapacity`.

Единственное отличие класса `ArrayList` от класса `Vector` заключается в том, что класс `ArrayList` не синхронизован. Это означает, что одновременное изменение экземпляра этого класса несколькими подпроцессами приведет к непредсказуемым результатам.

Эти вопросы мы рассмотрим в *главе 17*.

Двунаправленный список

Класс `LinkedList` полностью реализует интерфейс `List` и содержит дополнительные методы, превращающие его в двунаправленный список. Он реализует итераторы типа `Iterator` и `ListIterator`.

Этот класс можно использовать для обработки элементов в стеке, деке или двунаправленном списке.

В классе `LinkedList` два конструктора:

- `LinkedList()` — создает пустой объект;
- `LinkedList(Collection coll)` — создает объект, содержащий все элементы коллекции `coll`.

Классы, создающие отображения

Класс `HashMap` полностью реализует интерфейс `Map`, а также итератор типа `Iterator`. Класс `HashMap` очень похож на класс `Hashtable` и может использоваться в тех же ситуациях. Он имеет тот же набор функций и такие же конструкторы:

- `HashMap()` — создает пустой объект с показателем загруженности 0,75;
- `HashMap(int capacity)` — создает пустой объект с начальной емкостью `capacity` и показателем загруженности 0,75;
- `HashMap(int capacity, float loadFactor)` — создает пустой объект с начальной емкостью `capacity` и показателем загруженности `loadFactor`;
- `HashMap(Map f)` — создает объект класса `HashMap`, содержащий все элементы отображения `f`, с емкостью, равной удвоенному числу элементов отображения `f`, но не менее 11, и показателем загруженности 0,75.

Класс `WeakHashMap` отличается от класса `HashMap` только тем, что в его объектах неиспользуемые элементы, на которые никто не ссылается, автоматически исключаются из объекта.

Упорядоченные отображения

Класс `TreeMap` полностью реализует интерфейс `SortedMap`. Он реализован как бинарное дерево поиска, значит, его элементы хранятся в упорядоченном виде. Это значительно ускоряет поиск нужного элемента.

Порядок задается либо естественным следованием элементов, либо объектом, реализующим интерфейс сравнения `Comparator`.

В этом классе четыре конструктора:

- `TreeMap()` — создает пустой объект с естественным порядком элементов;
- `TreeMap(Comparator c)` — создает пустой объект, в котором порядок задается объектом сравнения `c`;
- `TreeMap(Map f)` — создает объект, содержащий все элементы отображения `f`, с естественным порядком его элементов;
- `TreeMap(SortedMap sf)` — создает объект, содержащий все элементы отображения `sf`, в том же порядке.

Здесь надо пояснить, каким образом можно задать упорядоченность элементов коллекции.

Сравнение элементов коллекций

Интерфейс `Comparator` описывает два метода сравнения:

- `int compare(Object obj1, Object obj2)` — возвращает отрицательное число, если `obj1` в каком-то смысле меньше `obj2`; нуль, если они считаются равными; положительное число, если `obj1` больше `obj2`. Для читателей, знакомых с теорией множеств, скажем, что этот метод сравнения обладает свойствами тождества, антисимметричности и транзитивности;
- `boolean equals(Object obj)` — сравнивает данный объект с объектом `obj`, возвращая `true`, если объекты совпадают в каком-либо смысле, заданном этим методом.

Для каждой коллекции можно реализовать эти два метода, задав конкретный способ сравнения элементов, и определить объект класса `SortedMap` вторым конструктором. Элементы коллекции будут автоматически отсортированы в заданном порядке.

Листинг 6.6 показывает один из возможных способов упорядочения комплексных чисел — объектов класса `Complex` из листинга 2.4. Здесь описывается класс `ComplexCompare`, реализующий интерфейс `Comparator`. В листинге 6.7 он применяется для упорядоченного хранения множества комплексных чисел.

Листинг 6.6. Сравнение комплексных чисел

```
import java.util.*;

class ComplexCompare implements Comparator{
  public int compare(Object obj1, Object obj2){
    Complex z1 = (Complex)obj1, z2 = (Complex)obj2;
    double re1 = z1.getRe(), im1 = z1.getIm();
    double re2 = z2.getRe(), im2 = z2.getIm();
    if (re1 != re2) return (int)(re1 — re2);
    else if (im1 != im2) return (int)(im1 — im2);
         else return 0;
  }
  public boolean equals(Object z){
    return compare(this, z) == 0;
  }
}
```

Классы, создающие множества

Класс `HashSet` полностью реализует интерфейс `Set` и итератор типа `Iterator`. Класс `HashSet` используется в тех случаях, когда надо хранить только одну копию каждого элемента.

В классе `HashSet` четыре конструктора:

- `HashSet()` — создает пустой объект с показателем загруженности 0,75;
- `HashSet(int capacity)` — создает пустой объект с начальной емкостью `capacity` и показателем загруженности 0,75;
- `HashSet(int capacity, float loadFactor)` — создает пустой объект с начальной емкостью `capacity` и показателем загруженности `loadFactor`;
- `HashSet(Collection coll)` — создает объект, содержащий все элементы коллекции `coll`, с емкостью, равной удвоенному числу элементов коллекции `coll`, но не менее 11, и показателем загруженности 0,75.

Упорядоченные множества

Класс `TreeSet` полностью реализует интерфейс `SortedSet` и итератор типа `Iterator`. Класс `TreeSet` реализован как бинарное дерево поиска, значит, его элементы хранятся в упорядоченном виде. Это значительно ускоряет поиск нужного элемента.

Порядок задается либо естественным следованием элементов, либо объектом, реализующим интерфейс сравнения `Comparator`.

Этот класс удобен при поиске элемента во множестве, например, для проверки, обладает ли какой-либо элемент свойством, определяющим множество.

В классе `TreeSet` четыре конструктора:

- `TreeSet()` — создает пустой объект с естественным порядком элементов;
- `TreeSet(Comparator c)` — создает пустой объект, в котором порядок задается объектом сравнения `c`;
- `TreeSet(Collection coll)` — создает объект, содержащий все элементы коллекции `coll`, с естественным порядком ее элементов;
- `TreeSet(SortedMap sf)` — создает объект, содержащий все элементы отображения `sf`, в том же порядке.

В листинге 6.7 показано, как можно хранить комплексные числа в упорядоченном виде. Порядок задается объектом класса `ComplexCompare`, определенного в листинге 6.6.

Листинг 6.7. Хранение комплексных чисел в упорядоченном виде

```
TreeSet ts = new TreeSet(new ComplexCompare());
ts.add(new Complex(1.2, 3.4));
ts.add(new Complex(-1.25, 33.4));
ts.add(new Complex(1.23, -3.45));
ts.add(new Complex(16.2, 23.4));
```

```
    Iterator it = ts.iterator();
    while(it.hasNext())
      ((Complex)it.next()).pr();
```

Действия с коллекциями

Коллекции предназначены для хранения элементов в удобном для дальнейшей обработки виде. Очень часто обработка заключается в сортировке элементов и поиске нужного элемента. Эти и другие методы обработки собраны в класс `Collections`.

Методы класса *Collections*

Все методы класса `Collections` статические, ими можно пользоваться, не создавая экземпляры класса `Collections`.

Как обычно в статических методах, коллекция, с которой работает метод, задается его аргументом.

Сортировка может быть сделана только в упорядочиваемой коллекции, реализующей интерфейс `List`. Для сортировки в классе `Collections` есть два метода:

- `static void sort(List coll)` — сортирует в естественном порядке возрастания коллекцию `coll`, реализующую интерфейс `List`;
- `static void sort(List coll, Comparator c)` — сортирует коллекцию `coll` в порядке, заданном объектом `c`.

После сортировки можно осуществить бинарный поиск в коллекции:

- `static int binarySearch(List coll, Object element)` — отыскивает элемент `element` в отсортированной в естественном порядке возрастания коллекции `coll` и возвращает индекс элемента или отрицательное число, если элемент не найден; отрицательное число показывает индекс, с которым элемент `element` был бы вставлен в коллекцию, с обратным знаком;
- `static int binarySearch(List coll, Object element, Comparator c)` — то же, но коллекция отсортирована в порядке, определенном объектом `c`.

Четыре метода находят наибольший и наименьший элементы в упорядочиваемой коллекции:

- `static Object max(Collection coll)` — возвращает наибольший в естественном порядке элемент коллекции `coll`;
- `static Object max(Collection coll, Comparator c)` — то же в порядке, заданном объектом `c`;
- `static Object min(Collection coll)` — возвращает наименьший в естественном порядке элемент коллекции `coll`;

- `static Object min(Collection coll, Comparator c)` — то же в порядке, заданном объектом `c`.

Два метода "перемешивают" элементы коллекции в случайном порядке:

- `static void shuffle(List coll)` — случайные числа задаются по умолчанию;
- `static void shuffle(List coll, Random r)` — случайные числа определяются объектом `r`.

Метод `reverse(List coll)` меняет порядок расположения элементов на обратный.

Метод `copy(List from, List to)` копирует коллекцию `from` в коллекцию `to`.

Метод `fill(List coll, Object element)` заменяет все элементы существующей коллекции `coll` элементом `element`.

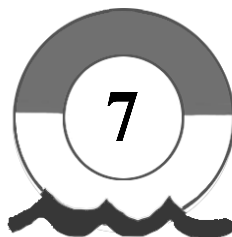
С остальными методами познакомимся по мере надобности.

Заключение

Итак, в данной главе мы выяснили, что язык Java предоставляет множество средств для работы с большими объемами информации. В большинстве случаев достаточно добавить в программу три-пять операторов, чтобы можно было проделать нетривиальную обработку информации.

В следующей главе мы рассмотрим аналогичные средства для работы с массивами, датами, для получения случайных чисел и прочих необходимых средств программирования.

ГЛАВА 7



Классы-утилиты

В этой главе описаны средства, полезные для создания программ: работа с массивами, датами, случайными числами.

Работа с массивами

В классе `Arrays` из пакета `java.util` собрано множество методов для работы с массивами. Их можно разделить на четыре группы.

Восемнадцать статических методов сортируют массивы с разными типами числовых элементов в порядке возрастания чисел или просто объекты в их естественном порядке.

Восемь из них имеют простой вид

```
static void sort(type[] a)
```

где `type` может быть один из семи примитивных типов `byte`, `short`, `int`, `long`, `char`, `float`, `double` или тип `Object`.

Восемь методов с теми же типами сортируют часть массива от индекса `from` включительно до индекса `to` исключительно:

```
static void sort(type[] a, int from, int to)
```

Оставшиеся два метода сортировки упорядочивают массив или его часть с элементами типа `Object` по правилу, заданному объектом `c`, реализующим интерфейс `Comparator`:

```
static void sort(Object[] a, Comparator c)
static void sort(Object[] a, int from, int to, Comparator c)
```

После сортировки можно организовать бинарный поиск в массиве одним из девяти статических методов поиска. Восемь методов имеют вид

```
static int binarySearch(type[] a, type element)
```

где `type` — один из тех же восьми типов.

Девятый метод поиска имеет вид

```
static int binarySearch(Object[] a, Object element, Comparator c).
```

Он отыскивает элемент `element` в массиве, отсортированном в порядке, заданном объектом `c`.

Методы поиска возвращают индекс найденного элемента массива. Если элемент не найден, то возвращается отрицательное число, означающее индекс, с которым элемент был бы вставлен в массив в заданном порядке, с обратным знаком.

Восемнадцать статических методов заполняют массив или часть массива указанным значением `value`:

```
static void fill(type[], type value)
static void fill(type[], int from, int to, type value)
```

где `type` — один из восьми примитивных типов или тип `Object`.

Наконец, девять статических логических методов сравнивают массивы:

```
static boolean equals(type[] a1, type[] a2)
```

где `type` — один из восьми примитивных типов или тип `Object`.

Массивы считаются равными, и возвращается `true`, если они имеют одинаковую длину и равны элементы массивов с одинаковыми индексами.

В листинге 7.1 приведен простой пример работы с этими методами.

Листинг 7.1. Применение методов класса `Arrays`

```
import java.util.*;

class ArraysTest{
  public static void main(String[] args){
    int[] a = {34, -45, 12, 67, -24, 45, 36, -56};
    Arrays.sort(a);
    for (int i = 0; i < a.length; i++)
      System.out.print(a[i] + " ");
    System.out.println();
    Arrays.fill(a, Arrays.binarySearch(a, 12), a.length, 0);
    for (int i = 0; i < a.length; i++)
      System.out.print(a[i] + " ");
    System.out.println();
  }
}
```

Локальные установки

Некоторые данные — даты, время — традиционно представляются в разных местностях по-разному. Например, дата в России выводится в формате число, месяц, год через точку: 27.06.01. В США принята запись месяц/число/год через наклонную черту: 06/27/01.

Совокупность таких форматов для данной местности, как говорят на жаргоне "локаль", хранится в объекте класса `Locale` из пакета `java.util`. Для создания такого объекта достаточно знать язык `language` и местность `country`. Иногда требуется третья характеристика — вариант `variant`, определяющая программный продукт, например, "`WIN`", "`MAC`", "`POSIX`".

По умолчанию местные установки определяются операционной системой и читаются из системных свойств. Посмотрите на строки (см. рис. 6.2):

```
user.language = ru        // Язык — русский
user.region = RU          // Местность — Россия
file.encoding = Cp1251    // Байтовая кодировка — CP1251
```

Они определяют русскую локаль и локальную кодировку байтовых символов. Локаль, установленную по умолчанию на той машине, где выполняется программа, можно выяснить статическим методом `Locale.getDefault()`.

Чтобы работать с другой локалью, ее надо прежде всего создать. Для этого в классе `Locale` есть два конструктора:

```
Locale(String language, String country)
Locale(String language, String country, String variant)
```

Параметр `language` — это строка из двух строчных букв, определенная стандартом ISO639, например, "`ru`", "`fr`", "`en`". Параметр `country` — строка из двух прописных букв, определенная стандартом ISO3166, например, "`RU`", "`US`", "`GB`". Параметр `variant` не определяется стандартом, это может быть, например, строка "`Traditional`".

Локаль часто указывают одной строкой "`ru_RU`", "`en_GB`", "`en_US`", "`en_CA`" и т. д.

После создания локали можно сделать ее локалью по умолчанию статическим методом:

```
Locale.setDefault(Locale newLocale);
```

Несколько статических методов класса `Locale` позволяют получить параметры локали по умолчанию, или локали, заданной параметром `locale`:

- `String getCountry()` — стандартный код страны из двух букв;
- `String getDisplayCountry()` — страна записывается словом, обычно выводящимся на экран;
- `String getDisplayCountry(Locale locale)` — то же для указанной локали.

Такие же методы есть для языка и варианта.

Можно просмотреть список всех локалей, определенных для данной JVM, и их параметров, выводимый в стандартном виде:

```
Locale[] getAvailableLocales()
String[] getISOCountries()
String[] getISOLanguages()
```

Установленная локаль в дальнейшем используется при выводе данных в местном формате.

Работа с датами и временем

Методы работы с датами и показаниями времени собраны в два класса: `Calendar` и `Date` из пакета `java.util`.

Объект класса `Date` хранит число миллисекунд, прошедших с 1 января 1970 г. 00:00:00 по Гринвичу. Это "день рождения" UNIX, он называется "Epoch".

Класс `Date` удобно использовать для отсчета промежутков времени в миллисекундах.

Получить текущее число миллисекунд, прошедших с момента Epoch на той машине, где выполняется программа, можно статическим методом

```
System.currentTimeMillis()
```

В классе `Date` два конструктора. Конструктор `Date()` заносит в создаваемый объект текущее время машины, на которой выполняется программа, по системным часам, а конструктор `Date(long millisec)` — указанное число.

Получить значение, хранящееся в объекте, можно методом `long getTime()`, установить новое значение — методом `setTime(long newTime)`.

Три логических метода сравнивают отсчеты времени:

- `boolean after(long when)` — возвращает `true`, если время `when` больше данного;
- `boolean before(long when)` — возвращает `true`, если время `when` меньше данного;
- `boolean after(Object when)` — возвращает `true`, если `when` — объект класса `Date` и времена совпадают.

Еще два метода, сравнивая отсчеты времени, возвращают отрицательное число типа `int`, если данное время меньше аргумента `when`; нуль, если времена совпадают; положительное число, если данное время больше аргумента `when`:

- `int compareTo(Date when)`;
- `int compareTo(Object when)` — если `when` не относится к объектам класса `Date`, создается исключительная ситуация.

Преобразование миллисекунд, хранящихся в объектах класса `Date`, в текущее время и дату производится методами класса `Calendar`.

Часовой пояс и летнее время

Методы установки и изменения часового пояса (time zone), а также летнего времени DST (Daylight Savings Time), собраны в абстрактном классе `TimeZone` из пакета `java.util`. В этом же пакете есть его реализация — подкласс `SimpleTimeZone`.

В классе `SimpleTimeZone` три конструктора, но чаще всего объект создается статическим методом `getDefault()`, возвращающим часовой пояс, установленный на машине, выполняющей программу.

В этих классах множество методов работы с часовыми поясами, но в большинстве случаев требуется только узнать часовой пояс на машине, выполняющей программу, статическим методом `getDefault()`, проверить, осуществляется ли переход на летнее время, логическим методом `useDaylightTime()`, и установить часовой пояс методом `setDefault(TimeZone zone)`.

Класс *Calendar*

Класс `Calendar` — абстрактный, в нем собраны общие свойства календарей: юлианского, григорианского, лунного. В Java API пока есть только одна его реализация — подкласс `GregorianCalendar`.

Поскольку `Calendar` — абстрактный класс, его экземпляры создаются четырьмя статическими методами по заданной локали и/или часовому поясу:

```
Calendar getInstance()
Calendar getInstance(Locale loc)
Calendar getInstance(TimeZone tz)
Calendar getInstance(TimeZone tz, Locale loc)
```

Для работы с месяцами определены целочисленные константы от `JANUARY` до `DECEMBER`, а для работы с днями недели — константы от `MONDAY` до `SUNDAY`.

Первый день недели можно узнать методом `int getFirstDayOfWeek()`, а установить — методом `setFirstDayOfWeek(int day)`, например:

```
setFirstDayOfWeek(Calendar.MONDAY)
```

Остальные методы позволяют просмотреть время и часовой пояс или установить их.

Подкласс *GregorianCalendar*

В григорианском календаре две целочисленные константы определяют эры: `BC` (before Christ) и `AD` (Anno Domini).

Семь конструкторов определяют календарь по времени, часовому поясу и/или локали:

```
GregorianCalendar()
GregorianCalendar(int year, int month, int date)
GregorianCalendar(int year, int month, int date, int hour, int minute)
GregorianCalendar(int year, int month, int date,
int hour, int minute, int second)
GregorianCalendar(Locale loc)
GregorianCalendar(TimeZone tz)
GregorianCalendar(TimeZone tz, Locale loc)
```

После создания объекта следует определить дату перехода с юлианского календаря на григорианский календарь методом `setGregorianChange(Date date)`. По умолчанию это 15 октября 1582 г. На территории России переход был осуществлен 14 февраля 1918 г., значит, создание объекта `greg` надо выполнить так:

```
GregorianCalendar greg = new GregorianCalendar();
greg.setGregorianChange(new
    GregorianCalendar(1918, Calendar.FEBRUARY, 14).getTime());
```

Узнать, является ли год високосным в григорианском календаре, можно логическим методом `isLeapYear()`.

Метод `get(int field)` возвращает элемент календаря, заданный аргументом `field`. Для этого аргумента в классе `Calendar` определены следующие статические целочисленные константы:

```
ERA          WEEK_OF_YEAR     DAY_OF_WEEK            SECOND
YEAR         WEEK_OF_MONTH    DAY_OF_WEEK_IN_MONTH   MILLISECOND
MONTH        DAY_OF_YEAR      HOUR_OF_DAY            ZONE_OFFSET
DATE         DAY_OF_MONTH     MINUTE                 DST_OFFSET
```

Несколько методов `set()`, использующих эти константы, устанавливают соответствующие значения.

Представление даты и времени

Различные способы представления дат и показаний времени можно осуществить методами, собранными в абстрактный класс `DateFormat` и его подкласс `SimpleDateFormat` из пакета `java.text`.

Класс `DateFormat` предлагает четыре стиля представления даты и времени:

- стиль `SHORT` представляет дату и время в коротком числовом виде: 27.04.01 17:32; в локали США: 4/27/01 5:32 PM;
- стиль `MEDIUM` задает год четырьмя цифрами и показывает секунды: 27.04.2001 17:32:45; в локали США месяц представляется тремя буквами;

- стиль LONG представляет месяц словом и добавляет часовой пояс: 27 апрель 2001 г. 17:32:45 GMT+03:00;
- стиль FULL в русской локали таков же, как и стиль LONG; в локали США добавляется еще день недели.

Есть еще стиль DEFAULT, совпадающий со стилем MEDIUM.

При создании объекта класса SimpleDateFormat можно задать в конструкторе шаблон, определяющий какой-либо другой формат, например:

```
SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy hh.mm");
System.out.println(sdf.format(new Date()));
```

Получим вывод в таком виде: 27-04-2001 17.32.

В шаблоне буква d означает цифру дня месяца, M — цифру месяца, y — цифру года, h — цифру часа, m — цифру минут. Остальные обозначения для шаблона указаны в документации по классу SimpleDateFormat.

Эти буквенные обозначения можно изменить с помощью класса DateFormatSymbols.

Не во всех локалях можно создать объект класса SimpleDateFormat. В таких случаях используются статические методы getInstance() класса DateFormat, возвращающие объект класса DateFormat. Параметрами этих методов служат стиль представления даты и времени и, может быть, локаль.

После создания объекта метод format() класса DateFormat возвращает строку с датой и временем, согласно заданному стилю. В качестве аргумента задается объект класса Date.

Например:

```
System.out.println("LONG: " + DateFormat.getDateTimeInstance(
    DateFormat.LONG, DateFormat.LONG).format(new Date()));
```

или

```
System.out.println("FULL: " + DateFormat.getDateTimeInstance(
    DateFormat.FULL,DateFormat.FULL, Locale.US).format(new Date()));
```

Получение случайных чисел

Получить случайное неотрицательное число, строго меньшее единицы, в виде типа double можно статическим методом random() из класса java.lang.Math.

При первом обращении к этому методу создается генератор псевдослучайных чисел, который используется потом при получении следующих случайных чисел.

Более серьезные действия со случайными числами можно организовать с помощью методов класса Random из пакета java.util. В классе два конструктора:

- `Random(long seed)` — создает генератор псевдослучайных чисел, использующий для начала работы число `seed`;
- `Random()` — выбирает в качестве начального значения текущее время.

Создав генератор, можно получать случайные числа соответствующего типа методами `nextBoolean()`, `nextDouble()`, `nextFloat()`, `nextGaussian()`, `nextInt()`, `nextLong()`, `nextInt(int max)` или записать сразу последовательность случайных чисел в заранее определенный массив байтов `bytes` методом `nextBytes(byte[] bytes)`.

Вещественные случайные числа равномерно располагаются в диапазоне от 0,0 включительно до 1,0 исключительно. Целые случайные числа равномерно распределяются по всему диапазону соответствующего типа за одним исключением: если в аргументе указано целое число `max`, то диапазон случайных чисел будет от нуля включительно до `max` исключительно.

Копирование массивов

В классе `System` из пакета `java.lang` есть статический метод копирования массивов, который использует сама исполняющая система Java. Этот метод действует быстро и надежно, его удобно применять в программах. Синтаксис:

```
static void arraycopy(Object src, int src_ind,
      Object dest, int dest_ind, int count)
```

Из массива, на который указывает ссылка `src`, копируется `count` элементов, начиная с элемента с индексом `src_ind`, в массив, на который указывает ссылка `dest`, начиная с его элемента с индексом `dest_ind`.

Все индексы должны быть заданы так, чтобы элементы лежали в массивах, типы массивов должны быть совместимы, а примитивные типы обязаны полностью совпадать. Ссылки на массивы не должны быть равны `null`.

Ссылки `src` и `dest` могут совпадать, при этом для копирования создается промежуточный буфер. Метод можно использовать, например, для сдвига элементов в массиве. После выполнения

```
int[] arr = {5, 6, 7, 8, 9, 1, 2, 3, 4, 5, -3, -7};
System.arraycopy(arr, 2, arr, 1, arr.length — 2);
```

получим `{5, 7, 8, 9, 1, 2, 3, 4, 5, -3, -7,-7}`.

Взаимодействие с системой

Класс `System` позволяет осуществить и некоторое взаимодействие с системой во время выполнения программы (run time). Но кроме него для этого есть специальный класс `Runtime`.

Класс `Runtime` содержит некоторые методы взаимодействия с JVM во время выполнения программы. Каждое приложение может получить только один экземпляр данного класса статическим методом `getRuntime()`. Все вызовы этого метода возвращают ссылку на один и тот же объект.

Методы `freeMemory()` и `totalMemory()` возвращают количество свободной и всей памяти, находящейся в распоряжении JVM для размещения объектов, в байтах, в виде числа типа `long`. Не стоит твердо опираться на эти числа, поскольку количество памяти меняется динамически.

Метод `exit(int status)` запускает процесс останова JVM и передает операционной системе статус завершения `status`. По соглашению, ненулевой статус означает ненормальное завершение. Удобнее использовать аналогичный метод класса `System`, который является статическим.

Метод `halt(int status)` осуществляет немедленный останов JVM. Он не завершает запущенные процессы нормально и должен использоваться только в аварийных ситуациях.

Метод `loadLibrary(String libName)` позволяет подгрузить динамическую библиотеку во время выполнения по ее имени `libName`.

Метод `load(String fileName)` подгружает динамическую библиотеку по имени файла `fileName`, в котором она хранится.

Впрочем, вместо этих методов удобнее использовать статические методы класса `System` с теми же именами и аргументами.

Метод `gc()` запускает процесс освобождения ненужной оперативной памяти (garbage collection). Этот процесс периодически запускается самой виртуальной машиной Java и выполняется на фоне с небольшим приоритетом, но можно его запустить и из программы. Опять-таки удобнее использовать статический метод `System.gc()`.

Наконец, несколько методов `exec()` запускают в отдельных процессах исполнимые файлы. Аргументом этих методов служит командная строка исполнимого файла.

Например, `Runtime.getRuntime().exec("notepad")` запускает программу Блокнот на платформе MS Windows.

Методы `exec()` возвращают экземпляр класса `Process`, позволяющего управлять запущенным процессом. Методом `destroy()` можно остановить процесс, методом `exitValue()` получить его код завершения. Метод `waitFor()` приостанавливает основной подпроцесс до тех пор, пока не закончится запущенный процесс. Три метода `getInputStream()`, `getOutputStream()` и `getErrorStream()` возвращают входной, выходной поток и поток ошибок запущенного процесса *(см. главу 18)*.

Часть III

Создание графического интерфейса пользователя и апплетов

Глава 8. Принципы построения графического интерфейса

Глава 9. Графические примитивы

Глава 10. Основные компоненты

Глава 11. Размещение компонентов

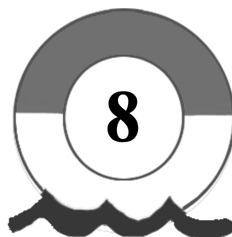
Глава 12. Обработка событий

Глава 13. Создание меню

Глава 14. Апплеты

Глава 15. Изображения и звук

ГЛАВА 8



Принципы построения графического интерфейса

В предыдущих главах мы писали программы, связанные с текстовым терминалом и запускающиеся из командной строки. Такие программы называются *консольными приложениями*. Они разрабатываются для выполнения на серверах, там, где не требуется интерактивная связь с пользователем.

Программы, тесно взаимодействующие с пользователем, воспринимающие сигналы от клавиатуры и мыши, работают в графической среде. Каждое приложение, предназначенное для работы в графической среде, должно создать хотя бы одно окно, в котором будет происходить его работа, и зарегистрировать его в графической оболочке операционной системы, чтобы окно могло взаимодействовать с операционной системой и другими окнами: перекрываться, перемещаться, менять размеры, сворачиваться в ярлык.

Есть много различных графических систем: MS Windows, X Window System, Macintosh. В каждой из них свои правила построения окон и их компонентов: меню, полей ввода, кнопок, списков, полос прокрутки. Эти правила сложны и запутанны. Графические API содержат сотни функций.

Для облегчения создания окон и их компонентов написаны библиотеки классов: MFC, Motif, OpenLook, Qt, Tk, Xview, OpenWindows и множество других. Каждый класс такой библиотеки описывает сразу целый графический компонент, управляемый методами этого и других классов.

В технологии Java дело осложняется тем, что приложения Java должны работать в любой или хотя бы во многих графических средах. Нужна библиотека классов, независимая от конкретной графической системы.

В первой версии JDK задачу решили следующим образом: были разработаны интерфейсы, содержащие методы работы с графическими объектами. Классы библиотеки AWT реализуют эти интерфейсы для создания приложений. Приложения Java используют данные методы для размещения и перемещения графических объектов, изменения их размеров, взаимодействия объектов.

С другой стороны, для работы с экраном в конкретной графической среде эти интерфейсы реализуются в каждой такой среде отдельно. В каждой графической оболочке это делается по-своему, средствами этой оболочки с помощью графических библиотек данной операционной системы. Такие интерфейсы были названы *peer-интерфейсами.*

Библиотека классов Java, основанных на peer-интерфейсах, получила название AWT (Abstract Window Toolkit). При выводе объекта, созданного в приложении Java и основанного на peer-интерфейсе, на экран создается парный ему (peer-to-peer) объект графической подсистемы операционной системы, который и отображается на экране. Эти объекты тесно взаимодействуют во время работы приложения. Поэтому графические объекты AWT в каждой графической среде имеют вид, характерный для этой среды: в MS Windows, Motif, OpenLook, OpenWindows, везде окна, созданные в AWT, выглядят как "родные" окна.

Именно из-за такой реализации peer-интерфейсов и других "родных" методов, написанных, главным образом, на языке C++, приходится для каждой платформы выпускать свой вариант JDK.

В версии JDK 1.1 библиотека AWT была переработана. В нее добавлена возможность создания компонентов, полностью написанных на Java и не зависящих от peer-интерфейсов. Такие компоненты стали называть *"легкими"* (lightweight) в отличие от компонентов, реализованных через peer-интерфейсы, названных *"тяжелыми"* (heavy).

"Легкие" компоненты везде выглядят одинаково, сохраняют заданный при создании вид (look and feel). Более того, приложение можно разработать таким образом, чтобы после его запуска можно было выбрать какой-то определенный вид: Motif, Metal, Windows 95 или какой-нибудь другой, и сменить этот вид в любой момент работы.

Эта интересная особенность "легких" компонентов получила название PL&F (Pluggable Look and Feel) или "plaf".

Была создана обширная библиотека "легких" компонентов Java, названная Swing. В ней были переписаны все компоненты библиотеки AWT, так что библиотека Swing может использоваться самостоятельно, несмотря на то, что все классы из нее расширяют классы библиотеки AWT.

Библиотека классов Swing поставлялась как дополнение к JDK 1.1. В состав Java 2 SDK она включена как основная графическая библиотека классов, реализующая идею "100% Pure Java", наряду с AWT.

В Java 2 библиотека AWT значительно расширена добавлением новых средств рисования, вывода текстов и изображений, получивших название Java 2D, и средств, реализующих перемещение текста методом DnD (Drag and Drop).

Кроме того, в Java 2 включены новые методы ввода/вывода Input Method Framework и средства связи с дополнительными устройствами ввода/вывода, такими как световое перо или клавиатура Бройля, названные Accessibility.

Все эти средства Java 2: AWT, Swing, Java 2D, DnD, Input Method Framework и Accessibility составили библиотеку графических средств Java, названную JFC (Java Foundation Classes).

Описание каждого из этих средств составит целую книгу, поэтому мы вынуждены ограничиться представлением только основных средств библиотеки AWT.

Компонент и контейнер

Основное понятие графического интерфейса пользователя (ГИП) — *компонент* (component) графической системы. В русском языке это слово подразумевает просто составную часть, элемент чего-нибудь, но в графическом интерфейсе это понятие гораздо конкретнее. Оно означает отдельный, полностью определенный элемент, который можно использовать в графическом интерфейсе независимо от других элементов. Например, это поле ввода, кнопка, строка меню, полоса прокрутки, радиокнопка. Само окно приложения — тоже его компонент. Компоненты могут быть и невидимыми, например, панель, объединяющая компоненты, тоже является компонентом.

Вы не удивитесь, узнав, что в AWT компонентом считается объект класса `Component` или объект всякого класса, расширяющего класс `Component`. В классе `Component` собраны общие методы работы с любым компонентом графического интерфейса пользователя. Этот класс — центр библиотеки AWT.

Каждый компонент перед выводом на экран помещается в *контейнер* (container). Контейнер "знает", как разместить компоненты на экране. Разумеется, в языке Java контейнер — это объект класса `Container` или всякого его расширения. Прямой наследник этого класса — класс `JComponent` — вершина иерархии многих классов библиотеки Swing.

Создав компонент — объект класса `Component` или его расширения, следует добавить его к предварительно созданному объекту класса `Container` или его расширения одним из методов `add()`.

Класс `Container` сам является невидимым компонентом, он расширяет класс `Component`. Таким образом, в контейнер наряду с компонентами можно помещать контейнеры, в которых находятся какие-то другие компоненты, достигая тем самым большой гибкости расположения компонентов.

Основное окно приложения, активно взаимодействующее с операционной системой, необходимо построить по правилам графической системы. Оно должно перемещаться по экрану, изменять размеры, реагировать на дейст-

вия мыши и клавиатуры. В окне должны быть, как минимум, следующие стандартные компоненты.

- *Строка заголовка* (title bar), с левой стороны которой необходимо разместить кнопку контекстного меню, а с правой — кнопки сворачивания и разворачивания окна и кнопку закрытия приложения.
- Необязательная *строка меню* (menu bar) с выпадающими пунктами меню.
- Горизонтальная и вертикальная *полосы прокрутки* (scrollbars).
- Окно должно быть окружено *рамкой* (border), реагирующей на действия мыши.

Окно с этими компонентами в готовом виде описано в классе `Frame`. Чтобы создать окно, достаточно сделать свой класс расширением класса `Frame`, как показано в листинге 8.1. Всего восемь строк текста и окно готово.

Листинг 8.1. Слишком простое окно приложения

```
import java.awt.*;

class TooSimpleFrame extends Frame{
  public static void main(String[] args){
    Frame fr = new TooSimpleFrame();
    fr.setSize(400, 150);
    fr.setVisible(true);
  }
}
```

Класс `TooSimpleFrame` обладает всеми свойствами класса `Frame`, являясь его расширением. В нем создается экземпляр окна `fr`, и устанавливаются размеры окна на экране — 400×150 пикселов — методом `setSize()`. Если не задать размер окна, то на экране появится окно минимального размера — только строка заголовка. Конечно, потом его можно растянуть с помощью мыши до любого размера.

Затем окно выводится на экран методом `setVisible(true)`. Дело в том, что, с точки зрения библиотеки AWT, создать окно значит выделить область оперативной памяти, заполненную нужными пикселами, а вывести содержимое этой области на экран — уже другая задача, которую и решает метод `setVisible(true)`.

Конечно, такое окно непригодно для работы. Не говоря уже о том, что у него нет заголовка и поэтому окно нельзя закрыть. Хотя его можно перемещать по экрану, менять размеры, сворачивать на панель задач и раскрывать, но команду завершения приложения мы не запрограммировали. Окно нельзя закрыть ни щелчком кнопки мыши на кнопке с крестиком в правом верхнем углу окна, ни комбинацией клавиш <Alt>+<F4>. Приходится за-

вершать работу приложения средствами операционной системы, например, комбинацией клавиш <Ctrl>+<C>.

В листинге 8.2 к программе листинга 8.1 добавлены заголовок окна и обращение к методу, позволяющему завершить приложение.

Листинг 8.2. Простое окно приложения

```
import java.awt.*;
import java.awt.event.*;

class SimpleFrame extends Frame{
  SimpleFrame(String s){
    super(s);
    setSize(400, 150);
    setVisible(true);
    addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
  public static void main(String[] args){
    new SimpleFrame("  Моя программа");
  }
}
```

В программу добавлен конструктор класса `SimpleFrame`, обращающийся к конструктору своего суперкласса `Frame`, который записывает свой аргумент `s` в строку заголовка окна.

В конструктор перенесена установка размеров окна, вывод его на экран и добавлено обращение к методу `addWindowListener()`, реагирующему на действия с окном. В качестве аргумента этому методу передается экземпляр безымянного внутреннего класса, расширяющего класс `WindowAdapter`. Этот безымянный класс реализует метод `windowClosing()`, обрабатывающий попытку закрытия окна. Данная реализация очень проста — приложение завершается статическим методом `exit()` класса `System`. Окно при этом закрывается автоматически.

Все это мы подробно разберем в *главе 12*, а пока просто добавляйте эти строчки во все ваши программы для закрытия окна и завершения работы приложения.

Итак, окно готово. Но оно пока пусто. Выведем в него, по традиции, приветствие "Hello, World!", правда, слегка измененное. В листинге 8.3 приведена полная программа этого вывода, а рис. 8.1 демонстрирует окно.

Рис. 8.1. Окно программы-приветствия

Листинг 8.3. Графическая программа с приветствием

```
import java.awt.*;
import java.awt.event.*;

class HelloWorldFrame extends Frame{
  HelloWorldFrame(String s){
    super(s);
  }
  public void paint(Graphics g){
    g.setFont(new Font("Serif", Font.ITALIC|Font.BOLD, 30));
    g.drawString("Hello, XXI century World!", 20, 100);
  }
  public static void main(String[] args){
    Frame f = new HelloWorldFrame("Здравствуй, мир XXI века!");
    f.setSize(400, 150);
    f.setVisible(true);
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

Для вывода текста мы переопределяем метод `paint()` класса `Component`. Класс `Frame` наследует этот метод с пустой реализацией.

Метод `paint()` получает в качестве аргумента экземпляр `g` класса `Graphics`, умеющего, в частности, выводить текст методом `drawString()`. В этом методе кроме текста мы указываем положение начала строки в окне — 20 пикселов от левого края и 100 пикселов сверху. Эта точка — левая нижняя точка первой буквы текста `H`.

Кроме того, мы установили новый шрифт `"Serif"` большего размера — 30 пунктов, полужирный, курсив. Всякий шрифт — объект класса `Font`, а задается он методом `setFont()` класса `Graphics`.

Работу со шрифтами мы рассмотрим в следующей главе.

В листинге 8.3, для разнообразия, мы вынесли вызовы методов установки размеров окна, вывода его на экран и завершения программы в метод `main()`.

Как вы видите из этого простого примера, библиотека AWT большая и разветвленная, в ней множество классов, взаимодействующих друг с другом. Рассмотрим иерархию некоторых наиболее часто используемых классов AWT.

Иерархия классов AWT

На рис. 8.2 показана иерархия основных классов AWT. Основу ее составляют готовые компоненты: `Button`, `Canvas`, `Checkbox`, `Choice`, `Container`, `Label`, `List`, `Scrollbar`, `TextArea`, `TextField`, `Menubar`, `Menu`, `PopupMenu`, `MenuItem`, `CheckboxMenuItem`. Если этого набора не хватает, то от класса `Canvas` можно породить собственные "тяжелые" компоненты, а от класса `Component` — "легкие" компоненты.

Основные контейнеры — это классы `Panel`, `ScrollPane`, `Window`, `Frame`, `Dialog`, `FileDialog`. Свои "тяжелые" контейнеры можно породить от класса `Panel`, а "легкие" — от класса `Container`.

Целый набор классов помогает размещать компоненты, задавать цвет, шрифт, рисунки и изображения, реагировать на сигналы от мыши и клавиатуры.

На рис. 8.2 показаны и начальные классы иерархии библиотеки Swing — классы `JComponent`, `JWindow`, `JFrame`, `JDialog`, `JApplet`.

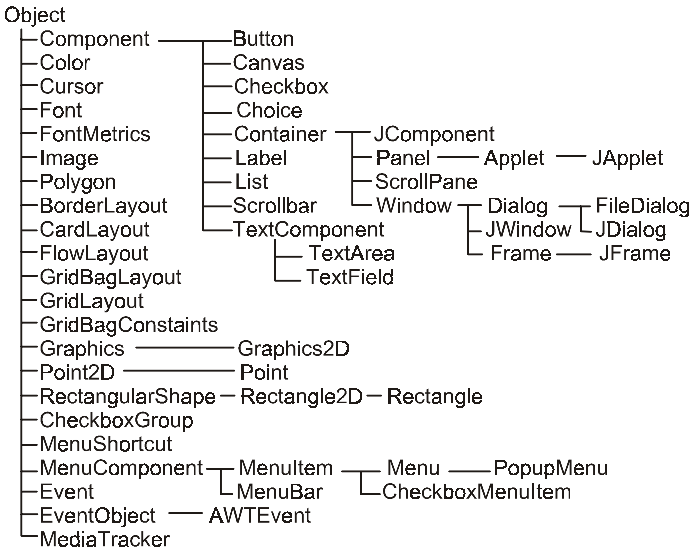


Рис. 8.2. Иерархия основных классов AWT

Заключение

Как видите, библиотека графических классов AWT очень велика и детально проработана. Это многообразие классов только отражает многообразие задач построения графического интерфейса. Стремление улучшить интерфейс безгранично. Оно приводит к созданию все новых библиотек классов и расширению существующих. Независимыми производителями создано уже много графических библиотек Java: KL Group, JBCL, и появляются все новые и новые библиотеки. Сведения о них можно получить на сайтах, указанных во *введении*.

В следующих главах мы подробно рассмотрим, как можно использовать библиотеку AWT для создания собственных приложений с графическим интерфейсом пользователя, изображениями, анимацией и звуком.

ГЛАВА 9

Графические примитивы

При создании компонента, т. е. объекта класса `Component`, автоматически формируется его *графический контекст* (graphics context). В контексте размещается область рисования и вывода текста и изображений. Контекст содержит текущий и альтернативный цвет рисования и цвет фона — объекты класса `Color`, текущий шрифт для вывода текста — объект класса `Font`.

В контексте определена система координат, начало которой с координатами (0, 0) расположено в верхнем левом углу области рисования, ось Ох направлена вправо, ось Оу — вниз. Точки координат находятся между пикселами.

Управляет контекстом класс `Graphics` или новый класс `Graphics2D`, введенный в Java 2. Поскольку графический контекст сильно зависит от конкретной графической платформы, эти классы сделаны абстрактными. Поэтому нельзя непосредственно создать экземпляры класса `Graphics` или `Graphics2D`.

Однако каждая виртуальная машина Java реализует методы этих классов, создает их экземпляры для компонента и предоставляет объект класса `Graphics` методом `getGraphics()` класса `Component` или как аргумент методов `paint()` и `update()`.

Посмотрим сначала, какие методы работы с графикой и текстом предоставляет нам класс `Graphics`.

Методы класса *Graphics*

При создании контекста в нем задается текущий цвет для рисования, обычно черный, и цвет фона области рисования — белый или серый. Изменить текущий цвет можно методом `setColor(Color newColor)`, аргумент `newColor` которого — объект класса `Color`.

Узнать текущий цвет можно методом `getColor()`, возвращающим объект класса `Color`.

Как задать цвет

Цвет, как и все в Java, — объект определенного класса, а именно, класса `Color`. Основу класса составляют семь конструкторов цвета. Самый простой конструктор:

```
Color(int red, int green, int blue)
```

создает цвет, получающийся как смесь красной `red`, зеленой `green` и синей `blue` составляющих. Эта цветовая модель называется RGB. Каждая составляющая меняется от 0 (отсутствие составляющей) до 255 (полная интенсивность этой составляющей). Например:

```
Color pureRed = new Color(255, 0, 0);
Color pureGreen = new Color(0, 255, 0);
```

определяют чистый ярко-красный `pureRed` и чистый ярко-зеленый `pureGreen` цвета.

Во втором конструкторе интенсивность составляющих можно изменять более гладко вещественными числами от 0.0 (отсутствие составляющей) до 1.0 (полная интенсивность составляющей):

```
Color(float red, float green, float blue)
```

Например:

```
Color someColor = new Color(0.05f, 0.4f, 0.95f);
```

Третий конструктор

```
Color(int rgb)
```

задает все три составляющие в одном целом числе. В битах 16—23 записывается красная составляющая, в битах 8—15 — зеленая, а в битах 0—7 — синяя составляющая цвета. Например:

```
Color c = new Color(0xFF8F48FF);
```

Здесь красная составляющая задана с интенсивностью `0x8F`, зеленая — `0x48`, синяя — `0xFF`.

Следующие три конструктора

```
Color(int red, int green, int blue, int alpha)
Color(float red, float green, float blue, float alpha)
Color(int rgb, boolean hasAlpha)
```

вводят четвертую составляющую цвета, так называемую "альфу", определяющую прозрачность цвета. Эта составляющая проявляет себя при наложении одного цвета на другой. Если альфа равна 255 или 1,0, то цвет совершенно непрозрачен, предыдущий цвет не просвечивает сквозь него. Если альфа равна 0 или 0,0, то цвет абсолютно прозрачен, для каждого пиксела виден только предыдущий цвет.

Последний из этих конструкторов учитывает составляющую альфа, находящуюся в битах 24—31, если параметр `hasAlpha` равен `true`. Если же `hasAlpha` равно `false`, то составляющая альфа считается равной 255, независимо от того, что записано в старших битах параметра `rgb`.

Первые три конструктора создают непрозрачный цвет с альфой, равной 255 или 1,0.

Седьмой конструктор

```
Color(ColorSpace cspace, float[] components, float alpha)
```

позволяет создавать цвет не только в цветовой модели (color model) RGB, но и в других моделях: CMYK, HSB, CIEXYZ, определенных объектом класса `ColorSpace`.

Для создания цвета в модели HSB можно воспользоваться статическим методом

```
getHSBColor(float hue, float saturation, float brightness).
```

Если нет необходимости тщательно подбирать цвета, то можно просто воспользоваться одной из тринадцати статических констант типа `Color`, имеющихся в классе `Color`. Вопреки соглашению "Code Conventions" они записываются строчными буквами: `black`, `blue`, `cyan`, `darkGray`, `gray`, `green`, `lightGray`, `magenta`, `orange`, `pink`, `red`, `white`, `yellow`.

Методы класса `Color` позволяют получить составляющие текущего цвета: `getRed()`, `getGreen()`, `getBlue()`, `getAlpha()`, `getRGB()`, `getColorSpace()`, `getComponents()`.

Два метода создают более яркий `brighter()` и более темный `darker()` цвета по сравнению с текущим цветом. Они полезны, если надо выделить активный компонент или, наоборот, показать неактивный компонент бледнее остальных компонентов.

Два статических метода возвращают цвет, преобразованный из цветовой модели RGB в HSB и обратно:

```
float[] RGBtoHSB(int red, int green, int blue, float[] hsb)
int HSBtoRGB(int hue, int saturation, int brightness)
```

Создав цвет, можно рисовать им в графическом контексте.

Как нарисовать чертеж

Основной метод рисования

```
drawLine(int x1, int y1, int x2, int y2)
```

вычерчивает текущим цветом отрезок прямой между точками с координатами `(x1, y1)` и `(x2, y2)`.

Одного этого метода достаточно, чтобы нарисовать любую картину по точкам, вычерчивая каждую точку с координатами `(x, y)` методом `drawLine(x, y, x, y)` и меняя цвета от точки к точке. Но никто, разумеется, не станет этого делать.

Другие графические примитивы:

- `drawRect(int x, int y, int width, int height)` — чертит прямоугольник со сторонами, параллельными краям экрана, задаваемый координатами верхнего левого угла `(x, y)`, шириной `width` пикселов и высотой `height` пикселов;
- `draw3DRect(int x, int y, int width, int height, boolean raised)` — чертит прямоугольник, как будто выделяющийся из плоскости рисования, если аргумент `raised` равен `true`, или как будто вдавленный в плоскость, если аргумент `raised` равен `false`;
- `drawOval(int x, int y, int width, int height)` — чертит овал, вписанный в прямоугольник, заданный аргументами метода. Если `width == height`, то получится окружность;
- `drawArc(int x, int y, int width, int height, int startAngle, int arc)` — чертит дугу овала, вписанного в прямоугольник, заданный первыми четырьмя аргументами. Дуга имеет величину `arc` градусов и отсчитывается от угла `startAngle`. Угол отсчитывается в градусах от оси Ох. Положительный угол отсчитывается против часовой стрелки, отрицательный — по часовой стрелке;
- `drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)` — чертит прямоугольник с закругленными краями. Закругления вычерчиваются четвертинками овалов, вписанных в прямоугольники шириной `arcWidth` и высотой `arcHeight`, построенные в углах основного прямоугольника;
- `drawPolyline(int[] xPoints, int[] yPoints, int nPoints)` — чертит ломаную с вершинами в точках `(xPoints[i], yPoints[i])` и числом вершин `nPoints`;
- `drawPolygon(int[] xPoints, int[] yPoints, int nPoints)` — чертит замкнутую ломаную, проводя замыкающий отрезок прямой между первой и последней точкой;
- `drawPolygon(Polygon p)` — чертит замкнутую ломаную, вершины которой заданы объектом `p` класса `Polygon`.

Класс `Polygon` рассмотрим подробнее.

Класс *Polygon*

Этот класс предназначен для работы с многоугольником, в частности, с треугольниками и произвольными четырехугольниками.

Объекты этого класса можно создать двумя конструкторами:

- Polygon() — создает пустой объект;
- Polygon(int[] xPoints, int[] yPoints, int nPoints) — задаются вершины многоугольника (xPoints[i], yPoints[i]) и их число nPoints.

После создания объекта в него можно добавлять вершины методом

```
addPoint(int x, int y)
```

Логические методы contains() позволяют проверить, не лежит ли в многоугольнике заданная аргументами метода точка, отрезок прямой или целый прямоугольник со сторонами, параллельными сторонам экрана.

Логические методы intersects() позволяют проверить, не пересекается ли с данным многоугольником отрезок прямой, заданный аргументами метода, или прямоугольник со сторонами, параллельными сторонам экрана.

Методы getBounds() и getBounds2D() возвращают прямоугольник, целиком содержащий в себе данный многоугольник.

Вернемся к методам класса Graphics. Несколько методов вычерчивают фигуры, залитые текущим цветом: fillRect(), fill3DRect(), fillArc(), fillOval(), fillPolygon(), fillRoundRect(). У них такие же аргументы, как и у соответствующих методов, вычерчивающих незаполненные фигуры.

Например, если вы хотите изменить цвет фона области рисования, то установите новый текущий цвет и начертите им заполненный прямоугольник величиной во всю область:

```
public void paint(Graphics g){
  Color initColor = g.getColor();       // Сохраняем исходный цвет
  g.setColor(new Color(0, 0, 255));     // Устанавливаем цвет фона
  // Заливаем область рисования
  g.fillRect(0, 0, getSize().width — 1, getSize().height — 1);
  g.setColor(initColor);                // Восстанавливаем исходный цвет
  // Дальнейшие действия
}
```

Как видите, в классе Graphics собраны только самые необходимые средства рисования. Нет даже метода, задающего цвет фона (хотя можно задать цвет фона компонента методом setBackground() класса Component). Средства рисования, вывода текста в область рисования и вывода изображений значительно дополнены и расширены в подклассе Graphics2D, входящем в систему Java 2D. Например, в нем есть метод задания цвета фона setBackground(Color c).

Перед тем как обратиться к классу Graphics2D, рассмотрим средства класса Graphics для вывода текста.

Как вывести текст

Для вывода текста в область рисования текущим цветом и шрифтом, начиная с точки `(x, y)`, в классе `Graphics` есть несколько методов:

- `drawString(String s, int x, int y)` — выводит строку `s`;
- `drawBytes(byte[] b, int offset, int length, int x, int y)` — выводит `length` элементов массива байтов `b`, начиная с индекса `offset`;
- `drawChars(char[] ch, int offset, int length, int x, int y)` — выводит `length` элементов массива символов `ch`, начиная с индекса `offset`.

Четвертый метод выводит текст, занесенный в объект класса, реализующего интерфейс `AttributedCharacterIterator`. Это позволяет задавать свой шрифт для каждого выводимого символа:

```
drawString(AttributedCharacterIterator iter, int x, int y)
```

Точка `(x, y)` — это левая нижняя точка первой буквы текста на базовой линии (baseline) вывода шрифта.

Как установить шрифт

Метод `setFont(Font newFont)` класса `Graphics` устанавливает текущий шрифт для вывода текста.

Метод `getFont()` возвращает текущий шрифт.

Как и все в языке Java, шрифт — это объект класса `Font`. Посмотрим, какие возможности предоставляет этот класс.

Как задать шрифт

Объекты класса `Font` хранят начертания (glyphs) символов, образующие шрифт. Их можно создать двумя конструкторами:

- `Font(Map attributes)` — задает шрифт с заданными аргументом `attributes` атрибутами. Ключи атрибутов и некоторые их значения задаются константами класса `TextAttribute` из пакета `java.awt.font`. Этот конструктор характерен для Java 2D и будет рассмотрен далее в настоящей главе.
- `Font(String name, int style, int size)` — задает шрифт по имени `name`, со стилем `style` и размером `size` типографских пунктов. Этот конструктор характерен для JDK 1.1, но широко используется и в Java 2D в силу своей простоты.

Типографский пункт в России и некоторых европейских странах равен 0,376 мм, точнее, 1/72 части французского дюйма. В англо-американской системе мер пункт равен 1/72 части английского дюйма, 0,351 мм. Этот-то пункт и применяется в компьютерной графике.

Имя шрифта `name` может быть строкой с физическим именем шрифта, например, `"Courier New"`, или одна из строк `"Dialog"`, `"DialogInput"`, `"Monospaced"`, `"Serif"`, `"SansSerif"`, `"Symbol"`. Это так называемые *логические имена шрифтов* (logical font names). Если `name == null`, то задается шрифт по умолчанию.

Стиль шрифта `style` — это одна из констант класса `Font`:

- `BOLD` — полужирный;
- `ITALIC` — курсив;
- `PLAIN` — обычный.

Полужирный курсив (bolditalic) можно задать операцией побитового сложения, `Font.BOLD|Font.ITALIC`, как это сделано в листинге 8.3.

При выводе текста логическим именам шрифтов и стилям сопоставляются *физические имена шрифтов* (font face name) или *имена семейств шрифтов* (font name). Это имена реальных шрифтов, имеющихся в графической подсистеме операционной системы.

Например, логическому имени `"Serif"` может быть сопоставлено имя семейства (family) шрифтов Times New Roman, а в сочетании со стилями — конкретные физические имена Times New Roman Bold, Times New Roman Italic. Эти шрифты должны находиться в составе шрифтов графической системы той машины, на которой выполняется приложение.

Список имен доступных шрифтов можно просмотреть следующими операторами:

```
Font[] fnt = Toolkit.getGraphicsEnvironment.getAllFonts();
for (int i = 0; i< fnt.length; i++)
    System.out.println(fnt[i].getFontName());
```

В состав SUN J2SDK входит семейство шрифтов Lucida. Установив SDK, вы можете быть уверены, что эти шрифты есть в вашей системе.

Таблицы сопоставления логических и физических имен шрифтов находятся в файлах с именами

- font.properties;
- font.properties.ar;
- font.properties.ja;
- font.properties.ru.

и т. д. Эти файлы должны быть расположены в JDK в каталоге jdk1.3\jre\lib или каком-либо другом подкаталоге lib корневого каталога JDK той машины, на которой выполняется приложение.

Нужный файл выбирается виртуальной машиной Java по окончании имени файла. Это окончание совпадает с международным кодом языка, установ-

ленного в локали или в системном свойстве user.language (см. рис. 6.2). Если у вас установлена русская локаль с международным кодом языка "ru", то для сопоставления будет выбран файл font.properties.ru. Если такой файл не найден, то применяется файл font.properties, не соответствующий никакой конкретной локали.

Поэтому можно оставить в системе только один файл font.properties, переписав в него содержимое нужного файла или создав файл заново. Для любой локали будет использоваться этот файл.

В листинге 9.1 показано сокращенное содержимое файла font.properties.ru из JDK 1.3 для платформы MS Windows.

Листинг 9.1. Примерный файл font.properties.ru

```
# %W% %E%
# Это просто комментарии
# AWT Font default Properties for Russian Windows
#
# Три сопоставления логическому имени "Dialog":
dialog.0=Arial,RUSSIAN_CHARSET
dialog.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
dialog.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
# По три сопоставления стилям ITALIC, BOLD, ITALIC+BOLD:
dialog.italic.0=Arial Italic,RUSSIAN_CHARSET
dialog.italic.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
dialog.italic.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED

dialog.bold.0=Arial Bold,RUSSIAN_CHARSET
dialog.bold.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
dialog.bold.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED

dialog.bolditalic.0=Arial Bold Italic,RUSSIAN_CHARSET
dialog.bolditalic.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
dialog.bolditalic.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
# По три сопоставления имени "DialogInput" и стилям:
dialoginput.0=Courier New,RUSSIAN_CHARSET
dialoginput.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
dialoginput.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED

dialoginput.italic.0=Courier New Italic,RUSSIAN_CHARSET
# И так далее
#
# По три сопоставления имени "Serif" и стилям:
serif.0=Times New Roman,RUSSIAN_CHARSET
serif.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
serif.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
```

```
serif.italic.0=Times New Roman Italic,RUSSIAN_CHARSET
# И так далее
# Прочие логические имена
sansserif.0=Arial,RUSSIAN_CHARSET
sansserif.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
sansserif.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED

sansserif.italic.0=Arial Italic,RUSSIAN_CHARSET
# И так далее
#
monospaced.0=Courier New,RUSSIAN_CHARSET
monospaced.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
monospaced.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED

monospaced.italic.0=Courier New Italic,RUSSIAN_CHARSET
# И так далее

# Default font definition
#
default.char=2751

# for backword compatibility
# Старые логические имена версии JDK 1.0
timesroman.0=Times New Roman,RUSSIAN_CHARSET
helvetica.0=Arial,RUSSIAN_CHARSET
courier.0=Courier New,RUSSIAN_CHARSET
zapfdingbats.0=WingDings,SYMBOL_CHARSET

# font filenames for reduced initialization time
# Файлы со шрифтами
filename.Arial=ARIAL.TTF
filename.Arial_Bold=ARIALBD.TTF
filename.Arial_Italic=ARIALI.TTF
filename.Arial_Bold_Italic=ARIALBI.TTF
filename.Courier_New=COUR.TTF
filename.Courier_New_Bold=COURBD.TTF
filename.Courier_New_Italic=COURI.TTF
filename.Courier_New_Bold_Italic=COURBI.TTF
filename.Times_New_Roman=TIMES.TTF
filename.Times_New_Roman_Bold=TIMESBD.TTF
filename.Times_New_Roman_Italic=TIMESI.TTF
filename.Times_New_Roman_Bold_Italic=TIMESBI.TTF
filename.WingDings=WINGDING.TTF
filename.Symbol=SYMBOL.TTF
# name aliases
# Псевдонимы логических имен закомментированы
# alias.timesroman=serif
```

```
# alias.helvetica=sansserif
# alias.courier=monospaced
# Static FontCharset info.
#
# Классы преобразования символов в байты
fontcharset.dialog.0=sun.io.CharToByteCP1251
fontcharset.dialog.1=sun.awt.windows.CharToByteWingDings
fontcharset.dialog.2=sun.awt.CharToByteSymbol

fontcharset.dialoginput.0=sun.io.CharToByteCP1251
fontcharset.dialoginput.1=sun.awt.windows.CharToByteWingDings
fontcharset.dialoginput.2=sun.awt.CharToByteSymbol

fontcharset.serif.0=sun.io.CharToByteCP1251
fontcharset.serif.1=sun.awt.windows.CharToByteWingDings
fontcharset.serif.2=sun.awt.CharToByteSymbol

fontcharset.sansserif.0=sun.io.CharToByteCP1251
fontcharset.sansserif.1=sun.awt.windows.CharToByteWingDings
fontcharset.sansserif.2=sun.awt.CharToByteSymbol

fontcharset.monospaced.0=sun.io.CharToByteCP1251
fontcharset.monospaced.1=sun.awt.windows.CharToByteWingDings
fontcharset.monospaced.2=sun.awt.CharToByteSymbol
# Exclusion Range info.
#
# Не просматривать в этом шрифте указанные диапазоны
exclusion.dialog.0=0100-0400,0460-ffff
exclusion.dialoginput.0=0100-0400,0460-ffff
exclusion.serif.0=0100-0400,0460-ffff
exclusion.sansserif.0=0100-0400,0460-ffff
exclusion.monospaced.0=0100-0400,0460-ffff
# charset for text input
#
# Вводимые байтовые символы кодируются в кириллический диапазон
# кодировки Unicode
inputtextcharset=RUSSIAN_CHARSET
```

Большая часть этого файла занята сопоставлениями логических и физических имен. Вы видите, что под номером 0:

- логическому имени `"Dialog"` сопоставлено имя семейства Arial;
- логическому имени `"DialogInput"` сопоставлено имя семейства Courier New;
- логическому имени `"Serif"` сопоставлено имя семейства Times New Roman;
- логическому имени `"SansSerif"` сопоставлено имя семейства Arial;
- логическому имени `"Monospaced"` сопоставлено имя семейства Courier New.

Там, где указан стиль: `dialog.italic`, `dialog.bold` и т. д., подставлен соответствующий физический шрифт.

В строках листинга 9.1, начинающихся со слова `filename`, указаны файлы с соответствующими физическими шрифтами, например:

```
filename.Arial=ARIAL.TTF
```

Эти строки необязательны, но они ускоряют поиск файлов со шрифтами.

Теперь посмотрите на последние строки листинга 9.1. Строка

```
exclusion.dialog.0=0100-0400,0460-ffff
```

означает, что в шрифте, сопоставленном логическому имени `"Dialog"` под номером 0, а именно, Arial, не станут отыскиваться начертания (glyphs) символов с кодами в диапазонах `'\u0100'`—`'\u0400'` и `'\u0460'`—`'\uFFFF'`. Они будут взяты из шрифта, сопоставленного этому имени под номером 1, а именно, WingDings.

То же самое будет происходить, если нужные начертания не найдены в шрифте, сопоставленному логическому имени под номером 0. Не все файлы со шрифтами Unicode содержат начертания (glyphs) всех символов.

Если нужные начертания не найдены и в сопоставлении 1 (в данном примере в шрифте WingDings), они будут отыскиваться в сопоставлении 2 (т. е. в шрифте Symbol) и т. д. Подобных сопоставлений можно написать сколько угодно.

Таким образом, каждому логическому имени шрифта можно сопоставить разные диапазоны различных реальных шрифтов, а также застраховаться от отсутствия начертаний некоторых символов в шрифтах Unicode.

Все сопоставления под номерами 0, 1, 2, 3, 4 следует повторить для всех стилей: bold, italic, bolditalic.

Если в графической системе используются шрифты Unicode, как, например, в MS Windows NT/2000, то больше ни о чем беспокоиться не надо.

Если же графическая система использует байтовые ASCII-шрифты как, например, MS Windows 95/98/ME, то следует позаботиться об их правильной перекодировке в Unicode и обратно.

Для этого на платформе MS Windows используются константы Win32 API `RUSSIAN_CHARSET`, `SYMBOL_CHARSET`, `ANSI_CHARSET`, `OEM_CHARSET` и др., показывающие, какую кодовую таблицу использовать при перекодировке, так же, как это отмечалось в *главе 5* при создании строки из массива байтов.

Если логическим именам сопоставлены байтовые ASCII-шрифты (в примере это шрифты WingDings и Symbol), то необходимость перекодировки отмечается константой `NEED_CONVERTED`.

Перекодировкой занимаются методы специальных классов `CharToByteCP1251`, `CharToByteWingDings`, `CharToByteSymbol`. Они указываются для каждого сопос-

тавления имен в строках, начинающихся со слова `fontcharset`. Эти строки обязательны для всех шрифтов, помеченных константой `NEED_CONVERTED`.

В последней строке файла указана кодовая страница для перекодировки в Unicode символов, вводимых в поля ввода:

```
inputtextcharset = RUSSIAN_CHARSET
```

Эта запись задает кодовую таблицу CP1251.

Итак, собираясь выводить строку `str` в графический контекст методом `drawString()`, мы создаем текущий шрифт конструктором класса `Font`, указывая в нем логическое имя шрифта, например, `"Serif"`. Исполняющая система Java отыскивает в файле font.properties, соответствующем локальному языку, сопоставленный этому логическому имени физический шрифт операционной системы, например, Times New Roman. Если это Unicode-шрифт, то из него извлекаются начертания символов строки `str` по их кодировке Unicode и отображаются в графический контекст. Если это байтовый ASCII-шрифт, то строка `str` предварительно перекодируется в массив байтов методами класса, указанного в одной из строк `fontcharset`, например, `CharToByteCP1251`.

Хорошие примеры файлов font.properties.ru собраны на странице Сергея Астахова, указанной во введении.

Обсуждение этих вопросов и примеры файлов font.properties для X Window System даны в документации SUN J2SDK в файле docs/guide/intl /fontprop.html.

Завершая обсуждение логических и физических имен шрифтов, следует сказать, что в JDK 1.0 использовались логические имена `"Helvetica"`, `"TimesRoman"`, `"Courier"`, замененные в JDK 1.1 на `"SansSerif"`, `"Serif"`, `"Monospaced"`, соответственно, из лицензионных соображений. Старые имена остались в файлах font.properties для совместимости.

При выводе строки в окно приложения очень часто возникает необходимость расположить ее определенным образом относительно других элементов изображения: центрировать, вывести над или под другим графическим объектом. Для этого надо знать метрику строки: ее высоту и ширину. Для измерения размеров отдельных символов и строки в целом разработан класс `FontMetrics`.

В Java 2D класс `FontMetrics` заменен классом `TextLayout`. Его мы рассмотрим в конце этой главы, а сейчас выясним, какую пользу можно извлечь из методов класса `FontMetrics`.

Класс *FontMetrics*

Класс `FontMetrics` является абстрактным, поэтому нельзя воспользоваться его конструктором. Для получения объекта класса `FontMetrics`, содержащего

набор метрических характеристик шрифта `f`, надо обратиться к методу `getFontMetrics(f)` класса `Graphics` или класса `Component`.

Подробно с характеристиками компьютерных шрифтов можно познакомиться по книге [12].

Класс `FontMetrics` позволяет узнать ширину отдельного символа `ch` в пикселах методом `charWidth(ch)`, общую ширину всех символов массива или подмассива символов или байтов методами `getChars()` и `getBytes()`, ширину целой строки `str` в пикселах методом `stringWidth(str)`.

Несколько методов возвращают в пикселах вертикальные размеры шрифта.

Интерлиньяж (leading) — расстояние между нижней точкой свисающих элементов таких букв, как `p`, `y` и верхней точкой выступающих элементов таких букв, как `б`, `й`, в следующей строке — возвращает метод `getLeading()`.

Среднее расстояние от базовой линии шрифта до верхней точки прописных букв и выступающих элементов той же строки (ascent) возвращает метод `getAscent()`, а максимальное — метод `getMaxAscent()`.

Среднее расстояние свисающих элементов от базовой линии той же строки (descent) возвращает метод `getDescent()`, а максимальное — метод `getMaxDescent()`.

Наконец, высоту шрифта (height) — сумму ascent + descent + leading — возвращает метод `getHeight()`. Высота шрифта равна расстоянию между базовыми линиями соседних строк.

Эти элементы показаны на рис. 9.1.

Рис. 9.1. Элементы шрифта

Дополнительные характеристики шрифта можно определить методами класса `LineMetrics` из пакета `java.awt.font`. Объект этого класса можно получить несколькими методами `getLineMetrics()` класса `FontMetrics`.

Листинг 9.2 показывает применение графических примитивов и шрифтов, а рис. 9.2 — результат выполнения программы из этого листинга.

Листинг 9.2. Использование графических примитивов и шрифтов

```
import java.awt.*;
import java.awt.event.*;
```

```
class GraphTest extends Frame{
  GraphTest(String s){
    super(s);
    setBounds(0, 0, 500, 300);
    setVisible(true);
  }
  public void paint(Graphics g){
    Dimension d = getSize();
    int dx = d.width / 20, dy = d.height / 20;
    g.drawRect(dx, dy + 20,
        d.width — 2 * dx, d.height — 2 * dy — 20);
    g.drawRoundRect(2 * dx, 2 * dy + 20,
        d.width — 4 * dx, d.height — 4 * dy — 20, dx, dy);
    g.fillArc(d.width / 2 — dx, d.height — 2 * dy + 1,
        2 * dx, dy — 1, 0, 360);
    g.drawArc(d.width / 2 — 3 * dx, d.height — 3 * dy / 2 — 5,
        dx, dy / 2, 0, 360);
    g.drawArc(d.width / 2 + 2 * dx, d.height — 3 * dy / 2 — 5,
        dx, dy / 2, 0, 360);
    Font f1 = new Font("Serif", Font.BOLD|Font.ITALIC, 2 * dy);
    Font f2 = new Font("Serif", Font.BOLD, 5 * dy / 2);
    FontMetrics fm1 = getFontMetrics(f1);
    FontMetrics fm2 = getFontMetrics(f2);
    String s1 = "Всякая последняя ошибка";
    String s2 = "является предпоследней.";
    String s3 = "Закон отладки";
    int firstLine = d.height / 3;
    int nextLine = fm1.getHeight();
    int secondLine = firstLine + nextLine / 2;
    g.setFont(f2);
    g.drawString(s3, (d.width-fm2.stringWidth(s3)) / 2, firstLine);
    g.drawLine(d.width / 4, secondLine — 2,
               3 * d.width / 4, secondLine — 2);
    g.drawLine(d.width / 4, secondLine — 1,
               3 * d.width / 4, secondLine — 1);
    g.drawLine(d.width / 4, secondLine,
               3 * d.width / 4, secondLine);
    g.setFont(f1);
    g.drawString(s1, (d.width — fm1.stringWidth(s1)) / 2,
                 firstLine + 2 * nextLine);
    g.drawString(s2, (d.width — fm1.stringWidth(s2)) / 2,
                 firstLine + 3 * nextLine);
  }
  public static void main(String[] args){
    GraphTest f = new GraphTest(" Пример рисования");
```

```
    f.addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent ev){
        System.exit(0);
      }
    });
  }
}
```

В листинге 9.2 использован простой класс `Dimension`, главная задача которого — хранить ширину и высоту прямоугольного объекта в своих полях `width` и `height`. Метод `getSize()` класса `Component` возвращает размеры компонента в виде объекта класса `Dimension`. В листинге 9.2 размеры компонента `f` типа `GraphTest` установлены в конструкторе методом `setBounds()` равными 500×300 пикселов.

Еще одна особенность листинга 9.2 — для вычерчивания толстой линии, отделяющей заголовок от текста, пришлось провести три параллельные прямые на расстоянии один пиксел друг от друга.

Как вы увидели из обзора класса `Graphics` и сопутствующих ему классов, средства рисования и вывода текста в этом классе весьма ограничены. Линии можно проводить только сплошные и только толщиной в один пиксел, текст выводится только горизонтально и слева направо, не учитываются особенности устройства вывода, например, разрешение экрана.

Эти ограничения можно обойти разными хитростями: чертить несколько параллельных линий, прижатых друг к другу, как в листинге 9.2, или узкий заполненный прямоугольник, выводить текст по одной букве, получить разрешение экрана методом `getScreenSize()` класса `java.awt.Toolkit` и использовать его в дальнейшем. Но все это затрудняет программирование, лишает его стройности и естественности, нарушает принцип KISS.

В Java 2 класс `Graphics`, в рамках системы Java 2D, значительно расширен классом `Graphics2D`.

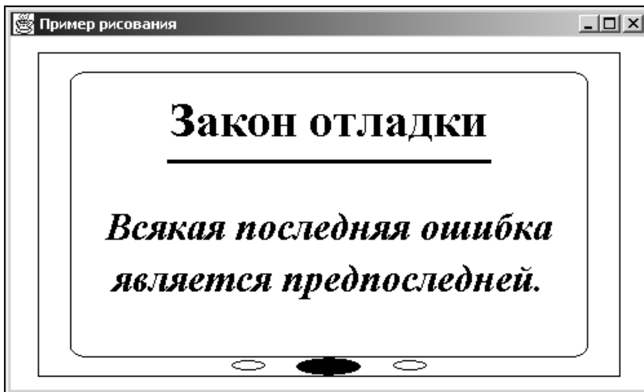


Рис. 9.2. Пример использования класса `Graphics`

Возможности Java 2D

В систему пакетов и классов Java 2D, основа которой — класс `Graphics2D` пакета `java.awt`, внесено несколько принципиально новых положений.

- Кроме координатной системы, принятой в классе `Graphics` и названной координатным пространством пользователя (User Space), введена еще система координат устройства вывода (Device Space): экрана монитора, принтера. Методы класса `Graphics2D` автоматически переводят (transform) систему координат пользователя в систему координат устройства при выводе графики.
- Преобразование координат пользователя в координаты устройства можно задать "вручную", причем преобразованием способно служить любое аффинное преобразование плоскости, в частности, поворот на любой угол и/или сжатие/растяжение. Оно определяется как объект класса `AffineTransform`. Его можно установить как преобразование по умолчанию методом `setTransform()`. Возможно выполнять преобразование "на лету" методами `transform()` и `translate()` и делать композицию преобразований методом `concatenate()`.
- Поскольку аффинное преобразование вещественно, координаты задаются вещественными, а не целыми числами.
- Графические примитивы: прямоугольник, овал, дуга и др., реализуют теперь новый интерфейс `Shape` пакета `java.awt`. Для их вычерчивания можно использовать новый единый для всех фигур метод `draw()`, аргументом которого способен служить любой объект, реализовавший интерфейс `Shape`. Введен метод `fill()`, заполняющий фигуры — объекты класса, реализовавшего интерфейс `Shape`.
- Для вычерчивания (stroke) линий введено понятие пера (pen). Свойства пера описывает интерфейс `Stroke`. Класс `BasicStroke` реализует этот интерфейс. Перо обладает четырьмя характеристиками:
  - оно имеет толщину (width) в один (по умолчанию) или несколько пикселов;
  - оно может закончить линию (end cap) закруглением — статическая константа `CAP_ROUND`, прямым обрезом — `CAP_SQUARE` (по умолчанию), или не фиксировать определенный способ окончания — `CAP_BUTT`;
  - оно может сопрягать линии (line joins) закруглением — статическая константа `JOIN_ROUND`, отрезком прямой — `JOIN_BEVEL`, или просто состыковывать — `JOIN_MITER` (по умолчанию);
  - оно может чертить линию различными пунктирами (dash) и штрихпунктирами, длины штрихов и промежутков задаются в массиве, элементы массива с четными индексами задают длину штриха, с нечетными индексами — длину промежутка между штрихами.

- Методы заполнения фигур описаны в интерфейсе `Paint`. Три класса реализуют этот интерфейс. Класс `Color` реализует его сплошной (solid) заливкой, класс `GradientPaint` — градиентным (gradient) заполнением, при котором цвет плавно меняется от одной заданной точки к другой заданной точке, класс `TexturePaint` — заполнением по предварительно заданному образцу (pattern fill).
- Буквы текста понимаются как фигуры, т. е. объекты, реализующие интерфейс `Shape`, и могут вычерчиваться методом `draw()` с использованием всех возможностей этого метода. При их вычерчивании применяется перо, все методы заполнения и преобразования.
- Кроме имени, стиля и размера, шрифт получил много дополнительных атрибутов, например, преобразование координат, подчеркивание или перечеркивание текста, вывод текста справа налево. Цвет текста и его фона являются теперь атрибутами самого текста, а не графического контекста. Можно задать разную ширину символов шрифта, надстрочные и подстрочные индексы. Атрибуты устанавливаются константами класса `TextAttribute`.
- Процесс визуализации (rendering) регулируется правилами (hints), определенными константами класса `RenderingHints`.

С такими возможностями Java 2D стала полноценной системой рисования, вывода текста и изображений. Посмотрим, как реализованы эти возможности, и как ими можно воспользоваться.

Преобразование координат

Правило преобразования координат пользователя в координаты графического устройства (transform) задается автоматически при создании графического контекста так же, как цвет и шрифт. В дальнейшем его можно изменить методом `setTransform()` так же, как меняется цвет или шрифт. Аргументом этого метода служит объект класса `AffineTransform` из пакета `java.awt.geom`, подобно объектам класса `Color` или `Font` при задании цвета или шрифта.

Рассмотрим подробнее класс `AffineTransform`.

Класс *AffineTransform*

Аффинное преобразование координат задается двумя основными конструкторами класса `AffineTransform`:

```
AffineTransform(double a, double b, double c,
                double d, double e, double  f)
AffineTransform(float a, float b, float c, float d, float e, float f)
```

При этом точка с координатами `(x, y)` в пространстве пользователя перейдет в точку с координатами `(a*x+c*y+e, b*x+d*y+f)` в пространстве графического устройства.

Такое преобразование не искривляет плоскость — прямые линии переходят в прямые, углы между линиями сохраняются. Примерами аффинных преобразований служат повороты вокруг любой точки на любой угол, параллельные сдвиги, отражения от осей, сжатия и растяжения по осям.

Следующие два конструктора используют в качестве аргумента массив `{a, b, c, d, e, f}` или `{a, b, c, d}`, если `e = f = 0`, составленный из таких же коэффициентов в том же порядке:

```
AffineTransform(double[] arr)
AffineTransform(float[] arr)
```

Пятый конструктор создает новый объект по другому, уже имеющемуся, объекту:

```
AffineTransform(AffineTransform at)
```

Шестой конструктор — конструктор по умолчанию — создает тождественное преобразование:

```
AffineTransform()
```

Эти конструкторы математически точны, но неудобны при задании конкретных преобразований. Попробуйте рассчитать коэффициенты поворота на 57° вокруг точки с координатами (20, 40) или сообразить, как будет преобразовано пространство пользователя после выполнения методов:

```
AffineTransform at =
    new AffineTransform(-1.5, 4.45, -0.56, 34.7, 2.68, 0.01);
g.setTransform(at);
```

Во многих случаях удобнее создать преобразование статическими методами, возвращающими объект класса `AffineTransform`.

- `getRotateInstance(double angle)` — возвращает поворот на угол `angle`, заданный в радианах, вокруг начала координат. Положительное направление поворота таково, что точки оси Ox поворачиваются в направлении к оси Oy. Если оси координат пользователя не менялись преобразованием отражения, то положительное значение `angle` задает поворот по часовой стрелке.
- `getRotateInstance(double angle, double x, double y)` — такой же поворот вокруг точки с координатами `(x, y)`.
- `getScaleInstance(double sx, double sy)` — изменяет масштаб по оси Ox в `sx` раз, по оси Oy — в `sy` раз.

- `getShareInstance(double shx, double shy)` — преобразует каждую точку `(x, y)` в точку `(x+shx*y, shy*x+y)`.
- `getTranslateInstance(double tx, double ty)` — сдвигает каждую точку `(x, y)` в точку `(x+tx, y+ty)`.

Метод `createInverse()` возвращает преобразование, обратное текущему преобразованию.

После создания преобразования его можно изменить методами:

```
setTransform(AffineTransform at)
setTransform(double a, double b, double c, double d, double e, double f)
setToIdentity()
setToRotation(double angle)
setToRotation(double angle, double x, double y)
setToScale(double sx, double sy)
setToShare(double shx, double shy)
setToTranslate(double tx, double ty)
```

сделав текущим преобразование, заданное одним из этих методов.

Преобразования, заданные методами:

```
concatenate(AffineTransform at)
rotate(double angle)
rotate(double angle, double x, double y)
scale(double sx, double sy)
shear(double shx, double shy)
translate(double tx, double ty)
```

выполняются перед текущим преобразованием, образуя композицию преобразований.

Преобразование, заданное методом `preConcatenate(AffineTransform at)`, напротив, осуществляется после текущего преобразования.

Прочие методы класса `AffineTransform` производят преобразования различных фигур в пространстве пользователя.

Пора привести пример. Добавим в начало метода `paint()` в листинге 9.2 четыре оператора, как записано в листинге 9.3.

Листинг 9.3. Преобразование пространства пользователя

```
// Начало листинга 9.2…
public void paint(Graphics gr){
  Graphics2D g = (Graphics2D)gr;
  AffineTransform at =
     AffineTransform.getRotateInstance(-Math.PI/4.0, 250.0,150.0);
```

```
    at.concatenate(
       new AffineTransform(0.5, 0.0, 0.0, 0.5, 100.0, 60.0));
    g.setTransform(at);
    Dimension d = getSize();
    // Продолжение листинга 9.2
```

Метод `paint()` начинается с получения экземпляра `g` класса `Graphics2D` простым приведением аргумента `gr` к типу `Graphics2D`. Затем, методом `getRotateInstance()` определяется поворот на 45° против часовой стрелки вокруг точки `(250.0, 150.0)`. Это преобразование — экземпляр `at` класса `AffineTransform`. Метод `concatenate()`, выполняемый объектом `at`, добавляет к этому преобразованию сжатие в два раза по обеим осям координат и перенос начала координат в точку `(100.0, 60.0)`. Наконец, композиция этих преобразований устанавливается как текущее преобразование объекта `g` методом `setTransform()`.

Преобразование выполняется в следующем порядке. Сначала пространство пользователя сжимается в два раза вдоль обеих осей, затем начало координат пользователя — левый верхний угол — переносится в точку (100.0, 60.0) пространства графического устройства. Потом картинка поворачивается на угол 45° против часовой стрелки вокруг точки (250.0, 150.0).

Результат этих преобразований показан на рис. 9.3.

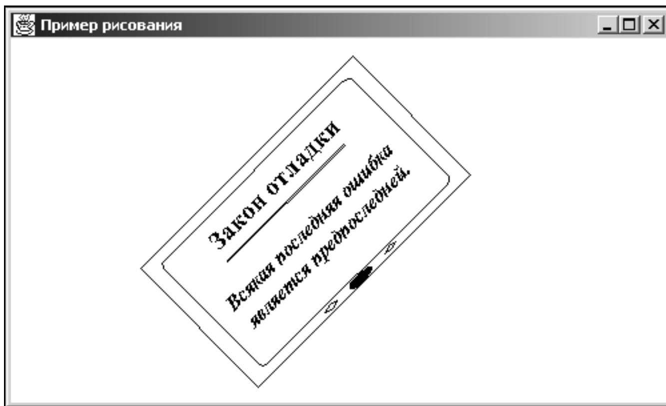


Рис. 9.3. Преобразование координат

Рисование фигур средствами Java 2D

Характеристики пера для рисования фигур описаны в интерфейсе `Stroke`. В Java 2D есть пока только один класс, реализующий этот интерфейс — класс `BasicStroke`.

Класс *BasicStroke*

Конструкторы класса `BasicStroke` определяют характеристики пера. Основной конструктор

```
BasicStroke(float width, int cap, int join, float miter,
 float[] dash, float dashBegin)
```

задает:

- толщину пера `width` в пикселах;
- оформление конца линии `cap`; это одна из констант:
  - `CAP_ROUND` — закругленный конец линии;
  - `CAP_SQUARE` — квадратный конец линии;
  - `CAP_BUTT` — оформление отсутствует;
- способ сопряжения линий `join`; это одна из констант:
  - `JOIN_ROUND` — линии сопрягаются дугой окружности;
  - `JOIN_BEVEL` — линии сопрягаются отрезком прямой, перпендикулярным биссектрисе угла между линиями;
  - `JOIN_MITER` — линии просто стыкуются;
- расстояние между линиями `miter`, начиная с которого применяется сопряжение `JOIN_MITER`;
- длину штрихов и промежутков между штрихами — массив `dash`; элементы массива с четными индексами задают длину штриха в пикселах, элементы с нечетными индексами — длину промежутка; массив перебирается циклически;
- индекс `dashBegin`, начиная с которого перебираются элементы массива `dash`.

Остальные конструкторы задают некоторые характеристики по умолчанию:

- `BasicStroke(float width, int cap, int join, float miter)` — сплошная линия;
- `BasicStroke(float width, int cap, int join)` — сплошная линия с сопряжением `JOIN_ROUND` или `JOIN_BEVEL`; для сопряжения `JOIN_MITER` задается значение `miter = 10.0f`;
- `BasicStroke(float width)` — прямой обрез `CAP_SQUARE` и сопряжение `JOIN_MITER` со значением `miter = 10.0f`;
- `BasicStroke()` — ширина `1.0f`.

Лучше один раз увидеть, чем сто раз прочитать. В листинге 9.4 определено пять перьев с разными характеристиками, рис. 9.4 показывает, как они рисуют.

Листинг 9.4. Определение перьев

```
import java.awt.*;
import java.awt.geom.*;
import java.awt.event.*;

class StrokeTest extends Frame{
  StrokeTest(String s){
    super(s);
    setSize(500, 400);
    setVisible(true);
    addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
  public void paint(Graphics gr){
   Graphics2D g = (Graphics2D)gr;
   g.setFont(new Font("Serif", Font.PLAIN, 15));
   BasicStroke pen1 = new BasicStroke(20, BasicStroke.CAP_BUTT,
      BasicStroke.JOIN_MITER,30);
   BasicStroke pen2 = new BasicStroke(20, BasicStroke.CAP_ROUND,
      BasicStroke.JOIN_ROUND);
   BasicStroke pen3 = new BasicStroke(20, BasicStroke.CAP_SQUARE,
      BasicStroke.JOIN_BEVEL);
   float[] dash1 = {5, 20};
   BasicStroke pen4 = new BasicStroke(10, BasicStroke.CAP_ROUND,
      BasicStroke.JOIN_BEVEL, 10, dash1, 0);
   float[] dash2 = {10, 5, 5, 5};
   BasicStroke pen5 = new BasicStroke(10, BasicStroke.CAP_BUTT,
      BasicStroke.JOIN_BEVEL, 10, dash2, 0);

   g.setStroke(pen1);
   g.draw(new Rectangle2D.Double(50, 50, 50, 50));
   g.draw(new Line2D.Double(50, 180, 150, 180));
   g.setStroke(pen2);
   g.draw(new Rectangle2D.Double(200, 50, 50, 50));
   g.draw(new Line2D.Double(50, 230, 150, 230));
   g.setStroke(pen3);
   g.draw(new Rectangle2D.Double(350, 50, 50, 50));
   g.draw(new Line2D.Double(50, 280, 150, 280));

   g.drawString("JOIN_MITER", 40, 130);
   g.drawString("JOIN_ROUND", 180, 130);
   g.drawString("JOIN_BEVEL", 330, 130);
```

```
    g.drawString("CAP_BUTT",   170, 190);
    g.drawString("CAP_ROUND",  170, 240);
    g.drawString("CAP_SQUARE", 170, 290);

    g.setStroke(pen5);
    g.draw(new Line2D.Double(50, 330, 250, 330));
    g.setStroke(pen4);
    g.draw(new Line2D.Double(50, 360, 250, 360));
    g.drawString("{10, 5, 5, 5,...}", 260, 335);
    g.drawString("{5, 10,...}", 260, 365);
    }
  public static void main(String[] args){
    new StrokeTest("Моя программа");
  }
}
```

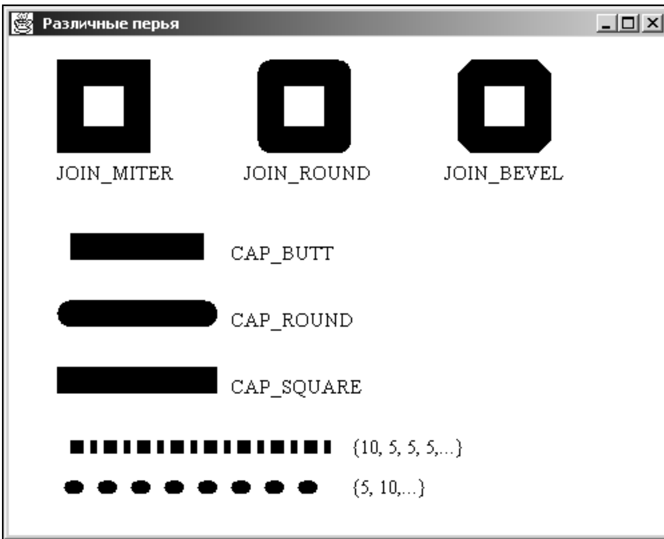


Рис. 9.4. Перья с различными характеристиками

После создания пера одним из конструкторов и установки пера методом `setStroke()` можно рисовать различные фигуры методами `draw()` и `fill()`.

Общие свойства фигур, которые можно нарисовать методом `draw()` класса `Graphics2D`, описаны в интерфейсе `Shape`. Этот интерфейс реализован для создания обычного набора фигур — прямоугольников, прямых, эллипсов, дуг, точек — классами `Rectangle2D`, `RoundRectangle2D`, `Line2D`, `Ellipse2D`, `Arc2D`, `Point2D` пакета `java.awt.geom`. В этом пакете есть еще классы `CubicCurve2D` и `QuadCurve2D` для создания кривых третьего и второго порядка.

Все эти классы абстрактные, но существуют их реализации — вложенные классы `Double` и `Float` для задания координат числами соответствующего

типа. В листинге 9.4 использованы классы Rectangle2D.Double и Line2d.Double для вычерчивания прямоугольников и отрезков.

В пакете java.awt.geom есть еще один интересный класс — GeneralPath. Объекты этого класса могут содержать сложные конструкции, составленные из отрезков прямых или кривых линий и прочих фигур, соединенных или не соединенных между собой. Более того, поскольку этот класс реализует интерфейс Shape, его экземпляры сами являются фигурами и могут быть элементами других объектов класса GeneralPath.

Класс *GeneralPath*

Вначале создается пустой объект класса GeneralPath конструктором по умолчанию GeneralPath() или объект, содержащий одну фигуру, конструктором GeneralPath(Shape sh).

Затем к этому объекту добавляются фигуры методом

```
append(Shape sh, boolean connect)
```

Если параметр connect равен true, то новая фигура соединяется с предыдущими фигурами с помощью текущего пера.

В объекте есть текущая точка. Вначале ее координаты (0, 0), затем ее можно переместить в точку (x, y) методом moveTo(float x, float y).

От текущей точки к точке (x, y) можно провести:

- отрезок прямой методом lineTo(float x, float y);
- отрезок квадратичной кривой методом quadTo(float x1, float y1, float x, float y);
- кривую Безье методом curveTo(float x1, float y1, float x2, float y2, float x, float y).

Текущей точкой после этого становится точка (x, y). Начальную и конечную точки можно соединить методом closePath(). Вот как можно создать треугольник с заданными вершинами:

```
GeneralPath p = new GeneralPath();
p.moveTo(x1, y1);     // Переносим текущую точку в первую вершину,
p.lineTo(x2, y2);     // проводим сторону треугольника до второй вершины,
p.lineTo(x3, y3);     // проводим вторую сторону,
p.closePath();        // проводим третью сторону до первой вершины
```

Способы заполнения фигур определены в интерфейсе Paint. В настоящее время Java 2D содержит три реализации этого интерфейса — классы Color, GradientPaint и TexturePaint. Класс Color нам известен, посмотрим, какие способы заливки предлагают классы GradientPaint и TexturePaint.

Классы *GradientPaint* и *TexturePaint*

Класс `GradientPaint` предлагает сделать заливку следующим образом.

В двух точках `M` и `N` устанавливаются разные цвета. В точке `M(x1, y1)` задается цвет `c1`, в точке `N(x2, y2)` — цвет `c2`. Цвет заливки гладко меняется от `c1` к `c2` вдоль прямой, соединяющей точки `M` и `N`, оставаясь постоянным вдоль каждой прямой, перпендикулярной прямой `MN`. Такую заливку создает конструктор

```
GradientPaint(float x1, float y1, Color c1,
              float x2, float y2, Color c2)
```

При этом вне отрезка `MN` цвет остается постоянным: за точкой `M` — цвет `c1`, за точкой `N` — цвет `c2`.

Второй конструктор

```
GradientPaint(float x1, float y1, Color c1,
       float x2, float y2, Color c2, boolean cyclic)
```

при задании параметра `cyclic == true` повторяет заливку полосы `MN` во всей заливаемой фигуре.

Еще два конструктора задают точки как объекты класса `Point2D`.

Класс `TexturePaint` поступает сложнее. Сначала создается буфер — объект класса `BufferedImage` из пакета `java.awt.image`. Это большой сложный класс. Мы с ним еще встретимся в *главе 15*, а пока нам понадобится только его графический контекст, управляемый экземпляром класса `Graphics2D`. Этот экземпляр можно получить методом `createGraphics()` класса `BufferedImage`. Графический контекст буфера заполняется фигурой, которая будет служить образцом заполнения.

Затем по буферу создается объект класса `TexturePaint`. При этом еще задается прямоугольник, размеры которого будут размерами образца заполнения. Конструктор выглядит так:

```
TexturePaint(BufferedImage buffer, Rectangle2D anchor)
```

После создания заливки — объекта класса `Color`, `GradientPaint` или `TexturePaint` — она устанавливается в графическом контексте методом `setPaint(Paint p)` и используется в дальнейшем методом `fill(Shape sh)`.

Все это демонстрирует листинг 9.5 и рис. 9.5.

Листинг 9.5. Способы заливки

```
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;
import java.awt.event.*;
```

```
class PaintTest extends Frame{
  PaintTest(String s){
    super(s);
    setSize(300, 300);
    setVisible(true);
    addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
  public void paint(Graphics gr){
    Graphics2D g = (Graphics2D)gr;
    BufferedImage bi =
       new BufferedImage(20, 20, BufferedImage.TYPE_INT_RGB);
    Graphics2D big = bi.createGraphics();
    big.draw(new Line2D.Double(0.0,  0.0, 10.0, 10.0));
    big.draw(new Line2D.Double(0.0, 10.0, 10.0,  0.0));
    TexturePaint tp = new TexturePaint(bi,
           new Rectangle2D.Double(0.0, 0.0, 10.0, 10.0));
    g.setPaint(tp);
    g.fill(new Rectangle2D.Double(50,50, 200, 200));
    GradientPaint gp =
         new GradientPaint(100, 100, Color.white,
                           150, 150, Color.black, true);
    g.setPaint(gp);
    g.fill(new Ellipse2D.Double(100, 100, 200, 200));
   }
  public static void main(String[] args){
    new PaintTest(" Способы заливки");
  }
}
```

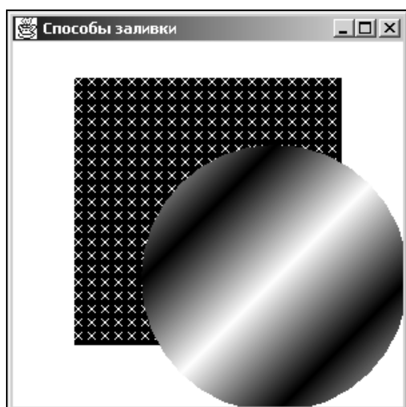


Рис. 9.5. Способы заливки

Вывод текста средствами Java 2D

Шрифт — объект класса `Font` — кроме имени, стиля и размера имеет еще полтора десятка атрибутов: подчеркивание, перечеркивание, наклон, цвет шрифта и цвет фона, ширину и толщину символов, аффинное преобразование, расположение слева направо или справа налево.

Атрибуты шрифта задаются как статические константы класса `TextAttribute`. Наиболее используемые атрибуты перечислены в табл. 9.1.

Таблица 9.1. *Атрибуты шрифта*

Атрибут	**Значение**
`BACKGROUND`	Цвет фона. Объект, реализующий интерфейс `Paint`
`FOREGROUND`	Цвет текста. Объект, реализующий интерфейс `Paint`
`BIDI_EMBEDDED`	Уровень вложенности просмотра текста. Целое от 1 до 15
`CHAR_REPLACEMENT`	Фигура, заменяющая символ. Объект `GraphicAttribute`
`FAMILY`	Семейство шрифта. Строка типа `String`
`FONT`	Шрифт. Объект класса `Font`
`JUSTIFICATION`	Допуск при выравнивании абзаца. Объект класса `Float` со значениями от 0,0 до 1,0. Есть две константы: `JUSTIFICATION_FULL` и `JUSTIFICATION_NONE`
`POSTURE`	Наклон шрифта. Объект класса `Float`. Есть две константы: `POSTURE_OBLIQUE` и `POSTURE_REGULAR`
`RUN_DIRECTION`	Просмотр текста: `RUN_DIRECTION_LTR` — слева направо, `RUN_DIRECTION_RTL` — справа налево
`SIZE`	Размер шрифта в пунктах. Объект класса `Float`
`STRIKETHROUGH`	Перечеркивание шрифта. Задается константой `STRIKETHROUGH_ON`, по умолчанию перечеркивания нет
`SUPERSCRIPT`	Подстрочные или надстрочные индексы. Константы: `SUPERSCRIPT_NONE`, `SUPERSCRIPT_SUB`, `SUPERSCRIPT_SUPER`
`SWAP_COLORS`	Замена местами цвета текста и цвета фона. Константа `SWAP_COLORS_ON`, по умолчанию замены нет
`TRANSFORM`	Преобразование шрифта. Объект класса `AffineTransform`
`UNDERLINE`	Подчеркивание шрифта. Константы: `UNDERLINE_ON`, `UNDERLINE_LOW_DASHED`, `UNDERLINE_LOW_DOTTED`, `UNDERLINE_LOW_GRAY`, `UNDERLINE_LOW_ONE_PIXEL`, `UNDERLINE_LOW_TWO_PIXEL`

Таблица 9.1 *(окончание)*

Атрибут	Значение
WEIGHT	Толщина шрифта. Константы: WEIGHT_ULTRA_LIGHT, WEIGHT__EXTRA_LIGHT, WEIGHT__LIGHT, WEIGHT__DEMILIGHT, WEIGHT__REGULAR, WEIGHT__SEMIBOLD, WEIGHT__MEDIUM, WEIGHT__DEMIBOLD, WEIGHT__BOLD, WEIGHT__HEAVY, WEIGHT__EXTRABOLD, WEIGHT__ULTRABOLD
WIDTH	Ширина шрифта. Константы: WIDTH_CONDENSED, WIDTH_SEMI_CONDENSED, WIDTH_REGULAR, WIDTH_SEMI_EXTENDED, WIDTH_EXTENDED

К сожалению, не все шрифты позволяют задать все атрибуты. Посмотреть список допустимых атрибутов для данного шрифта можно методом getAvailableAttributes() класса Font.

В классе Font есть конструктор Font(Map attributes), которым можно сразу задать нужные атрибуты создаваемому шрифту. Это требует предварительной записи атрибутов в специально созданный для этой цели объект класса, реализующего интерфейс Map: класса HashMap, WeakHashMap или Hashtable *(см. главу 7)*. Например:

```
HashMap hm = new HashMap();
hm.put(TextAttribute.SIZE, new Float(60.0f));
hm.put(TextAttribute.POSTURE, TextAttribute.POSTURE_OBLIQUE);
Font f = new Font(hm);
```

Можно создать шрифт и вторым конструктором, которым мы пользовались в листинге 9.2, а потом добавлять и изменять атрибуты методами deriveFont() класса Font.

Текст в Java 2D обладает собственным контекстом — объектом класса FontRenderContext, хранящим всю информацию, необходимую для вывода текста. Получить его можно методом getFontRenderContext() класса Graphics2D.

Вся информация о тексте, в том числе и об его контексте, собирается в объекте класса TextLayout. Этот класс в Java 2D заменяет класс FontMetrics.

В конструкторе класса TextLayout задается текст, шрифт и контекст. Начало метода paint() со всеми этими определениями может выглядеть так:

```
public void paint(Graphics gr){
   Graphics2D g = (Graphics2D)gr;
   FontRenderContext frc = g.getFontRenderContext();
   Font f = new Font("Serif", Font.BOLD, 15);
   String s = "Какой-то текст";
```

```
    TextLayout tl = new TextLayout(s, f, frc);
    // Продолжение метода
}
```

В классе `TextLayout` есть не только более двадцати методов `get`*`XXX`*`()`, позволяющих узнать различные сведения о тексте, его шрифте и контексте, но и метод

```
draw(Graphics2D g, float x, float y)
```

вычерчивающий содержимое объекта класса `TextLayout` в графический области `g`, начиная с точки `(x, y)`.

Еще один интересный метод

```
getOutline(AffineTransform at)
```

возвращает контур шрифта в виде объекта `Shape`. Этот контур можно затем заполнить по какому-нибудь образцу или вывести только контур, как показано в листинге 9.6.

Листинг 9.6. Вывод текста средствами Java 2D

```
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.awt.event.*;

class StillText extends Frame{
  StillText(String s){
    super(s);
    setSize(400, 200);
    setVisible(true);
    addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
  public void paint(Graphics gr){
    Graphics2D g = (Graphics2D)gr;
    int w = getSize().width, h = getSize().height;
    FontRenderContext frc = g.getFontRenderContext();
    String s = "Тень";
    Font f = new Font("Serif", Font.BOLD, h/3);
    TextLayout tl = new TextLayout(s, f, frc);

    AffineTransform at = new AffineTransform();
    at.setToTranslation(w/2-tl.getBounds().getWidth()/2, h/2);
```

```
    Shape sh = tl.getOutline(at);
    g.draw(sh);

    AffineTransform atsh =
        new AffineTransform(1, 0.0, 1.5, -1, 0.0, 0.0);
    g.transform(at);
    g.transform(atsh);
    Font df = f.deriveFont(atsh);
    TextLayout dtl = new TextLayout(s, df, frc);
    Shape sh2 = dtl.getOutline(atsh);
    g.fill(sh2);
  }
  public static void main(String[] args){
    new StillText("  Эффект тени");
  }
}
```

На рис. 9.6 показан вывод этой программы.

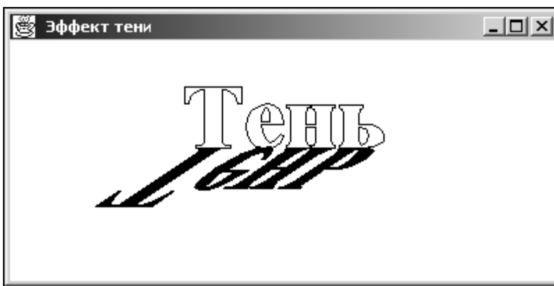


Рис. 9.6. Вывод текста средствами Java 2D

Еще одна возможность создать текст с атрибутами — определить объект класса `AttributedString` из пакета `java.text`. Конструктор этого класса

`AttributedString(String text, Map attributes)`

задает сразу и текст, и его атрибуты. Затем можно добавить или изменить характеристики текста одним их трех методов `addAttibute()`.

Если текст занимает несколько строк, то встает вопрос его форматирования. Для этого вместо класса `TextLayout` используется класс `LineBreakMeasurer`, методы которого позволяют отформатировать абзац. Для каждого сегмента текста можно получить экземпляр класса `TextLayout` и вывести текст, используя его атрибуты.

Для редактирования текста необходимо отслеживать курсором (caret) текущую позицию в тексте. Это осуществляется методами класса `TextHitInfo`, а методы класса `TextLayout` позволяют получить позицию курсора, выделить блок текста и подсветить его.

Наконец, можно задать отдельные правила для вывода каждого символа текста. Для этого надо получить экземпляр класса `GlyphVector` методом `createGlyphVector()` класса `Font`, изменить позицию символа методом `setGlyphPosition()`, задать преобразование символа, если это допустимо для данного шрифта, методом `setGlyphTransform()`, и вывести измененный текст методом `drawGlyphVector()` класса `Graphics2D`. Все это показано в листинге 9.7 и на рис. 9.7 — выводе программы листинга 9.7.

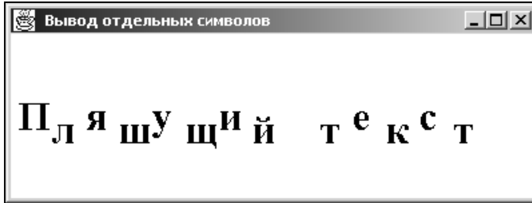


Рис. 9.7. Вывод отдельных символов

Листинг 9.7. Вывод отдельных символов

```
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.awt.event.*;

class GlyphTest extends Frame{
  GlyphTest(String s){
    super(s);
    setSize(400, 150);
    setVisible(true);
    addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
  public void paint(Graphics gr){
    int h = 5;
    Graphics2D g = (Graphics2D)gr;
    FontRenderContext frc = g.getFontRenderContext();
    Font f = new Font("Serif", Font.BOLD, 30);
    GlyphVector gv = f.createGlyphVector(frc, "Пляшущий текст");
    int len = gv.getNumGlyphs();
    for (int i = 0; i < len; i++){
      Point2D.Double p = new Point2D.Double(25 * i, h = -h);
      gv.setGlyphPosition(i, p);
    }
```

```
    g.drawGlyphVector(gv, 10, 100);
  }
  public static void main(String[] args){
    new GlyphTest("  Вывод отдельных символов");
  }
}
```

Методы улучшения визуализации

Визуализацию (rendering) созданной графики можно усовершенствовать, установив один из методов (hint) улучшения одним из методов класса `Graphics2D`:

```
setRenderingHints(RenderingHints.Key key, Object value)
setRenderingHints(Map hints)
```

Ключи — методы улучшения — и их значения задаются константами класса `RenderingHints`, перечисленными в табл. 9.2.

Таблица 9.2. *Методы визуализации и их значения*

Метод	Значение
KEY_ANTIALIASING	Размывание крайних пикселов линий для гладкости изображения; три значения, задаваемые константами VALUE_ANTIALIAS_DEFAULT, VALUE_ANTIALIAS_ON, VALUE_ANTIALIAS_OFF
KEY_TEXT_ANTIALIASING	То же для текста. Константы: VALUE_TEXT_ANTIALIASING_DEFAULT, VALUE_TEXT_ANTIALIASING_ON, VALUE_TEXT_ANTIALIASING_OFF
KEY_RENDERING	Три типа визуализации. Константы: VALUE_RENDER_SPEED, VALUE_RENDER_QUALITY, VALUE_RENDER_DEFAULT
KEY_COLOR_RENDERING	То же для цвета. Константы: VALUE_COLOR_RENDER_SPEED, VALUE_COLOR_RENDER_QUALITY, VALUE_COLOR_RENDER_DEFAULT
KEY_ALPHA_INTERPOLATION	Плавное сопряжение линий. Константы: VALUE_ALPHA_INTERPOLATION_SPEED, VALUE_ALPHA_INTERPOLATION_QUALITY, VALUE_ALPHA_INTERPOLATION_DEFAULT
KEY_INTERPOLATION	Способы сопряжения. Константы: VALUE_INTERPOLATION_BILINEAR, VALUE_INTERPOLATION_BICUBIC, VALUE_INTERPOLATION_NEAREST_NEIGHBOR

Таблица 9.2 *(окончание)*

Метод	**Значение**
KEY_DITHERING	Замена близких цветов. Константы: VALUE_DITHER_ENABLE, VALUE_DITHER_DISABLE, VALUE_DITHER_DEFAULT

Не все графические системы обеспечивают выполнение этих методов, поэтому задание указанных атрибутов не означает, что определяемые ими методы будут применяться на самом деле.

Вот как может выглядеть начало метода `paint()` с указанием методов улучшения визуализации:

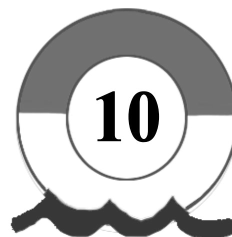
```
public void paint(Graphics gr){
  Graphics2D g = (Graphics2D)gr;
  g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON);
  g.setRenderingHint(RenderingHints.KEY_RENDERING,
        RenderingHints.VALUE_RENDER_QUALITY);
  // Продолжение метода
}
```

Заключение

В этой главе мы, разумеется, не смогли подробно разобрать все возможности Java 2D. Мы не коснулись моделей задания цвета и смешивания цветов, печати графики и текста, динамической загрузки шрифтов, изменения области рисования. В *главе 15* мы рассмотрим средства Java 2D для работы с изображениями, в *главе 18* — средства печати.

В документации SUN J2SDK, в каталоге docs\guide\2d\spec, есть руководство Java 2 API Guide с обзором всех возможностей Java 2D. Там помещены ссылки на руководства и пособия по Java 2D. В каталоге demo\jfc\Java2D\src приведены исходные тексты программ, использующих Java 2D.

ГЛАВА 10



Основные компоненты

Графическая библиотека AWT предлагает более двадцати готовых компонентов. Они показаны на рис. 8.2. Наиболее часто используются подклассы класса `Component`: классы `Button`, `Canvas`, `Checkbox`, `Choice`, `Container`, `Label`, `List`, `Scrollbar`, `TextArea`, `TextField`, `Panel`, `ScrollPane`, `Window`, `Dialog`, `FileDialog`, `Frame`.

Еще одна группа компонентов — это компоненты меню — классы `MenuItem`, `MenuBar`, `Menu`, `PopupMenu`, `CheckboxMenuItem`. Мы рассмотрим их в *главе 13*.

Забегая вперед, для каждого компонента перечислим события, которые в нем происходят. Обработку событий мы разберем в *главе 12*.

Начнем изучать эти компоненты от простых компонентов к сложным и от наиболее часто используемых к применяемым реже. Но сначала посмотрим на то общее, что есть во всех этих компонентах, на сам класс `Component`.

Класс *Component*

Класс `Component` — центр библиотеки AWT — очень велик и обладает большими возможностями. В нем пять статических констант, определяющих размещение компонента внутри пространства, выделенного для компонента в содержащем его контейнере: `BOTTOM_ALIGNMENT`, `CENTER_ALIGNMENT`, `LEFT_ALIGNMENT`, `RIGHT_ALIGNMENT`, `TOP_ALIGNMENT`, и около сотни методов.

Большинство методов — это методы доступа `get`*`XXX`*`()`, `is`*`XXX`*`()`, `set`*`XXX`*`()`. Изучать их нет смысла, надо просто посмотреть, как они используются в подклассах.

Конструктор класса недоступен — он защищенный (protected), потому, что класс `Component` абстрактный, он не может использоваться сам по себе, применяются только его подклассы.

Компонент всегда занимает прямоугольную область со сторонами, параллельными сторонам экрана и в каждый момент времени имеет определенные размеры, измеряемые в пикселах, которые можно узнать методом `getSize()`, возвращающим объект класса `Dimension`, или целочисленными методами `getHeight()` и `getWidth()`, возвращающими высоту и ширину прямоугольника. Новый размер компонента можно установить из программы методами `setSize(Dimension d)` или `setSize(int width, int height)`, если это допускает менеджер размещения контейнера, содержащего компонент.

У компонента есть предпочтительный размер, при котором компонент выглядит наиболее пропорционально. Его можно получить методом `getPreferredSize()` в виде объекта `Dimension`.

Компонент обладает минимальным и максимальным размерами. Их возвращают методы `getMinimumSize()` и `getMaximumSize()` в виде объекта `Dimension`.

В компоненте есть система координат. Ее начало — точка с координатами (0, 0) — находится в левом верхнем углу компонента, ось Ox идет вправо, ось Oy — вниз, координатные точки расположены между пикселами.

В компоненте хранятся координаты его левого верхнего угла в системе координат объемлющего контейнера. Их можно узнать методами `getLocation()`, а изменить — методами `setLocation()`, переместив компонент в контейнере, если это позволит менеджер размещения компонентов.

Можно выяснить сразу и положение, и размер прямоугольной области компонента методом `getBounds()`, возвращающим объект класса `Rectangle`, и изменить разом и положение, и размер компонента методами `setBounds()`, если это позволит сделать менеджер размещения.

Компонент может быть недоступен для действий пользователя, тогда он выделяется на экране обычно светло-серым цветом. Доступность компонента можно проверить логическим методом `isEnabled()`, а изменить — методом `setEnabled(boolean enable)`.

Для многих компонентов определяется графический контекст — объект класса `Graphics`, — который управляется методом `paint()`, описанным в предыдущей главе, и который можно получить методом `getGraphics()`.

В контексте есть текущий цвет и цвет фона — объекты класса `Color`. Цвет фона можно получить методом `getBackground()`, а изменить — методом `setBackground(Color color)`. Текущий цвет можно получить методом `getForeground()`, а изменить — методом `setForeground(Color color)`.

В контексте есть шрифт — объект класса `Font`, возвращаемый методом `getFont()` и изменяемый методом `setFont(Font font)`.

В компоненте определяется локаль — объект класса `Locale`. Его можно получить методом `getLocale()`, изменить — методом `setLocale(Locale locale)`.

В компоненте существует курсор, показывающий положение мыши, — объект класса Cursor. Его можно получить методом getCursor(), изменяется форма курсора в "тяжелых" компонентах с помощью метода setCursor(Cursor cursor). Остановимся на этом классе подробнее.

Класс *Cursor*

Основа класса — статические константы, определяющие форму курсора:

- CROSSHAIR_CURSOR — курсор в виде креста, появляется обычно при поиске позиции для размещения какого-то элемента;
- DEFAULT_CURSOR — обычная форма курсора — стрелка влево вверх;
- HAND_CURSOR — "указующий перст", появляется обычно при выборе какого-то элемента списка;
- MOVE_CURSOR — крест со стрелками, возникает обычно при перемещении элемента;
- TEXT_CURSOR — вертикальная черта, появляется в текстовых полях;
- WAIT_CURSOR — изображение часов, появляется при ожидании.

Следующие курсоры появляются обычно при приближении к краю или углу компонента:

- E_RESIZE_CURSOR — стрелка вправо с упором;
- N_RESIZE_CURSOR — стрелка вверх с упором;
- NE_RESIZE_CURSOR — стрелка вправо вверх, упирающаяся в угол;
- NW_RESIZE_CURSOR — стрелка влево вверх, упирающаяся в угол;
- S_RESIZE_CURSOR — стрелка вниз с упором;
- SE_RESIZE_CURSOR — стрелка вправо вниз, упирающаяся в угол;
- SW_RESIZE_CURSOR — стрелка влево вниз, упирающаяся в угол;
- W_RESIZE_CURSOR — стрелка влево с упором.

Перечисленные константы являются аргументом type в конструкторе класса Cursor(int type).

Вместо конструктора можно обратиться к статическому методу getPredefinedCursor(int type), создающему объект класса Cursor и возвращающему ссылку на него.

Получить курсор по умолчанию можно статическим методом getDefaultCursor(). Затем созданный курсор надо установить в компонент. Например, после выполнения:

```
Cursor curs = new Cursor(Cursor.WAIT_CURSOR);
someComp.setCursor(curs);
```

при появлении указателя мыши в компоненте someComp указатель примет вид часов.

Как создать свой курсор

Кроме этих предопределенных курсоров можно задать свою собственную форму курсора. Ее тип носит название CUSTOM_CURSOR. Сформировать свой курсор можно методом

```
createCustomCursor(Image cursor, Point hotspot, String name)
```

создающим объект класса Cursor и возвращающим ссылку на него. Перед этим следует создать изображение курсора cursor — объект класса Image. Как это сделать, рассказывается в *главе 15*. Аргумент name задает имя курсора, можно написать просто null. Аргумент hotspot задает точку фокуса курсора. Эта точка должна быть в пределах изображения курсора, точнее, в пределах, показываемых методом

```
getBestCursorSize(int desiredWidth, int desiredHeight)
```

возвращающим ссылку на объект класса Dimension. Аргументы метода означают желаемый размер курсора. Если графическая система не допускает создание курсоров, возвращается (0, 0). Этот метод показывает приблизительно размер того курсора, который создаст графическая система, например, (32, 32). Изображение cursor будет подогнано под этот размер, при этом возможны искажения.

Третий метод — getMaximumCursorColors() — возвращает наибольшее количество цветов, например, 256, которое можно использовать в изображении курсора.

Это методы класса java.awt.Toolkit, с которым мы еще не работали. Класс Toolkit содержит некоторые методы, связывающие приложение Java со средствами платформы, на которой выполняется приложение. Поэтому нельзя создать экземпляр класса Toolkit конструктором, для его получения следует выполнить статический метод Toolkit.getDefaultToolkit().

Если приложение работает в окне Window или его расширениях, например, Frame, то можно получить экземпляр Toolkit методом getToolkit() класса Window.

Соберем все это вместе:

```
Toolkit tk = Toolkit.getDefaultToolkit();
int colorMax = tk.getMaximumCursorColors();  // Наибольшее число цветов
Dimension d = tk.getBestCursorSize(50, 50);  // d — размер изображения
int w = d.width, h = d.height, k = 0;
Point p = new Point(0, 0);                   // Фокус курсора будет
                                             // в его верхнем левом углу
```

```
int[] pix = new int[w * h];          // Здесь будут пикселы изображения
for(int i = 0; i < w; i++)
   for(int j = 0; j < h; j++)
      if (j < i) pix[k++] = 0xFFFF0000;   // Левый нижний угол — красный
      else pix[k++] = 0;             // Правый верхний угол — прозрачный
  // Создается прямоугольное изображение размером (w, h),
  // заполненное массивом пикселов pix, с длиной строки w
Image im = createImage(new MemoryImageSource(w, h, pix, 0, w));
Cursor curs = tk.createCustomCursor(im, p, null);
someComp.setCursor(curs);
```

В этом примере создается курсор в виде красного прямоугольного треугольника с катетами размером 32 пиксела и устанавливается в каком-то компоненте `someComp`.

События

Событие `ComponentEvent` происходит при перемещении компонента, изменении его размера, удалении с экрана и появлении на экране.

Событие `FocusEvent` возникает при получении или потере фокуса.

Событие `KeyEvent` проявляется при каждом нажатии и отпускании клавиши, если компонент имеет фокус ввода.

Событие `MouseEvent` происходит при манипуляциях мыши на компоненте.

Каждый компонент перед выводом на экран помещается в контейнер — подкласс класса `Container`. Познакомимся с этим классом.

Класс *Container*

Класс `Container` — прямой подкласс класса `Component`, и наследует все его методы. Кроме них основу класса составляют методы добавления компонентов в контейнер:

- `add(Component comp)` — компонент `comp` добавляется в конец контейнера;
- `add(Component comp, int index)` — компонент `comp` добавляется в позицию `index` в контейнере, если `index == -1`, то компонент добавляется в конец контейнера;
- `add(Component comp, Object constraints)` — менеджеру размещения контейнера даются указания объектом `constraints`;
- `add(String name, Component comp)` — компонент получает имя `name`.

Два метода удаляют компоненты из контейнера:

- `remove(Component comp)` — удаляет компонент с именем `comp`;
- `remove(int index)` — удаляет компонент с индексом `index` в контейнере.

Один из компонентов в контейнере получает *фокус ввода* (input focus), на него направляется ввод с клавиатуры. Фокус можно переносить с одного компонента на другой клавишами <Tab> и <Shift>+<Tab>. Компонент может запросить фокус методом `requestFocus()` и передать фокус следующему компоненту методом `transferFocus()`. Компонент может проверить, имеет ли он фокус, своим логическим методом `hasFocus()`. Это методы класса `Component`.

Для облегчения размещения компонентов в контейнере определяется *менеджер размещения* (layout manager) — объект, реализующий интерфейс `LayoutManager` или его подынтерфейс `LayoutManager2`. Каждый менеджер размещает компоненты в каком-то своем порядке: один менеджер расставляет компоненты в таблицу, другой норовит растащить компоненты по сторонам, третий просто располагает их один за другим, как слова в тексте. Менеджер определяет смысл слов "добавить в конец контейнера" и "добавить в позицию `index`".

В контейнере в любой момент времени может быть установлен только один менеджер размещения. В каждом контейнере есть свой менеджер по умолчанию, установка другого менеджера производится методом

```
setLayout(LayoutManager manager)
```

Менеджеры размещения мы рассмотрим подробно в следующей главе. В данной главе мы будем размещать компоненты вручную, отключив менеджер по умолчанию методом `setLayout(null)`.

События

Кроме событий класса `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent` при добавлении и удалении компонентов в контейнере происходит событие `ContainerEvent`.

Перейдем к рассмотрению конкретных компонентов. Самый простой компонент описывает класс `Label`.

Компонент *Label*

Компонент `Label` — это просто строка текста, оформленная как графический компонент для размещения в контейнере. Текст можно поменять только методом доступа `setText(String text)`, но не вводом пользователя с клавиатуры или с помощью мыши.

Создается объект этого класса одним из трех конструкторов:

- `Label()` — пустой объект без текста;
- `Label(String text)` — объект с текстом `text`, который прижимается к левому краю компонента;

- Label(String text, int alignment) — объект с текстом text и определенным размещением в компоненте текста, задаваемого одной из трех констант: CENTER, LEFT, RIGHT.

Размещение можно изменить методом доступа setAlignment(int alignment).

Остальные методы, кроме методов, унаследованных от класса Component, позволяют получить текст getText() и размещение getAlignment().

События

В классе Label происходят события классов Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent.

Немногим сложнее класс Button.

Компонент *Button*

Компонент Button — это кнопка стандартного для данной графической системы вида с надписью, умеющая реагировать на щелчок кнопки мыши — при нажатии она "вдавливается" в плоскость контейнера, при отпускании — становится "выпуклой".

Два конструктора Button() и Button(String label) создают кнопку без надписи и с надписью label соответственно.

Методы доступа getLabel() и setLabel(String label) позволяют получить и изменить надпись на кнопке.

Главная функция кнопки — реагировать на щелчки мыши, и прочие методы класса обрабатывают эти действия. Мы рассмотрим их в *главе 12.*

События

Кроме событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, при воздействии на кнопку происходит событие ActionEvent.

Немного сложнее класса Label класс Checkbox, создающий кнопки выбора.

Компонент *Checkbox*

Компонент Checkbox — это надпись справа от небольшого квадратика, в котором в некоторых графических системах появляется галочка после щелчка кнопкой мыши — компонент переходит в состояние (state) on. После следующего щелчка галочка пропадает — это состояние off. В других графических системах состояние on отмечается "вдавливанием" квадратика. В компоненте Checkbox состояния on/off отмечаются логическими значениями true/false соответственно.

Три конструктора `Checkbox()`, `Checkbox(String label)`, `Checkbox(String label, boolean state)` создают компонент без надписи, с надписью `label` в состоянии off, и в заданном состоянии `state`.

Методы доступа `getLabel()`, `setLabel(String label)`, `getState()`, `setState(boolean state)` возвращают и изменяют эти параметры компонента.

Компоненты `Checkbox` удобны для быстрого и наглядного выбора из списка, целиком расположенного на экране, как показано на рис. 10.1. Там же продемонстрирована ситуация, в которой нужно выбрать только один пункт из нескольких. В таких ситуациях образуется группа так называемых *радиокнопок* (radio buttons). Они помечаются обычно кружком или ромбиком, а не квадратиком, выбор обозначается жирной точкой в кружке или "вдавливанием" ромбика.

События

В классе `Checkbox` происходят события класса `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, а при изменении состояния кнопки возникает событие `ItemEvent`.

В библиотеке AWT радиокнопки не образуют отдельный компонент. Вместо этого несколько компонентов `Checkbox` объединяются в группу с помощью объекта класса `CheckboxGroup`.

Класс *CheckboxGroup*

Класс `CheckboxGroup` очень мал, поскольку его задача — просто дать общее имя всем объектам `Checkbox`, образующим одну группу. В него входит один конструктор по умолчанию `CheckboxGroup()` и два метода доступа:

- `getSelectedCheckbox()`, возвращающий выбранный объект `Checkbox`;
- `setSelectedCheckbox(Checkbox box)`, задающий выбор.

Как создать группу радиокнопок

Чтобы организовать группу радиокнопок, надо сначала сформировать объект класса `CheckboxGroup`, а затем создавать кнопки конструкторами

```
Checkbox(String label, CheckboxGroup group, boolean state)
Checkbox(String label, boolean state, CheckboxGroup group)
```

Эти конструкторы идентичны, просто при записи конструктора можно не думать о порядке следования его аргументов.

Только одна радиокнопка в группе может иметь состояние `state == true`.

Пора привести пример. В листинге 10.1 приведена программа, помещающая в контейнер `Frame` две метки `Label` сверху, под ними слева три объекта `Checkbox`, справа — группу радиокнопок. Внизу — три кнопки `Button`. Результат выполнения программы показан на рис. 10.1.

Листинг 10.1. Размещение компонентов

```
import java.awt.*;
import java.awt.event.*;

class SimpleComp extends Frame{
  SimpleComp(String s){
    super(s);
    setLayout(null);
    Font f = new Font("Serif", Font.BOLD, 15);
    setFont(f);

    Label l1 = new Label("Выберите товар:", Label.CENTER);
    l1.setBounds(10, 50, 120, 30); add(l1);
    Label l2 = new Label("Выберите способ оплаты:");
    l2.setBounds(160, 50, 200, 30); add(l2);

    Checkbox ch1 = new Checkbox("Книги");
    ch1.setBounds(20, 90, 100, 30); add(ch1);
    Checkbox ch2 = new Checkbox("Диски");
    ch2.setBounds(20, 120, 100, 30); add(ch2);
    Checkbox ch3 = new Checkbox("Игрушки");
    ch3.setBounds(20, 150, 100, 30); add(ch3);

    CheckboxGroup grp = new  CheckboxGroup();
    Checkbox chg1 = new Checkbox("Почтовым переводом", grp, true);
    chg1.setBounds(170, 90, 200, 30); add(chg1);
    Checkbox chg2 = new Checkbox("Кредитной картой", grp, false);
    chg2.setBounds(170, 120, 200, 30); add(chg2);

    Button b1 = new Button("Продолжить");
    b1.setBounds( 30, 220, 100, 30); add(b1);
    Button b2 = new Button("Отменить");
    b2.setBounds(140, 220, 100, 30); add(b2);
    Button b3 = new Button("Выйти");
    b3.setBounds(250, 220, 100, 30); add(b3);

    setSize(400, 300);
    setVisible(true);
  }
  public static void main(String[] args){
    Frame f = new SimpleComp("  Простые компоненты");
```

```
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

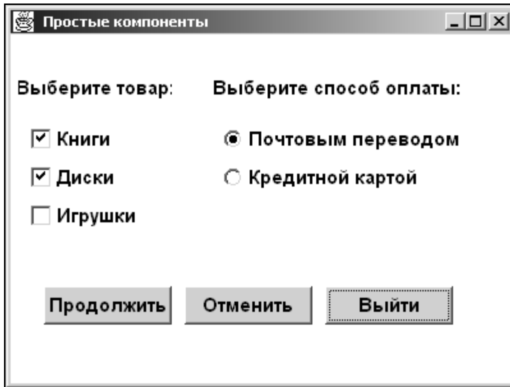


Рис. 10.1. Простые компоненты

Заметьте, что каждый создаваемый компонент следует заносить в контейнер, в данном случае `Frame`, методом `add()`. Левый верхний угол компонента помещается в точку контейнера с координатами, указанными первыми двумя аргументами метода `setBounds()`. Размер компонента задается последними двумя параметрами этого метода.

Если нет необходимости отображать весь список на экране, то вместо группы радиокнопок можно создать раскрывающийся список — объект класса `Choice`.

Компонент *Choice*

Компонент `Choice` — это раскрывающийся список, один, выбранный, пункт (item) которого виден в поле, а другие появляются при щелчке кнопкой мыши на небольшой кнопке справа от поля компонента.

Вначале конструктором `Choice()` создается пустой список.

Затем, методом `add(String text)`, в список добавляются новые пункты с текстом `text`. Они располагаются в порядке написания методов `add()` и нумеруются от нуля.

Вставить новый пункт в нужное место можно методом `insert(String text, int position)`.

Выбор пункта можно произвести из программы методом `select(String text)` или `select(int position)`.

Удалить один пункт из списка можно методом `remove(String text)` или `remove(int position)`, а все пункты сразу — методом `removeAll()`.

Число пунктов в списке можно узнать методом `getItemCount()`.

Выяснить, какой пункт находится в позиции `pos` можно методом `getItem(int pos)`, возвращающим строку.

Наконец, определение выбранного пункта производится методом `getSelectedIndex()`, возвращающим позицию этого пункта, или методом `getSelectedItem()`, возвращающим выделенную строку.

События

В классе `Choice` происходят события класса `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, а при выборе пункта возникает событие `ItemEvent`.

Если надо показать на экране несколько пунктов списка, то создайте объект класса `List`.

Компонент *List*

Компонент `List` — это список с полосой прокрутки, в котором можно выделить один или несколько пунктов. Количество видимых на экране пунктов определяется конструктором списка и размером компонента.

В классе три конструктора:

- `List()` — создает пустой список с четырьмя видимыми пунктами;
- `List(int rows)` — создает пустой список с `rows` видимыми пунктами;
- `List(int rows, boolean multiple)` — создает пустой список, в котором можно отметить несколько пунктов, если `multiple == true`.

После создания объекта в список добавляются пункты с текстом `item`:

- метод `add(String item)` — добавляет новый пункт в конец списка;
- метод `add(String item, int position)` — добавляет новый пункт в позицию `position`.

Позиции нумеруются по порядку, начиная с нуля.

Удалить пункт можно методами `remove(String item)`, `remove(int position)`, `removeAll()`.

Метод `replaceItem(String newItem, int pos)` позволяет заменить текст пункта в позиции `pos`.

Количество пунктов в списке возвращает метод `getItemCount()`.

Выделенный пункт можно получить методом `getSelectedItem()`, а его позицию — методом `getSelectedIndex()`.

Если список позволяет осуществить множественный выбор, то выделенные пункты в виде массива типа `String[]` можно получить методом `getSelectedItems()`, позиции выделенных пунктов в виде массива типа `int[]` — методом `getSelectedIndexes()`.

Кроме этих необходимых методов класс `List` содержит множество других, позволяющих манипулировать пунктами списка и получать его характеристики.

События

Кроме событий класса `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, при двойном щелчке кнопкой мыши на выбранном пункте происходит событие `ActionEvent`.

В листинге 10.2 задаются компоненты, аналогичные компонентам листинга 10.1, с помощью классов `Choice` и `List`, а рис. 10.2 показывает, как изменится при этом интерфейс.

Листинг 10.2. Использование списков

```
import java.awt.*;
import java.awt.event.*;

class ListTest extends Frame{
  ListTest(String s){
    super(s);
    setLayout(null);
    setFont(new Font("Serif", Font.BOLD, 15));

    Label l1 = new Label("Выберите товар:", Label.CENTER);
    l1.setBounds(10, 50, 120, 30); add(l1);
    Label l2 = new Label("Выберите способ оплаты:");
    l2.setBounds(170, 50, 200, 30); add(l2);

    List l = new List(2, true);
    l.add("Книги");
    l.add("Диски");
    l.add("Игрушки");
    l.setBounds(20, 90, 100, 40); add(l);
```

```
    Choice ch = new Choice();
    ch.add("Почтовым переводом");
    ch.add("Кредитной картой");
    ch.setBounds(170, 90, 200,30); add(ch);

    Button b1 = new Button("Продолжить");
    b1.setBounds( 30, 150, 100, 30); add(b1);
    Button b2 = new Button("Отменить");
    b2.setBounds(140, 150, 100, 30); add(b2);
    Button b3 = new Button("Выйти");
    b3.setBounds(250, 150, 100, 30); add(b3);

    setSize(400, 200);
    setVisible(true);
  }
  public static void main(String[] args){
    Frame f = new ListTest("  Простые компоненты");
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

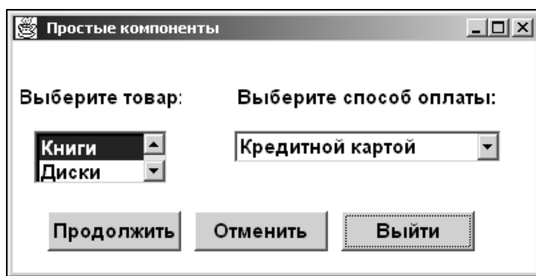


Рис. 10.2. Использование списков

Компоненты для ввода текста

В библиотеке AWT есть два компонента для ввода текста с клавиатуры: `TextField`, позволяющий ввести только одну строку, и `TextArea`, в который можно ввести множество строк.

Оба класса расширяют класс `TextComponent`, в котором собраны их общие методы, такие как выделение текста, позиционирование курсора, получение текста.

Класс *TextComponent*

В классе `TextComponent` нет конструктора, этот класс не используется самостоятельно.

Основной метод класса — метод `getText()` — возвращает текст, находящийся в поле ввода, в виде строки `String`.

Поле ввода может быть нередактируемым, в этом состоянии текст в поле нельзя изменить с клавиатуры или мышью. Узнать состояние поля можно логическим методом `isEditable()`, изменить значения в нем — методом `setEditable(boolean editable)`.

Текст, находящийся в поле, хранится как объект класса `String`, поэтому у каждого символа есть индекс (у первого — индекс 0). Индекс используется для определения позиции курсора (caret) методом `getCaretPosition()`, для установки позиции курсора методом `setCaretPosition(int ind)` и для выделения текста.

Текст выделяется, как обычно, мышью или клавишами со стрелками при нажатой клавише <Shift>, но можно выделить его из программы методом `select(int begin, int end)`. При этом помечается текст от символа с индексом `begin` включительно, до символа с индексом `end` исключительно.

Весь текст выделяет метод `selectAll()`. Можно отметить начало выделения методом `setSelectionStart(int ind)` и конец выделения методом `setSelectionEnd(int ind)`.

Важнее все-таки не задать, а получить выделенный текст. Его возвращает метод `getSelectedText()`, а начальный и конечный индекс выделения возвращают методы `getSelectionStart()` и `getSelectionEnd()`.

События

Кроме событий класса `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, при изменении текста пользователем происходит событие `TextEvent`.

Компонент *TextField*

Компонент `TextField` — это поле для ввода одной строки текста. Ширина поля измеряется в колонках (column). Ширина колонки — это средняя ширина символа в шрифте, которым вводится текст. Нажатие клавиши <Enter> заканчивает ввод и служит сигналом к началу обработки введенного текста, т. е. при этом происходит событие `ActionEvent`.

В классе четыре конструктора:

- `TextField()` — создает пустое поле шириной в одну колонку;
- `TextField(int columns)` — создает пустое поле с числом колонок `columns`;

- `TextField(String text)` — создает поле с текстом `text`;
- `TextField(String text, int columns)` — создает поле с текстом `text` и числом колонок `columns`.

К методам, унаследованным от класса `TextComponent`, добавляются еще методы `getColumns()` и `setColumns(int col)`.

Интересная разновидность поля ввода — поле для ввода пароля. В таком поле вместо вводимых символов появляется какой-нибудь особый эхо-символ, чаще всего звездочка, чтобы пароль никто не подсмотрел через плечо.

Данное поле ввода получается выполнением метода `setEchoChar(char echo)`. Аргумент `echo` — это символ, который будет появляться в поле. Проверить, установлен ли эхо-символ, можно логическим методом `echoCharIsSet()`, получить эхо-символ — методом `getEchoChar()`.

Чтобы вернуть поле ввода в обычное состояние, достаточно выполнить метод `setEchoChar(0)`.

События

Кроме событий класса `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, при изменении текста пользователем происходит событие `TextEvent`, а при нажатии на клавишу <Enter> — событие `ActionEvent`.

Компонент *TextArea*

Компонент `TextArea` — это область ввода с произвольным числом строк. Нажатие клавиши <Enter> просто переводит курсор в начало следующей строки. В области ввода могут быть установлены линейки прокрутки, одна или обе.

Основной конструктор класса

```
TextArea(String text, int rows, int columns, int scrollbars)
```

создает область ввода с текстом `text`, числом видимых строк `rows`, числом колонок `columns`, и заданием полос прокрутки `scrollbars` одной из четырех констант: `SCROLLBARS_NONE`, `SCROLLBARS_HORIZONTAL_ONLY`, `SCROLLBARS_VERTICAL_ONLY`, `SCROLLBARS_BOTH`.

Остальные конструкторы задают некоторые параметры по умолчанию:

- `TextArea(String text, int rows, int columns)` — присутствуют обе полосы прокрутки;
- `TextArea(int rows, int columns)` — в поле пустая строка;
- `TextArea(String text)` — размеры устанавливает контейнер;
- `TextArea()` — конструктор по умолчанию.

Среди методов класса TextArea наиболее важны методы:

- append(String text), добавляющий текст text в конец уже введенного текста;
- insert(String text, int pos), вставляющий текст в указанную позицию pos;
- replaceRange(String text, int begin, int end), удаляющий текст, начиная с индекса begin включительно по end исключительно, и помещающий вместо него текст text.

Другие методы позволяют изменить и получить количество видимых строк.

События

Кроме событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, при изменении текста пользователем происходит событие TextEvent.

В листинге 10.3 создаются три поля: tf1, tf2, tf3 для ввода имени пользователя, его пароля и заказа, и нередактируемая область ввода, в которой накапливается заказ. В поле ввода пароля tf2 появляется эхо-символ *. Результат показан на рис. 10.3.

Листинг 10.3. Поля ввода

```
import java.awt.*;
import java.awt.event.*;

class TextTest extends Frame{
  TextTest(String s){
    super(s);
    setLayout(null);
    setFont(new Font("Serif", Font.PLAIN, 14));

    Label l1 = new Label("Ваше имя:", Label.RIGHT);
    l1.setBounds(20, 30, 70, 25); add(l1);
    Label l2 = new Label("Пароль:", Label.RIGHT);
    l2.setBounds(20, 60, 70, 25); add(l2);

    TextField tf1 = new TextField(30);
    tf1.setBounds(100, 30, 160, 25); add(tf1);

    TextField tf2 = new TextField(30);
    tf2.setBounds(100, 60, 160, 25); add(tf2);
    tf2.setEchoChar('*');

    TextField tf3 = new TextField("Введите сюда Ваш заказ", 30);
    tf3.setBounds(10, 100, 250, 30); add(tf3);
```

```
    TextArea ta = new TextArea("Ваш заказ:", 5, 50,
                              TextArea.SCROLLBARS_NONE);
    ta.setEditable(false);
    ta.setBounds(10, 150, 250, 140); add(ta);

    Button b1 = new Button("Применить");
    b1.setBounds(280, 180, 100, 30); add(b1);
    Button b2 = new Button("Отменить");
    b2.setBounds(280, 220, 100, 30); add(b2);
    Button b3 = new Button("Выйти");
    b3.setBounds(280, 260, 100, 30); add(b3);

    setSize(400, 300);
    setVisible(true);
  }
  public static void main(String[] args){
    Frame f = new TextTest("  Поля ввода");
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

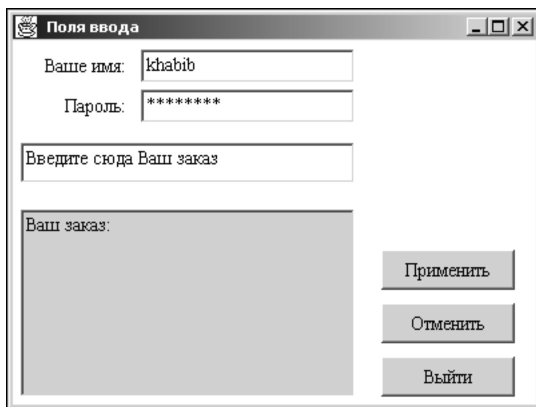


Рис. 10.3. Поля ввода

Компонент *Scrollbar*

Компонент `Scrollbar` — это полоса прокрутки, но в библиотеке AWT класс `Scrollbar` используется еще и для организации ползунка (slider). Объект может располагаться горизонтально или вертикально, обычно полосы прокрутки размещают внизу и справа.

Каждая полоса прокрутки охватывает некоторый диапазон значений и хранит текущее значение из этого диапазона. В линейке прокрутки есть пять элементов управления для перемещения по диапазону. Две стрелки на концах линейки вызывают перемещение на одну единицу (unit) в соответствующем направлении при щелчке на стрелке кнопкой мыши. Положение движка или бегунка (bubble, thumb) показывает текущее значение из диапазона и может его изменять при перемещении бегунка с помощью мыши. Два промежутка между движком и стрелками позволяют переместиться на один блок (block) щелчком кнопки мыши.

Смысл понятий "единица" и "блок" зависит от объекта, с которым работает полоса прокрутки. Например, для вертикальной полосы прокрутки при просмотре текста это может быть строка и страница или строка и абзац.

Методы работы с данным компонентом описаны в интерфейсе `Adjustable`, который реализован классом `Scrollbar`.

В классе `Scrollbar` три конструктора:

- `Scrollbar()` — создает вертикальную полосу прокрутки с диапазоном 0—100, текущим значением 0 и блоком 10 единиц;
- `Scrollbar(int orientation)` — ориентация `orientation` задается одной из двух констант `HORIZONTAL` или `VERTICAL`;
- `Scrollbar(int orientation, int value, int visible, int min, int max)` — задает, кроме ориентации, еще начальное значение `value`, размер блока `visible`, диапазон значений `min`—`max`.

Аргумент `visible` определяет еще и длину движка — она устанавливается пропорционально диапазону значений и длине полосы прокрутки. Например, конструктор по умолчанию задаст длину движка равной 0,1 длины полосы прокрутки.

Основной метод класса — `getValue()` — возвращает значение текущего положения движка на полосе прокрутки. Остальные методы доступа позволяют узнать и изменить характеристики объекта, примеры их использования показаны в листинге 12.6.

События

Кроме событий класса `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, при изменении значения пользователем происходит событие `AdjustmentEvent`.

В листинге 10.4 создаются три вертикальные полосы прокрутки — красная, зеленая и синяя, позволяющие выбрать какое-нибудь значение соответствующего цвета в диапазоне 0—255, с начальным значением 127. Кроме них создается область, заполняемая получившимся цветом, и две кнопки. Ли-

нейки прокрутки, их заголовок и масштабные метки помещены в отдельный контейнер `p` типа `Panel`. Об этом чуть позже в данной главе.

Как все это выглядит, показано на рис. 10.4.

В листинге 12.6 мы "оживим" эту программу.

Листинг 10.4. Линейки прокрутки для выбора цвета

```
import java.awt.*;
import java.awt.event.*;

class ScrollTest extends Frame{
  Scrollbar sbRed   = new Scrollbar(Scrollbar.VERTICAL, 127, 10, 0, 255);
  Scrollbar sbGreen = new Scrollbar(Scrollbar.VERTICAL, 127, 10, 0, 255);
  Scrollbar sbBlue  = new Scrollbar(Scrollbar.VERTICAL, 127, 10, 0, 255);
  Color mixedColor  = new Color(127, 127, 127);
  Label  lm = new Label();
  Button b1 = new Button("Применить");
  Button b2 = new Button("Отменить");

  ScrollTest(String s){
    super(s);
    setLayout(null);
    setFont(new Font("Serif", Font.BOLD, 15));

    Panel p = new Panel();
    p.setLayout(null);
    p.setBounds(10,50, 150, 260); add(p);

    Label lc = new Label("Подберите цвет");
    lc.setBounds(20, 0, 120, 30); p.add(lc);

    Label lmin = new Label("0", Label.RIGHT);
    lmin.setBounds(0, 30, 30, 30); p.add(lmin);
    Label lmiddle = new Label("127", Label.RIGHT);
    lmiddle.setBounds(0, 120, 30, 30); p.add(lmiddle);
    Label lmax = new Label("255", Label.RIGHT);
    lmax.setBounds(0, 200, 30, 30); p.add(lmax);

    sbRed.setBackground(Color.red);
    sbRed.setBounds(40, 30, 20, 200); p.add(sbRed);

    sbGreen.setBackground(Color.green);
    sbGreen.setBounds(70, 30, 20, 200); p.add(sbGreen);

    sbBlue.setBackground(Color.blue);
    sbBlue.setBounds(100, 30, 20, 200); p.add(sbBlue);
```

```
Label lp = new Label("Образец:");
    lp.setBounds(250, 50, 120, 30); add(lp);

    lm.setBackground(new Color(127, 127, 127));
    lm.setBounds(220,  80, 120, 80); add(lm);
    b1.setBounds(240, 200, 100, 30); add(b1);
    b2.setBounds(240, 240, 100, 30); add(b2);

    setSize(400, 300);
    setVisible(true);
  }
  public static void main(String[] args){
    Frame f = new ScrollTest("  Выбор цвета");
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

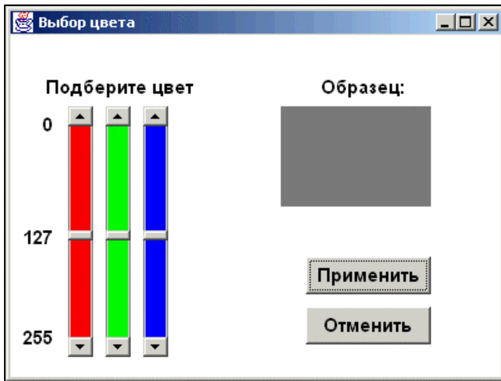


Рис. 10.4. Полосы прокрутки для выбора цвета

В листинге 10.4 использован контейнер `Panel`. Рассмотрим возможности этого класса.

Контейнер *Panel*

Контейнер `Panel` — это невидимый компонент графического интерфейса, служащий для объединения нескольких других компонентов в один объект типа `Panel`.

Класс `Panel` очень прост, но важен. В нем всего два конструктора:

- `Panel()` — создает контейнер с менеджером размещения по умолчанию `FlowLayout`;

- `Panel(LayoutManager layout)` — создает контейнер с указанным менеджером размещения компонентов `layout`.

После создания контейнера в него добавляются компоненты унаследованным методом `add()`:

```
Panel p = new Panel();
p.add(comp1);
p.add(comp2);
```

и т. д. Размещает компоненты в контейнере его менеджер размещения, о чем мы поговорим в следующей главе.

Контейнер `Panel` используется очень часто. Он удобен для создания группы компонентов.

В листинге 10.4 три полосы прокрутки вместе с заголовком "Подберите цвет" и масштабными метками **0**, **127** и **255** образуют естественную группу. Если мы захотим переместить ее в другое место окна, нам придется переносить каждый из семи компонентов, входящих в указанную группу. При этом придется следить за тем, чтобы их взаимное положение не изменилось. Вместо этого мы создали панель `p` и разместили на ней все семь элементов. Метод `setBounds()` каждого из рассматриваемых компонентов указывает в данном случае положение и размер компонента в системе координат панели `p`, а не окна `Frame`. В окно мы поместили сразу целую панель, а не ее отдельные компоненты.

Теперь для перемещения всей группы компонентов достаточно переместить панель, и находящиеся на ней объекты автоматически переместятся вместе с ней, не изменив своего взаимного положения.

Контейнер *ScrollPane*

Контейнер `ScrollPane` может содержать только один компонент, но зато такой, который не помещается целиком в окне. Контейнер обеспечивает средства прокрутки для просмотра большого компонента. В контейнере можно установить полосы прокрутки либо постоянно, константой `SCROLLBARS_ALWAYS`, либо так, чтобы они появлялись только при необходимости (если компонент действительно не помещается в окно) константой `SCROLLBARS_AS_NEEDED`.

Если полосы прокрутки не установлены, это задает константа `SCROLLBARS_NEVER`, то перемещение компонента для просмотра нужно обеспечить из программы одним из методов `setScrollPosition()`.

В классе два конструктора:

- `ScrollPane()` — создает контейнер, в котором полосы прокрутки появляются по необходимости;

- `ScrollPane(int scrollbars)` — создает контейнер, в котором появление линеек прокрутки задается одной из трех указанных выше констант.

Конструкторы создают контейнер размером 100×100 пикселов, в дальнейшем можно изменить размер унаследованным методом `setSize(int width, int height)`.

Ограничение, заключающееся в том, что `ScrollPane` может содержать только один компонент, легко обходится. Всегда можно сделать этим единственным компонентом объект класса `Panel`, разместив на панели что угодно.

Среди методов класса интересны те, что позволяют прокручивать компонент в `ScrollPane`:

- методы `getHAdjustable()` и `getVAdjustable()` возвращают положение линеек прокрутки в виде интерфейса `Adjustable`;
- метод `getScrollPosition()` показывает координаты `(x, y)` точки компонента, находящейся в левом верхнем углу панели `ScrollPane`, в виде объекта класса `Point`;
- метод `setScrollPosition(Point p)` или `setScrollPosition(int x, int y)` прокручивает компонент в позицию `(x, y)`.

Контейнер *Window*

Контейнер `Window` — это пустое окно, без внутренних элементов: рамки, строки заголовка, строки меню, полос прокрутки. Это просто прямоугольная область на экране. Окно типа `Window` самостоятельно, не содержится ни в каком контейнере, его не надо заносить в контейнер методом `add()`. Однако оно не связано с оконным менеджером графической системы. Следовательно, нельзя изменить его размеры, переместить в другое место экрана. Поэтому оно может быть создано только каким-нибудь уже существующим окном, ***владельцем*** (owner) или ***родителем*** (parent) окна `Window`. Когда окно-владелец убирается с экрана, вместе с ним убирается и порожденное окно. Владелец окна указывается в конструкторе:

- `Window(Frame f)` — создает окно, владелец которого — фрейм `f`;
- `Window(Window owner)` — создает окно, владелец которого — уже имеющееся окно или подкласс класса `Window`.

Созданное конструктором окно не выводится на экран автоматически. Его следует отобразить методом `show()`. Убрать окно с экрана можно методом `hide()`, а проверить, видно ли окно на экране — логическим методом `isShowing()`.

Окно типа `Window` возможно использовать для создания всплывающих окон предупреждения, сообщения, подсказки. Для создания диалоговых окон есть подкласс `Dialog`, всплывающих меню — класс `PopupMenu` *(см. главу 13)*.

Видимое на экране окно выводится на передний план методом `toFront()` или, наоборот, помещается на задний план методом `toBack()`.

Уничтожить окно, освободив занимаемые им ресурсы, можно методом `dispose()`.

Менеджер размещения компонентов в окне по умолчанию — `BorderLayout`.

Окно создает свой экземпляр класса `Toolkit`, который возможно получить методом `getToolkit()`.

События

Кроме событий класса `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, при изменении размеров окна, его перемещении или удалении с экрана, а также показа на экране происходит событие `WindowEvent`.

Контейнер *Frame*

Контейнер `Frame` — это полноценное готовое окно со строкой заголовка, в которую помещены кнопки контекстного меню, сворачивания окна в ярлык и разворачивания во весь экран и кнопка закрытия приложения. Заголовок окна записывается в конструкторе или методом `setTitle(String title)`. Окно окружено рамкой. В него можно установить строку меню методом `setMenuBar(MenuBar mb)`. Это мы обсудим в *главе 13*.

На кнопке контекстного меню в левой части строки заголовка изображена дымящаяся чашечка кофе — логотип Java. Вы можете установить там другое изображение методом `setIconImage(Image icon)`, создав предварительно изображение `icon` в виде объекта класса `Image`. Как это сделать, объясняется в *главе 15*.

Все элементы окна `Frame` вычерчиваются графической оболочкой операционной системы по правилам этой оболочки. Окно `Frame` автоматически регистрируется в оконном менеджере графической оболочки и может перемещаться, менять размеры, сворачиваться в панель задач (task bar) с помощью мыши или клавиатуры, как "родное" окно операционной системы.

Создать окно типа `Frame` можно следующими конструкторами:

- `Frame()` — создает окно с пустой строкой заголовка;
- `Frame(String title)` — записывает аргумент `title` в строку заголовка.

Методы класса `Frame` осуществляют доступ к элементам окна, но не забывайте, что класс `Frame` наследует около двухсот методов классов `Component`, `Container` и `Window`. В частности, наследуется менеджер размещения по умолчанию — `BorderLayout`.

События

Кроме событий класса `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, при изменении размеров окна, его перемещении или удалении с экрана, а также показа на экране происходит событие `WindowEvent`.

Программа листинга 10.5 создает два окна типа `Frame`, в которые помещаются строки — метки `Label`. При закрытии основного окна щелчком по соответствующей кнопке в строке заголовка или комбинацией клавиш <Alt>+<F4> выполнение программы завершается обращением к методу `System.exit(0)`, и закрываются оба окна. При закрытии второго окна происходит обращение к методу `dispose()`, и закрывается только это окно.

Листинг 10.5. Создание двух окон

```
import java.awt.*;
import java.awt.event.*;

class TwoFrames{
  public static void main(String[] args){
    Fr1 f1 = new Fr1("  Основное окно");
    Fr2 f2 = new Fr2("  Второе окно");
  }
}
class Fr1 extends Frame{
  Fr1(String s){
    super(s);
    setLayout(null);
    Font f = new Font("Serif", Font.BOLD, 15);
    setFont(f);
    Label l = new Label("Это главное окно", Label.CENTER);
    l.setBounds(10, 30, 180, 30);
    add(l);
    setSize(200, 100);
    setVisible(true);
    addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
class Fr2 extends Frame{
  Fr2(String s){
    super(s);
    setLayout(null);
```

```
    Font f = new Font("Serif", Font.BOLD, 15);
    setFont(f);
    Label l = new Label("Это второе окно", Label.CENTER);
    l.setBounds(10, 30, 180, 30);
    add(l);
    setBounds(50, 50, 200, 100);
    setVisible(true);
    addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         dispose();
       }
    });
  }
}
```

На рис. 10.5 показан вывод этой программы. Взаимное положение окон определяется оконным менеджером операционной системы и может быть не таким, какое показано на рисунке.

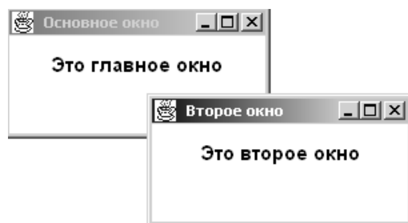


Рис. 10.5. Программа с двумя окнами

Контейнер *Dialog*

Контейнер `Dialog` — это окно обычно фиксированного размера, предназначенное для ответа на сообщения приложения. Оно автоматически регистрируется в оконном менеджере графической оболочки, следовательно, его можно перемещать по экрану, менять его размеры. Но окно типа `Dialog`, как и его суперкласс — окно типа `Window`, — обязательно имеет владельца `owner`, который указывается в конструкторе. Окно типа `Dialog` может быть *модальным* (modal), в котором надо обязательно выполнить все предписанные действия, иначе из окна нельзя будет выйти.

В классе семь конструкторов. Из них:

- `Dialog(Dialog owner)` — создает немодальное диалоговое окно с пустой строкой заголовка;
- `Dialog(Dialog owner, String title)` — создает немодальное диалоговое окно со строкой заголовка `title`;

- `Dialog(Dialog owner, String title, boolean modal)` — создает диалоговое окно, которое будет модальным, если `modal == true`.

Четыре других конструктора аналогичны, но создают диалоговые окна, принадлежащие окну типа `Frame`:

```
Dialog(Frame owner)
Dialog(Frame owner, String title)
Dialog(Frame owner, boolean modal)
Dialog(Frame owner, String title, Boolean modal)
```

Среди методов класса интересны методы: `isModal()`, проверяющий состояние модальности, и `setModal(boolean modal)`, меняющий это состояние.

События

Кроме событий класса `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, при изменении размеров окна, его перемещении или удалении с экрана, а также показа на экране происходит событие `WindowEvent`.

В листинге 10.6. создается модальное окно доступа, в которое вводится имя и пароль. Пока не будет сделан правильный ввод, другие действия невозможны. На рис. 10.6 показан вид этого окна.

Листинг 10.6. Модальное окно доступа

```
import java.awt.*;
import java.awt.event.*;

class LoginWin extends Dialog{
  LoginWin(Frame f, String s){
    super(f, s, true);
    setLayout(null);
    setFont(new Font("Serif", Font.PLAIN, 14));

    Label l1 = new Label("Ваше имя:", Label.RIGHT);
    l1.setBounds(20, 30, 70, 25); add(l1);

    Label l2 = new Label("Пароль:", Label.RIGHT);
    l2.setBounds(20, 60, 70, 25); add(l2);

    TextField tf1 = new TextField(30);
    tf1.setBounds(100, 30, 160, 25); add(tf1);

    TextField tf2 = new TextField(30);
    tf2.setBounds(100, 60, 160, 25); add(tf2);
    tf2.setEchoChar('*');
```

```
    Button b1 = new Button("Применить");
    b1.setBounds(50, 100, 100, 30); add(b1);

    Button b2 = new Button("Отменить");
    b2.setBounds(160, 100, 100, 30); add(b2);

    setBounds(50, 50, 300, 150);
  }
}
class DialogTest extends Frame{
  DialogTest(String s){
    super(s);
    setLayout(null);
    setSize(200, 100);
    setVisible(true);

    Dialog d = new LoginWin(this, "  Окно входа");
    d.setVisible(true);
  }
  public static void main(String[] args){
    Frame f = new DialogTest("  Окно-владелец");
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

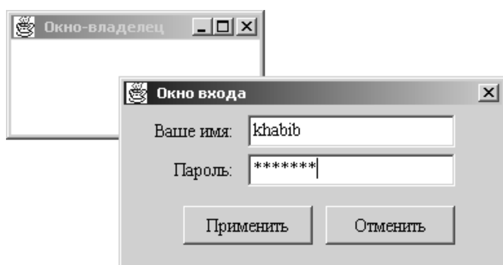


Рис. 10.6. Модальное окно доступа

Контейнер *FileDialog*

Контейнер `FileDialog` — это модальное окно с владельцем типа `Frame`, содержащее стандартное окно выбора файла операционной системы для открытия (константа `LOAD`) или сохранения (константа `SAVE`). Окна операционной системы создаются и помещаются в объект класса `FileDialog` автоматически.

В классе три конструктора:

- `FileDialog(Frame owner)` — создает окно с пустым заголовком для открытия файла;
- `FileDialog(Frame owner, String title)` — создает окно открытия файла с заголовком `title`;
- `FileDialog(Frame owner, String title, int mode)` — создает окно открытия или сохранения документа; аргумент `mode` имеет два значения: `FileDialog.LOAD` и `FileDialog.SAVE`.

Методы класса `getDirectory()` и `getFile()` возвращают только выбранный каталог и имя файла в виде строки `String`. Загрузку или сохранение файла затем нужно производить методами классов ввода/вывода, как рассказано в *главе 18*, там же приведены примеры использования класса `FileDialog`.

Можно установить начальный каталог для поиска файла и имя файла методами `setDirectory(String dir)` и `setFile(String fileName)`.

Вместо конкретного имени файла `fileName` можно написать шаблон, например, `*.java` (первые символы — звездочка и точка), тогда в окне будут видны только имена файлов, заканчивающиеся точкой и словом `java`.

Метод `setFilenameFilter(FilenameFilter filter)` устанавливает шаблон `filter` для имени выбираемого файла. В окне будут видны только имена файлов, подходящие под шаблон. Этот метод не реализован в SUN JDK на платформе MS Windows.

События

Кроме событий класса `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, при изменении размеров окна, его перемещении или удалении с экрана, а также показа на экране происходит событие `WindowEvent`.

Создание собственных компонентов

Создать свой компонент, дополняющий свойства и методы уже существующих компонентов AWT, очень просто — надо лишь образовать свой класс как расширение существующего класса `Button`, `TextField` или другого класса-компонента.

Если надо скомбинировать несколько компонентов в один, новый, компонент, то достаточно расширить класс `Panel`, расположив компоненты на панели.

Если же требуется создать совершенно новый компонент, то AWT предлагает две возможности: создать "тяжелый" или "легкий" компонент. Для создания собственных "тяжелых" компонентов в библиотеке AWT есть класс

Canvas — пустой компонент, для которого создается свой peer-объект графической системы.

Компонент *Canvas*

Компонент Canvas — это пустой компонент. Класс Canvas очень прост — в нем только конструктор по умолчанию Canvas() и пустая реализация метода paint(Graphics g).

Чтобы создать свой "тяжелый" компонент, необходимо расширить класс Canvas, дополнив его нужными полями и методами, и при необходимости переопределить метод paint().

Например, как вы заметили, на стандартной кнопке Button можно написать только одну текстовую строку. Нельзя написать несколько строк или отобразить на кнопке рисунок. Создадим свой "тяжелый" компонент — кнопку с рисунком.

В листинге 10.7 кнопка с рисунком — класс FlowerButton. Рисунок задается методом drawFlower(), а рисуется методом paint(). Метод paint(), кроме того, чертит по краям кнопки внизу и справа отрезки прямых, изображающих тень, отбрасываемую "выпуклой" кнопкой. При нажатии кнопки мыши на компоненте такие же отрезки чертятся вверху и слева — кнопка "вдавилась". При этом рисунок сдвигается на два пиксела вправо вниз — он "вдавливается" в плоскость окна.

Кроме этого, в классе FlowerButton задана реакция на нажатие и отпускание кнопки мыши. Это мы обсудим в *главе 12*, а пока скажем, что при каждом нажатии и отпускании кнопки меняется значение поля isDown и кнопка перечерчивается методом repaint(). Это достигается выполнением методов mousePressed() и mouseReleased().

Для сравнения рядом помещена стандартная кнопка типа Button того же размера. Рис. 10.7 демонстрирует вид этих кнопок.

Листинг 10.7. Кнопка с рисунком

```
import java.awt.*;
import java.awt.event.*;

class FlowerButton extends Canvas implements MouseListener{
  private boolean isDown=false;
  public FlowerButton(){
     super();
     setBackground(Color.lightGray);
     addMouseListener(this);
  }
```

```
  public void drawFlower(Graphics g, int x, int y, int w, int h){
     g.drawOval(x + 2*w/5 — 6, y, w/5, w/5);
     g.drawLine(x + w/2 — 6, y + w/5, x + w/2 — 6, y + h — 4);
     g.drawOval(x + 3*w/10 — 6, y + h/3 — 4, w/5, w/5);
     g.drawOval(x + w/2 — 6, y + h/3 — 4, w/5, w/5);
  }
  public void paint(Graphics g){
     int w = getSize().width, h = getSize().height;
     if (isDown){
       g.drawLine(0, 0, w — 1, 0);
       g.drawLine(1, 1, w — 1, 1);
       g.drawLine(0, 0, 0, h — 1);
       g.drawLine(1, 1, 1, h — 1);
       drawFlower(g, 8, 10, w, h);
     }else{
       g.drawLine(0, h — 2, w — 2, h — 2);
       g.drawLine(1, h — 1, w — 1, h — 1);
       g.drawLine(w — 2, h — 2, w — 2, 0);
       g.drawLine(w — 1, h — 1, w — 1, 1);
       drawFlower(g, 6, 8, w, h);
     }
  }
  public void mousePressed(MouseEvent e){
    isDown=true;
    repaint();
  }
  public void mouseReleased(MouseEvent e){
    isDown=false;
    repaint();
  }
  public void mouseEntered(MouseEvent e){}
  public void mouseExited(MouseEvent e) {}
  public void mouseClicked(MouseEvent e){}
}
class DrawButton extends Frame{
  DrawButton(String s){
    super(s);
    setLayout(null);

    Button b = new Button("OK");
    b.setBounds(200, 50, 100, 60); add(b);

    FlowerButton d = new FlowerButton();
    d.setBounds(50, 50, 100, 60); add(d);

    setSize(400, 150);
    setVisible(true);
  }
```

```
  public static void main(String[] args){
    Frame f= new DrawButton("  Кнопка с рисунком");
    f.addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent ev){
        System.exit(0);
      }
    });
  }
}
```

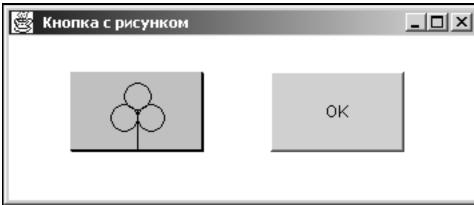


Рис. 10.7. Кнопка с рисунком

Создание "легкого" компонента

"Легкий" компонент, не имеющий своего peer-объекта в графической системе, создается как прямое расширение класса `Component` или `Container`. При этом необходимо задать те действия, которые в "тяжелых" компонентах выполняет peer-объект.

Например, заменив в листинге 10.7 заголовок класса `FlowerButton` строкой

```
class FlowerButton extends Component implements MouseListener{
```

а затем перекомпилировав и выполнив программу, вы получите "легкую" кнопку, но увидите, что ее фон стал белым, потому что метод `setBackground(Color.lightGray)` не сработал.

Это объясняется тем, что теперь всю черную работу по изображению кнопки на экране выполняет не peer-двойник кнопки, а "тяжелый" контейнер, в котором расположена кнопка, в нашем случае класс `Frame`. Контейнер же ничего не знает о том, что надо обратиться к методу `setBackground()`, он рисует только то, что записано в методе `paint()`. Придется убрать метод `setBackground()` из конструктора и заливать фон серым цветом вручную в методе `paint()`, как показано в листинге 10.8.

"Легкий" контейнер не умеет рисовать находящиеся в нем "легкие" компоненты, поэтому в конце метода `paint()` "легкого" контейнера нужно обратиться к методу `paint()` суперкласса:

```
super.paint(g);
```

Тогда рисованием займется "тяжелый" суперкласс-контейнер. Он нарисует и лежащий в нем "легкий" контейнер, и размещенные в контейнере "легкие" компоненты.

Совет

Завершайте метод `paint()` "легкого" контейнера обращением к методу `paint()` суперкласса.

Предпочтительный размер "тяжелого" компонента устанавливается peer-объектом, а для "легких" компонентов его надо задать явно, переопределив метод `getPreferredSize()`, иначе некоторые менеджеры размещения, например `FlowLayout()`, установят нулевой размер, и компонент не будет виден на экране.

Совет

Переопределяйте метод `getPreferredSize()`.

Интересная особенность "легких" компонентов — они изначально рисуются прозрачными, не закрашенная часть прямоугольного объекта не будет видна. Это позволяет создать компонент любой видимой формы. Листинг 10.8 показывает, как можно изменить метод `paint()` листинга 10.7 для создания круглой кнопки и задать дополнительные методы, а рис. 10.8 демонстрирует ее вид.

Листинг 10.8. Создание круглой кнопки

```
public void paint(Graphics g){
   int w = getSize().width, h = getSize().height;
   int d = Math.min(w, h);           // Диаметр круга
   Color c = g.getColor();           // Сохраняем текущий цвет
   g.setColor(Color.lightGray);      // Устанавливаем серый цвет
   g.fillArc(0, 0, d, d, 0, 360);    // Заливаем круг серым цветом
   g.setColor(c);                    // Восстанавливаем текущий цвет
   if (isDown){
     g.drawArc(0, 0, d, d, 43, 180);
     g.drawArc(1, 1, d — 2, d — 2, 43, 180);
     drawFlower(g, 8, 10, d, d);
   }else{
     g.drawArc(0, 0, d, d, 229, 162);
     g.drawArc(1, 1, d — 2, d — 2, 225, 170);
     drawFlower(g, 6, 8, d, d);
   }
}
public Dimension getPreferredSize(){
   return new Dimension(30,30);
}
```

```
  public Dimension getMinimumSize(){
     return getPreferredSize();
  }
  public Dimension getMaximumSize(){
     return getPreferredSize();
  }
```

Рис. 10.8. Круглая кнопка

Сразу же надо дать еще одну рекомендацию. "Легкие" контейнеры не занимаются обработкой событий без специального указания. Поэтому в конструктор "легкого" компонента следует включить обращение к методу `enableEvents()` для каждого типа событий. В нашем примере в конструктор класса `FlowerButton` полезно добавить строку

```
enableEvents(AWTEvent.MOUSE_EVENT_MASK);
```

на случай, если кнопка окажется в "легком" контейнере. Подробнее об этом мы поговорим в *главе 12*.

В документации есть хорошие примеры создания "легких" компонентов, посмотрите страницу docs\guide\awt\demos\lightweight\index.html.

ГЛАВА 11

Размещение компонентов

В предыдущей главе мы размещали компоненты "вручную", задавая их размеры и положение в контейнере абсолютными координатами в координатной системе контейнера. Для этого мы применяли метод `setBounds()`.

Такой способ размещает компоненты с точностью до пиксела, но не позволяет перемещать их. При изменении размеров окна с помощью мыши компоненты останутся на своих местах, привязанными к левому верхнему углу контейнера. Кроме того, нет гарантии, что все мониторы отобразят компоненты так, как вы задумали.

Чтобы учесть изменение размеров окна, надо задать размеры и положение компонента относительно размеров контейнера, например, так:

```
int w = getSize().width;        // Получаем ширину
int h = getSize().height;       // и высоту контейнера
Button b = new Button("OK");    // Создаем кнопку
b.setBounds(9*w/20, 4*h/5, w/10, h/10);
```

и при всяком изменении размеров окна задавать расположение компонента заново.

Чтобы избавить программиста от этой кропотливой работы, в библиотеку AWT внесены два интерфейса: `LayoutManager` и порожденный от него интерфейс `LayoutManager2`, а также пять реализаций этих интерфейсов: классы `BorlerLayout`, `CardLayout`, `FlowLayout`, `GridLayout` и `GridBagLayout`. Эти классы названы *менеджерами размещения* (layout manager) компонентов.

Каждый программист может создать свои менеджеры размещения, реализовав интерфейсы `LayoutManager` или `LayoutManager2`.

Посмотрим, как размещают компоненты эти классы.

Менеджер *FlowLayout*

Наиболее просто поступает менеджер размещения `FlowLayout`. Он укладывает в контейнер один компонент за другим слева направо как кирпичи, переходя от верхних рядов к нижним. При изменении размера контейнера "кирпичи" перестраиваются, как показано на рис. 11.1. Компоненты поступают в том порядке, в каком они заданы в методах `add()`.

В каждом ряду компоненты могут прижиматься к левому краю, если в конструкторе аргумент `align` равен `FlowLayout.LEFT`, к правому краю, если этот аргумент `FlowLayout.RIGHT`, или собираться в середине ряда, если `FlowLayout.CENTER`.

Между компонентами можно оставить промежутки (gap) по горизонтали `hgap` и вертикали `vgap`. Это задается в конструкторе:

```
FlowLayout(int align, int hgap, int vgap)
```

Второй конструктор задает промежутки размером 5 пикселов:

```
FlowLayout(int align)
```

Третий конструктор определяет выравнивание по центру и промежутки 5 пикселов:

```
FlowLayout()
```

После формирования объекта эти параметры можно изменить методами:

```
setHgap(int hgap)
setVgap(int vgap)
setAlignment(int align)
```

В листинге 11.1 создаются кнопка `Button`, метка `Label`, кнопка выбора `Checkbox`, раскрывающийся список `Choice`, поле ввода `TextField` и все это размещается в контейнере `Frame`. Рис. 11.1 содержит вид этих компонентов при разных размерах контейнера.

Листинг 11.1. Менеджер размещения `FlowLayout`

```
import java.awt.*;
import java.awt.event.*;

class FlowTest extends Frame{
  FlowTest(String s){
    super(s);
    setLayout(new FlowLayout(FlowLayout.LEFT, 10, 10));
    add(new Button("Кнопка"));
    add(new Label("Метка"));
    add(new Checkbox("Выбор"));
```

```
    add(new Choice());
    add(new TextField("Справка", 10));
    setSize(300, 100);
    setVisible(true);
  }
  public static void main(String[] args){
    Frame f= new FlowTest("  Менеджер FlowLayout");
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

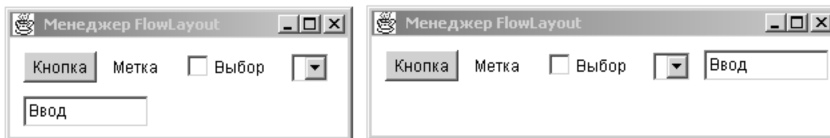


Рис. 11.1. Размещение компонентов с помощью `FlowLayout`

Менеджер *BorderLayout*

Менеджер размещения `BorderLayout` делит контейнер на пять неравных областей, полностью заполняя каждую область одним компонентом, как показано на рис. 11.2. Области получили географические названия `NORTH`, `SOUTH`, `WEST`, `EAST` и `CENTER`.

Метод `add()` в случае применения `BorderLayout` имеет два аргумента: ссылку на компонент `comp` и область `region`, в которую помещается компонент — одну из перечисленных выше констант:

```
add(Component comp, String region)
```

Обычный метод `add(Component comp)` с одним аргументом помещает компонент в область `CENTER`.

В классе два конструктора:

- `BorderLayout()` — между областями нет промежутков;
- `BorderLayout(int hgap int vgap)` — между областями остаются горизонтальные `hgap` и вертикальные `vgap` промежутки, задаваемые в пикселах.

Если в контейнер помещается менее пяти компонентов, то некоторые области не используются и не занимают места в контейнере, как можно заме-

тить на рис. 11.3. Если не занята область `CENTER`, то компоненты прижимаются к границам контейнера.

В листинге 11.2 создаются пять кнопок, размещаемых в контейнере. Заметьте отсутствие установки менеджера в контейнере `setLayout()` — менеджер `BorderLayout` установлен в контейнере `Frame` по умолчанию. Результат размещения показан на рис. 11.2.

Листинг 11.2. Менеджер размещения `BorderLayout`

```
import java.awt.*;
import java.awt.event.*;

class BorderTest extends Frame{
  BorderTest(String s){
    super(s);
    add(new Button("North"), BorderLayout.NORTH);
    add(new Button("South"), BorderLayout.SOUTH);
    add(new Button("West"), BorderLayout.WEST);
    add(new Button("East"), BorderLayout.EAST);
    add(new Button("Center"));
    setSize(300, 200);
    setVisible(true);
  }
  public static void main(String[] args){
    Frame f= new BorderTest("  Менеджер BorderLayout");
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

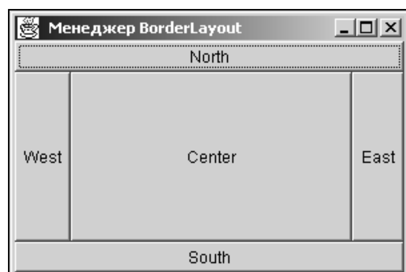


Рис. 11.2. Области размещения `BorderLayout`

Менеджер размещения `BorderLayout` кажется неудобным: он располагает не больше пяти компонентов, последние растекаются по всей области, области

имеют странный вид. Но дело в том, что в каждую область можно поместить не компонент, а панель, и размещать компоненты на ней, как сделано в листинге 11.3 и показано на рис. 11.3. Напомним, что на панели `Panel` менеджер размещения по умолчанию `FlowLayout`.

Листинг 11.3. Сложная компоновка

```
import java.awt.*;
import java.awt.event.*;

class BorderPanelTest extends Frame{
  BorderPanelTest(String s){
    super(s);
        // Создаем панель p2 с тремя кнопками
    Panel p2 = new Panel();
    p2.add(new Button("Выполнить"));
    p2.add(new Button("Отменить"));
    p2.add(new Button("Выйти"));

    Panel p1 = new Panel();
    p1.setLayout(new BorderLayout());
        // Помещаем панель p2 с кнопками на "юге" панели p1
    p1.add(p2, BorderLayout.SOUTH);
        // Поле ввода помещаем на "севере"
    p1.add(new TextField("Поле ввода", 20), BorderLayout.NORTH);
        // Область ввода помещается в центре
    p1.add(new TextArea("Область ввода", 5, 20,
       TextArea.SCROLLBARS_NONE), BorderLayout.CENTER);

    add(new Scrollbar(Scrollbar.HORIZONTAL), BorderLayout.SOUTH);
    add(new Scrollbar(Scrollbar.VERTICAL), BorderLayout.EAST);
        // Панель p1 помещаем в "центре" контейнера
    add(p1, BorderLayout.CENTER);
    setSize(300, 200);
    setVisible(true);
  }
  public static void main(String[] args){
    Frame f= new BorderPanelTest("  Сложная компоновка");
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

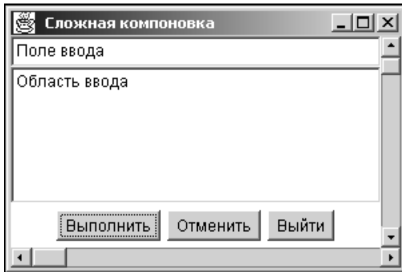


Рис. 11.3. Компоновка с помощью `FlowLayout` и `BorderLayout`

Менеджер *GridLayout*

Менеджер размещения `GridLayout` расставляет компоненты в таблицу с заданным в конструкторе числом строк `rows` и столбцов `columns`:

```
GridLayout(int rows, int columns)
```

Все компоненты получают одинаковый размер. Промежутков между компонентами нет.

Второй конструктор позволяет задать промежутки между компонентами в пикселах по горизонтали `hgap` и вертикали `vgap`:

```
GridLayout(int rows, int columns, int hgap, int vgap)
```

Конструктор по умолчанию `GridLayout()` задает таблицу размером 0×0 без промежутков между компонентами. Компоненты будут располагаться в одной строке.

Компоненты размещаются менеджером `GridLayout` слева направо по строкам созданной таблицы в том порядке, в котором они заданы в методах `add()`.

Нулевое количество строк или столбцов означает, что менеджер сам создаст нужное их число.

В листинге 11.4 выстраиваются кнопки для калькулятора, а рис. 11.4 показывает, как выглядит это размещение.

Листинг 11.4. Менеджер `GridLayout`

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

class GridTest extends Frame{
  GridTest(String s){
    super(s);
    setLayout(new GridLayout(4, 4, 5, 5));
```

```
    StringTokenizer st =
       new StringTokenizer("7 8 9 / 4 5 6 * 1 2 3 — 0 . = +");
    while(st.hasMoreTokens())
       add(new Button(st.nextToken()));

    setSize(200, 200);
    setVisible(true);
  }
  public static void main(String[] args){
    Frame f= new GridTest("  Менеджер GridLayout");
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

Рис. 11.4. Размещение кнопок менеджером `GridLayout`

Менеджер *CardLayout*

Менеджер размещения `CardLayout` своеобразен — он показывает в контейнере только один, первый (first), компонент. Остальные компоненты лежат под первым в определенном порядке как игральные карты в колоде. Их расположение определяется порядком, в котором написаны методы `add()`. Следующий компонент можно показать методом `next(Container c)`, предыдущий — методом `previous(Container c)`, последний — методом `last(Container c)`, первый — методом `first(Container c)`. Аргумент этих методов — ссылка на контейнер, в который помещены компоненты, обычно `this`.

В классе два конструктора:

- `CardLayout()` — не отделяет компонент от границ контейнера;
- `CardLayout(int hgap, int vgap)` — задает горизонтальные `hgap` и вертикальные `vgap` поля.

Менеджер `CardLayout` позволяет организовать и произвольный доступ к компонентам. Метод `add()` для менеджера `CardLayout` имеет своеобразный вид:

```
add(Component comp, Object constraints)
```

Здесь аргумент `constraints` должен иметь тип `String` и содержать имя компонента.

Нужный компонент с именем `name` можно показать методом:

```
show(Container parent, String name)
```

В листинге 11.5 менеджер размещения `cl` работает с панелью `p`, помещенной в "центр" контейнера `Frame`. Панель `p` указывается как аргумент `parent` в методах `next()` и `show()`. На "север" контейнера `Frame` отправлена панель `p2` с меткой и раскрывающимся списком `ch`. Рис. 11.5 демонстрирует результат работы программы.

Листинг 11.5. Менеджер `CardLayout`

```
import java.awt.*;
import java.awt.event.*;

class CardTest extends Frame{
  CardTest(String s){
    super(s);

    Panel p = new Panel();
    CardLayout cl = new CardLayout();
    p.setLayout(cl);
    p.add(new Button("Русская страница"),"page1");
    p.add(new Button("English page"),     "page2");
    p.add(new Button("Deutsche Seite"),   "page3");
    add(p);
    cl.next(p);
    cl.show(p, "page1");

    Panel p2 = new Panel();
    p2.add(new Label("Выберите язык:"));

    Choice ch = new Choice();
    ch.add("Русский");
    ch.add("Английский");
    ch.add("Немецкий");

    p2.add(ch);
    add(p2, BorderLayout.NORTH);

    setSize(400, 300);
    setVisible(true);
  }
```

```
  public static void main(String[] args){
    Frame f= new CardTest("  Менеджер CardLayout");
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

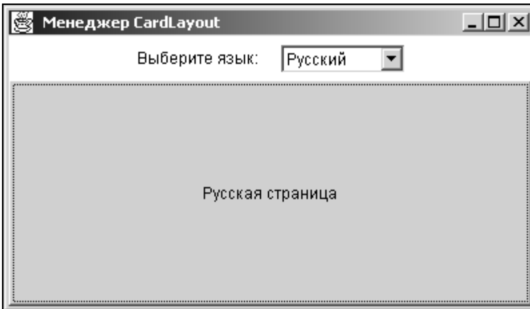


Рис. 11.5. Менеджер размещения `CardLayout`

Менеджер *GridBagLayout*

Менеджер размещения `GridBagLayout` расставляет компоненты наиболее гибко, позволяя задавать размеры и положение каждого компонента. Но он оказался очень сложным и применяется редко.

В классе `GridBagLayout` есть только один конструктор по умолчанию, без аргументов. Менеджер класса `GridBagLayout`, в отличие от других менеджеров размещения, не содержит правил размещения. Он играет только организующую роль. Ему передаются ссылка на компонент и правила расположения этого компонента, а сам он помещает данный компонент по указанным правилам в контейнер. Все правила размещения компонентов задаются в объекте другого класса, `GridBagConstraints`.

Менеджер размещает компоненты в таблице с неопределенным заранее числом строк и столбцов. Один компонент может занимать несколько ячеек этой таблицы, заполнять ячейку целиком, располагаться в ее центре, углу или прижиматься к краю ячейки.

Класс `GridBagConstraints` содержит одиннадцать полей, определяющих размеры компонентов, их положение в контейнере и взаимное положение, и несколько констант — значений некоторых полей. Они перечислены в табл. 11.1. Эти параметры определяются конструктором, имеющим одиннадцать аргументов. Второй конструктор — конструктор по умолчанию — присваивает параметрам значения, заданные по умолчанию.

Таблица 11.1. Поля класса `GridBagConstraints`

Поле	Значение
anchor	Направление размещения компонента в контейнере. Константы: CENTER, NORTH, EAST, NORTHEAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, и NORTHWEST; по умолчанию CENTER
fill	Растяжение компонента для заполнения ячейки. Константы: NONE, HORIZONTAL, VERTICAL, BOTH; по умолчанию NONE
gridheight	Количество ячеек в колонке, занимаемых компонентом. Целое типа int, по умолчанию 1. Константа REMAINDER означает, что компонент займет остаток колонки, RELATIVE — будет следующим по порядку в колонке
gridwidth	Количество ячеек в строке, занимаемых компонентом. Целое типа int, по умолчанию 1. Константа REMAINDER означает, что компонент займет остаток строки, RELATIVE — будет следующим в строке по порядку
gridx	Номер ячейки в строке. Самая левая ячейка имеет номер 0. По умолчанию константа RELATIVE, что означает: следующая по порядку
gridy	Номер ячейки в столбце. Самая верхняя ячейка имеет номер 0. По умолчанию константа RELATIVE, что означает: следующая по порядку
insets	Поля в контейнере. Объект класса Insets; по умолчанию объект с нулями
ipadx, ipady	Горизонтальные и вертикальные поля вокруг компонентов; по умолчанию 0
weightx, weighty	Пропорциональное растяжение компонентов при изменении размера контейнера; по умолчанию 0,0

Как правило, объект класса `GridBagConstraints` создается конструктором по умолчанию, затем значения нужных полей меняются простым присваиванием новых значений, например:

```
GridBagConstraints gbc = new GridBagConstraints();
gbc.weightx = 1.0;
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.gridheight = 2;
```

После создания объекта `gbc` класса `GridBagConstraints` менеджеру размещения указывается, что при помещении компонента `comp` в контейнер следует применять правила, занесенные в объект `gbc`. Для этого применяется метод

```
add(Component comp, GridBagConstraints gbc)
```

Итак, схема применения менеджера `GridBagLayout` такова:

```
GridBagLayout gbl = new GridBagLayout(); // Создаем менеджер
setLayout(gbl);                          // Устанавливаем его в контейнер
// Задаем правила размещения по умолчанию
GridBagConstraints c = new GridBagConstraints();
Button b1 = new Button();        // Создаем компонент
c.gridwidth = 2;                 // Меняем правила размещения
add(b1, c);                      // Помещаем компонент b1 в контейнер
                                 // по указанным правилам размещения c
Button b2 = new Button();        // Создаем следующий компонент
c.gridwidth = 1;                 // Меняем правила для его размещения
add(b2, c);                      // Помещаем в контейнер
```

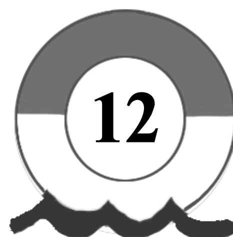
и т. д.

В документации к классу `GridBagLayout` приведен хороший пример использования этого менеджера размещения.

Заключение

Все менеджеры размещения написаны полностью на языке Java, в состав SUN J2SDK входят их исходные тексты. Если вы решили написать свой менеджер размещения, реализовав интерфейс `LayoutManager` или `LayoutManager2`, то посмотрите эти исходные тексты.

ГЛАВА 12



Обработка событий

В двух предыдущих главах мы написали много программ, создающих интерфейсы, но, собственно, интерфейса, т. е. взаимодействия с пользователем, эти программы не обеспечивают. Можно щелкать по кнопке на экране, она будет "вдавливаться" в плоскость экрана, но больше ничего не будет происходить. Можно ввести текст в поле ввода, но он не станет восприниматься и обрабатываться программой. Все это происходит из-за того, что мы не задали обработку действий пользователя, обработку событий.

Событие (event) в библиотеке AWT возникает при воздействии на компонент какими-нибудь манипуляциями мышью, при вводе с клавиатуры, при перемещении окна, изменении его размеров.

Объект, в котором произошло событие, называется *источником* (source) события.

Все события в AWT классифицированы. При возникновении события исполняющая система Java автоматически создает объект соответствующего событию класса. Этот объект не производит никаких действий, он только хранит все сведения о событии.

Во главе иерархии классов-событий стоит класс `EventObject` из пакета `java.util` — непосредственное расширение класса `Object`. Его расширяет абстрактный класс `AWTEvent` из пакета `java.awt` — глава классов, описывающих события библиотеки AWT. Дальнейшая иерархия классов-событий показана на рис. 12.1. Все классы, отображенные на рисунке, кроме класса `AWTEvent`, собраны в пакет `java.awt.event`.

События типа `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent` возникают во всех компонентах.

А события типа `ContainerEvent` — только в контейнерах: `Container`, `Dialog`, `FileDialog`, `Frame`, `Panel`, `ScrollPane`, `Window`.

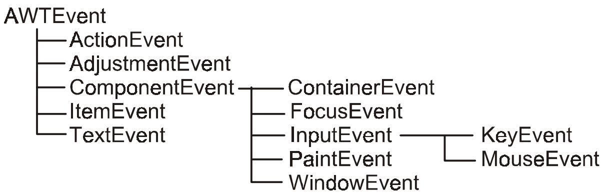


Рис. 12.1. Иерархия классов, описывающих события AWT

События типа `WindowEvent` возникают только в окнах: `Frame`, `Dialog`, `FileDialog`, `Window`.

События типа `TextEvent` генерируются только в контейнерах `TextComponent`, `TextArea`, `TextField`.

События типа `ActionEvent` проявляются только в контейнерах `Button`, `List`, `TextField`.

События типа `ItemEvent` возникают только в контейнерах `Checkbox`, `Choice`, `List`.

Наконец, события типа `AdjustmentEvent` возникают только в контейнере `Scrollbar`.

Узнать, в каком объекте произошло событие, можно методом `getSource()` класса `EventObject`. Этот метод возвращает тип `Object`.

В каждом из этих классов-событий определен метод `paramString()`, возвращающий содержимое объекта данного класса в виде строки `String`. Кроме того, в каждом классе есть свои методы, предоставляющие те или иные сведения о событии. В частности, метод `getID()` возвращает *идентификатор* (identifier) события — целое число, обозначающее тип события. Идентификаторы события определены в каждом классе-событии как константы.

Методы обработки событий описаны в интерфейсах-*слушателях* (listener). Для каждого показанного на рис. 12.1 типа событий, кроме `InputEvent` (это событие редко используется самостоятельно), есть свой интерфейс. Имена интерфейсов составляются из имени события и слова Listener, например, `ActionListener`, `MouseListener`. Методы интерфейса "слушают", что происходит в потенциальном источнике события. При возникновении события эти методы автоматически выполняются, получая в качестве аргумента объект-событие и используя при обработке сведения о событии, содержащиеся в этом объекте.

Чтобы задать обработку события определенного типа, надо реализовать соответствующий интерфейс. Классы, реализующие такой интерфейс, классы-обработчики (handlers) события, называются *слушателями* (listeners): они "слушают", что происходит в объекте, чтобы отследить возникновение события и обработать его.

Чтобы связаться с обработчиком события, классы-источники события должны получить ссылку на экземпляр `eventHandler` класса-обработчика события

одним из методов `addXxxListener(XxxEvent eventHandler)`, где *Xxx* — имя события.

Такой способ регистрации, при котором слушатель оставляет "визитную карточку" источнику для своего вызова при наступлении события, называется *обратный вызов* (callback). Им часто пользуются студенты, которые, звоня родителям и не желая платить за телефонный разговор, говорят: "Перезвони мне по такому-то номеру".

Обратное действие — отказ от обработчика, прекращение прослушивания — выполняется методом `removeXxxListener()`.

Таким образом, компонент-источник, в котором произошло событие, не занимается его обработкой. Он обращается к экземпляру класса-слушателя, умеющего обрабатывать события, *делегирует* (delegate) ему полномочия по обработке.

Такая схема получила название схемы *делегирования* (delegation). Она удобна тем, что мы можем легко сменить класс-обработчик и обработать событие по-другому или назначить несколько обработчиков одного и того же события. С другой стороны, мы можем один обработчик назначить на прослушивание нескольких объектов-источников событий.

Эта схема кажется слишком сложной, но мы ей часто пользуемся в жизни. Допустим, мы решили оборудовать квартиру. Мы помещаем в нее, как в контейнер, разные компоненты: мебель, сантехнику, электронику, антиквариат. Мы предполагаем, что может произойти неприятное событие — квартиру посетят воры, — и хотим его обработать. Мы знаем, что классы-обработчики этого события — охранные агентства, — и обращаемся к некоторому экземпляру такого класса. Компоненты-источники события, т. е. те, которые могут быть украдены, присоединяют к себе датчики методом `addXxxListener()`. Затем экземпляр-обработчик "слушает", что происходит в объектах, к которым он подключен. Он реагирует на наступление только одного события — похищения прослушиваемого объекта, — прочие события, например, короткое замыкание или обрыв водопроводной трубы, его не интересуют. При наступлении "своего" события он действует по контракту, записанному в методе обработки.

Замечание

В JDK 1.0 была принята другая модель обработки событий. Не удивляйтесь, читая старые книги и просматривая исходные тексты старых программ, но и не пользуйтесь старой моделью.

Приведем пример. Пусть в контейнер типа `Frame` помещено поле ввода `tf` типа `TextField`, не редактируемая область ввода `ta` типа `TextArea` и кнопка `b` типа `Button`. В поле `tf` вводится строка, после нажатия клавиши <Enter> или щелчка кнопкой мыши по кнопке `b` строка переносится в область `ta`. После этого можно снова вводить строку в поле `tf` и т. д.

Здесь и при нажатии клавиши <Enter> и при щелчке кнопкой мыши возникает событие класса `ActionEvent`, причем оно может произойти в двух компонентах-источниках: поле `tf` или кнопке `b`. Обработка события в обоих случаях заключается в получении строки текста из поля `tf` (например, методом `tf.getText()`) и помещения ее в область `ta` (скажем, методом `ta.append()`). Значит, можно написать один обработчик события `ActionEvent`, реализовав соответствующий интерфейс, который называется `ActionListener`. В этом интерфейсе всего один метод `actionPerformed()`.

Итак, пишем:

```
class TextMove implements ActionListener{
  private TextField tf;
  private TextArea ta;
  TextMove(TextField tf, TextArea ta){
    this.tf = tf; this.ta = ta;
  }
  public void actionPerformed(ActionEvent ae){
     ta.append(tf.getText()+"\n");
  }
}
```

Обработчик событий готов. При наступлении события типа `ActionEvent` будет создан экземпляр класса-обработчика `TextMove`, конструктор получит ссылки на конкретные поля объекта-источника, метод `actionPerformed()`, автоматически включившись в работу, перенесет текст из одного поля в другое.

Теперь напишем класс-контейнер, в котором находятся источники `tf` и `b` события `ActionEvent`, и подключим к ним слушателя этого события `TextMove`, передав им ссылки на него методом `addActionListener()`, как показано в листинге 12.1.

Листинг 12.1. Обработка события `ActionEvent`

```
import java.awt.*;
import java.awt.event.*;

class MyNotebook extends Frame{
  MyNotebook(String title){
    super(title);

    TextField tf = new TextField("Вводите текст", 50);
    add(tf, BorderLayout.NORTH);

    TextArea ta = new TextArea();
    ta.setEditable(false);
    add(ta);
```

```
    Panel p = new Panel();
    add(p, BorderLayout.SOUTH);

    Button b = new Button("Перенести");
    p.add(b);

    tf.addActionListener(new TextMove(tf, ta));
     b.addActionListener(new TextMove(tf, ta));

    setSize(300, 200);
    setVisible(true);
  }
  public static void main(String[] args){
    Frame f = new MyNotebook("  Обработка ActionEvent");
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
// Текст класса TextMove
// ...
```

На рис. 12.2 показан результат работы с этой программой.

В листинге 12.1 в методах `addActionListener()` создаются два экземпляра класса `TextMove` — для прослушивания поля `tf` и для прослушивания кнопки `b`. Можно создать один экземпляр класса `TextMove`, он будет прослушивать оба компонента:

```
TextMove tml = new TextMove(tf, ta);
tf.addActionListener(tml);
b.addActionListener(tml);
```

Но в первом случае экземпляры создаются после наступления события в соответствующем компоненте, а во втором — независимо от того, наступило событие или нет, что приводит к расходу памяти, даже если событие не произошло. Решайте сами, что лучше.

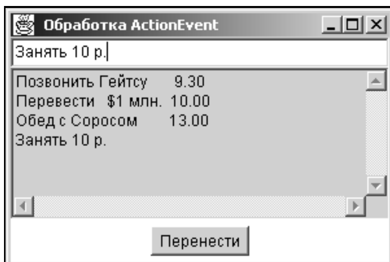


Рис. 12.2. Обработка события `ActionEvent`

Класс, содержащий источники события, может сам обрабатывать его. Вы можете самостоятельно прослушивать компоненты в своей квартире, установив пульт сигнализации у кровати.

Для этого достаточно реализовать соответствующий интерфейс прямо в классе-контейнере, как показано в листинге 12.2.

Листинг 12.2. Самообработка события `ActionEvent`

```
import java.awt.*;
import java.awt.event.*;

class MyNotebook extends Frame implements ActionListener{
  private TextField tf;
  private TextArea ta;
  MyNotebook(String title){
    super(title);

    tf = new TextField("Вводите текст", 50);
    add(tf, BorderLayout.NORTH);

    ta = new TextArea();
    ta.setEditable(false);
    add(ta);

    Panel p = new Panel();
    add(p, BorderLayout.SOUTH);

    Button b = new Button("Перенести");
    p.add(b);

    tf.addActionListener(this);
     b.addActionListener(this);

    setSize(300, 200);
    setVisible(true);
  }
  public void actionPerformed(ActionEvent ae){
     ta.append(tf.getText()+"\n");
  }
  public static void main(String[] args){
    Frame f = new MyNotebook("  Обработка ActionEvent");
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

Здесь `tf` и `ta` уже не локальные переменные, а переменные экземпляра, поскольку они используются и в конструкторе, и в методе `actionPerformed()`. Этот метод теперь — один из методов класса `MyNotebook`. Класс `MyNotebook` стал классом-обработчиком события `ActionEvent` — он реализует интерфейс `ActionListener`. В методе `addActionListener()` указывается аргумент `this` — класс сам слушает свои компоненты.

Рассмотренная схема, кажется, проще и удобнее, но она предоставляет меньше возможностей. Если вы захотите изменить обработку, например заносить записи в поле `ta` по алфавиту или по времени выполнения заданий, то придется переписать и перекомпилировать класс `MyNotebook`.

Еще один вариант — сделать обработчик вложенным классом. Это позволяет обойтись без переменных экземпляра и конструктора в классе-обработчике `TextMove`, как показано в листинге 12.3.

Листинг 12.3. Обработка вложенным классом

```
import java.awt.*;
import java.awt.event.*;

class MyNotebook extends Frame{
  private TextField tf;
  private TextArea ta;
  MyNotebook(String title){
    super(title);
    tf = new TextField("Вводите текст", 50);
    add(tf, BorderLayout.NORTH);
    ta = new TextArea();
    ta.setEditable(false);
    add(ta);
    Panel p = new Panel();
    add(p, BorderLayout.SOUTH);
    Button b = new Button("Перенести");
    p.add(b);

    tf.addActionListener(new TextMove());
    b.addActionListener(new TextMove());

    setSize(300, 200);
    setVisible(true);
  }
  public static void main(String[] args){
    Frame f = new MyNotebook("  Обработка ActionEvent");
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
```

```
        System.exit(0);
      }
    });
  }
  // Вложенный класс
  class TextMove implements ActionListener{
    public void actionPerformed(ActionEvent ae){
       ta.append(tf.getText()+"\n");
    }
  }
}
```

Наконец, можно создать безымянный вложенный класс, что мы и делали в этой и предыдущих главах, обрабатывая нажатие комбинации клавиш <Alt>+<F4> или щелчок кнопкой мыши по кнопке закрытия окна. При этом возникает событие типа `WindowEvent`, для его обработки мы обращались к методу `windowClosing()`, реализуя его обращением к методу завершения приложения `System.exit(0)`. Но для этого нужно иметь суперкласс определяемого безымянного класса, такой как `WindowAdapter`. Такими суперклассами могут быть классы-адаптеры, о них речь пойдет чуть ниже.

Перейдем к детальному рассмотрению разных типов событий.

Событие *ActionEvent*

Это простое событие означает, что надо выполнить какое-то действие. При этом неважно, что вызвало событие: щелчок мыши, нажатие клавиши или что-то другое.

В классе `ActionEvent` есть два полезных метода:

- метод `getActionCommand()` возвращает в виде строки `String` надпись на кнопке `Button`, точнее, то, что установлено методом `setActionCommand(String s)` класса `Button`, выбранный пункт списка `List`, или что-то другое, зависящее от компонента;
- метод `getModifiers()` возвращает код клавиш <Alt>, <Ctrl>, <Meta> или <Shift>, если какая-нибудь одна или несколько из них были нажаты, в виде числа типа `int`; узнать, какие именно клавиши были нажаты, можно сравнением со статическими константами этого класса `ALT_MASK`, `CTRL_MASK`, `META_MASK`, `SHIFT_MASK`.

Примечание

Клавиши <Meta> на PC-клавиатуре нет, ее действие часто назначается на клавишу <Esc> или левую клавишу <Alt>.

Например:

```
public void actionPerformed(ActionEvent ae){
   if (ae.getActionCommand() == "Open" &&
            (ae.getModifiers() | ActionEvent.ALT_MASK) != 0){
      // Какие-то действия
   }
}
```

Обработка действий мыши

Событие MouseEvent возникает в компоненте по любой из семи причин:

- нажатие кнопки мыши — идентификатор MOUSE_PRESSED;
- отпускание кнопки мыши — идентификатор MOUSE_RELEASED;
- щелчок кнопкой мыши — идентификатор MOUSE_CLICKED (нажатие и отпускание не различаются);
- перемещение мыши — идентификатор MOUSE_MOVED;
- перемещение мыши с нажатой кнопкой — идентификатор MOUSE_DRAGGED;
- появление курсора мыши в компоненте — идентификатор MOUSE_ENTERED;
- выход курсора мыши из компонента — идентификатор MOUSE_EXITED.

Для их обработки есть семь методов в двух интерфейсах:

```
public interface MouseListener extends EventListener{
    public void mouseClicked(MouseEvent e);
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
}
public interface MouseMotionListener extends EventListener{
    public void mouseDragged(MouseEvent e);
    public void mouseMoved(MouseEvent e);
}
```

Эти методы могут получить от аргумента e координаты курсора мыши в системе координат компонента методами e.getX(), e.getY(), или одним методом e.getPoint(), возвращающим экземпляр класса Point.

Двойной щелчок кнопкой мыши можно отследить методом e.getClickCount(), возвращающим количество щелчков. При перемещении мыши возвращается 0.

Узнать, какая кнопка была нажата, можно с помощью метода e.getModifiers() класса InputEvent сравнением со следующими статическими константами класса InputEvent:

- BUTTON1_MASK — нажата первая кнопка, обычно левая;
- BUTTON2_MASK — нажата вторая кнопка, обычно средняя, или одновременно нажаты обе кнопки на двухкнопочной мыши;
- BUTTON3_MASK — нажата третья кнопка, обычно правая.

Приведем пример, уже ставший классическим. В листинге 12.4 представлен простейший вариант "рисовалки" — класс Scribble. При нажатии первой кнопки мыши методом mousePressed() запоминаются координаты курсора мыши. При протаскивании мыши вычерчиваются отрезки прямых между текущим и предыдущим положением курсора мыши методом mouseDragged(). На рис. 12.3 показан пример работы с этой программой.

Листинг 12.4. Простейшая программа рисования

```
import java.awt.*;
import java.awt.event.*;

public class ScribbleTest extends Frame{
  public ScribbleTest(String s){
    super(s);

    ScrollPane pane = new ScrollPane();
    pane.setSize(300, 300);
    add(pane, BorderLayout.CENTER);

    Scribble scr = new Scribble(this, 500, 500);
    pane.add(scr);

    Panel p = new Panel();
    add(p, BorderLayout.SOUTH);

    Button b1 = new Button("Красный");  p.add(b1);
    b1.addActionListener(scr);
    Button b2 = new Button("Зеленый");  p.add(b2);
    b2.addActionListener(scr);
    Button b3 = new Button("Синий");    p.add(b3);
    b3.addActionListener(scr);
    Button b4 = new Button("Черный");   p.add(b4);
    b4.addActionListener(scr);
    Button b5 = new Button("Очистить"); p.add(b5);
    b5.addActionListener(scr);

    addWindowListener(new WindowAdapter() {
       public void windowClosing(WindowEvent e){
         System.exit(0);
       }
    });
```

```
    pack();
    setVisible(true);
  }
  public static void main(String[] args){
     new ScribbleTest("  \"Рисовалка\"");
  }
}

class Scribble extends Component implements
        ActionListener, MouseListener, MouseMotionListener{
  protected int lastX, lastY, w, h;
  protected Color currColor = Color.black;
  protected Frame f;

  public Scribble(Frame frame, int width, int height){
    f = frame; w = width; h = height;
    enableEvents(AWTEvent.MOUSE_EVENT_MASK |
                 AWTEvent.MOUSE_MOTION_EVENT_MASK);
    addMouseListener(this);
    addMouseMotionListener(this);
  }
  public Dimension getPreferredSize(){
    return new Dimension(w, h);
  }
  public void actionPerformed(ActionEvent event){
    String s = event.getActionCommand();
    if (s.equals("Очистить")) repaint();
    else if (s.equals("Красный")) currColor = Color.red;
    else if (s.equals("Зеленый")) currColor = Color.green;
    else if (s.equals("Синий"))   currColor = Color.blue;
    else if (s.equals("Черный"))  currColor = Color.black;
  }
  public void mousePressed(MouseEvent e){
    if ((e.getModifiers() & MouseEvent.BUTTON1_MASK) == 0) return;
    lastX = e.getX(); lastY = e.getY();
  }
  public void mouseDragged(MouseEvent e){
    if ((e.getModifiers() & MouseEvent.BUTTON1_MASK) == 0) return;
    Graphics g = getGraphics();
    g.setColor(currColor);
    g.drawLine(lastX, lastY, e.getX(), e.getY());
    lastX = e.getX(); lastY = e.getY();
  }
  public void mouseReleased(MouseEvent e){}
  public void mouseClicked(MouseEvent e){}
```

```
  public void mouseEntered(MouseEvent e){}
  public void mouseExited(MouseEvent e){}
  public void mouseMoved(MouseEvent e){}
}
```

Рис. 12.3. Пример работы с программой рисования

При создании класса-слушателя `Scribble` и реализации интерфейсов `MouseListener` и `MouseMotionListener` пришлось реализовать все их семь методов, хотя мы отслеживали только нажатие и перемещение мыши, и нам нужны были только методы `mousePressed()` и `mouseDragged()`. Для остальных методов мы задали пустые реализации.

Чтобы облегчить задачу реализации интерфейсов, имеющих более одного метода, созданы классы-адаптеры.

Классы-адаптеры

Классы-адаптеры представляют собой пустую реализацию интерфейсов-слушателей, имеющих более одного метода. Их имена составляются из имени события и слова Adapter. Например, для действий с мышью есть два класса-адаптера. Выглядят они очень просто:

```
public abstract class MouseAdapter implements MouseListener{
    public void mouseClicked(MouseEvent e){}
    public void mousePressed(MouseEvent e){}
    public void mouseReleased(MouseEvent e){}
    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}
}
public abstract class MouseMotionAdapter implements MouseMotionListener{
    public void mouseDragged(MouseEvent e){}
    public void mouseMoved(MouseEvent e){}
}
```

Вместо того чтобы реализовать интерфейс, можно расширять эти классы. Не бог весть что, но полезно для создания безымянного вложенного класса, как у нас и делалось для закрытия окна. Там мы использовали класс-адаптер `WindowAdapter`.

Классов-адаптеров всего семь. Кроме уже упомянутых трех классов, это классы `ComponentAdapter`, `ContainerAdapter`, `FocusAdapter` и `KeyAdapter`.

Обработка действий клавиатуры

Событие `KeyEvent` происходит в компоненте по любой из трех причин:

- нажата клавиша — идентификатор `KEY_PRESSED`;
- отпущена клавиша — идентификатор `KEY_RELEASED`;
- введен символ — идентификатор `KEY_TYPED`.

Последнее событие возникает из-за того, что некоторые символы вводятся нажатием нескольких клавиш, например, заглавные буквы вводятся комбинацией клавиш <Shift>+<буква>. Вспомните еще <Alt>-ввод в MS Windows. Нажатие функциональных клавиш, например <F1>, не вызывает событие `KEY_TYPED`.

Обрабатываются эти события тремя методами, описанными в интерфейсе:

```
public interface KeyListener extends EventListener{
    public void keyTyped(KeyEvent e);
    public void keyPressed(KeyEvent e);
    public void keyReleased(KeyEvent e);
}
```

Аргумент `e` этих методов может дать следующие сведения.

Метод `e.getKeyChar()` возвращает символ Unicode типа `char`, связанный с клавишей. Если с клавишей не связан никакой символ, то возвращается константа `CHAR_UNDEFINED`.

Метод `e.getKeyCode()` возвращает код клавиши в виде целого числа типа `int`. В классе `KeyEvent` определены коды всех клавиш в виде констант, называемых ***виртуальными кодами*** клавиш (virtual key codes), например, `VK_F1`, `VK_SHIFT`, `VK_A`, `VK_B`, `VK_PLUS`. Они перечислены в документации к классу `KeyEvent`. Фактическое значение виртуального кода зависит от языка и раскладки клавиатуры. Чтобы узнать, какая клавиша была нажата, надо сравнить результат выполнения метода `getKeyCode()` с этими константами. Если кода клавиши нет, как происходит при наступлении события `KEY_TYPED`, то возвращается значение `VK_UNDEFINED`.

Чтобы узнать, не нажата ли одна или несколько клавиш-модификаторов <Alt>, <Ctrl>, <Meta>, <Shift>, надо воспользоваться унаследованным от

класса InputEvent методом getModifiers() и сравнить его результат с константами ALT_MASK, CTRL_MASK, META_MASK, SHIFT_MASK. Другой способ — применить логические методы isAltDown(), isControlDown(), isMetaDown(), isShiftDown().

Добавим в листинг 12.3 возможность очистки поля ввода tf после нажатия клавиши <Esc>. Для этого перепишем вложенный класс-слушатель TextMove:

```
class TextMove implements ActionListener, KeyListener{
  public void actionPerformed(ActionEvent ae){
     ta.append(tf.getText()+"\n");
  }
  public void keyPressed(KeyEvent ke){
     if (ke.getKeyCode() == KeyEvent.VK_ESCAPE) tf.setText("");
  }
  public void keyReleased(KeyEvent ke){}
  public void keyTyped(KeyEvent ke){}
}
```

Событие *TextEvent*

Событие TextEvent происходит только по одной причине — изменению текста — и отмечается идентификатором TEXT_VALUE_CHANGED.

Соответствующий интерфейс имеет только один метод:

```
public interface TextListener extends EventListener{
    public void textValueChanged(TextEvent e);
}
```

От аргумента e этого метода можно получить ссылку на объект-источник события методом getSource(), унаследованным от класса EventObject, например, так:

```
TextComponent tc = (TextComponent)e.getSource();
String s = tc.getText();
// Дальнейшая обработка
```

Обработка действий с окном

Событие WindowEvent может произойти по семи причинам:

- окно открылось — идентификатор WINDOW_OPENED;
- окно закрылось — идентификатор WINDOW_CLOSED;
- попытка закрытия окна — идентификатор WINDOW_CLOSING;

- окно получило фокус — идентификатор `WINDOW_ACTIVATED`;
- окно потеряло фокус — идентификатор `WINDOW_DEACTIVATED`;
- окно свернулось в ярлык — идентификатор `WINDOW_ICONIFIED`;
- окно развернулось — идентификатор `WINDOW_DEICONIFIED`.

Соответствующий интерфейс содержит семь методов:

```
public interface WindowListener extends EventListener {
    public void windowOpened(WindowEvent e);
    public void windowClosing(WindowEvent e);
    public void windowClosed(WindowEvent e);
    public void windowIconified(WindowEvent e);
    public void windowDeiconified(WindowEvent e);
    public void windowActivated(WindowEvent e);
    public void windowDeactivated(WindowEvent e);
}
```

Аргумент `e` этих методов дает ссылку типа `Window` на окно-источник методом `e.getWindow()`.

Чаще всего эти события используются для перерисовки окна методом `repaint()` при изменении его размеров и для остановки приложения при закрытии окна.

Событие *ComponentEvent*

Данное событие происходит в компоненте по четырем причинам:

- компонент перемещается — идентификатор `COMPONENT_MOVED`;
- компонент меняет размер — идентификатор `COMPONENT_RESIZED`;
- компонент убран с экрана — идентификатор `COMPONENT_HIDDEN`;
- компонент появился на экране — идентификатор `COMPONENT_SHOWN`.

Соответствующий интерфейс содержит описания четырех методов:

```
public interface ComponentListener extends EventListener{
    public void componentResized(ComponentEvent e);
    public void componentMoved(ComponentEvent e);
    public void componentShown(ComponentEvent e);
    public void componentHidden(ComponentEvent e);
}
```

Аргумент `e` методов этого интерфейса предоставляет ссылку на компонент-источник события методом `e.getComponent()`.

Событие *ContainerEvent*

Это событие происходит по двум причинам:

- в контейнер добавлен компонент — идентификатор `COMPONENT_ADDED`;
- из контейнера удален компонент — идентификатор `COMPONENT_REMOVED`.

Этим причинам соответствуют методы интерфейса:

```
public interface ContainerListener extends EventListener{
    public void componentAdded(ContainerEvent e);
    public void componentRemoved(ContainerEvent e);
}
```

Аргумент `e` предоставляет ссылку на компонент, чье добавление или удаление из контейнера вызвало событие, методом `e.getChild()`, и ссылку на контейнер — источник события методом `e.getContainer()`. Обычно при наступлении данного события контейнер перемещает свои компоненты.

Событие *FocusEvent*

Событие возникает в компоненте, когда он получает фокус ввода — идентификатор `FOCUS_GAINED`, или теряет фокус — идентификатор `FOCUS_LOST`.

Соответствующий интерфейс:

```
public interface FocusListener extends EventListener{
    public void focusGained(FocusEvent e);
    public void focusLost(FocusEvent e);
}
```

Обычно при потере фокуса компонент перечерчивается бледным цветом, для этого применяется метод `brighter()` класса `Color`, при получении фокуса становится ярче, что достигается применением метода `darker()`. Это приходится делать самостоятельно при создании своего компонента.

Событие *ItemEvent*

Это событие возникает при выборе или отказе от выбора элемента в списке `List`, `Choice` или флажка `Checkbox` и отмечается идентификатором `ITEM_STATE_CHANGED`.

Соответствующий интерфейс очень прост:

```
public interface ItemListener extends EventListener{
    void itemStateChanged(ItemEvent e);
}
```

Аргумент `e` предоставляет ссылку на источник методом `e.getItemSelectable()`, ссылку на выбранный пункт методом `e.getItem()` в виде `Object`.

Метод `e.getStateChange()` позволяет уточнить, что произошло: значение `SELECTED` указывает на то, что элемент был выбран, значение `DESELECTED` — произошел отказ от выбора.

В следующей главе мы рассмотрим примеры использования этого события.

Событие *AdjustmentEvent*

Это событие возникает для полосы прокрутки `Scrollbar` при всяком изменении ее бегунка и отмечается идентификатором `ADJUSTMENT_VALUE_CHANGED`.

Соответствующий интерфейс описывает один метод:

```
public interface AdjustmentListener extends EventListener{
    public void adjustmentValueChanged(AdjustmentEvent e);
}
```

Аргумент `e` этого метода предоставляет ссылку на источник события методом `e.getAdjustable()`, текущее значение положения движка полосы прокрутки методом `e.getValue()`, и способ изменения его значения методом `e.getAdjustmentType()`, возвращающим следующие значения:

- `UNIT_INCREMENT` — увеличение на одну единицу;
- `UNIT_DECREMENT` — уменьшение на одну единицу;
- `BLOCK_INCREMENT` — увеличение на один блок;
- `BLOCK_DECREMENT` — уменьшение на один блок;
- `TRACK` — процесс передвижения бегунка полосы прокрутки.

"Оживим" программу создания цвета, приведенную в листинге 10.4, добавив необходимые действия. Результат этого приведен в листинге 12.5.

Листинг 12.5. Программа создания цвета

```
import java.awt.*;
import java.awt.event.*;

class ScrollTest1 extends Frame{
  private Scrollbar
    sbRed   = new Scrollbar(Scrollbar.VERTICAL, 127, 16, 0, 271),
    sbGreen = new Scrollbar(Scrollbar.VERTICAL, 127, 16, 0, 271),
    sbBlue  = new Scrollbar(Scrollbar.VERTICAL, 127, 16, 0, 271);
  private Color c  = new Color(127, 127, 127);
  private Label lm = new Label();
```

```
private Button
  b1 = new Button("Применить"),
  b2 = new Button("Отменить");

ScrollTest1(String s){
  super(s);
  setLayout(null);
  setFont(new Font("Serif", Font.BOLD, 15));

  Panel p = new Panel();
  p.setLayout(null);
  p.setBounds(10,50, 150, 260); add(p);

  Label lc = new Label("Подберите цвет");
  lc.setBounds(20, 0, 120, 30); p.add(lc);
  Label lmin = new Label("0", Label.RIGHT);
  lmin.setBounds(0, 30, 30, 30); p.add(lmin);
  Label lmiddle = new Label("127", Label.RIGHT);
  lmiddle.setBounds(0, 120, 30, 30); p.add(lmiddle);
  Label lmax = new Label("255", Label.RIGHT);
  lmax.setBounds(0, 200, 30, 30); p.add(lmax);

  sbRed.setBackground(Color.red);
  sbRed.setBounds(40, 30, 20, 200); p.add(sbRed);
  sbRed.addAdjustmentListener(new ChColor());

  sbGreen.setBackground(Color.green);
  sbGreen.setBounds(70, 30, 20, 200); p.add(sbGreen);
  sbGreen.addAdjustmentListener(new ChColor());

  sbBlue.setBackground(Color.blue);
  sbBlue.setBounds(100, 30, 20, 200); p.add(sbBlue);
  sbBlue.addAdjustmentListener(new ChColor());

  Label lp = new Label("Образец:");
  lp.setBounds(250, 50, 120, 30); add(lp);

  lm.setBackground(new Color(127, 127, 127));
  lm.setBounds(220, 80, 120, 80); add(lm);

  b1.setBounds(240, 200, 100, 30); add(b1);
  b1.addActionListener(new ApplyColor());

  b2.setBounds(240, 240, 100, 30); add(b2);
  b2.addActionListener(new CancelColor());

  setSize(400, 300);
  setVisible(true);
}
```

```
  class ChColor implements AdjustmentListener{
    public void adjustmentValueChanged(AdjustmentEvent e){
      int red = c.getRed(), green = c.getGreen(), blue = c.getBlue();
      if (e.getAdjustable() == sbRed)          red = e.getValue();
      else if (e.getAdjustable() == sbGreen) green = e.getValue();
      else if (e.getAdjustable() == sbBlue)   blue = e.getValue();
      c = new Color(red, green, blue);
      lm.setBackground(c);
    }
  }
  class ApplyColor implements ActionListener{
     public void actionPerformed(ActionEvent ae){
       setBackground(c);
     }
  }
  class CancelColor implements ActionListener{
     public void actionPerformed(ActionEvent ae){
       c = new Color(127, 127, 127);
         sbRed.setValue(127);
       sbGreen.setValue(127);
        sbBlue.setValue(127);
       lm.setBackground(c);
       setBackground(Color.white);
     }
  }
  public static void main(String[] args){
    Frame f = new ScrollTest1(" Выбор цвета");
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

Несколько слушателей одного источника

В начале этой главы, в листингах 12.1—12.3, мы привели пример класса `TextMove`, слушающего сразу два компонента: поле ввода `tf` типа `TextField` и кнопку `b` типа `Button`.

Чаще встречается обратная ситуация — несколько слушателей следят за одним компонентом. В том же примере кнопка `b` в ответ на щелчок по ней

кнопки мыши совершала еще и собственные действия — она "вдавливалась", а при отпускании кнопки мыши становилась "выпуклой". В классе `Button` эти действия выполняет peer-объект.

В классе `FlowerButton` листинга 10.6 такие же действия выполняет метод `paint()` этого класса.

В данной модели реализован design pattern под названием `Observer`.

К каждому компоненту можно присоединить сколько угодно слушателей одного и того же события или разных типов событий. Однако при этом не гарантируется какой-либо определенный порядок их вызова, хотя чаще всего слушатели вызываются в порядке написания методов add*Xxx*Listener().

Если нужно задать определенный порядок вызовов слушателей для обработки события, то придется обращаться к ним друг из друга или создавать объект, вызывающий слушателей в нужном порядке.

Ссылки на присоединенные методами add*Xxx*Listener() слушатели можно было бы хранить в любом классе-коллекции, например, `Vector`, но в пакет `java.awt` специально для этого введен класс `AWTEventMulticaster`. Он реализует все одиннадцать интерфейсов *Xxx*Listener, значит, сам является слушателем любого события. Основу класса составляют своеобразные статические методы `add()`, написанные для каждого типа событий, например:

```
add(ActionListener a, ActionListener b)
```

Своеобразие этих методов двоякое: они возвращают ссылку на тот же интерфейс, в данном случае, `ActionListener`, и присоединяют объект `a` к объекту `b`, создавая совокупность слушателей одного и того же типа. Это позволяет использовать их наподобие операций `a += b`. Заглянув в исходный текст класса `Button`, вы увидите, что метод `addActionListener()` очень прост:

```
public synchronized void addActionListener(ActionListener l){
   if (l == null){ return; }
   actionListener = AWTEventMulticaster.add(actionListener, l);
   newEventsOnly = true;
}
```

Он добавляет к совокупности слушателей `actionListener` нового слушателя `l`.

Для событий типа `InputEvent`, а именно, `KeyEvent` и `MouseEvent`, есть возможность прекратить дальнейшую обработку события методом `consume()`. Если записать вызов этого метода в класс-слушатель, то ни peer-объекты, ни следующие слушатели не будут обрабатывать событие. Этим способом обычно пользуются, чтобы отменить стандартные действия компонента, например, "вдавливание" кнопки.

Диспетчеризация событий

Если вам понадобится обработать просто действие мыши, не важно, нажатие это, перемещение или еще что-нибудь, то придется включать эту обработку во все семь методов двух классов-слушателей событий мыши.

Эту работу можно облегчить, выполнив обработку не в слушателе, а на более ранней стадии. Дело в том, что прежде чем событие дойдет до слушателя, оно обрабатывается несколькими методами.

Чтобы в компоненте произошло событие AWT, должно быть выполнено хотя бы одно из двух условий: к компоненту присоединен слушатель или в конструкторе компонента определена возможность появления события методом `enableEvents()`. В аргументе этого метода через операцию побитового сложения перечисляются константы класса `AWTEvent`, задающие события, которые могут произойти в компоненте, например:

```
enableEvents(AWTEvent.MOUSE_MOTION_EVENT_MASK |
   AWTEvent.MOUSE_EVENT_MASK | AWTEvent.KEY_EVENT_MASK)
```

При появлении события создается объект соответствующего класса *Xxx*`Event`. Метод `dispatchEvent()` определяет, где появилось событие — в компоненте или одном из его подкомпонентов, — и передает объект-событие методу `processEvent()` компонента-источника.

Метод `processEvent()` определяет тип события и передает его специализированному методу `process`*Xxx*`Event()`. Вот начало этого метода:

```
protected void processEvent(AWTEvent e){
   if (e instanceof FocusEvent){
       processFocusEvent((FocusEvent)e);
   }else if (e instanceof MouseEvent){
       switch(e.getID()){
          case MouseEvent.MOUSE_PRESSED:
          case MouseEvent.MOUSE_RELEASED:
          case MouseEvent.MOUSE_CLICKED:
          case MouseEvent.MOUSE_ENTERED:
          case MouseEvent.MOUSE_EXITED:
             processMouseEvent((MouseEvent)e);
          break;
          case MouseEvent.MOUSE_MOVED:
          case MouseEvent.MOUSE_DRAGGED:
             processMouseMotionEvent((MouseEvent)e);
          break;
       }
   }else if (e instanceof KeyEvent){
```

```
        processKeyEvent((KeyEvent)e);
    }
// ...
```

Затем в дело вступает специализированный метод, например, `processKeyEvent()`. Он-то и передает объект-событие слушателю. Вот исходный текст этого метода:

```
protected void processKeyEvent(KeyEvent e){
   KeyListener listener = keyListener;
   if (listener != null){
      int id = e.getID();
      switch(id){
         case KeyEvent.KEY_TYPED: listener.keyTyped(e);
         break;
         case KeyEvent.KEY_PRESSED: listener.keyPressed(e);
         break;
         case KeyEvent.KEY_RELEASED: listener.keyReleased(e);
         break;
      }
   }
}
```

Из этого описания видно, что если вы хотите обработать любое событие типа `AWTEvent`, то вам надо переопределить метод `processEvent()`, а если более конкретное событие, например, событие клавиатуры, — переопределить более конкретный метод `processKeyEvent()`. Если вы не переопределяете весь метод целиком, то не забудьте в конце обратиться к методу суперкласса, например,

```
super.processKeyEvent(e);
```

Замечание

Не забывайте обращаться к методу `process`*`Xxx`*`Event()` суперкласса.

В следующей главе мы применим такое переопределение в листинге 13.2 для вызова всплывающего меню.

Создание собственного события

Вы можете создать собственное событие и определить источник и условия его возникновения.

В листинге 12.6 приведен пример создания события `MyEvent`, любезно предоставленный Вячеславом Педаком.

Событие `MyEvent` говорит о начале работы программы (`START`) и окончании ее работы (`STOP`).

Листинг 12.6. Создание собственного события

```
// 1. Создаем свой класс события:
public class MyEvent extends java.util.EventObject{
  protected int id;
  public static final int START = 0, STOP = 1;
  public MyEvent(Object source, int id){
    super(source);
    this.id = id;
  }
  public int getID(){ return id; }
}
// 2. Описываем Listener:
public interface MyListener extends java.util.EventListener{
  public void start(MyEvent e);
  public void stop(MyEvent e);
}
// 3. В теле нужного класса создаем метод fireEvent():
  protected Vector listeners = new Vector();
  public void fireEvent( MyEvent e){
    Vector list = (Vector) listeners.clone();
    for(int i = 0; i < list.size(); i++){
      MyListener listener = (MyListener)list.elementAt(i);
      switch(e.getID()){
        case MyEvent.START: listener.start(e); break;
        case MyEvent.STOP:  listener.stop(e);  break;
      }
    }
  }
```

Все, теперь при запуске программы делаем

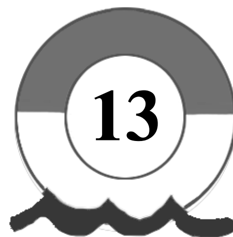
```
fireEvent(this, MyEvent.START);
```

а при окончании

```
fireEvent(this, MyEvent.STOP);
```

При этом все зарегистрированные слушатели получат экземпляры событий.

ГЛАВА 13



Создание меню

В контейнер типа `Frame` заложена возможность установки стандартной *строки меню* (menu bar), располагаемой ниже строки заголовка, как показано на рис. 13.1. Эта строка — объект класса `MenuBar`.

Все, что нужно сделать для установки строки меню в контейнере `Frame` — это создать объект класса `MenuBar` и обратиться к методу `setMenuBar()`:

```
Frame f = new Frame("Пример меню");
MenuBar mb = new MenuBar();
f.setMenuBar(mb);
```

Если имя `mb` не понадобится, можно совместить два последних обращения к методам:

```
f.setMenuBar(new MenuBar());
```

Разумеется, строка меню еще пуста и пункты меню не созданы.

Каждый элемент строки меню — *выпадающее меню* (drop-down menu) — это объект класса `Menu`. Создать эти объекты и занести их в строку меню ничуть не сложнее, чем создать строку меню:

```
Menu mFile = new Menu("Файл");
mb.add(mFile);
Menu mEdit = new Menu("Правка");
mb.add(mEdit);
Menu mView = new Menu("Вид");
mb.add(mView);
Menu mHelp = new Menu("Справка");
mb.setHelpMenu(mHelp);
```

и т. д. Элементы располагаются слева направо в порядке обращения к методам `add()`, как показано на рис. 13.1. Во многих графических системах при-

нято меню **Справка** (**Help**) прижимать к правому краю строки меню. Это достигается обращением к методу `setHelpMenu()`, но фактическое положение меню **Справка** определяется графической оболочкой.

Рис. 13.1. Система меню

Затем определяем каждое выпадающее меню, создавая его пункты. Каждый пункт меню — это объект класса `MenuItem`. Схема его создания и добавления к меню точно такая же, как и самого меню:

```
MenuItem create = new MenuItem("Создать");
mFile.add(create);
MenuItem open = new MenuItem("Открыть...");
mFile.add(open);
```

и т. д. Пункты меню будут расположены сверху вниз в порядке обращения к методам `add()`.

Часто пункты меню объединяются в группы. Одна группа от другой отделяется горизонтальной чертой. На рис. 13.1 черта проведена между командами **Открыть** и **Отправить**. Эта черта создается методом `addSeparator()` класса `Menu` или определяется как пункт меню с надписью специального вида — дефисом:

```
mFile.add(new MenuItem("-"));
```

Интересно, что класс `Menu` расширяет класс `MenuItem`, а не наоборот. Это означает, что меню само является пунктом меню, и позволяет задавать меню в качестве пункта другого меню, тем самым организуя вложенные подменю:

```
Menu send = new Menu("Отправить");
mFile.add(send);
```

Здесь меню `send` добавляется в меню `mFile` как один из его пунктов. Подменю `send` заполняется пунктами меню как обычное меню.

Часто команды меню создаются для выбора из них каких-то возможностей, подобно компонентам `Checkbox`. Такие пункты можно выделить щелчком кнопки мыши или отменить выделение повторным щелчком. Эти команды — объекты класса `CheckboxMenuItem`:

```
CheckboxMenuItem disk = new CheckboxMenuItem("Диск A:", true);
send.add(disk);
send.add(new CheckboxMenuItem("Архив"));
```

и т. д.

Все, что получилось в результате перечисленных действий, показано на рис. 13.1.

Многие графические оболочки, но не MS Windows, позволяют создавать *отсоединяемые* (tear-off) меню, которые можно перемещать по экрану. Это указывается в конструкторе

```
Menu(String label, boolean tearOff)
```

Если `tearOff == true` и графическая оболочка умеет создавать отсоединяемое меню, то оно будет создано. В противном случае этот аргумент просто игнорируется.

Наконец, надо назначить действия командам меню. Команды меню типа `MenuItem` порождают события типа `ActionEvent`, поэтому нужно присоединить к ним объект класса-слушателя как к обычным компонентам, записав что-то вроде

```
create.addActionListener(new SomeActionEventHandler())
open.addActionListener(new AnotherActionEventHandler())
```

Пункты типа `CheckboxMenuItem` порождают события типа `ItemEvent`, поэтому надо обращаться к объекту-слушателю этого события:

```
disk.addItemListener(new SomeItemEventHandler())
```

Очень часто действия, записанные в командах меню, вызываются не только щелчком кнопки мыши, но и "горячими" клавишами-акселераторами (shortcut), действующими чаще всего при нажатой клавише <Ctrl>. На экране в пунктах меню, которым назначены "горячие" клавиши, появляются подсказки вида **Ctrl+N**, **Ctrl+O**, как на рис. 13.1. "Горячая" клавиша определяется объектом класса `MenuShortcut` и указывается в его конструкторе константой класса `KeyEvent`, например:

```
MenuShortcut keyCreate = new MenuShortcut(KeyEvent.VK_N);
```

После этого "горячей" будет комбинация клавиш <Ctrl>+<N>. Затем полученный объект указывается в конструкторе класса `MenuItem`:

```
MenuItem create = new MenuItem("Создать", keyCreate);
```

Нажатие <Ctrl>+<N> будет вызывать окно создания. Эти действия, разумеется, можно совместить, например,

```
MenuItem open = new MenuItem("Открыть...",
                  new MenuShortcut(KeyEvent.VK_O));
```

Можно добавить еще нажатие клавиши <Shift>. Действие пункта меню будет вызываться нажатием комбинации клавиш <Shift>+<Ctrl>+<X>, если воспользоваться вторым конструктором:

```
MenuShortcut(int key, boolean useShift)
```

с аргументом `useShift == true`.

Программа рисования, созданная в листинге 12.4 и показанная на рис. 12.3, явно перегружена кнопками. Перенесем их действия в пункты меню. Добавим возможность манипуляции файлами и команду завершения работы. Это сделано в листинге 13.1. Класс `Scribble` не изменялся и в листинге не приведен. Результат показан на рис. 13.2.

Листинг 13.1. Программа рисования с меню

```
import java.awt.*;
import java.awt.event.*;

public class MenuScribble extends Frame{
  public MenuScribble(String s){
    super(s);

    ScrollPane pane = new ScrollPane();
    pane.setSize(300, 300);
    add(pane, BorderLayout.CENTER);

    Scribble scr = new Scribble(this, 500, 500);
    pane.add(scr);

    MenuBar mb = new MenuBar();
    setMenuBar(mb);
    Menu f = new Menu("Файл");
    Menu v = new Menu("Вид");
    mb.add(f); mb.add(v);

    MenuItem open = new MenuItem("Открыть...",
                        new MenuShortcut(KeyEvent.VK_O));
    MenuItem save = new MenuItem("Сохранить",
                        new MenuShortcut(KeyEvent.VK_S));
    MenuItem saveAs = new MenuItem("Сохранить как...");
    MenuItem exit = new MenuItem("Выход",
                        new MenuShortcut(KeyEvent.VK_Q));
    f.add(open); f.add(save); f.add(saveAs);
    f.addSeparator(); f.add(exit);

    open.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e){
```

```
        FileDialog fd = new FileDialog(new Frame(),
                        " Загрузить", FileDialog.LOAD);
        fd.setVisible(true);
      }
    });
    saveAs.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e){
        FileDialog fd = new FileDialog(new Frame(),
                        " Сохранить", FileDialog.SAVE);
        fd.setVisible(true);
      }
    });
    exit.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e){
        System.exit(0);
      }
    });
    Menu c = new Menu("Цвет");
    MenuItem clear = new MenuItem("Очистить",
                          new MenuShortcut(KeyEvent.VK_D));
    v.add(c); v.add(clear);
    MenuItem red   = new MenuItem("Красный");
    MenuItem green = new MenuItem("Зеленый");
    MenuItem blue  = new MenuItem("Синий");
    MenuItem black = new MenuItem("Черный");
    c.add(red); c.add(green); c.add(blue); c.add(black);

      red.addActionListener(scr);
    green.addActionListener(scr);
     blue.addActionListener(scr);
    black.addActionListener(scr);
    clear.addActionListener(scr);

    addWindowListener(new WinClose());
    pack();
    setVisible(true);
  }
  class WinClose extends WindowAdapter{
     public void windowClosing(WindowEvent e){
       System.exit(0);
     }
  }
  public static void main(String[] args){
     new MenuScribble(" \"Рисовалка\" с меню");
  }
}
```

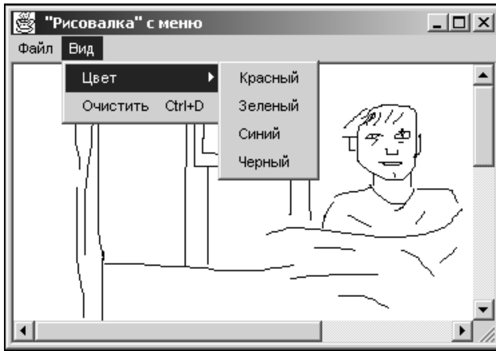


Рис. 13.2. Программа рисования с меню

Всплывающее меню

Всплывающее меню (popup menu) появляется обычно при нажатии или отпускании правой или средней кнопки мыши и является *контекстным* (context) меню. Его команды зависят от компонента, на котором была нажата кнопка мыши. В языке Java всплывающее меню — объект класса `PopupMenu`. Этот класс расширяет класс `Menu`, следовательно, наследует все свойства меню и пункта меню `MenuItem`. Всплывающее меню присоединяется не к строке меню типа `MenuBar` или к меню типа `Menu` в качестве подменю, а к определенному компоненту. Для этого в классе `Component` есть метод `add(PopupMenu menu)`.

У некоторых компонентов, например `TextField` и `TextArea`, уже существует всплывающее меню. Подобные меню нельзя переопределить.

Присоединить всплывающее меню можно только к одному компоненту. Если надо использовать всплывающее меню с несколькими компонентами в контейнере, то его присоединяют к контейнеру, а нужный компонент определяют с помощью метода `getComponent()` класса `MouseEvent`, как показано в листинге 13.2.

Кроме унаследованных свойств и методов, в классе `PopupMenu` есть метод `show(Component comp, int x, int y)`, показывающий всплывающее меню на экране так, что его левый верхний угол располагается в точке `(x, y)` в системе координат компонента `comp`. Чаще всего это компонент, на котором нажата кнопка мыши, возвращаемый методом `getComponent()`. Компонент `comp` должен быть внутри контейнера, к которому присоединено меню, иначе возникнет исключительная ситуация.

Всплывающее меню появляется в MS Windows при отпускании правой кнопки мыши, в Motif — при нажатии средней кнопки, а в других графических системах могут быть иные правила. Чтобы учесть эту разницу, в класс `MouseEvent` введен логический метод `isPopupTrigger()`, показывающий, что

возникшее событие мыши вызывает появление всплывающего меню. Его нужно вызывать при возникновении всякого события мыши, чтобы проверять, не является ли оно сигналом к появлению всплывающего меню, т. е. обращению к методу `show()`. Было бы слишком неудобно включать такую проверку во все семь методов классов-слушателей событий мыши. Поэтому метод `isPopupTrigger()` лучше вызывать в методе `processMouseEvent()`.

Переделаем еще раз программу рисования из листинга 12.4, введя в класс `Scribble` всплывающее меню для выбора цвета рисования и очистки окна и изменив обработку событий мыши. Для простоты уберем строку меню, хотя ее можно было оставить. Результат показан в листинге 13.2, а на рис. 13.3 — вид всплывающего меню в MS Windows.

Листинг 13.2. Программа рисования с всплывающим меню

```
import java.awt.*;
import java.awt.event.*;

public class PopupMenuScribble extends Frame{
  public PopupMenuScribble(String s){
    super(s);
    ScrollPane pane = new ScrollPane();
    pane.setSize(300, 300);
    add(pane, BorderLayout.CENTER);

    Scribble scr = new Scribble(this, 500, 500);
    pane.add(scr);

    addWindowListener(new WinClose());
    pack();
    setVisible(true);
  }
  class WinClose extends WindowAdapter{
     public void windowClosing(WindowEvent e){
       System.exit(0);
     }
  }
  public static void main(String[] args){
     new PopupMenuScribble("  \"Рисовалка\" с всплывающим меню");
  }
}

class Scribble extends Component implements ActionListener{
  protected int lastX, lastY, w, h;
  protected Color currColor = Color.black;
  protected Frame f;
  protected PopupMenu c;
```

```
public Scribble(Frame frame, int width, int height){
  f = frame; w = width; h = height;
  enableEvents(AWTEvent.MOUSE_EVENT_MASK |
               AWTEvent.MOUSE_MOTION_EVENT_MASK);

  c = new PopupMenu("Цвет");
  add(c);
  MenuItem clear = new MenuItem("Очистить",
                   new MenuShortcut(KeyEvent.VK_D));
  MenuItem red   = new MenuItem("Красный");
  MenuItem green = new MenuItem("Зеленый");
  MenuItem blue  = new MenuItem("Синий");
  MenuItem black = new MenuItem("Черный");
  c.add(red); c.add(green); c.add(blue); c.add(black);
  c.addSeparator(); c.add(clear);

    red.addActionListener(this);
  green.addActionListener(this);
   blue.addActionListener(this);
  black.addActionListener(this);
  clear.addActionListener(this);
}
public Dimension getPreferredSize(){
  return new Dimension(w, h);
}

public void actionPerformed(ActionEvent event){
  String s = event.getActionCommand();
  if (s.equals("Очистить")) repaint();
  else if (s.equals("Красный")) currColor = Color.red;
  else if (s.equals("Зеленый")) currColor = Color.green;
  else if (s.equals("Синий"))   currColor = Color.blue;
  else if (s.equals("Черный"))  currColor = Color.black;
}

public void processMouseEvent(MouseEvent e){
  if (e.isPopupTrigger())
    c.show(e.getComponent(), e.getX(), e.getY());
  else if (e.getID() == MouseEvent.MOUSE_PRESSED){
    lastX = e.getX(); lastY = e.getY();
  }
  else super.processMouseEvent(e);
}

public void processMouseMotionEvent(MouseEvent e){
  if (e.getID() == MouseEvent.MOUSE_DRAGGED){
    Graphics g = getGraphics();
```

```
      g.setColor(currColor);
      g.drawLine(lastX, lastY, e.getX(), e.getY());
      lastX = e.getX(); lastY = e.getY();
    }
    else super.processMouseMotionEvent(e);
  }
}
```

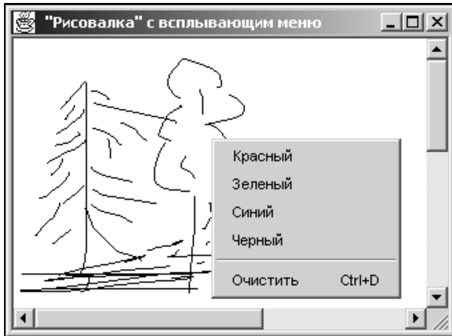
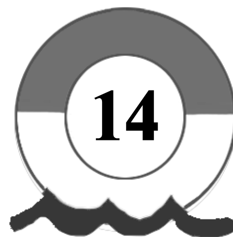


Рис. 13.3. Программа рисования с всплывающим меню

ГЛАВА 14



Апплеты

До сих пор мы создавали приложения (applications), работающие самостоятельно (standalone) в JVM под управлением графической оболочки операционной системы. Эти приложения имели собственное окно верхнего уровня типа `Frame`, зарегистрированное в оконном менеджере (window manager) графической оболочки.

Кроме приложений, язык Java позволяет создавать *апплеты* (applets). Это программы, работающие в среде другой программы — браузера. Апплеты не нуждаются в окне верхнего уровня — им служит окно браузера. Они не запускаются JVM — их загружает браузер, который сам запускает JVM для выполнения апплета. Эти особенности отражаются на написании программы апплета.

С точки зрения языка Java, апплет — это всякое расширение класса `Applet`, который, в свою очередь, расширяет класс `Panel`. Таким образом, апплет — это панель специального вида, контейнер для размещения компонентов с дополнительными свойствами и методами. Менеджером размещения компонентов по умолчанию, как и в классе `Panel`, служит `FlowLayout`. Класс `Applet` находится в пакете `java.applet`, в котором кроме него есть только три интерфейса, реализованные в браузере. Надо заметить, что не все браузеры реализуют эти интерфейсы полностью.

Поскольку JVM не запускает апплет, отпадает необходимость в методе `main()`, его нет в апплетах.

В апплетах редко встречается конструктор. Дело в том, что при запуске первого создается его контекст. Во время выполнения конструктора контекст еще не сформирован, поэтому не все начальные значения удается определить в конструкторе.

Начальные действия, обычно выполняемые в конструкторе и методе `main()`, в апплете записываются в метод `init()` класса `Applet`. Этот метод автомати-

чески запускается исполняющей системой Java браузера сразу же после загрузки апплета. Вот как он выглядит в исходном коде класса `Applet`:

```
public void init(){}
```

Негусто! Метод `init()` не имеет аргументов, не возвращает значения и должен переопределяться в каждом апплете — подклассе класса `Applet`.

Обратные действия — завершение работы, освобождение ресурсов — записываются при необходимости в метод `destroy()`, тоже выполняющийся автоматически при выгрузке апплета. В классе `Applet` есть пустая реализация этого метода.

Кроме методов `init()` и `destroy()` в классе `Applet` присутствуют еще два пустых метода, выполняющихся автоматически. Браузер должен обращаться к методу `start()` при каждом появлении апплета на экране и обращаться к методу `stop()`, когда апплет уходит с экрана. В методе `stop()` можно определить действия, приостанавливающие работу апплета, в методе `start()` — возобновляющие ее. Надо сразу же заметить, что не все браузеры обращаются к этим методам как должно. Работу указанных методов можно пояснить простым житейским примером.

Приехав весной на дачный участок, вы прокладываете водопроводные трубы, прикручиваете краны, протягиваете шланги — выполняете метод `init()` для своей оросительной системы. После этого, приходя на участок, включаете краны — запускаете метод `start()`, а уходя, выключаете их — выполняете метод `stop()`. Наконец, осенью вы разбираете оросительную систему, отвинчиваете краны, просушиваете и укладываете водопроводные трубы — выполняете метод `destroy()`.

Все эти методы в апплете необязательны. В листинге 14.1 записан простейший апплет, выполняющий вечную программу `HelloWorld`.

Листинг 14.1. Апплет `HelloWorld`

```
import java.awt.*;
import java.applet.*;

public class HelloWorld extends Applet{
   public void paint(Graphics g){
     g.drawString("Hello, XXI century World!", 10, 30);
   }
}
```

Эта программа записывается в файл HelloWorld.java и компилируется как обычно:

```
javac HelloWorld.java
```

Компилятор создает файл HelloWorld.class, но воспользоваться для его выполнения интерпретатором `java` теперь нельзя — нет метода `main()`. Вместо интерпретации надо дать указание браузеру для запуска апплета.

Все указания браузеру даются пометками, *тегами* (tags), на языке HTML (HyperText Markup Language). В частности, указание на запуск апплета дается в теге `<applet>`. В нем обязательно задается имя файла с классом апплета параметром `code`, ширина `width` и высота `height` панели апплета в пикселах. Полностью текст HTML для нашего апплета приведен в листинге 14.2.

Листинг 14.2. Файл HTML для загрузки апплета `HelloWorld`

```
<html>
  <head><title> Applet</title></head>
  <body>
    Ниже выполняется апплет.<br>
    <applet code = "HelloWorld.class" width = "200" height = "100">
    </applet>
  </body>
</html>
```

Этот текст заносится в файл с расширением html или htm, например, HelloWorld.html. Имя файла произвольно, никак не связано с апплетом или классом апплета.

Оба файла — HelloWorld.html и HelloWorld.class — помещаются в один каталог на сервере, и файл HelloWorld.html загружается в браузер, который может находиться в любом месте Internet. Браузер, просматривая HTML-файл, выполнит тег `<applet>` и загрузит апплет. После загрузки апплет появится в окне браузера, как показано на рис. 14.1.

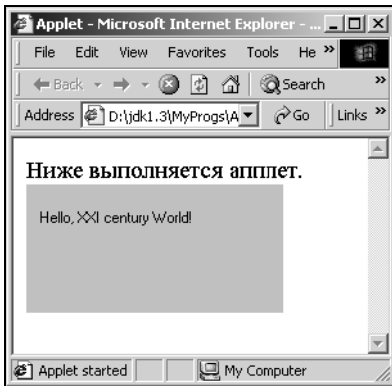


Рис. 14.1. Апплет `HelloWorld` в окне Internet Explorer

В этом простом примере можно заметить еще две особенности апплетов. Во-первых, размер апплета задается не в нем, а в теге `<applet>`. Это очень удобно, можно менять размер апплета, не компилируя его заново. Можно

организовать апплет невидимым, сделав его размером в один пиксел. Кроме того, размер апплета разрешается задать в процентах по отношению к размеру окна браузера, например,

```
<applet code = "HelloWorld.class" width = "100%" height = "100%">
```

Во-вторых, как видно на рис. 14.1, у апплета серый фон. Такой фон был в первых браузерах, и апплет не выделялся из текста в окне браузера. Теперь в браузерах принят белый фон, его можно установить обычным для компонентов методом `setBackground(Color.white)`, обратившись к нему в методе `init()`.

В состав JDK любой версии входит программа `appletviewer`. Это простейший браузер, предназначенный для запуска апплетов в целях отладки. Если под рукой нет Internet-браузера, можно воспользоваться им. `Appletviewer` запускается из командной строки:

```
appletviewer HelloWorld.html
```

На рис. 14.2 `appletviewer` показывает апплет `HelloWorld`.

Рис. 14.2. Апплет `HelloWorld` в окне программы `appletviewer`

Приведем пример невидимого апплета. В нижней строке браузера — *строке состояния* (status bar) — отражаются сведения о загрузке файлов. Апплет может записать в нее любую строку `str` методом `showStatus(String str)`. В листинге 14.3 приведен апплет, записывающий в строку состояния браузера "бегущую строку", а в листинге 14.4 — соответствующий HTML-файл.

Листинг 14.3. Бегущая строка в строке состояния браузера

```
// Файл RunningString.java
import java.awt.*;
import java.applet.*;

public class RunningString extends Applet{
   private boolean go;
   public void start(){
    go = true;
    sendMessage("Эта строка выводится апплетом");
   }
```

```
   public void sendMessage(String s){
     String s1 = s+"  ";
     while(go){
        showStatus(s);
        try{
           Thread.sleep(200);
        }catch(Exception e){}
        s = s1.substring(1)+s.charAt(0);
        s1 =s;
     }
   }
  public void stop(){
    go = false;
  }
}
```

Листинг 14.4. Файл RunningString.html

```
<html>
  <head><title> Applet</title></head>
  <body>
    Здесь работает апплет.<br>
    <applet code = "RunningString.class" width = "1" height = "1">
    </applet>
  </body>
</html>
```

К сожалению, нет строгого стандарта на выполнение апплетов, и браузеры могут запускать их по-разному. Программа `appletviewer` способна показать апплет не так, как браузеры. Приходится проверять апплеты на всех имеющихся в распоряжении браузерах, добиваясь одинакового выполнения.

Приведем более сложный пример. Апплет `ShowWindow` создает окно `SomeWindow` типа `Frame`, в котором расположено поле ввода типа `TextField`. В него вводится текст, и после нажатия клавиши <Enter> переносится в поле ввода апплета. В апплете присутствует кнопка. После щелчка кнопкой мыши по ней окно `SomeWindow` то скрывается с экрана, то вновь появляется на нем. То же самое должно происходить при уходе и появлении апплета в окне браузера в результате прокрутки, как записано в методах `stop()` и `start()`, но будет ли? Программа приведена в листингах 14.5 и 14.6, результат — на рис. 14.3.

Листинг 14.5. Апплет, создающий окно

```
// Файл ShowWindow.java
import java.awt.*;
```

```
import java.awt.event.*;
import java.applet.*;

public class ShowWindow extends Applet{
  private SomeWindow sw = new SomeWindow();
  private TextField tf = new TextField(30);
  private Button b = new Button("Скрыть");
  public void init(){
    add(tf);  add(b); sw.pack();
    b.addActionListener(new ActShow());
    sw.tf.addActionListener(new ActShow());
  }
  public void start(){ sw.setVisible(true); }
  public void stop(){ sw.setVisible(false); }
  public void destroy(){
    sw.dispose(); sw = tf = b = null;
  }
  public class ActShow implements ActionListener{
    public void actionPerformed(ActionEvent ae){
      if (ae.getSource() == sw.tf)
         tf.setText(sw.tf.getText());
      else if (b.getActionCommand() == "Показать"){
         sw.setVisible(true);
         b.setLabel("Скрыть");
      }else{
         sw.setVisible(false);
         b.setLabel("Показать");
      }
    }
  }
}
class SomeWindow extends Frame{
  public TextField tf = new TextField(50);
  SomeWindow(){
    super(" Окно ввода");
    add(new Label("Введите, пожалуйста, свое имя"), "North");
    add(tf, "Center");
  }
}
```

Листинг 14.6. Файл ShowWindow.html

```
<html>
  <head><title> ShowWindow Applet</title></head>
  <body>
```

```
    Здесь появится Ваше имя.<br>
    <applet code = "ShowWindow.class" width = "400" height = "50">
    </applet>
  </body>
</html>
```

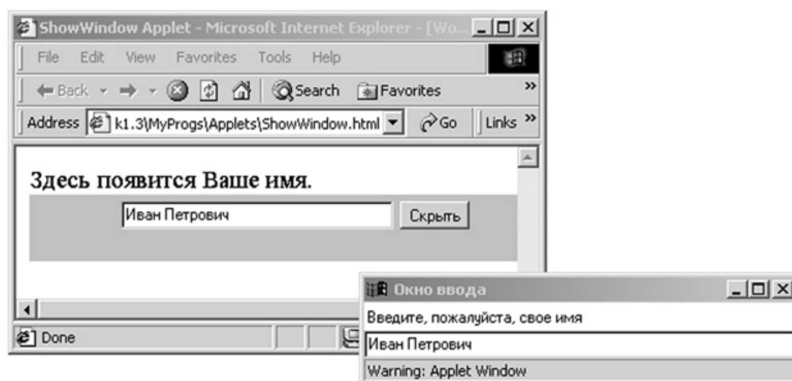


Рис. 14.3. Апплет, создающий окно

Замечание по отладке

Браузеры помещают загруженные апплеты в свой кэш, поэтому после щелчка кнопкой мыши по кнопке **Refresh** или **Reload** запускается старая копия апплета из кэша. Для загрузки новой копии надо при щелчке по кнопке **Refresh** в IE (Internet Explorer) держать нажатой клавишу <Ctrl>, а при щелчке по кнопке **Reload** в NC (Netscape Communicator) — клавишу <Shift>. Иногда и это не помогает. Не спасает даже перезапуск браузера. Тогда следует очистить оба кэша: и дисковый, и кэш в памяти. В IE это выполняется кнопкой **Delete Files** в окне, вызываемом выбором команды **Tools** | **Internet Options**. В NC необходимо открыть окно **Cache** командой **Edit** | **Preferences | Advanced**.

При запуске приложения интерпретатором `java` из командной строки в него можно передать параметры в виде аргумента метода `main(String[] args)`. В апплеты также передаются параметры, но другим путем.

Передача параметров

Передача параметров в апплет производится с помощью тегов `<param>`, располагаемых между открывающим тегом `<applet>` и закрывающим тегом `</applet>` в HTML-файле. В тегах `<param>` указывается название параметра `name` и его значение `value`.

Передадим, например, в наш апплет `HelloWorld` параметры шрифта. В листинге 14.7 показан измененный файл HelloWorld.html.

Листинг 14.7. Параметры для передачи в апплет

```
<html>
  <head><title> Applet</title></head>
  <body>
    Ниже выполняется апплет.<br>
    <applet code = "HelloWorld.class" width = "400" height = "50">
      <param name = "fontName"  value = "Serif">
      <param name = "fontStyle" value = "2">
      <param name = "fontSize"  value = "30">
    </applet>
  </body>
</html>
```

В апплете для приема каждого параметра надо воспользоваться методом `getParameter(String name)` класса `Applet`, возвращающим строку типа `String`. В качестве аргумента этого метода задается значение параметра `name` в виде строки, причем здесь не различается регистр букв, а метод возвращает значение параметра `value` тоже в виде строки.

Замечание по отладке

Операторы `System.out.println()`, обычно записываемые в апплет для отладки, выводят указанные в них аргументы в специальное окно браузера **Java Console**. Сначала надо установить возможность показа этого окна. В Internet Explorer это делается установкой флажка **Java Console enabled** выбором команды **Tools** | **Internet Options** | **Advanced**. После перезапуска IE в меню **View** появляется команда **Java Console**.

В листинге 14.8 показан переработанный апплет `HelloWorld`. В нем назначен белый фон, а шрифт устанавливается с параметрами, извлеченными из HTML-файла.

Листинг 14.8. Апплет, принимающий параметры

```
import java.awt.*;
import java.applet.*;

public class HelloWorld extends Applet{
   public void init(){
     setBackground(Color.white);
     String font = "Serif";
     int style = Font.PLAIN, size = 10;
     font = getParameter("fontName");
     style = Integer.parseInt(getParameter("fontStyle"));
     size  = Integer.parseInt(getParameter("fontSize"));
```

```
      setFont(new Font(font, style, size));
    }
    public void paint(Graphics g){
      g.drawString("Hello, XXI century World!", 10, 30);
    }
}
```

Совет

Надеясь на то, что параметры будут заданы в HTML-файле, все-таки присвойте начальные значения переменным в апплете, как это сделано в листинге 14.8.

На рис. 14.4 показан работающий апплет.

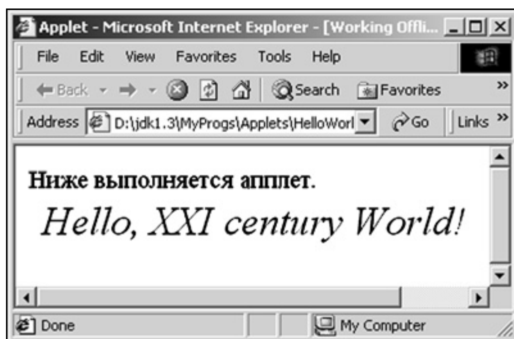


Рис. 14.4. Апплет с измененным шрифтом

Правила хорошего тона рекомендуют описать параметры, передаваемые апплету, в виде массива, каждый элемент которого — массив из трех строк, соответствующий одному параметру. Данная структура представляется в виде "имя", "тип", "описание". Для нашего примера можно написать:

```
String[][] pinfo = {
  {"fontName", "String", "font name"},
  {"fontStyle", "int", "font style"},
  {"fontSize", "int", "font size"}
};
```

Затем переопределяется метод `getParameterInfo()`, возвращающий указанный массив. Это пустой метод класса `Applet`. Любой объект, желающий узнать, что передать апплету, может вызвать этот метод. Для нашего примера переопределение выглядит так:

```
public String[][] getParameterInfo(){
  return pinfo;
}
```

Кроме того, правила хорошего тона предписывают переопределить метод `getAppletInfo()`, возвращающий строку, в которой записано имя автора,

версия апплета и прочие сведения об апплете, которые вы хотите предоставить всем желающим. Например:

```
public String getAppletInfo(){
   return "MyApplet v.1.5 P.S.Ivanov";
}
```

Посмотрим теперь, какие еще параметры можно задать в теге `<applet>`.

Параметры тега *<applet>*

Перечислим все параметры тега `<applet>`.

Обязательные параметры:

- `code` — URL-адрес файла с классом апплета или архивного файла;
- `width` и `height` — ширина и высота апплета в пикселах.

Необязательные параметры:

- `codebase` — URL-адрес каталога, в котором расположен файл класса апплета. Если этот параметр отсутствует, браузер будет искать файл в том же каталоге, где размещен соответствующий HTML-файл;
- `archive` — файлы всех классов, составляющих апплет, могут быть упакованы архиватором ZIP или специальным архиватором JAR в один или несколько архивных файлов. Параметр задает URL-адреса этих файлов через запятую;
- `align` — выравнивание апплета в окне браузера. Этот параметр имеет одно из следующих значений: `ABSBOTTOM`, `ABSMIDDLE`, `BASELINE`, `BOTTOM`, `CENTER`, `LEFT`, `MIDDLE`, `RIGHT`, `TEXTTOP`, `TOP`;
- `hspace` и `vspace` — горизонтальные и вертикальные поля, отделяющие апплет от других объектов в окне браузера в пикселах;
- `download` — задает порядок загрузки изображений апплетом. Имена изображений перечисляются через запятую в порядке загрузки;
- `name` — имя апплета. Параметр нужен, если загружаются несколько апплетов с одинаковыми значениями `code` и `codebase`;
- `style` — информация о стиле CSS (Cascading Style Sheet);
- `title` — текст, отображаемый в процессе выполнения апплета;
- `alt` — текст, выводимый вместо апплета, если браузер не может загрузить его;
- `mayscript` — не имеет значения. Это слово указывает на то, что апплет будет обращаться к тексту JavaScript.

Между тегами <applet> и </applet> можно написать текст, который будет выведен, если браузер не сможет понять тег <applet>. Вот полный пример:

```
<applet name = "AnApplet" code = "AnApplet.class"
   archive = "anapplet.zip, myclasses.zip"
   codebase = "http://www.some.com/public/applets"
   width = "300" height = "200" align = "TOP"
   vspace = "5" hspace = "5" mayscript
   alt = "If you have a java-enabled browser,
          you would see an applet here.">
<hr>If your browser recognized the applet tag,
    you would see an applet here.<hr>
</applet>
```

Совет

Обязательно упаковывайте все классы апплета в zip- и rar-архивы и указывайте их в параметре archive в HTML-файле. Это значительно ускорит загрузку апплета.

Следует еще сказать, что, начиная с версии HTML 4.0, есть тег <object>, предназначенный для загрузки и апплетов, и других объектов, например, ActiveX. Кроме того, некоторые браузеры могут использовать для загрузки апплетов тег <embed>.

Мы уже упоминали, что при загрузке апплета браузер создает контекст, в котором собирает все сведения, необходимые для выполнения апплета. Некоторые сведения из контекста можно передать в апплет.

Сведения об окружении апплета

Метод getCodeBase() возвращает URL-адрес каталога, в котором лежит файл класса апплета.

Метод getDocumentBase() возвращает URL-адрес каталога, в котором лежит HTML-файл, вызвавший апплет.

Браузер реализует интерфейс AppletContext, находящийся в пакете java.applet. Апплет может получить ссылку на этот интерфейс методом getAppletContext().

С помощью методов getApplet(String name) и getApplets() интерфейса AppletContext можно получить ссылку на указанный аргументом name апплет или на все апплеты, загруженные в браузер.

Метод showDocument(URL address) загружает в браузер HTML-файл с адреса address.

Метод showDocument(URL address, String target) загружает файл во фрейм, указанный вторым аргументом target. Этот аргумент может принимать следующие значения:

- _self — то же окно и тот же фрейм, в котором работает апплет;
- _parent — родительский фрейм апплета;
- _top — фрейм верхнего уровня окна апплета;
- _blank — новое окно верхнего уровня;
- name — фрейм или окно с именем name, если оно не существует, то будет создано.

Изображение и звук

Изображение в Java — это объект класса Image, представляющий прямоугольный массив пикселов. Его могут показать на экране логические методы drawImage() класса Graphics. Мы рассмотрим их подробно в следующей главе, а пока нам понадобятся два логических метода:

```
drawImage(Image img, int x, int y, ImageObserver obs)
drawImage(Image img, int x, int y, int width, int height,
          ImageObserver obs)
```

Методы начинают рисовать изображение, не дожидаясь окончания загрузки изображения img. Более того, загрузка не начнется, пока не вызван метод drawImage(). Методы возвращают false, пока загрузка не закончится.

Аргументы (x, y) задают координаты левого верхнего угла изображения img; width и height — ширину высоту изображения на экране; obs — ссылку на объект, реализующий интерфейс ImageObserver, следящий за процессом загрузки изображения. Последнему аргументу можно дать значение this.

Первый метод задает на экране такие же размеры изображения, как и у объекта класса Image, без изменений. Получить эти размеры можно методами getWidth(), getHeight() класса Image.

Интерфейс ImageObserver, реализованный классом Component, а значит, и классом Applet, описывает только один логический метод imageUpdate(), выполняющийся при каждом изменении изображения. Именно этот метод побуждает перерисовывать компонент на экране при каждом его изменении. Посмотрим, как его можно использовать в процессе загрузки файлов из Internet.

Слежение за процессом загрузки

Если вы хотя бы раз видели, как изображение загружается из Internet, то заметили, что оно появляется на экране по частям по мере загрузки. Это

происходит в том случае, когда системное свойство `awt.image.incrementalDraw` имеет значение `true`.

При поступлении каждой порции изображения браузер вызывает логический метод `imageUpdate()` интерфейса `ImageObserver`. Аргументы этого метода содержат информацию о процессе загрузки изображения `img`. Рассмотрим их:

```
imageUpdate(Image img, int status, int x, int y, int width, int height);
```

Аргумент `status` содержит информацию о загрузке в виде одного целого числа, которое можно сравнить со следующими константами интерфейса `ImageObserver`:

- `WIDTH` — ширина уже загруженной части изображения известна, и может быть получена из аргумента `width`;
- `HEIGHT` — высота уже загруженной части изображения известна, и может быть получена из аргумента `height`;
- `PROPERTIES` — свойства изображения уже известны, их можно получить методом `getProperties()` класса `Image`;
- `SOMEBITS` — получены пикселы, достаточные для рисования масштабированной версии изображения; аргументы `x`, `y`, `width`, `height` определены;
- `FRAMEBITS` — получен следующий кадр изображения, содержащего несколько кадров; аргументы `x`, `y`, `width`, `height` не определены;
- `ALLBITS` — все изображение получено, аргументы `x`, `y`, `width`, `height` не содержат информации;
- `ERROR` — загрузка прекращена, рисование прервано, определен бит `ABORT`;
- `ABORT` — загрузка прервана, рисование приостановлено до прихода следующей порции изображения.

Вы можете переопределить этот метод в своем апплете и использовать его аргументы для слежения за процессом загрузки и определения момента полной загрузки.

Другой способ отследить окончание загрузки — воспользоваться методами класса `MediaTracker`. Они позволяют проверить, не окончена ли загрузка, или приостановить работу апплета до окончания загрузки. Один экземпляр класса `MediaTracker` может следить за загрузкой нескольких зарегистрированных в нем изображений.

Класс *MediaTracker*

Сначала конструктором `MediaTracker(Component comp)` создается объект класса для указанного аргументом компонента. Аргумент конструктора чаще всего `this`.

Затем методом `addImage(Image img, int id)` регистрируется изображение `img` под порядковым номером `id`. Несколько изображений можно зарегистрировать под одним номером.

После этого логическими методами `checkID(int id)`, `checkID(int id, boolean load)` и `checkAll()` проверяется, загружено ли изображение с порядковым номером `id` или все зарегистрированные изображения. Методы возвращают `true`, если изображение уже загружено, `false` — в противном случае. Если аргумент `load` равен `true`, то производится загрузка всех еще не загруженных изображений.

Методы `statusID(int id)`, `statusID(int id, boolean load)` и `statusAll()` возвращают целое число, которое можно сравнить со статическими константами `COMPLETE`, `ABORTED`, `ERRORED`.

Наконец, методы `waitForID(int id)` и `waitForAll()` ожидают окончания загрузки изображения.

В следующей главе в листинге 15.5 мы применим эти методы для ожидания загрузки изображения.

Изображение, находящееся в объекте класса `Image` можно создать непосредственно по пикселам, а можно получить из графического файла, типа GIF или JPEG, одним из двух методов класса `Applet`:

- `getImage(URL address)` — задается URL-адрес графического файла;
- `getImage(URL address, String fileName)` — задается адрес каталог `address` и имя графического файла `filename`.

Аналогично, звуковой файл в апплетах представляется в виде объекта, реализующего интерфейс `AudioClip`, и может быть получен из файла типа AU, AIFF, WAVE или MIDI одним из трех методов класса `Applet` с такими же аргументами:

```
getAudioClip(URL address)
getAudioClip(URL address, String fileName)
newAudioClip(URL address)
```

Последний метод статический, его можно использовать не только в апплетах, но и в приложениях.

Интерфейс `AudioClip` из пакета `java.applet` очень прост. В нем всего три метода без аргументов. Метод `play()` проигрывает мелодию один раз. Метод `loop()` бесконечно повторяет мелодию. Метод `stop()` прекращает проигрывание.

Этот интерфейс реализуется браузером. Конечно, перед проигрыванием звуковых файлов браузер должен быть связан со звуковой системой компьютера.

В листинге 14.9 приведен простой пример загрузки изображения и звука из файлов, находящихся в том же каталоге, что и HTML-файл. На рис. 14.5 показано, как выглядит изображение, увеличенное в два раза.

Листинг 14.9. Звук и изображение в апплете

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class SimpleAudioImage extends Applet{
   private Image img;
   private AudioClip ac;
   public void init(){
      img = getImage(getDocumentBase(), "javalogo52x88.gif");
      ac = getAudioClip(getDocumentBase(), "yesterday.au");
   }
   public void start(){ ac.loop(); }
   public void paint(Graphics g){
      int w = img.getWidth(this), h = img.getHeight(this);
      g.drawImage(img, 0, 0, 2 * w, 2 * h, this);
   }
   public void stop(){ ac.stop(); }
}
```

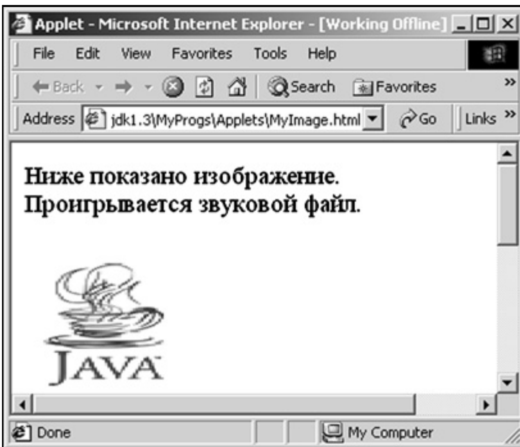


Рис. 14.5. Вывод изображения

Перед выводом на экран изображение можно преобразовать, но об этом поговорим в следующей главе.

Как видите, апплету в браузере позволено очень немного. Это не случайно. Апплет, появившийся в браузере откуда-то из Internet, может натворить

много бед. Он может быть вызван из файла с увлекательным текстом, невидимо обыскать файловую систему и похитить секретные сведения, или, напротив, открыть окно, неотличимое от окна, в которое вы вводите пароль, и перехватить его.

Поэтому браузер сообщает при загрузке апплета: "Applet started", а в строке состояния окна, открытого апплетом, появляется надпись: "Warning: Applet Window".

Но это не единственная защита от апплета. Рассмотрим данную проблему подробнее.

Защита от апплета

Браузер может вообще отказаться от загрузки апплетов. В Netscape Communicator это делается с помощью флажка **Enable Java** в окне, вызываемом командой **Edit** | **Preferences** | **Advanced**, в Internet Explorer — в окне после выбора команды **Tools** | **Internet Options** | **Security**. В таком случае говорить в этой книге больше не о чем.

Если браузер загружает апплет, то создает ему ограничения, так называемую *"песочницу"* (sandbox), в которой резвится апплет, но выйти из которой не может. Каждый браузер создает свои ограничения, но обычно они заключаются в том, что апплет:

- не может обращаться к файловой системе машины, на которой он выполняется, даже для чтения файлов или просмотра каталогов;
- может связаться по сети только с тем сайтом, с которого он был загружен;
- не может прочитать системные свойства, как это делает, например, приложение в листинге 6.4;
- не может печатать на принтере, подключенном к тому компьютеру, на котором он выполняется;
- не может воспользоваться буфером обмена (clipboard);
- не может запустить приложение методом `exec()`;
- не может использовать "родные" методы или загрузить библиотеку методом `load()`;
- не может остановить JVM методом `exit()`;
- не может создавать классы в пакетах `java.*`, а классы пакетов `sun.*` не может даже загружать.

Браузеры могут усилить или ослабить эти ограничения, например, разрешить локальным апплетам, загруженным с той же машины, где они выполняются, доступ к файловой системе. Наименьшие ограничения имеют *дове-*

ренные (trusted) апплеты, снабженные электронной подписью с помощью классов из пакетов `java.security.*`.

При создании приложения, загружающего апплеты, необходимо обеспечить средства проверки апплета и задать ограничения. Их предоставляет класс `SecurityManager`. Экземпляр этого класса или его наследника устанавливается в JVM при запуске виртуальной машины статическим методом `setSecurityManager(SecurityManager sm)` класса `System`. Обычные приложения не могут использовать данный метод.

Каждый браузер расширяет класс `SecurityManager` по-своему, устанавливая те или иные ограничения. Единственный экземпляр этого класса создается при запуске JVM в браузере и не может быть изменен.

Заключение

Апплеты были первоначальным практическим применением Java. За первые два года существования Java были написаны тысячи очень интересных и красивых апплетов, ожививших WWW. Масса апплетов разбросана по Internet, хорошие примеры апплетов собраны в JDK в каталоге demo\applets.

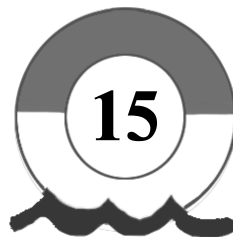
В JDK вошел целый пакет `java.applet`, в который фирма SUN собиралась заносить классы, развивающие и улучшающие апплеты.

С увеличением скорости и улучшением качества компьютерных сетей значение апплетов сильно упало. Теперь вся обработка данных, прежде выполняемая апплетами, переносится на сервер, браузер только загружает и показывает результаты этой обработки, становится "тонким клиентом".

С другой стороны, появилось много специализированных программ, в том числе написанных на Java, загружающих информацию из Internet. Такая возможность есть сейчас у всех музыкальных и видеопроигрывателей.

Фирма SUN больше не развивает пакет `java.applet`. В нем так и остался один класс и три интерфейса. В библиотеку Swing вошел класс `JApplet`, расширяющий класс `Applet`. В нем есть дополнительные возможности, например, можно установить систему меню. Он способен использовать все классы библиотеки Swing. Но большинство браузеров еще не имеют Swing в своем составе, поэтому приходится загружать классы Swing с сервера или включать их в jar-архив вместе с классами апплета.

ГЛАВА 15



Изображения и звук

Как уже упоминалось в предыдущей главе, изображение в Java — это объект класса `Image`. Там же показано, как в апплетах применяются методы `getImage()` для создания этих объектов из графических файлов.

Приложения тоже могут применять аналогичные методы `getImage()` класса `Toolkit` из пакета `java.awt` с одним аргументом типа `String` или `URL`. Обращение к этим методам из компонента выполняется через метод `getToolkit()` класса `Component` и выглядит так:

```
Image img = getToolkit().getImage("C:\\images\\Ivanov.gif");
```

В общем случае обращение можно сделать через статический метод `getDefaultToolkit()` класса `Toolkit`:

```
Image img = Toolkit.getDefaultToolkit().getImage("C:\\images\\Ivanov.gif");
```

Но, кроме этих методов, класс `Toolkit` содержит пять методов `createImage()`, возвращающих ссылку на объект типа `Image`:

- `createImage(String fileName)` — создает изображение из содержимого графического файла `filename`;
- `createImage(URL address)` — создает изображение из содержимого графического файла по адресу `address`;
- `createImage(byte[] imageData)` — создает изображение из массива байтов `imageData`, данные в котором должны иметь формат GIF или JPEG;
- `createImage(byte[] imageData, int offset, int length)` — создает изображение из части массива `imageData`, начинающейся с индекса `offset` длиной `length` байтов;
- `createImage(ImageProducer producer)` — создает изображение, полученное от поставщика `producer`.

Последний метод есть и в классе `Component`. Он использует модель "поставщик-потребитель" и требует подробного объяснения.

Модель обработки "поставщик-потребитель"

Очень часто изображение перед выводом на экран подвергается обработке: меняются цвета отдельных пикселов или целых участков изображения, выделяются и преобразуются какие-то фрагменты изображения.

В библиотеке AWT применяются две модели обработки изображения. Одна модель реализует давно известную в программировании общую модель "поставщик-потребитель" (Producer-Consumer). Согласно этой модели один объект, "поставщик", генерирует сам или преобразует полученную из другого места продукцию, в данном случае, набор пикселов, и передает другим объектам. Эти объекты, "потребители", принимают продукцию и тоже преобразуют ее при необходимости. Только после этого создается объект класса `Image` и изображение выводится на экран. У одного поставщика может быть несколько потребителей, которые должны быть зарегистрированы поставщиком. Поставщик и потребитель активно взаимодействуют, обращаясь к методам друг друга.

В AWT эта модель описана в двух интерфейсах: `ImageProducer` и `ImageConsumer` пакета `java.awt.image`.

Интерфейс `ImageProducer` описывает пять методов:

- `addConsumer(ImageConsumer ic)` — регистрирует потребителя `ic`;
- `removeConsumer(ImageConsumer ic)` — отменяет регистрацию;
- `isConsumer(ImageConsumer ic)` — логический метод, проверяет, зарегистрирован ли потребитель `ic`;
- `startProduction(ImageConsumer ic)` — регистрирует потребителя `ic` и начинает поставку изображения всем зарегистрированным потребителям;
- `requestTopDownLeftRightResend(ImageConsumer ic)` — используется потребителем для того, чтобы затребовать изображение еще раз в порядке "сверху-вниз, слева-направо" для методов обработки, применяющих именно такой порядок.

С каждым экземпляром класса `Image` связан объект, реализующий интерфейс `ImageProducer`. Его можно получить методом `getSource()` класса `Image`.

Самая простая реализация интерфейса `ImageProducer` — класс `Memory-ImageSource` — создает пикселы в оперативной памяти по массиву байтов или целых чисел. Вначале создается массив `pix`, содержащий цвет каждой точки. Затем одним из шести конструкторов создается объект класса

`MemoryImageSource`. Он может быть обработан потребителем или прямо преобразован в тип `Image` методом `createImage()`.

В листинге 15.1 приведена простая программа, выводящая на экран квадрат размером 100×100 пикселов. Левый верхний угол квадрата синий, левый нижний — красный, правый верхний — зеленый, а к центру квадрата цвета перемешиваются.

Листинг 15.1. Изображение, построенное по точкам

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;

class InMemory extends Frame{
  private int w = 100, h = 100;
  private int[] pix = new int[w * h];
  private Image img;
  InMemory(String s){
    super(s);
    int i = 0;
    for (int y = 0; y < h; y++){
      int red = 255 * y / (h — 1);
      for (int x = 0; x < w; x++){
        int green = 255 * x / (w — 1);
        pix[i++] = (255 << 24)|(red << 16)|(green << 8)| 128;
      }
    }
    setSize(250, 200);
    setVisible(true);
  }
  public void paint(Graphics gr){
    if (img == null)
      img = createImage(new MemoryImageSource(w, h, pix, 0, w));
    gr.drawImage(img, 50, 50, this);
  }
  public static void main(String[] args){
    Frame f= new InMemory("  Изображение в памяти");
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

В листинге 15.1 в конструктор класса-поставщика `MemoryImageSource(w, h, pix, 0, w)` заносится ширина `w` и высота `h` изображения, массив `pix`, смещение в этом массиве `0` и длина строки `w`. Потребителем служит изображение `img`, которое создается методом `createImage()` и выводится на экран методом `drawImage(img, 50, 50, this)`. Левый верхний угол изображения `img` располагается в точке `(50, 50)` контейнера, а последний аргумент `this` показывает, что роль `ImageObserver` играет сам класс `InMemory`. Это заставляет включить в метод `paint()` проверку `if (img == null)`, иначе изображение будет постоянно перерисовываться. Другой способ избежать этого — переопределить метод `imageUpdate()`, о чем говорилось в *главе 14*, просто написав в нем `return true`.

Рис. 15.1 демонстрирует вывод этой программы. К сожалению, черно-белая печать не передает смешение цветов.

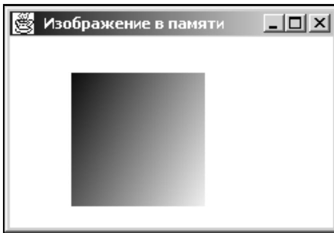


Рис. 15.1. Изображение, созданное по точкам

Интерфейс `ImageConsumer` описывает семь методов, самыми важными из которых являются два метода `setPixels()`. Первый:

```
setPixels(int x, int y, int width, int height, ColorModel model,
          byte[] pix, int offset, int scansize);
```

Второй метод отличается только тем, что массив `pix` содержит элементы типа `int`.

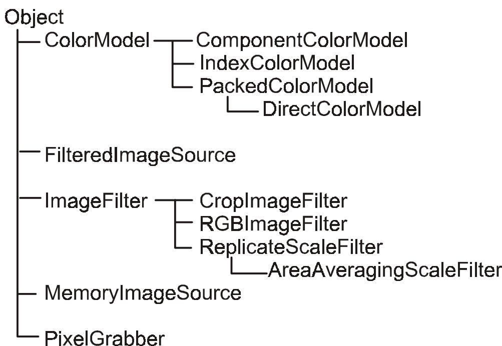


Рис. 15.2. Классы, реализующие модель "поставщик-потребитель"

К этим методам обращается поставщик для передачи пикселов потребителю. Передается прямоугольник шириной `width` и высотой `height` с заданным

верхним левым углом (x, y), заполняемый пикселами из массива pix, начиная с индекса offset. Каждая строка занимает scansize элементов массива pix. Цвета пикселов определяются в цветовой модели model (обычно это модель RGB).

На рис. 15.2 показана иерархия классов, реализующих модель "поставщик-потребитель".

Классы-фильтры

Интерфейс ImageConsumer нет нужды реализовывать, обычно используется его готовая реализация — класс ImageFilter. Несмотря на название, этот класс не производит никакой фильтрации, он передает изображение без изменений. Для преобразования изображений данный класс следует расширить, переопределив метод setPixels(). Результат преобразования следует передать потребителю, роль которого играет поле consumer этого класса.

В пакете java.awt.image есть четыре расширения класса ImageFilter:

- CropImageFilter(int x, int y, int w, int h) — выделяет фрагмент изображения, указанный в приведенном конструкторе;
- RGBImageFilter — позволяет изменять отдельные пикселы; это абстрактный класс, он требует расширения и переопределения своего метода filterRGB();
- ReplicateScaleFilter(int w, int h) — изменяет размеры изображения на указанные в приведенном конструкторе, дублируя строки и/или столбцы при увеличении размеров или убирая некоторые из них при уменьшении;
- AreaAveragingScaleFilter(int w, int h) — расширение предыдущего класса; использует более сложный алгоритм изменения размеров изображения, усредняющий значения соседних пикселов.

Применяются эти классы совместно со вторым классом-поставщиком, реализующим интерфейс ImageProducer — классом FilteredImageSource. Этот класс преобразует уже готовую продукцию, полученную от другого поставщика producer, используя для преобразования объект filter класса-фильтра ImageFilter или его подкласса. Оба объекта задаются в конструкторе

```
FilteredImageSource(ImageProducer producer, ImageFilter filter)
```

Все это кажется очень запутанным, но схема применения фильтров всегда одна и та же. Она показана в листингах 15.2—15.4.

Как выделить фрагмент изображения

В листинге 15.2 выделяется фрагмент изображения и выводится на экран в увеличенном виде. Кроме того, ниже выводятся изображения, увеличенные с помощью классов ReplicateScaleFilter и AreaAveragingScaleFilter.

Листинг 15.2. Примеры масштабирования изображения

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;

class CropTest extends Frame{
  private Image img, cropimg, replimg, averimg;
  CropTest(String s){
    super(s);
        // 1. Создаем изображение — объект класса Image
    img = getToolkit().getImage("javalogo52x88.gif");
        // 2. Создаем объекты-фильтры:
        //    а) выделяем левый верхний угол размером 30x30
    CropImageFilter crp =
                   new CropImageFilter(0, 0, 30, 30);
        //    б) увеличиваем изображение в два раза простым методом
    ReplicateScaleFilter rsf =
                   new ReplicateScaleFilter(104, 176);
        //    в) увеличиваем изображение в два раза с усреднением
    AreaAveragingScaleFilter asf =
                   new AreaAveragingScaleFilter(104, 176);
        // 3. Создаем измененные изображения
    cropimg = createImage(new FilteredImageSource(img.getSource(), crp));
    replimg = createImage(new FilteredImageSource(img.getSource(), rsf));
    averimg = createImage(new FilteredImageSource(img.getSource(), asf));
    setSize(400, 350);
    setVisible(true);
  }
  public void paint(Graphics g){
    g.drawImage(img, 10, 40, this);
    g.drawImage(cropimg, 150,  40, 100, 100, this);
    g.drawImage(replimg,  10, 150, this);
    g.drawImage(averimg, 150, 150, this);
  }
  public static void main(String[] args){
    Frame f= new CropTest("  Масштабирование");
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

На рис. 15.3 слева сверху показано исходное изображение, справа — увеличенный фрагмент, внизу — изображение, увеличенное двумя способами.

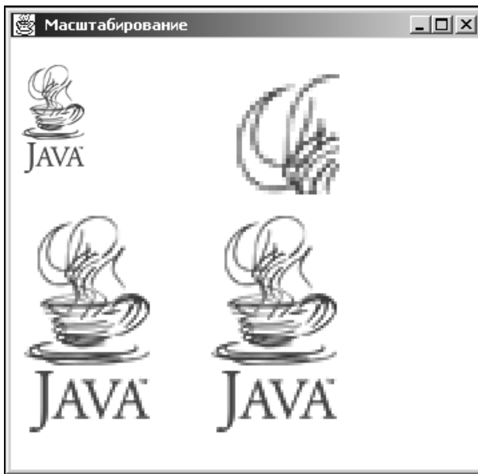


Рис. 15.3. Масштабированное изображение

Как изменить цвет изображения

В листинге 15.3 меняются цвета каждого пиксела изображения. Это достигается просто сдвигом `rgb >> 1` содержимого пиксела на один бит вправо в методе `filterRGB()`. При этом усиливается красная составляющая цвета. Метод `filterRGB()` переопределен в расширении `ColorFilter` класса `RGBImageFilter`.

Листинг 15.3. Изменение цвета всех пикселов

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;

class RGBTest extends Frame{
  private Image img, newimg;
  RGBTest(String s){
    super(s);
    img = getToolkit().getImage("javalogo52x88.gif");
    RGBImageFilter rgb = new ColorFilter();
    newimg = createImage(new FilteredImageSource(img.getSource(), rgb));
    setSize(400, 350);
    setVisible(true);
  }
  public void paint(Graphics g){
    g.drawImage(img, 10, 40, this);
    g.drawImage(newimg, 150,  40, this);
  }
```

```
  public static void main(String[] args){
    Frame f= new RGBTest("  Изменение цвета");
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
class ColorFilter extends RGBImageFilter{
   ColorFilter(){
     canFilterIndexColorModel = true;
   }
   public int filterRGB(int x, int y, int rgb){
     return rgb >> 1;
   }
}
```

Как переставить пикселы изображения

В листинге 15.4 определяется преобразование пикселов изображения. Создается новый фильтр — расширение `ShiftFilter` класса `ImageFilter`, сдвигающее изображение циклически вправо на указанное в конструкторе число пикселов. Все, что для этого нужно, — это переопределить метод `setPixels()`.

Листинг 15.4. Циклический сдвиг изображения

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;

class ShiftImage extends Frame{
  private Image img, newimg;
  ShiftImage(String s){
    super(s);
       // 1. Получаем изображение из файла
    img = getToolkit().getImage("javalogo52x88.gif");
       // 2. Создаем экземпляр фильтра
    ImageFilter imf = new ShiftFilter(26);       // Сдвиг на 26 пикселов
       // 3. Получаем новые пикселы с помощью фильтра
    ImageProducer ip = new FilteredImageSource(img.getSource(), imf);
       // 4. Создаем новое изображение
    newimg = createImage(ip);
    setSize(300, 200);
```

```
    setVisible(true);
  }
  public void paint(Graphics gr){
    gr.drawImage(img,     20, 40, this);
    gr.drawImage(newimg, 100, 40, this);
  }
  public static void main(String[] args){
    Frame f= new ShiftImage("  Циклический сдвиг изображения");
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
     // Класс-фильтр
class ShiftFilter extends ImageFilter{
  private int sh; // Сдвиг на sh пикселов вправо.
  public ShiftFilter(int shift){ sh = shift; }
  public void setPixels(int x, int y, int w, int h,
             ColorModel m, byte[] pix, int off, int size){
     for (int k = x; k < x+w; k++){
        if (k+sh <= w)
           consumer.setPixels(k, y, 1, h, m, pix, off+sh+k, size);
        else
           consumer.setPixels(k, y, 1, h, m, pix, off+sh+k-w, size);
     }
   }
}
```

Как видно из листинга 15.4, переопределение метода `setPixels()` заключается в том, чтобы изменить аргументы этого метода, переставив, тем самым, пикселы изображения, и передать их потребителю `consumer` — полю класса `ImageFilter` методом `setPixels()` потребителя. На рис. 15.4 показан результат выполнения этой программы.

Вторая модель обработки изображения введена в Java 2D. Она названа моделью *прямого доступа* (immediate mode model).

Рис. 15.4. Перестановка пикселов изображения

Модель обработки прямым доступом

Подобно тому, как вместо класса `Graphics` система Java 2D использует его расширение `Graphics2D`, описанное в *главе 9*, вместо класса `Image` в Java 2D употребляется его расширение — класс `BufferedImage`. В конструкторе этого класса

```
BufferedImage(int width,  int height, int imageType)
```

задаются размеры изображения и способ хранения точек — одна из констант:

```
TYPE_INT_RGB        TYPE_4BYTE_ABRG       TYPE_USHORT_565_RGB
TYPE_INT_ARGB       TYPE_4BYTE_ABRG_PRE   TYPE_USHORT_555_RGB
TYPE_INT_ARGB_PRE   TYPE_BYTE_GRAY        TYPE_USHORT_GRAY
TYPE_INT_BRG        TYPE_BYTE_BINARY
TYPE_3BYTE_BRG      TYPE_BYTE_INDEXED
```

Как видите, каждый пиксел может занимать 4 байта — `INT`, `4BYTE`, или 2 байта — `USHORT`, или 1 байт — `BYTE`. Может использоваться цветовая модель `RGB`, или добавлена альфа-составляющая — `ARGB`, или задан другой порядок расположения цветовых составляющих — `BRG`, или заданы градации серого цвета — `GRAY`. Каждая составляющая цвета может занимать один байт, 5 битов или 6 битов.

Экземпляры класса `BufferedImage` редко создаются конструкторами. Для их создания чаще обращаются к методам `createImage()` класса `Component` с простым приведением типа:

```
BufferedImage bi = (BufferedImage)createImage(width, height)
```

При этом экземпляр `bi` получает характеристики компонента: цвет фона и цвет рисования, способ хранения точек.

Расположение точек в изображении регулируется классом `Raster` или его подклассом `WritableRaster`. Эти классы задают систему координат изображения, предоставляют доступ к отдельным пикселам методами `getPixel()`, позволяют выделять фрагменты изображения методами `getPixels()`. Класс `WritableRaster` дополнительно разрешает изменять отдельные пикселы методами `setPixel()` или целые фрагменты изображения методами `setPixels()` и `setRect()`.

Начало системы координат изображения — левый верхний угол — имеет координаты `(minX, minY)`, не обязательно равные нулю.

При создании экземпляра класса `BufferedImage` автоматически формируется связанный с ним экземпляр класса `WritableRaster`.

Точки изображения хранятся в скрытом буфере, содержащем одномерный или двумерный массив точек. Вся работа с буфером осуществляется методами одного из классов DataBufferByte, DataBufferInt, DataBufferShort, DataBufferUShort в зависимости от длины данных. Общие свойства этих классов собраны в их абстрактном суперклассе DataBuffer. В нем определены типы данных, хранящихся в буфере: TYPE_BYTE, TYPE_USHORT, TYPE_INT, TYPE_UNDEFINED.

Методы класса DataBuffer предоставляют прямой доступ к данным буфера, но удобнее и безопаснее обращаться к ним методами классов Raster и WritableRaster.

При создании экземпляра класса Raster или класса WritableRaster создается экземпляр соответствующего подкласса класса DataBuffer.

Чтобы отвлечься от способа хранения точек изображения, Raster может обращаться не к буферу DataBuffer, а к подклассам абстрактного класса SampleModel, рассматривающим не отдельные байты буфера, а составляющие (samples) цвета. В модели RGB — это красная, зеленая и синяя составляющие. В пакете java.awt.image есть пять подклассов класса SampleModel:

- ComponentSampleModel — каждая составляющая цвета хранится в отдельном элементе массива DataBuffer;
- BandedSampleModel — данные хранятся по составляющим, составляющие одного цвета хранятся обычно в одном массиве, а DataBuffer содержит двумерный массив: по массиву для каждой составляющей; данный класс расширяет класс ComponentSampleModel;
- PixelInterleavedSampleModel — все составляющие цвета одного пиксела хранятся в соседних элементах единственного массива DataBuffer; данный класс расширяет класс ComponentSampleModel;
- MultiPixelPackedSampleModel — цвет каждого пиксела содержит только одну составляющую, которая может быть упакована в один элемент массива DataBuffer;
- SinglePixelPackedSampleModel — все составляющие цвета каждого пиксела хранятся в одном элементе массива DataBuffer.

На рис. 15.5 представлена иерархия классов Java 2D, реализующая модель прямого доступа.

Итак, Java 2D создает сложную и разветвленную трехслойную систему DataBuffer — SampleModel — Raster управления данными изображения BufferedImage. Вы можете манипулировать точками изображения, используя их координаты в методах классов Raster или спуститься на уровень ниже и обращаться к составляющим цвета пиксела методами классов SampleModel. Если же вам надо работать с отдельными байтами, воспользуйтесь классами DataBuffer.

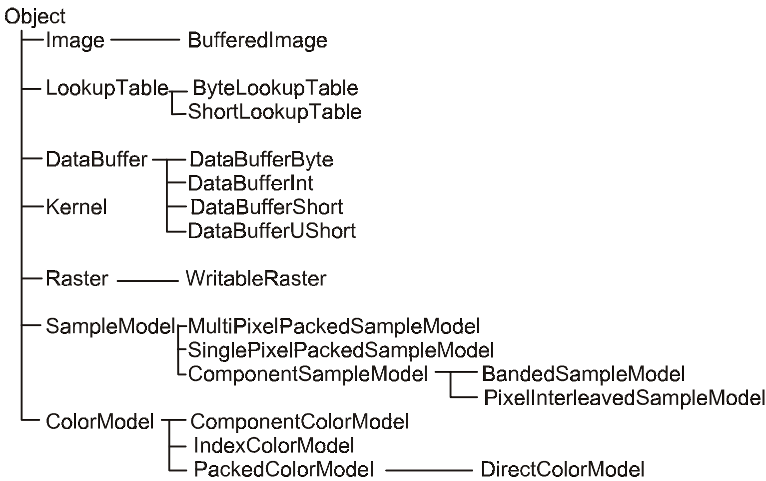


Рис. 15.5. Классы, реализующие модель прямого доступа

Применять эту систему приходится редко, только при создании своего способа преобразования изображения. Стандартные же преобразования выполняются очень просто.

Преобразование изображения в Java 2D

Преобразование изображения `source`, хранящегося в объекте класса `BuffredImage`, в новое изображение `destination` выполняется методом

```
filter(BuffredImage source, BuffredImage destination)
```

описанным в интерфейсе `BuffredImageOp`. Указанный метод возвращает ссылку на новый, измененный объект `destination` класса `BuffredImage`, что позволяет задать цепочку последовательных преобразований.

Можно преобразовать только координатную систему изображения методом

```
filter(Raster source, WritableRaster destination)
```

возвращающим ссылку на измененный объект класса `WritableRaster`. Данный метод описан в интерфейсе `RasterOp`.

Способ преобразования определяется классом, реализующим эти интерфейсы, а параметры преобразования задаются в конструкторе класса.

В пакете `java.awt.image` есть шесть классов, реализующих интерфейсы `BuffredImageOp` и `RasterOp`:

- `AffineTransformOp` — выполняет аффинное преобразование изображения: сдвиг, поворот, отражение, сжатие или растяжение по осям;
- `RescaleOp` — изменяет интенсивность изображения;

- LookupOp — изменяет отдельные составляющие цвета изображения;
- BandCombineOp — меняет составляющие цвета в Raster;
- ColorConvertOp — изменяет цветовую модель изображения;
- ConvolveOp — выполняет свертку, позволяющую изменить контраст и/или яркость изображения, создать эффект "размытости" и другие эффекты.

Рассмотрим, как можно применить эти классы для преобразования изображения.

Аффинное преобразование изображения

Класс AffineTransform и его использование подробно разобраны в *главе 9*, здесь мы только применим его для преобразования изображения.

В конструкторе класса AffineTransformOp указывается предварительно созданное аффинное преобразование at и способ интерполяции interp и/или правила визуализации hints:

```
AffineTransformOp(AffineTransform at, int interp);
AffineTransformOp(AffineTransform at, RenderingHints hints);
```

Способ интерполяции — это одна из двух констант: TYPE_NEAREST_NEIGHBOR (по умолчанию во втором конструкторе) или TYPE_BILINEAR.

После создания объекта класса AffineTransformOp применяется метод filter(). При этом изображение преобразуется внутри новой области типа BufferedImage, как показано на рис. 15.6, справа. Сама область выделена черным цветом.

Другой способ аффинного преобразования изображения — применить метод

```
drawImage(BufferedImage img, BufferedImageOp op, int x, int y)
```

класса Graphics2D. При этом преобразуется вся область img, как продемонстрировано на рис. 15.6, посередине.

В листинге 15.5 показано, как задаются преобразования, представленные на рис. 15.6.

Обратите внимание на особенности работы с BufferedImage. Надо создать графический контекст изображения и вывести в него изображение. Эти действия кажутся лишними, но удобны для двойной буферизации, которая сейчас стала стандартом перерисовки изображений, а в библиотеке Swing выполняется автоматически.

Листинг 15.5. Аффинное преобразование изображения

```
import java.awt.*;
import java.awt.geom.*;
```

```
import java.awt.image.*;
import java.awt.event.*;

public class AffOp extends Frame{
  private BufferedImage bi;
  public AffOp(String s){
     super(s);
          // Загружаем изображение img
     Image img = getToolkit().getImage("javalogo52x88.gif");
          // В этом блоке организовано ожидание загрузки
     try{
        MediaTracker mt = new MediaTracker(this);
        mt.addImage(img, 0);
        mt.waitForID(0);       // Ждем окончания загрузки
     }catch(Exception e){}
          // Размеры создаваемой области bi совпадают
          // с размерами изображения img
     bi = new BufferedImage(img.getWidth(this), img.getHeight(this),
           BufferedImage.TYPE_INT_RGB);
          // Создаем графический контекст big изображения bi
     Graphics2D big = bi.createGraphics();
          // Выводим изображение img в графический контекст big
     big.drawImage(img, 0, 0, this);
   }
  public void paint(Graphics g){
     Graphics2D g2 = (Graphics2D)g;
     int w = getSize().width;
     int h = getSize().height;
     int bw = bi.getWidth(this);
     int bh = bi.getHeight(this);
          // Создаем аффинное преобразование at
     AffineTransform at = new AffineTransform();
     at.rotate(Math.PI/4);      // Задаем поворот на 45 градусов
          // по часовой стрелке вокруг левого верхнего угла.
          // Затем сдвигаем изображение вправо на величину bw
     at.preConcatenate(new AffineTransform(1, 0, 0, 1, bw, 0));
          // Определяем область хранения bimg преобразованного
          // изображения. Ее размер вдвое больше исходного
     BufferedImage bimg =
       new BufferedImage(2*bw, 2*bw, BufferedImage.TYPE_INT_RGB);
          // Создаем объект biop, содержащий преобразование at
     BufferedImageOp biop = new AffineTransformOp(at,
                    AffineTransformOp.TYPE_NEAREST_NEIGHBOR);
          // Преобразуем изображение, результат заносим в bimg
     biop.filter(bi, bimg);
```

```
        // Выводим исходное изображение.
    g2.drawImage(bi, null, 10, 30);
        // Выводим измененную преобразованием biop область bi
    g2.drawImage(bi, biop, w/4+3, 30);
        // Выводим преобразованное внутри области bimg изображение
    g2.drawImage(bimg, null, w/2+3, 30);
  }
  public static void main(String[] args){
    Frame f = new AffOp("  Аффинное преобразование");
    f.addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent e){
        System.exit(0);
      }
    });
    f.setSize(400, 200);
    f.setVisible(true);
  }
}
```

На рис. 15.6 показано исходное изображение, преобразованная область и преобразованное внутри области изображение.

Рис. 15.6. Аффинное преобразование изображения

Изменение интенсивности изображения

Изменение интенсивности изображения выражается математически в умножении каждой составляющей цвета на число `factor` и прибавлении к результату умножения числа `offset`. Результат приводится к диапазону значений составляющей. После этого интенсивность каждой составляющей цвета линейно изменяется в одном и том же масштабе.

Числа `factor` и `offset` постоянны для каждого пиксела и задаются в конструкторе класса вместе с правилами визуализации `hints`:

```
RescaleOp(float factor, float offset, RenderingHints hints)
```

После этого остается применить метод `filter()`.

На рис. 15.7 интенсивность каждого цвета уменьшена вдвое, в результате белый фон стал серым, а цвета — темнее. Затем интенсивность увеличена на 70 единиц. В листинге 15.6 приведена программа, выполняющая это преобразование.

Листинг 15.6. Изменение интенсивности изображения

```
import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;

public class Rescale extends Frame{
  private BufferedImage bi;
  public Rescale(String s){
     super(s);
     Image img = getToolkit().getImage("javalogo52x88.gif");
     try{
        MediaTracker mt = new MediaTracker(this);
        mt.addImage(img, 0);
        mt.waitForID(0);
     }catch(Exception e){}

     bi = new BufferedImage(img.getWidth(this), img.getHeight(this),
            BufferedImage.TYPE_INT_RGB);
     Graphics2D big = bi.createGraphics();
     big.drawImage(img, 0, 0, this);
  }
  public void paint(Graphics g){
     Graphics2D g2 = (Graphics2D)g;
     int w = getSize().width;
     int bw = bi.getWidth(this);
     int bh = bi.getHeight(this);
     BufferedImage bimg =
         new BufferedImage(bw, bh, BufferedImage.TYPE_INT_RGB);
//---------- Начало определения преобразования ------------
     RescaleOp rop = new RescaleOp(0.5f, 70.0f, null);
     rop.filter(bi, bimg);
//---------- Конец определения преобразования -------------
     g2.drawImage(bi, null, 10, 30);
     g2.drawImage(bimg, null, w/2+3, 30);
  }
  public static void main(String[] args){
     Frame f = new Rescale("  Изменение интенсивности");
     f.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
```

```
            System.exit(0);
        }
      });
      f.setSize(300, 200);
      f.setVisible(true);
   }
}
```

Рис. 15.7. Изменение интенсивности изображения

Изменение составляющих цвета

Чтобы изменить отдельные составляющие цвета, надо прежде всего посмотреть тип хранения элементов в `BufferedImage`, по умолчанию это `TYPE_INT_RGB`. Здесь три составляющие — красная, зеленая и синяя. Каждая составляющая цвета занимает один байт, все они хранятся в одном числе типа `int`. Затем надо составить таблицу новых значений составляющих. В листинге 15.7 это двумерный массив `samples`. Потом заполняем данный массив нужными значениями составляющих каждого цвета. В листинге 15.7 задается ярко-красный цвет рисования и белый цвет фона. По полученной таблице создаем экземпляр класса `ByteLookupTable`, который свяжет эту таблицу с буфером данных. Этот экземпляр используем для создания объекта класса `LookupOp`. Наконец, применяем метод `filter()` этого класса.

В листинге 15.7 приведен только фрагмент программы. Для получения полной программы его надо вставить в листинг 15.6 вместо выделенного в нем фрагмента. Логотип Java будет нарисован ярко-красным цветом.

Листинг 15.7. Изменение составляющих цвета

```
//---------------- Вставить в листинг 15.6 ---------------
    byte samples[][] = new byte[3][256];
    for (int j = 0; j < 255; j++){
      samples[0][j] = (byte)(255);        // Красная составляющая
      samples[1][j] = (byte)(0);          // Зеленая составляющая
      samples[2][j] = (byte)(0);          // Синяя составляющая
    }
```

```
        samples[0][255] = (byte)(255);    // Цвет фона — белый
        samples[1][255] = (byte)(255);
        samples[2][255] = (byte)(255);
    ByteLookupTable blut=new ByteLookupTable(0, samples);
    LookupOp lop = new LookupOp(blut, null);
    lop.filter(bi, bimg);
//------------------ Конец вставки -----------------------
```

Создание различных эффектов

Операция свертки (convolution) задает значение цвета точки в зависимости от цветов окружающих точек следующим образом.

Пусть точка с координатами `(x, y)` имеет цвет, выражаемый числом `A(x, y)`. Составляем массив из девяти вещественных чисел `w(0)`, `w(1)`, ... `w(8)`. Тогда новое значение цвета точки с координатами `(x, y)` будет равно:

```
w(0)*A(x-1, y-1)+w(1)*A(x, y-1)+w(2)*A(x+1, y-1)+
w(3)*A(x-1, y)+w(4)*A(x, y)+w(5)*A(x+1, y)+
w(6)*A(x-1, y+1)+w(7)*A(x, y+1)+w(8)*A(x+1, y+1)
```

Задавая различные значения весовым коэффициентам `w(i)`, будем получать различные эффекты, усиливая или уменьшая влияние соседних точек.

Если сумма всех девяти чисел `w(i)` равна `1.0f`, то интенсивность цвета останется прежней. Если при этом все веса равны между собой, т. е. равны `0.1111111f`, то получим эффект размытости, тумана, дымки. Если вес `w(4)` значительно больше остальных при общей сумме их `1.0f`, то возрастет контрастность, возникнет эффект графики, штрихового рисунка.

Можно свернуть не только соседние точки, но и следующие ряды точек, взяв массив весовых коэффициентов из 15 элементов (3×5, 5×3), 25 элементов (5×5) и больше.

В Java 2D свертка делается так. Сначала определяем массив весов, например:

```
float[] w = {0, -1, 0, -1, 5, -1, 0, -1, 0};
```

Затем создаем экземпляр класса `Kernel` — ядра свертки:

```
Kernel kern = new Kernel(3, 3, w);
```

Потом объект класса `ConvolveOp` с этим ядром:

```
ConvolveOp conv = new ConvolveOp(kern);
```

Все готово, применяем метод `filter()`:

```
conv.filter(bi, bimg);
```

В листинге 15.8 записаны действия, необходимые для создания эффекта "размытости".

Листинг 15.8. Создание различных эффектов

```
//-------------- Вставить в листинг 15.6 ------------------------
  float[] w1 = { 0.11111111f, 0.11111111f, 0.11111111f,
                 0.11111111f, 0.11111111f, 0.11111111f,
                 0.11111111f, 0.11111111f, 0.11111111f };
  Kernel kern = new Kernel(3, 3, w1);
  ConvolveOp cop = new ConvolveOp(kern, ConvolveOp.EDGE_NO_OP, null);
  cop1.filter(bi, bimg);
//------------- Конец вставки ------------------------------------
```

На рис 15.8 представлены слева направо исходное изображение и изображения, преобразованные весовыми матрицами `w1`, `w2` и `w3`, где матрица `w1` показана в листинге 15.8, а матрицы `w2` и `w3` выглядят так:

```
float[] w2 = { 0, -1, 0,-1, 4, -1, 0, -1, 0 };
float[] w3 = { -1, -1, -1,-1, 9, -1, -1, -1, -1 };
```

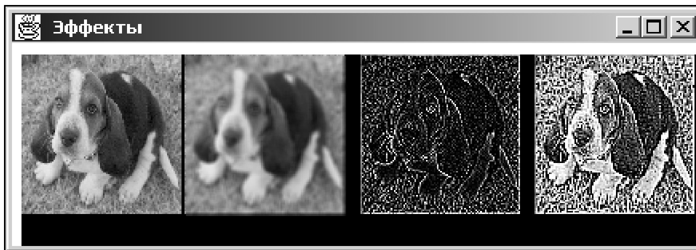


Рис. 15.8. Создание эффектов

Анимация

Есть несколько способов создать анимацию. Самый простой из них — записать заранее все необходимые кадры в графические файлы, загрузить их в оперативную память в виде объектов класса `Image` или `BufferedImage` и выводить по очереди на экран.

Это сделано в листинге 15.9. Заготовлено десять кадров в файлах run1.gif, run2.gif, , run10.gif. Они загружаются в массив `img[]` и выводятся на экран циклически 100 раз, с задержкой в 0,1 сек.

Листинг 15.9. Простая анимация

```
import java.awt.*;
import java.awt.event.*;
```

```
class SimpleAnim extends Frame{
  private Image[] img = new Image[10];
  private int count;
  SimpleAnim(String s){
    super(s);
    MediaTracker tr = new MediaTracker(this);
    for (int k = 0; k < 10; k++){
       img[k] = getToolkit().getImage("run"+(k+1)+".gif");
       tr.addImage(img[k], 0);
    }
    try{
       tr.waitForAll();      // Ждем загрузки всех изображений
    }catch(InterruptedException e){}
    setSize(400, 300);
    setVisible(true);
  }
  public void paint(Graphics g){
    g.drawImage(img[count % 10], 0, 0, this);
  }
//  public void update(Graphics g){ paint(g); }
  public void go(){
    while(count < 100){
      repaint();            // Выводим следующий кадр
      try{                  // Задержка в 0.1 сек
         Thread.sleep(100);
      }catch(InterruptedException e){}
      count++;
    }
  }
  public static void main(String[] args){
    SimpleAnim f = new SimpleAnim("  Простая анимация");
    f.go();
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

Обратите внимание на следующее важное обстоятельство. Мы не можем обратиться прямо к методу `paint()` для перерисовки окна компонента, потому что выполнение этого метода связано с операционной системой — метод `paint()` выполняется автоматически при каждом изменении содержимого окна, его перемещении и изменении размеров. Для запроса на перерисовку окна в классе `Component` есть метод `repaint()`.

Метод `repaint()` ждет, когда представится возможность перерисовать окно, и потом обращается к методу `update(Graphics g)`. При этом несколько обращений к `repaint()` могут быть произведены исполняющей системой Java за один раз.

Метод `update()` сначала обращается к методу `g.clearRect()`, заполняющему окно цветом фона, а уж затем к методу `paint(g)`. Полный исходный текст таков:

```
public void update(Graphics g){
   if ((this instanceof java.awt.Canvas) ||
       (this instanceof java.awt.Panel)  ||
       (this instanceof java.awt.Frame)  ||
       (this instanceof java.awt.Dialog) ||
       (this instanceof java.awt.Window)){
              g.clearRect(0, 0, width, height);
   }
   paint(g);
}
```

Если кадры анимации полностью перерисовывают окно, то его очистка методом `clearRect()` не нужна. Более того, она часто вызывает неприятное мерцание из-за появления на мгновение белого фона. В таком случае надо сделать следующее переопределение:

```
public void update(Graphics g){
    paint(g);
}
```

В листинге 15.9 это переопределение сделано как комментарий.

Для "легких" компонентов дело обстоит сложнее. Метод `repaint()` последовательно обращается к методам `repaint()` объемлющих "легких" контейнеров, пока не встретится "тяжелый" контейнер, чаще всего это экземпляр класса `Container`. В нем вызывается метод `update()`, очищающий и перерисовывающий контейнер. После этого идет обращение к методам `update()` всех "легких" компонентов в контейнере.

Отсюда следует, что для устранения мерцания "легких" компонентов необходимо переопределять метод `update()` первого объемлющего "тяжелого" контейнера, обращаясь в нем к методам `super.update(g)` или `super.paint(g)`.

Если кадры покрывают только часть окна, причем каждый раз новую, то очистка окна необходима, иначе старые кадры останутся в окне, появится "хвост". Чтобы устранить мерцание, используют прием, получивший название *"двойная буферизация"* (double buffering).

Улучшение изображения двойной буферизацией

Суть двойной буферизации в том, что в оперативной памяти создается буфер — объект класса `Image` или `BufferedImage`, и вызывается его графический контекст, в котором формируется изображение. Там же происходит очистка буфера, которая тоже не отражается на экране. Только после выполнения всех действий готовое изображение выводится на экран.

Все это происходит в методе `update()`, а метод `paint()` только обращается к `update()`. Листинги 15.10—15.11 разъясняют данный прием.

Листинг 15.10. Двойная буферизация с помощью класса `Image`

```
public void update(Graphics g){
  int w = getSize().width, h = getSize().height;
       // Создаем изображение-буфер в оперативной памяти
  Image offImg = createImage(w, h);
       // Получаем его графический контекст
  Graphics offGr = offImg.getGraphics();
       // Меняем текущий цвет буфера на цвет фона
  offGr.setColor(getBackground());
       // и заполняем им окно компонента, очищая буфер
  offGr.fillRect(0, 0, w, h);
       // Восстанавливаем текущий цвет буфера
  offGr.setColor(getForeground());
       // Для листинга 15.9 выводим в контекст изображение
  offGr.drawImage(img[count % 10], 0, 0, this);
       // Рисуем в графическом контексте буфера
       // (необязательное действие)
  paint(offGr);
       // Выводим изображение-буфер на экран
       // (можно перенести в метод paint())
  g.drawImage(offImg, 0, 0, this);
}
       // Метод paint() необязателен
public void paint(Graphics g){ update(g); }
```

Листинг 15.11. Двойная буферизация с помощью класса `BufferedImage`

```
public void update(Graphics g){
   Graphics2D g2 = (Graphics2D)g;
   int w = getSize().width, h = getSize().height;
      // Создаем изображение-буфер в оперативной памяти
   BufferedImage bi = (BufferedImage)createImage(w, h);
```

```
      // Создаем графический контекст буфера
   Graphics2D big = bi.createGraphics();
       // Устанавливаем цвет фона
   big.setColor(getBackground());
       // Очищаем буфер цветом фона
   big.clearRect(0, 0, w, h);
       // Восстанавливаем текущий цвет
   big.setColor(getForeground());
   // Выводим что-нибудь в графический контекст big
   // ...
   // Выводим буфер на экран
   g2.drawImage(bi, 0, 0, this);
}
```

Метод двойной буферизации стал фактическим стандартом вывода изменяющихся изображений, а в библиотеке Swing он применяется автоматически.

Данный метод удобен и при перерисовке отдельных частей изображения. В этом случае в изображении-буфере рисуется неизменяемая часть изображения, а в методе `paint()` — то, что меняется при каждой перерисовке.

В листинге 15.12 показан второй способ анимации — кадры изображения рисуются непосредственно в программе, в методе `update()`, по заданному закону изменения изображения. В результате красный мячик прыгает на фоне изображения.

Листинг 15.12. Анимация рисованием

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.awt.image.*;

class DrawAnim1 extends Frame{
  private Image img;
  private int count;

  DrawAnim1(String s){
    super(s);
    MediaTracker tr = new MediaTracker(this);
    img = getToolkit().getImage("back2.jpg");
    tr.addImage(img, 0);
    try{
        tr.waitForID(0);
    }catch(InterruptedException e){}

    setSize(400, 400);
    setVisible(true);
  }
```

```
  public void update(Graphics g){
    Graphics2D g2 = (Graphics2D)g;
    int w = getSize().width, h = getSize().height;
    BufferedImage bi = (BufferedImage)createImage(w, h);
    Graphics2D big = bi.createGraphics();
       // Заполняем фон изображением img
    big.drawImage(img, 0, 0, this);
       // Устанавливаем цвет рисования
    big.setColor(Color.red);
       // Рисуем в графическом контексте буфера круг,
       // перемещающийся по синусоиде
    big.fill(new Arc2D.Double(4*count, 50+30*Math.sin(count),
                   50, 50, 0, 360, Arc2D.OPEN));
       // Меняем цвет рисования
    big.setColor(getForeground());
       // Рисуем горизонтальную прямую
    big.draw(new Line2D.Double(0, 125, w, 125));
       // Выводим изображение-буфер на экран
    g2.drawImage(bi, 0, 0, this);
  }
  public void go(){
    while(count < 100){
      repaint();
      try{
         Thread.sleep(10);
      }catch(InterruptedException e){}
      count++;
    }
  }
  public static void main(String[] args){
    DrawAnim1 f = new DrawAnim1("  Анимация");
    f.go();
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

Эффект мерцания, переливы цвета, затемнение и прочие эффекты, получающиеся заменой отдельных пикселов изображения, удобно создавать с помощью класса `MemoryImageSource`. Методы `newPixels()` этого класса вызывают немедленную перерисовку изображения даже без обращения к методу `repaint()`, если перед этим выполнен метод `setAnimated(true)`. Чаще всего применяются два метода:

- `newPixels(int x, int y, int width, int height)` — получателю посылается указанный аргументами прямоугольный фрагмент изображения;
- `newPixels()` — получателю посылается все изображение.

В листинге 15.13 показано применение этого способа. Квадрат, выведенный на экран, переливается разными цветами.

Листинг 15.13. Анимация с помощью `MemoryImageSource`

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;

class InMemory extends Frame{
  private int w = 100, h = 100, count;
  private int[] pix = new int[w * h];
  private Image img;
  MemoryImageSource mis;
  InMemory(String s){
    super(s);
    int i = 0;
    for(int y = 0; y < h; y++){
      int red = 255 * y / (h — 1);
      for(int x = 0; x < w; x++){
        int green = 255 * x / (w — 1);
        pix[i++] = (255 << 24)|(red << 16)|(green << 8)| 128;
      }
    }
    mis = new MemoryImageSource(w, h, pix, 0, w);
         // Задаем возможность анимации
    mis.setAnimated(true);
    img = createImage(mis);
    setSize(350, 300);
    setVisible(true);
  }
  public void paint(Graphics gr){
    gr.drawImage(img, 10, 30, this);
  }
  public void update(Graphics g){ paint(g); }

  public void go(){
    while(count < 100){
      int i = 0;
            // Изменяем массив пикселов по некоторому закону
      for(int y = 0; y < h; y++)
```

```
         for(int x = 0; x < w; x++)
            pix[i++] = (255 << 24)|(255 + 8 * count << 16)|
                    (8*count << 8)| 255 + 8 * count;
            // Уведомляем потребителя об изменении
      mis.newPixels();
      try{
         Thread.sleep(100);
      }catch(InterruptedException e){}
      count++;
    }
  }
  public static void main(String[] args){
    InMemory f= new InMemory("  Изображение в памяти");
    f.go();
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

Вот и все средства для анимации, остальное — умелое их применение. Комбинируя рассмотренные способы, можно добиться удивительных эффектов. В документации SUN J2SDK, в каталогах demo\applets и demo\jfc\Java2D \src, приведено много примеров апплетов и приложений с анимацией.

Звук

Как было указано в предыдущей главе, в апплетах реализуется интерфейс `AudioClip`. Экземпляр объекта, реализующего этот интерфейс можно получить методом `getAudioClip()`, который, кроме того, загружает звуковой файл, а затем пользоваться методами `play()`, `loop()` и `stop()` этого интерфейса для проигрывания музыки.

Для применения данного же приема в приложениях в класс `Applet` введен статический метод `newAudioClip(URL address)`, загружающий звуковой файл, находящийся по адресу `address`, и возвращающий объект, реализующий интерфейс `AudioClip`. Его можно использовать для проигрывания звука в приложении, если конечно звуковая система компьютера уже настроена.

В листинге 15.14 приведено простейшее консольное приложение, бесконечно проигрывающее звуковой файл doom.mid, находящийся в текущем каталоге. Для завершения приложения требуется применить средства операционной системы, например, комбинацию клавиш <Ctrl>+<C>.

Листинг 15.14. Простейшее аудиоприложение

```
import java.applet.*;
import java.net.*;

class SimpleAudio{
  SimpleAudio(){
    try{
    AudioClip ac = Applet.newAudioClip(new URL("file:doom.mid"));
    ac.loop();
    }catch(Exception e){}
  }
  public static void main(String[] args){
    new SimpleAudio();
  }
}
```

Таким способом можно проигрывать звуковые файлы типов AU, WAVE, AIFF, MIDI без сжатия.

В состав виртуальной машины Java, входящей в SUN J2SDK начиная с версии 1.3, включено устройство, проигрывающее звук, записанный в одном из форматов AU, WAVE, AIFF, MIDI, преобразующее, микширующее и записывающее звук в тех же форматах.

Для работы с этим устройством созданы классы, собранные в пакеты `javax.sound.sampled`, `javax.sound.midi`, `javax.sound.sampled.spi` и `javax.sound.midi.spi`. Перечисленный набор классов для работы со звуком получил название Java Sound API.

Проигрывание звука в Java 2

Проигрыватель звука, встроенный в JVM, рассчитан на два способа записи звука: моно и стерео оцифровку (digital audio) с частотой дискретизации (sample rate) от 8 000 до 48 000 Гц и аппроксимацией (quantization) 8 и 16 битов, и MIDI-последовательности (sequences) типа 0 и 1.

Оцифрованный звук должен храниться в файлах типа AU, WAVE и AIFF. Его можно проигрывать двумя способами.

Первый способ описан в интерфейсе `Clip`. Он рассчитан на воспроизведение небольших файлов или неоднократное проигрывание файла и заключается в том, что весь файл целиком загружается в оперативную память, а затем проигрывается.

Второй способ описан в интерфейсе `SourceDataLine`. Согласно этому способу файл загружается в оперативную память по частям в буфер, размер которого можно задать произвольно.

Перед загрузкой файла надо задать формат записи звука в объекте класса `AudioFormat`. Конструктор этого класса:

```
AudioFormat(float sampleRate, int sampleSize, int  channels,
            boolean signed, boolean bigEndian)
```

требует знания частоты дискретизации `sampleRate` (по умолчанию 44 100 Гц), аппроксимации `sampleSize`, заданной в битах (по умолчанию 16), числа каналов `channels` (1 — моно, по умолчанию 2 — стерео), запись чисел со знаком, `signed == true`, или без знака, и порядка расположения байтов в числе `bigEndian`. Такие сведения обычно неизвестны, поэтому их получают косвенным образом из файла. Это осуществляется в два шага.

На первом шаге получаем формат файла статическим методом `getAudioFileFormat()` класса `AudioSystem`, на втором — формат записи звука методом `getFormat()` класса `AudioFileFormat`. Это описано в листинге 15.15. После того как формат записи определен и занесен в объект класса `AudioFormat`, в объекте класса `DataLine.Info` собирается информация о входной линии (line) и способе проигрывания `Clip` или `SourceDataLine`. Далее следует проверить, сможет ли проигрыватель обслуживать линию с таким форматом. Затем надо связать линию с проигрывателем статическим методом `getLine()` класса `AudioSystem`. Потом создаем поток данных из файла — объект класса `AudioInputStream`. Из этого потока тоже можно извлечь объект класса `AudioFormat` методом `getFormat()`. Данный вариант выбран в листинге 15.16. Открываем созданный поток методом `open()`.

У-фф! Все готово, теперь можно начать проигрывание методом `start()`, завершить методом `stop()`, "перемотать" в начало методом `setFramePosition(0)` или `setMillisecondPosition(0)`.

Можно задать проигрывание `n` раз подряд методом `loop(n)` или бесконечное число раз методом `loop(Clip.LOOP_CONTINUOUSLY)`. Перед этим необходимо установить начальную `n` и конечную `m` позиции повторения методом `setLoopPoints(n, m)`.

По окончании проигрывания следует закрыть линию методом `close()`.

Вся эта последовательность действий показана в листинге 15.15.

Листинг 15.15. Проигрывание аудиоклипа

```
import javax.sound.sampled.*;
import java.io.*;

class PlayAudio{
  PlayAudio(String s){
    play(s);
  }
```

```
  public void play(String file){
    Clip line = null;
    try{
           // Создаем объект, представляющий файл
    File f = new File(file);
           // Получаем информацию о способе записи файла
       AudioFileFormat aff = AudioSystem.getAudioFileFormat(f);
           // Получаем информацию о способе записи звука
       AudioFormat af = aff.getFormat();
           // Собираем всю информацию вместе,
           // добавляя сведения о классе Class
       DataLine.Info info = new DataLine.Info(Clip.class, af);
           // Проверяем, можно ли проигрывать такой формат
       if (!AudioSystem.isLineSupported(info)){
          System.err.println("Line is not supported");
          System.exit(0);
       }
            // Получаем линию связи с файлом
       line = (Clip)AudioSystem.getLine(info);
            // Создаем поток байтов из файла
       AudioInputStream ais = AudioSystem.getAudioInputStream(f);
            // Открываем линию
       line.open(ais);
    }catch(Exception e){
      System.err.println(e);
    }
       // Начинаем проигрывание
    line.start();
       // Здесь надо сделать задержку до окончания проигрывания
       // или остановить его следующим методом:
    line.stop();
       // По окончании проигрывания закрываем линию
    line.close();
  }
  public static void main(String[] args){
    if (args.length != 1)
       System.out.println("Usage: java PlayAudio filename");
    new PlayAudio(args[0]);
  }
}
```

Как видите, методы Java Sound API выполняют элементарные действия, которые надо повторять из программы в программу. Как говорят, это методы "низкого уровня" (low level).

Второй способ, использующий методы интерфейса `SourceDataLine`, требует предварительного создания буфера произвольного размера.

Листинг 15.16. Проигрывание аудиофайла

```
import javax.sound.sampled.*;
import java.io.*;

class PlayAudioLine{
  PlayAudioLine(String s){
    play(s);
  }
  public void play(String file){
    SourceDataLine line = null;
    AudioInputStream ais = null;
    byte[] b = new byte[2048];          // Буфер данных
    try{
    File f = new File(file);
         // Создаем входной поток байтов из файла f
      ais = AudioSystem.getAudioInputStream(f);
         // Извлекаем из потока информацию о способе записи звука
      AudioFormat af = ais.getFormat();
         // Заносим эту информацию в объект info
      DataLine.Info info = new DataLine.Info(SourceDataLine.class, af);
         // Проверяем, приемлем ли такой способ записи звука
      if (!AudioSystem.isLineSupported(info)){
         System.err.println("Line is not supported");
         System.exit(0);
      }
         // Получаем входную линию
      line = (SourceDataLine)AudioSystem.getLine(info);
         // Открываем линию
      line.open(af);
         // Начинаем проигрывание
      line.start();    // Ждем появления данных в буфере
      int num = 0;
         // Раз за разом заполняем буфер
      while(( num = ais.read(b)) != -1)
          line.write(b, 0, num);
        // "Сливаем" буфер, проигрывая остаток файла
      line.drain();
        // Закрываем поток
      ais.close();
    }catch(Exception e){
       System.err.println(e);
    }
        // Останавливаем проигрывание
    line.stop();
```

```
      // Закрываем линию
    line.close();
  }
  public static void main(String[] args){
    String s = "mrmba.aif";
    if (args.length > 0) s = args[0];
    new PlayAudioLine(s);
  }
}
```

Управлять проигрыванием файла можно с помощью событий. Событие класса `LineEvent` происходит при открытии, `OPEN`, и закрытии, `CLOSE`, потока, при начале, `START`, и окончании, `STOP`, проигрывания. Характер события отмечается указанными константами. Соответствующий интерфейс `LineListener` описывает только один метод `update()`.

В MIDI-файлах хранится *последовательность* (sequence) команд для *секвенсора* (sequencer) — устройства для записи, проигрывания и редактирования MIDI-последовательности, которым может быть физическое устройство или программа. Последовательность состоит из нескольких *дорожек* (tracks), на которых записаны *MIDI-события* (events). Каждая дорожка загружается в своем *канале* (channel). Обычно дорожка содержит звучание одного музыкального инструмента или запись голоса одного исполнителя или запись нескольких исполнителей, микшированную *синтезатором* (synthesizer).

Для проигрывания MIDI-последовательности в простейшем случае надо создать экземпляр секвенсора, открыть его и направить в него последовательность, извлеченную из файла, как показано в листинге 15.17. После этого следует начать проигрывание методом `start()`. Закончить проигрывание можно методом `stop()`, "перемотать" последовательность на начало записи или на указанное время проигрывания — методами `setMicrosecondPosition(long mcs)` или `setTickPosition(long tick)`.

Листинг 15.17. Проигрывание MIDI-последовательности

```
import javax.sound.midi.*;
import java.io.*;

class PlayMIDI{
  PlayMIDI(String s){
    play(s);
  }
  public void play(String file){
    try{
       File f = new File(file);
```

```
            // Получаем секвенсор по умолчанию
       Sequencer sequencer = MidiSystem.getSequencer();
            // Проверяем, получен ли секвенсор
      if (sequencer == null) {
          System.err.println("Sequencer is not supported");
          System.exit(0);
      }
            // Открываем секвенсор
        sequencer.open();
            // Получаем MIDI-последовательность из файла
        Sequence seq = MidiSystem.getSequence(f);
            // Направляем последовательность в секвенсор
        sequencer.setSequence(seq);
            // Начинаем проигрывание
         sequencer.start();
            // Здесь надо сделать задержку на время проигрывания,
            // а затем остановить:
         sequencer.stop();
      }catch(Exception e){
         System.err.println(e);
    }
  }
  public static void main(String[] args){
    String s = "doom.mid";
    if (args.length > 0) s = args[0];
    new PlayMIDI(s);
  }
}
```

Синтез и запись звука в Java 2

Синтез звука заключается в создании MIDI-последовательности — объекта класса `Sequence` — каким-либо способом: с микрофона, линейного входа, синтезатора, из файла, или просто создать в программе, как это делается в листинге 15.18.

Сначала создается пустая последовательность одним из двух конструкторов:

```
Sequence(float divisionType, int resolution)
Sequence(float divisionType, int resolution, int numTracks)
```

Первый аргумент `divisionType` определяет способ отсчета моментов (ticks) MIDI-событий — это одна из констант:

- `PPQ` (Pulses Per Quarter note) — отсчеты замеряются в долях от длительности звука в четверть;

- SMPTE_24, SMPTE_25, SMPTE_30, SMPTE_30DROP (Society of Motion Picture and Television Engineers) — отсчеты в долях одного кадра, при указанном числе кадров в секунду.

Второй аргумент resolution задает количество отсчетов в указанную единицу, например,

```
Sequence seq = new Sequence(Sequence.PPQ, 10);
```

задает 10 отсчетов в звуке длительностью в четверть.

Третий аргумент numTracks определяет количество дорожек в MIDI-последовательности.

Потом, если применялся первый конструктор, в последовательности создается одна или несколько дорожек:

```
Track tr = seq.createTrack();
```

Если применялся второй конструктор, то надо получить уже созданные конструктором дорожки:

```
Track[] trs = seq.getTracks();
```

Затем дорожки заполняются MIDI-событиями с помощью MIDI-сообщений. Есть несколько типов сообщений для разных типов событий. Наиболее часто встречаются сообщения типа ShortMessage, которые создаются конструктором по умолчанию и потом заполняются методом setMessage():

```
ShortMessage msg = new ShortMessage();
msg.setMessage(ShortMessage.NOTE_ON, 60, 93);
```

Первый аргумент указывает тип сообщения: NOTE_ON — начать звучание, NOTE_OFF — прекратить звучание и т. д. Второй аргумент для типа NOTE_ON показывает высоту звука, в стандарте MIDI это числа от 0 до 127, 60 — нота "до" первой октавы. Третий аргумент означает "скорость" нажатия клавиши MIDI-инструмента и по-разному понимается различными устройствами.

Далее создается MIDI-событие:

```
MidiEvent me = new MidiEvent(msg, ticks);
```

Первый аргумент конструктора msg — это сообщение, второй аргумент ticks — время наступления события (в нашем примере проигрывания ноты "до") в единицах последовательности seq (в нашем примере в десятых долях четверти). Время отсчитывается от начала проигрывания последовательности.

Наконец, событие заносится на дорожку:

```
tr.add(me);
```

Указанные действия продолжаются, пока все дорожки не будут заполнены всеми событиями. В листинге 15.18 это делается в цикле, но обычно MIDI-события создаются в методах обработки нажатия клавиш на обычной или

специальной MIDI-клавиатуре. Еще один способ — вывести на экран изображение клавиатуры и создавать MIDI-события в методах обработки нажатий кнопки мыши на этой клавиатуре.

После создания последовательности ее можно проиграть, как в листинге 15.17, или записать в файл или выходной поток. Для этого вместо метода `start()` надо применить метод `startRecording()`, который одновременно и проигрывает последовательность, и подготавливает ее к записи, которую осуществляют статические методы:

```
write(Sequence in, int type, File out)
write(Sequence in, int type, OutputStream out)
```

Второй аргумент `type` задает тип MIDI-файла, который лучше всего определить для заданной последовательности `seq` статическим методом `getMidiFileTypes(seq)`. Данный метод возвращает массив возможных типов. Надо воспользоваться нулевым элементом массива. Все это показано в листинге 15.18.

Листинг 15.18. Создание MIDI-последовательности нот звукоряда

```
import javax.sound.midi.*;
import java.io.*;

class SynMIDI{
  SynMIDI(){
     play(synth());
  }
  public Sequence synth(){
     Sequence seq = null;
     try{
            // Последовательность будет отсчитывать по 10
            // MIDI-событий на звук длительностью в четверть
        seq = new Sequence(Sequence.PPQ, 10);
            // Создаем в последовательности одну дорожку
        Track tr = seq.createTrack();

        for (int k = 0; k < 100; k++){
           ShortMessage msg = new ShortMessage();
                // Пробегаем MIDI-ноты от номера 10 до 109
           msg.setMessage(ShortMessage.NOTE_ON, 10+k, 93);
                // Будем проигрывать ноты через каждые 5 отсчетов
           tr.add(new MidiEvent(msg, 5*k));
           msg = null;
        }
     }catch(Exception e){
        System.err.println("From synth(): "+e);
```

```
        System.exit(0);
     }
     return seq;
  }
  public void play(Sequence seq){
    try{
       Sequencer sequencer = MidiSystem.getSequencer();
       if (sequencer == null){
          System.err.println("Sequencer is not supported");
          System.exit(0);
      }
      sequencer.open();
      sequencer.setSequence(seq);
      sequencer.startRecording();
      int[] type = MidiSystem.getMidiFileTypes(seq);
      MidiSystem.write(seq, type[0], new File("gammas.mid"));
    }catch(Exception e){
        System.err.println("From play(): " + e);
    }
  }
  public static void main(String[] args){
    new SynMIDI();
  }
}
```

К сожалению, объем книги не позволяет коснуться темы о работе с синтезатором (synthesizer), микширования звука, работы с несколькими инструментами и прочих возможностей Java Sound API. В документации SUN J2SDK, в каталоге docs\guide\sound\prog_guide, есть подробное руководство программиста, а в каталоге demo\sound\src лежат исходные тексты синтезатора, использующего Java Sound API.

Часть IV

Необходимые конструкции Java

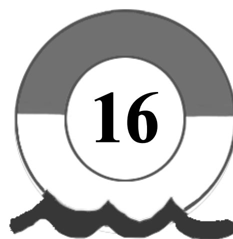
Глава 16. Обработка исключительных ситуаций

Глава 17. Подпроцессы

Глава 18. Потоки ввода/вывода

Глава 19. Сетевые средства Java

ГЛАВА 16



Обработка исключительных ситуаций

Исключительные ситуации (exceptions) могут возникнуть во время выполнения (runtime) программы, прервав ее обычный ход. К ним относится деление на нуль, отсутствие загружаемого файла, отрицательный или вышедший за верхний предел индекс массива, переполнение выделенной памяти и масса других неприятностей, которые могут случиться в самый неподходящий момент.

Конечно, можно предусмотреть такие ситуации и застраховаться от них как-нибудь так:

```
if (something == wrong){
   // Предпринимаем аварийные действия
}else{
   // Обычный ход действий
}
```

Но при этом много времени уходит на проверки, и программа превращается в набор этих проверок. Посмотрите любую штатную производственную программу, написанную на языке С или Pascal, и увидите, что она на 2/3 состоит из таких проверок.

В объектно-ориентированных языках программирования принят другой подход. При возникновении исключительной ситуации исполняющая система создает объект определенного класса, соответствующего возникшей ситуации, содержащий сведения о том, что, где и когда произошло. Этот объект передается на обработку программе, в которой возникло исключение. Если программа не обрабатывает исключение, то объект возвращается обработчику по умолчанию исполняющей системы. Обработчик поступает очень просто: выводит на консоль сообщение о произошедшем исключении и прекращает выполнение программы.

Приведем пример. В программе листинга 16.1 может возникнуть деление на нуль, если запустить ее с аргументом 0. В программе нет никаких средств обработки такой исключительной ситуации. Посмотрите на рис. 16.1, какие сообщения выводит исполняющая система Java.

Листинг 16.1. Программа без обработки исключений

```
class SimpleExt{
  public static void main(String[] args){
    int n = Integer.parseInt(args[0]);
    System.out.println("10 / n = " + (10 / n));
    System.out.println("After all actions");
  }
}
```

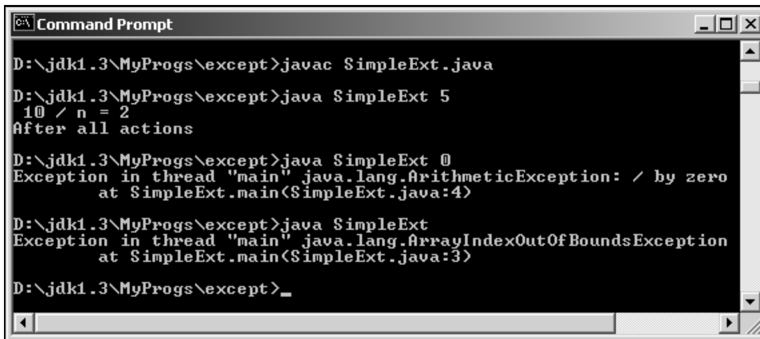


Рис. 16.1. Сообщения об исключительных ситуациях

Программа `SimpleExt` запущена три раза. Первый раз аргумент `args[0]` равен `5` и программа выводит результат: "`10 / n = 2`". После этого появляется второе сообщение: "`After all actions`".

Второй раз аргумент равен `0`, и вместо результата мы получаем сообщение о том, что в подпроцессе "`main`" произошло исключение класса `ArithmeticException` вследствие деления на нуль: "`/ by zero`". Далее уточняется, что исключение возникло при выполнении метода `main` класса `SimpleExt`, а в скобках указано, что действие, в результате которого возникла исключительная ситуация, записано в четвертой строке файла SimpleExt.java. Выполнение программы прекращается, заключительное сообщение не появляется.

Третий раз программа запущена вообще без аргумента. В массиве `args[]` нет элементов, его длина равна нулю, а мы пытаемся обратиться к элементу `args[0]`. Возникает исключительная ситуация класса `ArrayIndexOutOfBoundsException` вследствие действия, записанного в третьей

строке файла SimpleExt.java. Выполнение программы прекращается, обращение к методу `println()` не происходит.

Блоки перехвата исключения

Мы можем перехватить и обработать исключение в программе. При описании обработки применяется бейсбольная терминология. Говорят, что исполняющая система или программа "выбрасывает" (throws) объект-исключение. Этот объект "пролетает" через всю программу, появившись сначала в том методе, где произошло исключение, а программа в одном или нескольких местах пытается (try) его "перехватить" (catch) и обработать. Обработку можно сделать полностью в одном месте, а можно обработать исключение в одном месте, выбросить снова, перехватить в другом месте и обрабатывать дальше.

Мы уже много раз в этой книге сталкивались с необходимостью обрабатывать различные исключительные ситуации, но не делали этого, потому что не хотели отвлекаться от основных конструкций языка. Не вводите это в привычку! Хорошо написанные объектно-ориентированные программы обязательно должны обрабатывать все возникающие в них исключительные ситуации.

Для того чтобы попытаться (try) перехватить (catch) объект-исключение, надо весь код программы, в котором может возникнуть исключительная ситуация, охватить оператором `try{}catch(){}`. Каждый блок `catch(){}` перехватывает исключение только одного типа, того, который указан в его аргументе. Но можно написать несколько блоков `catch(){}` для перехвата нескольких типов исключений.

Например, мы знаем, что в программе листинга 16.1 могут возникнуть исключения двух типов. Напишем блоки их обработки, как это сделано в листинге 16.2.

Листинг 16.2. Программа с блоками обработки исключений

```
class SimpleExt1{
  public static void main(String[] args){
    try{
       int n = Integer.parseInt(args[0]);
       System.out.println("After parseInt());
       System.out.println(" 10 / n = " + (10 / n));
       System.out.println("After results output");
    }catch(ArithmeticException ae){
       System.out.println("From Arithm.Exc. catch: "+ae);
    }catch(ArrayIndexOutOfBoundsException arre){
```

```
        System.out.println("From Array.Exc. catch: "+arre);
    }finally{
        System.out.println("From finally");
    }
    System.out.println("After all actions");
  }
}
```

В программу листинга 16.1 вставлен блок `try{}` и два блока перехвата `catch(){}` для каждого типа исключений. Обработка исключения здесь заключается просто в выводе сообщения и содержимого объекта-исключения, как оно представлено методом `toString()` соответствующего класса-исключения.

После блоков перехвата вставлен еще один, необязательный блок `finally{}`. Он предназначен для выполнения действий, которые надо выполнить обязательно, чтобы ни случилось. Все, что написано в этом блоке, будет выполнено и при возникновении исключения, и при обычном ходе программы, и даже если выход из блока `try{}` осуществляется оператором `return`.

Если в операторе обработки исключений есть блок `finally{}`, то блок `catch(){}` может отсутствовать, т. е. можно не перехватывать исключение, но при его возникновении все-таки проделать какие-то обязательные действия.

Кроме блоков перехвата в листинге 16.2 после каждого действия делается трассировочная печать, чтобы можно было проследить за порядком выполнения программы. Программа запущена три раза: с аргументом 5, с аргументом 0 и вообще без аргумента. Результат показан на рис. 16.2.

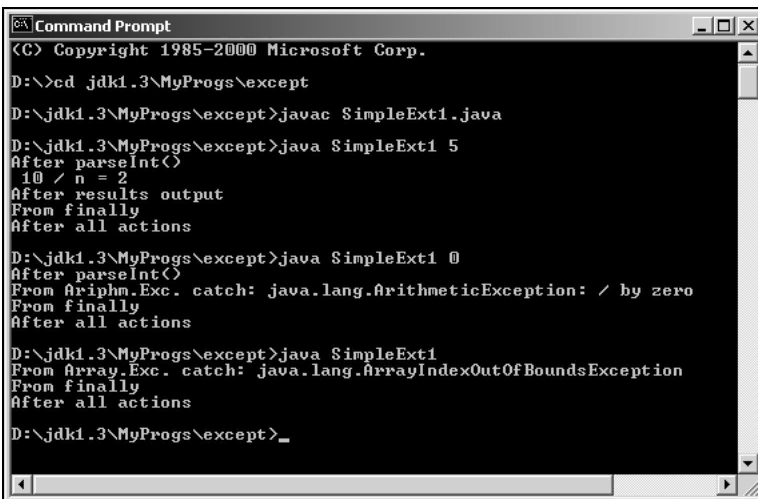


Рис. 16.2. Сообщения обработки исключений

После первого запуска, при обычном ходе программы, выводятся все сообщения.

После второго запуска, приводящего к делению на нуль, управление сразу же передается в соответствующий блок catch(ArithmeticException ae){}, потом выполняется то, что написано в блоке finally{}.

После третьего запуска управление после выполнения метода parseInt() передается в другой блок catch(ArrayIndexOutOfBoundsException arre){}, затем в блок finally{}.

Обратите внимание, что во всех случаях — и при обычном ходе программы, и после этих обработок — выводится сообщение "After all actions". Это свидетельствует о том, что выполнение программы не прекращается при возникновении исключительной ситуации, как это было в программе листинга 16.1, а продолжается после обработки и выполнения блока finally{}.

При записи блоков обработки исключений надо совершенно четко представлять себе, как будет передаваться управление во всех случаях. Поэтому изучите внимательно рис. 16.2.

Интересно, что пустой блок catch(){}, в котором между фигурными скобками нет ничего, даже пробела, тоже считается обработкой исключения и приводит к тому, что выполнение программы не прекратится. Именно так мы "обрабатывали" исключения в предыдущих главах.

Немного выше было сказано, что выброшенное исключение "пролетает" через всю программу. Что это означает? Изменим программу листинга 16.2, вынеся деление в отдельный метод f(). Получим листинг 16.3.

Листинг 16.3. Выбрасывание исключения из метода

```
class SimpleExt2{
  private static void f(int n){
    System.out.println(" 10 / n = " + (10 / n));
  }
  public static void main(String[] args){
    try{
       int n = Integer.parseInt(args[0]);
       System.out.println("After parseInt());
       f(n);
       System.out.println("After results output");
    }catch(ArithmeticException ae){
       System.out.println("From Arithm.Exc. catch: "+ae);
    }catch(ArrayIndexOutOfBoundsException arre){
       System.out.println("From Array.Exc. catch: "+arre);
    }finally{
```

```
        System.out.println("From finally");
      }
      System.out.println("After all actions");
    }
}
```

Откомпилировав и запустив программу листинга 16.3, убедимся, что вывод программы не изменился, он такой же, как на рис. 16.2. Исключение, возникшее при делении на нуль в методе `f()`, "пролетело" через этот метод, "вылетело" в метод `main()`, там перехвачено и обработано.

Часть заголовка метода *throws*

То обстоятельство, что метод не обрабатывает возникающее в нем исключение, а выбрасывает (throws) его, следует отмечать в заголовке метода служебным словом `throws` и указанием класса исключения:

```
private static void f(int n) throws ArithmeticException{
  System.out.println(" 10 / n = " + (10 / n));
}
```

Почему же мы не сделали это в листинге 16.3? Дело в том, что спецификация JLS делит все исключения на *проверяемые* (checked), те, которые проверяет компилятор, и *непроверяемые* (unchecked). При проверке компилятор замечает необработанные в методах и конструкторах исключения и считает ошибкой отсутствие в заголовке таких методов и конструкторов пометки `throws`. Именно для предотвращения подобных ошибок мы в предыдущих главах вставляли в листинги блоки обработки исключений.

Так вот, исключения класса `RuntimeException` и его подклассов, одним из которых является `ArithmeticException`, непроверяемые, для них пометка `throws` необязательна. Еще одно большое семейство непроверяемых исключений составляет класс `Error` и его расширения.

Почему компилятор не проверяет эти типы исключений? Причина в том, что исключения класса `RuntimeException` свидетельствуют об ошибках в программе, и единственно разумный метод их обработки — исправить исходный текст программы и перекомпилировать ее. Что касается класса `Error`, то эти исключения очень трудно локализовать и на стадии компиляции невозможно определить место их появления.

Напротив, возникновение проверяемого исключения показывает, что программа недостаточно продумана, не все возможные ситуации описаны. Такая программа должна быть доработана, о чем и напоминает компилятор.

Если метод или конструктор выбрасывает несколько исключений, то их надо перечислить через запятую после слова `throws`. Заголовок метода `main()`

листинга 16.1, если бы исключения, которые он выбрасывает, не были бы объектами подклассов класса RuntimeException, следовало бы написать так:

```
public static void main(String[] args)
         throws ArithmeticException, ArrayIndexOutOfBoundsException{
  // Содержимое метода
}
```

Перенесем теперь обработку деления на нуль в метод f() и добавим трассировочную печать, как это сделано в листинге 16.4. Результат — на рис. 16.3.

Листинг 16.4. Обработка исключения в методе

```
class SimpleExt3{
  private static void f(int n){ // throws ArithmeticException{
    try{
       System.out.println(" 10 / n = " + (10 / n));
       System.out.println("From f() after results output");
    }catch(ArithmeticException ae){
       System.out.println("From f() catch: " + ae);
//     throw ae;
    }finally{
       System.out.println("From f() finally");
    }
  }
  public static void main(String[] args){
    try{
       int n = Integer.parseInt(args[0]);
       System.out.println("After parseInt());
       f(n);
       System.out.println("After results output");
    }catch(ArithmeticException ae){
       System.out.println("From Arithm.Exc. catch: "+ae);
    }catch(ArrayIndexOutOfBoundsException arre){
       System.out.println("From Array.Exc. catch: "+arre);
    }finally{
       System.out.println("From finally");
    }
    System.out.println("After all actions");
  }
}
```

Внимательно проследите за передачей управления и заметьте, что исключение класса ArithmeticException уже не выбрасывается в метод main().

Оператор try{}catch(){} в методе f() можно рассматривать как вложенный в оператор обработки исключений в методе main().

При необходимости исключение можно выбросить оператором `throw ae`. В листинге 16.4 этот оператор показан как комментарий. Уберите символы комментария `//`, перекомпилируйте программу и посмотрите, как изменится ее вывод.

```
Command Prompt
D:\jdk1.3\MyProgs\except>javac SimpleExt3.java

D:\jdk1.3\MyProgs\except>java SimpleExt3 5
After parseInt()
 10 / n = 2
From f() after results output
From f() finally
After results output
From finally
After all actions

D:\jdk1.3\MyProgs\except>java SimpleExt3 0
After parseInt()
From f() catch: java.lang.ArithmeticException: / by zero
From f() finally
After results output
From finally
After all actions

D:\jdk1.3\MyProgs\except>java SimpleExt3
From Array.Exc. catch: java.lang.ArrayIndexOutOfBoundsException
From finally
After all actions

D:\jdk1.3\MyProgs\except>_
```

Рис. 16.3. Обработка исключения в методе

Оператор *throw*

Этот оператор очень прост: после слова `throw` через пробел записывается объект класса-исключения. Достаточно часто он создается прямо в операторе `throw`, например:

```
throw new ArithmeticException();
```

Оператор можно записать в любом месте программы. Он немедленно выбрасывает записанный в нем объект-исключение и дальше обработка этого исключения идет как обычно, будто бы здесь произошло деление на нуль или другое действие, вызвавшее исключение класса `ArithmeticException`.

Итак, каждый блок `catch(){}` перехватывает один определенный тип исключений. Если требуется одинаково обработать несколько типов исключений, то можно воспользоваться тем, что классы-исключения образуют иерархию. Изменим еще раз листинг 16.2, получив листинг 16.5.

Листинг 16.5. Обработка нескольких типов исключений

```
class SimpleExt4{
  public static void main(String[] args){
```

```
    try{
       int n = Integer.parseInt(args[0]);
       System.out.println("After parseInt());
       System.out.println(" 10 / n = " + (10 / n));
       System.out.println("After results output");
    }catch(RuntimeException ae){
       System.out.println("From Run.Exc. catch: "+ae);
    }finally{
       System.out.println("From finally");
    }
    System.out.println("After all actions");
  }
}
```

В листинге 16.5 два блока `catch(){}` заменены одним блоком, перехватывающим исключение класса `RuntimeException`. Как видно на рис. 16.4, этот блок перехватывает оба исключения. Почему? Потому что это исключения подклассов класса `RuntimeException`.

```
Command Prompt
D:\jdk1.3\MyProgs\except>javac SimpleExt4.java

D:\jdk1.3\MyProgs\except>java SimpleExt4 5
After parseInt()
 10 / n = 2
After results output
From finally
After all actions

D:\jdk1.3\MyProgs\except>java SimpleExt4 0
After parseInt()
From Run.Exc. catch: java.lang.ArithmeticException: / by zero
From finally
After all actions

D:\jdk1.3\MyProgs\except>java SimpleExt4
From Run.Exc. catch: java.lang.ArrayIndexOutOfBoundsException
From finally
After all actions

D:\jdk1.3\MyProgs\except>
```

Рис. 16.4. Перехват нескольких типов исключений

Таким образом, перемещаясь по иерархии классов-исключений, мы можем обрабатывать сразу более или менее крупные совокупности исключений. Рассмотрим подробнее иерархию классов-исключений.

Иерархия классов-исключений

Все классы-исключения расширяют класс `Throwable` — непосредственное расширение класса `Object`.

У класса `Throwable` и у всех его расширений по традиции два конструктора:

- `Throwable()` — конструктор по умолчанию;
- `Throwable(String message)` — создаваемый объект будет содержать произвольное сообщение `message`.

Записанное в конструкторе сообщение можно получить затем методом `getMessage()`. Если объект создавался конструктором по умолчанию, то данный метод возвратит `null`.

Метод `toString()` возвращает краткое описание события, именно он работал в предыдущих листингах.

Три метода выводят сообщения обо всех методах, встретившихся по пути "полета" исключения:

- `printStackTrace()` — выводит сообщения в стандартный вывод, как правило, это консоль;
- `printStackTrace(PrintStream stream)` — выводит сообщения в байтовый поток `stream`;
- `printStackTrace(PrintWriter stream)` — выводит сообщения в символьный поток `stream`.

У класса `Throwable` два непосредственных наследника — классы `Error` и `Exception`. Они не добавляют новых методов, а служат для разделения классов-исключений на два больших семейства — семейство классов-ошибок (error) и семейство собственно классов-исключений (exception).

Классы-ошибки, расширяющие класс `Error`, свидетельствуют о возникновении сложных ситуаций в виртуальной машине Java. Их обработка требует глубокого понимания всех тонкостей работы JVM. Ее не рекомендуется выполнять в обычной программе. Не советуют даже выбрасывать ошибки оператором `throw`. Не следует делать свои классы-исключения расширениями класса `Error` или какого-то его подкласса.

Имена классов-ошибок, по соглашению, заканчиваются словом `Error`.

Классы-исключения, расширяющие класс `Exception`, отмечают возникновение обычной нештатной ситуации, которую можно и даже нужно обработать. Такие исключения следует выбросить оператором `throw`. Классов-исключений очень много, более двухсот. Они разбросаны буквально по всем пакетам J2SDK. В большинстве случаев вы способны подобрать готовый класс-исключение для обработки исключительных ситуаций в своей программе. При желании можно создать и свой класс-исключение, расширив класс `Exception` или любой его подкласс.

Среди классов-исключений выделяется класс `RuntimeException` — прямое расширение класса `Exception`. В нем и его подклассах отмечаются исключения, возникшие при работе JVM, но не столь серьезные, как ошибки. Их можно обрабатывать и выбрасывать, расширять своими классами, но лучше

доверить это JVM, поскольку чаще всего это просто ошибка в программе, которую надо исправить. Особенность исключений данного класса в том, что их не надо отмечать в заголовке метода пометкой `throws`.

Имена классов-исключений, по соглашению, заканчиваются словом `Exception`.

Порядок обработки исключений

Блоки `catch(){}` перехватывают исключения в порядке написания этих блоков. Это правило приводит к интересным результатам.

В листинге 16.2 мы записали два блока перехвата `catch(){}` и оба блока выполнялись при возникновении соответствующего исключения. Это происходило потому, что классы-исключения `ArithmeticException` и `ArrayIndexOutOfBoundsException` находятся на разных ветвях иерархии исключений. Иначе обстоит дело, если блоки `catch(){}` перехватывают исключения, расположенные на одной ветви. Если в листинге 16.4 после блока, перехватывающего `RuntimeException`, поместить блок, обрабатывающий выход индекса за пределы:

```
try{
    // Операторы, вызывающие исключения
}catch(RuntimeException re){
    // Какая-то обработка
}catch(ArrayIndexOutOfBoundsException ae){
    // Никогда не будет выполнен!
}
```

то он не будет выполняться, поскольку исключение этого типа является, к тому же, исключением общего типа `RuntimeException` и будет перехватываться предыдущим блоком `catch(){}`.

Создание собственных исключений

Прежде всего, нужно четко определить ситуации, в которых будет возникать ваше собственное исключение, и подумать, не станет ли его перехват невольно перехватывать также и другие, не учтенные вами исключения.

Потом надо выбрать суперкласс создаваемого класса-исключения. Им может быть класс `Exception` или один из его многочисленных подклассов.

После этого можно написать класс-исключение. Его имя, по соглашению, должно завершаться словом `Exception`. Как правило, этот класс состоит только из двух конструкторов и переопределения методов `toString()` и `getMessage()`.

Рассмотрим простой пример. Пусть метод `handle(int cipher)` обрабатывает арабские цифры 0—9, которые передаются ему в аргументе `cipher` типа `int`.

Мы хотим выбросить исключение, если аргумент `cipher` выходит за диапазон 0—9.

Прежде всего, убедимся, что такого исключения нет в иерархии классов `Exception`. Ко всему прочему, не отслеживается и более общая ситуация попадания целого числа в какой-то диапазон. Поэтому будем расширять наш класс. Назовем его `CipherException`, прямо от класса `Exception`. Определим класс `CipherException`, как показано в листинге 16.6, и используем его в классе `ExceptDemo`. На рис. 16.5 продемонстрирован вывод этой программы.

Листинг 16.6. Создание класса-исключения

```
class CipherException extends Exception{
   private String msg;
   CipherException(){ msg = null;}
   CipherException(String s){ msg = s;}
   public String toString(){
      return "CipherException (" + msg + ");
   }
}
class ExceptDemo{
   static public void handle(int cipher) throws CipherException{
      System.out.println("handle()'s beginning");
      if (cipher < 0 || cipher > 9)
         throw new CipherException("" + cipher);
      System.out.println("handle()'s ending");
   }
   public static void main(String[] args){
      try{
         handle(1);
         handle(10);
      }catch(CipherException ce){
         System.out.println("caught " + ce);
         ce.printStackTrace();
      }
  }
}
```

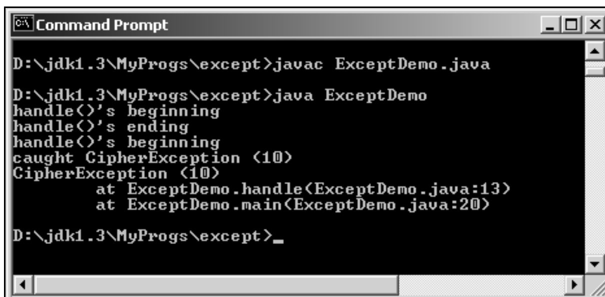


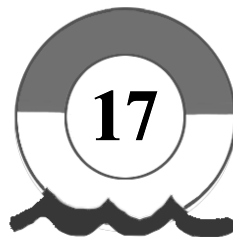
Рис. 16.5. Обработка собственного исключения

Заключение

Обработка исключительных ситуаций стала сейчас обязательной частью объектно-ориентированных программ. Применяя методы классов J2SDK и других пакетов, обращайте внимание на то, какие исключения они выбрасывают, и обрабатывайте их. Исключения резко меняют ход выполнения программы, делают его запутанным. Не увлекайтесь сложной обработкой, помните о принципе KISS.

Например, из блока `finally{}` можно выбросить исключение и обработать его в другом месте. Подумайте, что произойдет в этом случае с исключением, возникшем в блоке `try{}`? Оно нигде не будет перехвачено и обработано.

ГЛАВА 17



Подпроцессы

Основное понятие современных операционных систем — *процесс* (process). Как и все общие понятия, процесс трудно определить, да это и не входит в задачу книги. Можно понимать под процессом выполняющуюся (runnable) программу, но надо помнить о том, что у процесса есть несколько состояний. Процесс может в любой момент перейти к выполнению машинного кода другой программы, а также "заснуть" (sleep) на некоторое время, приостановив выполнение программы. Он может быть выгружен на диск. Количество состояний процесса и их особенности зависят от операционной системы.

Все современные операционные системы *многозадачные* (multitasking), они запускают и выполняют сразу несколько процессов. Одновременно может работать браузер, текстовый редактор, музыкальный проигрыватель. На экране дисплея открываются несколько окон, каждое из которых связано со своим работающим процессом.

Если на компьютере только один процессор, то он переключается с одного процесса на другой, создавая видимость одновременной работы. Переключение происходит по истечении одного или нескольких "тиков" (ticks). Размер тика зависит от тактовой частоты процессора и обычно имеет порядок 0,01 секунды. Процессам назначаются разные *приоритеты* (priority). Процессы с низким приоритетом не могут прервать выполнение процесса с более высоким приоритетом, они меньше занимают процессор и поэтому выполняются медленно, как говорят, "на фоне". Самый высокий приоритет у системных процессов, например, у *диспетчера* (scheduler), который как раз и занимается переключением процессора с процесса на процесс. Такие процессы нельзя прерывать, пока они не закончат работу, иначе компьютер быстро придет в хаотическое состояние.

Каждому процессу выделяется определенная область оперативной памяти для размещения кода программы и ее данных — его *адресное пространство*.

В эту же область записывается часть сведений о процессе, составляющая его *контекст* (context). Очень важно разделить адресные пространства разных процессов, чтобы они не могли изменить код и данные друг друга.

Операционные системы по-разному относятся к обеспечению защиты адресных пространств процессов. MS Windows NT/2000 тщательно разделяют адресные пространства, тратя на это много ресурсов и времени. Это повышает надежность выполнения программы, но затрудняет создание процесса. Такие операционные системы плохо справляются с управлением большого числа процессов.

Операционные системы семейства UNIX меньше заботятся о защите памяти, но легче создают процессы и способны управлять сотней одновременно работающих процессов.

Кроме управления работой процессов операционная система должна обеспечить средства их взаимодействия: обмен сигналами и сообщениями, создание разделяемых несколькими процессами областей памяти и разделяемого исполнимого кода программы. Эти средства тоже требуют ресурсов и замедляют работу компьютера.

Работу многозадачной системы можно упростить и ускорить, если разрешить взаимодействующим процессам работать в одном адресном пространстве. Такие процессы называются threads. В русской литературе предлагаются различные переводы этого слова. Буквальный перевод — "нить", но мы не занимаемся прядильным производством. Часто переводят thread как "поток", но в этой книге мы говорим о потоке ввода/вывода. Иногда просто говорят "тред", но в русском языке уже есть "тред-юнион". Встречается перевод "легковесный процесс", но в некоторых операционных системах, например, Solaris, есть и thread и lightweight process. Остановимся на слове "подпроцесс".

Подпроцессы создают новые трудности для операционной системы — надо очень внимательно следить за тем, чтобы они не мешали друг другу при записи в общие участки памяти, — но зато облегчают взаимодействие подпроцессов.

Создание подпроцессов и управление ими — это дело операционной системы, но в язык Java введены средства для выполнения этих действий. Поскольку программы, написанные на Java, должны работать во всех операционных системах, эти средства позволяют выполнять только самые общие действия.

Когда операционная система запускает виртуальную машину Java для выполнения приложения, она создает один процесс с несколькими подпроцессами. *Главный* (main) подпроцесс выполняет байт-коды программы, а именно, он сразу же обращается к методу `main()` приложения. Этот подпроцесс может породить новые подпроцессы, которые, в свою очередь, способны породить подпроцессы и т. д. Главным подпроцессом апплета является один

из подпроцессов браузера, в котором апплет выполняется. Главный подпроцесс не играет никакой особой роли, просто он создается первым.

Подпроцесс в Java создается и управляется методами класса `Thread`. После создания объекта этого класса одним из его конструкторов новый подпроцесс запускается методом `start()`.

Получить ссылку на текущий подпроцесс можно статическим методом

```
Thread.currentThread();
```

Класс `Thread` реализует интерфейс `Runnable`. Этот интерфейс описывает только один метод `run()`. Новый подпроцесс будет выполнять то, что записано в этом методе. Впрочем, класс `Thread` содержит только пустую реализацию метода `run()`, поэтому класс `Thread` не используется сам по себе, он всегда расширяется. При его расширении метод `run()` переопределяется.

Метод `run()` не содержит аргументов, т. к. некому передавать их значения в метод. Он не возвращает значения, его некуда передавать. К методу `run()` нельзя обратиться из программы, это всегда делается автоматически исполняющей системой Java при запуске нового подпроцесса методом `start()`.

Итак, задать действия создаваемого подпроцесса можно двумя способами: расширить класс `Thread` или реализовать интерфейс `Runnable`. Первый способ позволяет использовать методы класса `Thread` для управления подпроцессом. Второй способ применяется в тех случаях, когда надо только реализовать метод `run()`, или класс, создающий подпроцесс, уже расширяет какой-то другой класс.

Посмотрим, какие конструкторы и методы содержит класс `Thread`.

Класс *Thread*

В классе `Thread` семь конструкторов:

- `Thread(ThreadGroup group, Runnable target, String name)` — создает подпроцесс с именем `name`, принадлежащий группе `group` и выполняющий метод `run()` объекта `target`. Это основной конструктор, все остальные обращаются к нему с тем или иным параметром, равным `null`;
- `Thread()` — создаваемый подпроцесс будет выполнять свой метод `run()`;
- `Thread(Runnable target)`;
- `Thread(Runnable target, String name)`;
- `Thread(String name)`;
- `Thread(ThreadGroup group, Runnable target)`;
- `Thread(ThreadGroup group, String name)`.

Имя подпроцесса `name` не имеет никакого значения, оно не используется виртуальной машиной Java и применяется только для различения подпроцессов в программе.

После создания подпроцесса его надо запустить методом `start()`. Виртуальная машина Java начнет выполнять метод `run()` этого объекта-подпроцесса.

Подпроцесс завершит работу после выполнения метода `run()`. Для уничтожения объекта-подпроцесса вслед за этим он должен присвоить значение `null`.

Выполняющийся подпроцесс можно приостановить статическим методом `sleep(long ms)` на `ms` миллисекунд. Этот метод мы уже использовали в предыдущих главах. Если вычислительная система может отсчитывать наносекунды, то можно приостановить подпроцесс с точностью до наносекунд методом `sleep(long ms, int nanosec)`.

В листинге 17.1 приведен простейший пример. Главный подпроцесс создает два подпроцесса с именами `Thread 1` и `Thread 2`, выполняющих один и тот же метод `run()`. Этот метод просто выводит 20 раз текст на экран, а затем сообщает о своем завершении.

Листинг 17.1. Два подпроцесса, запущенных из главного подпроцесса

```
class OutThread extends Thread{
   private String msg;
   OutThread(String s, String name){
      super(name);  msg = s;
   }
   public void run(){
      for(int i = 0; i < 20; i++){
//          try{
//             Thread.sleep(100);
//          }catch(InterruptedException ie){}
          System.out.print(msg + " ");
      }
      System.out.println("End of " + getName());
   }
}
class TwoThreads{
  public static void main(String[] args){
     new OutThread("HIP", "Thread 1").start();
     new OutThread("hop", "Thread 2").start();
     System.out.println();
  }
}
```

На рис. 17.1 показан результат двух запусков программы листинга 17.1. Как видите, в первом случае подпроцесс `Thread 1` успел отработать полностью до переключения процессора на выполнение второго подпроцесса. Во втором случае работа подпроцесса `Thread 1` была прервана, процессор переключился на выполнение подпроцесса `Thread 2`, успел выполнить его полностью, а затем переключился обратно на выполнение подпроцесса `Thread 1` и завершил его.

```
Command Prompt
D:\jdk1.3\MyProgs\thread>java TwoThreads
HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP
End of Thread 1

hop hop hop hop hop hop hop hop hop hop hop hop hop hop hop hop hop hop hop hop
End of Thread 2

D:\jdk1.3\MyProgs\thread>java TwoThreads

HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP HIP hop hop hop hop
hop hop hop hop hop hop hop hop hop hop hop hop hop hop hop hop End of Thread 2
HIP HIP HIP HIP End of Thread 1

D:\jdk1.3\MyProgs\thread>_
```

Рис. 17.1. Два подпроцесса работают без задержки

Уберем в листинге 17.1 комментарии, задержав тем самым выполнение каждой итерации цикла на 0,1 секунды. Пустая обработка исключения `InterruptedException` означает, что мы игнорируем попытку прерывания работы подпроцесса. На рис. 17.2 показан результат двух запусков программы. Как видите, процессор переключается с одного подпроцесса на другой, но в одном месте регулярность переключения нарушается и ранее запущенный подпроцесс завершается позднее.

```
Command Prompt
D:\jdk1.3\MyProgs\thread>java TwoThreads

HIP hop HIP hop HIP hop HIP hop HIP hop HIP hop HIP hop HIP hop hop HIP hop HIP
hop HIP hop HIP hop HIP hop HIP hop HIP hop HIP HIP hop hop HIP hop HIP hop End
of Thread 2
HIP End of Thread 1

D:\jdk1.3\MyProgs\thread>java TwoThreads

HIP hop HIP hop HIP hop HIP hop HIP hop HIP hop HIP hop HIP hop hop HIP hop HIP
hop HIP hop HIP hop HIP hop HIP hop HIP hop HIP HIP hop hop HIP hop HIP hop End
of Thread 2
HIP End of Thread 1

D:\jdk1.3\MyProgs\thread>
```

Рис. 17.2. Подпроцессы работают с задержкой

Как же добиться согласованности, как говорят, *синхронизации* (synchronization) подпроцессов? Обсудим это ниже, а пока покажем еще два варианта создания той же самой программы.

В листинге 17.2 приведен второй вариант той же программы: сам класс `TwoThreads2` является расширением класса `Thread`, а метод `run()` реализуется прямо в нем.

Листинг 17.2. Класс расширяет `Thread`

```
class TwoThreads2 extends Thread{
   private String msg;
   TwoThreads2(String s, String name){
      super(name);  msg = s;
   }
   public void run(){
      for(int i = 0; i < 20; i++){
         try{
            Thread.sleep(100);
         }catch(InterruptedException ie){}
         System.out.print(msg + " ");
      }
      System.out.println("End of " + getName());
   }
  public static void main(String[] args){
     new TwoThreads2("HIP", "Thread 1").start();
     new TwoThreads2("hop", "Thread 2").start();
     System.out.println();
  }
}
```

Третий вариант: класс `TwoThreads3` реализует интерфейс `Runnable`. Этот вариант записан в листинге 17.3. Здесь нельзя использовать методы класса `Thread`, но зато класс `TwoThreads3` может быть расширением другого класса. Например, можно сделать его апплетом, расширив класс `Applet` или `JApplet`.

Листинг 17.3. Реализация интерфейса `Runnable`

```
class TwoThreads3 implements Runnable{
   private String msg;
   TwoThreads3(String s){ msg = s; }
   public void run(){
      for(int i = 0; i < 20; i++){
         try{
            Thread.sleep(100);
         }catch(InterruptedException ie){}
         System.out.print(msg + " ");
      }
```

```
      System.out.println("End of thread.");
   }
  public static void main(String[] args){
     new Thread(new TwoThreads3("HIP"), "Thread 1").start();
     new Thread(new TwoThreads3("hop"), "Thread 2").start();
     System.out.println();
  }
}
```

Чаще всего в новом подпроцессе задаются бесконечные действия, выполняющиеся на фоне основных действий: проигрывается музыка, на экране вращается анимированный логотип фирмы, бежит рекламная строка. Для реализации такого подпроцесса в методе `run()` задается бесконечный цикл, останавливаемый после того, как объект-подпроцесс получит значение `null`.

В листинге 17.4 показан четвертый вариант той же самой программы, в которой метод `run()` выполняется до тех пор, пока текущий объект-подпроцесс `th` совпадает с объектом `go`, запустившим текущий подпроцесс. Для прекращения его выполнения предусмотрен метод `stop()`, к которому обращается главный подпроцесс. Это стандартная конструкция, рекомендуемая документацией J2SDK. Главный подпроцесс в данном примере только создает объекты-подпроцессы, ждет одну секунду и останавливает их.

Листинг 17.4. Прекращение работы подпроцессов

```
class TwoThreads5 implements Runnable{
   private String msg;
   private Thread go;
   TwoThreads5(String s){
      msg = s;
      go = new Thread(this);
      go.start();
   }
   public void run(){
      Thread th = Thread.currentThread();
      while(go == th){
         try{
            Thread.sleep(100);
         }catch(InterruptedException ie){}
         System.out.print(msg + " ");
      }
      System.out.println("End of thread.");
   }
   public void stop(){ go = null; }
```

```
   public static void main(String[] args){
     TwoThreads5 th1 = new TwoThreads5("HIP");
     TwoThreads5 th2 = new TwoThreads5("hop");
     try{
        Thread.sleep(1000);
     }catch(InterruptedException ie){}
     th1.stop(); th2.stop();
    System.out.println();
  }
}
```

Синхронизация подпроцессов

Основная сложность при написании программ, в которых работают несколько подпроцессов — это согласовать совместную работу подпроцессов с общими ячейками памяти.

Классический пример — банковская транзакция, в которой изменяется остаток на счету клиента с номером `numDep`. Предположим, что для ее выполнения запрограммированы такие действия:

```
Deposit myDep = getDeposit(numDep);    // Получаем счет с номером numDep
int rest = myDep.getRest();            // Получаем остаток на счету myDep
Deposit newDep = myDep.operate(rest, sum);     // Изменяем остаток
                                               // на величину sum
myDep.setDeposit(newDep); // Заносим новый остаток на счет myDep
```

Пусть на счету лежит 1000 рублей. Мы решили снять со счета 500 рублей, а в это же время поступил почтовый перевод на 1500 рублей. Эти действия выполняют разные подпроцессы, но изменяют они один и тот же счет `myDep` с номером `numDep`. Посмотрев еще раз на рис. 17.1 и 17.2, вы поверите, что последовательность действий может сложиться так. Первый подпроцесс проделает вычитание 1000−500, в это время второй подпроцесс выполнит все три действия и запишет на счет 1000+1500 = 2500 рублей, после чего первый подпроцесс выполнит свое последнее действие и у нас на счету окажется 500 рублей. Вряд ли вам понравится такое выполнение двух транзакций.

В языке Java принят выход из этого положения, называемый в теории операционных систем *монитором* (monitor). Он заключается в том, что подпроцесс блокирует объект, с которым работает, чтобы другие подпроцессы не могли обратиться к данному объекту, пока блокировка не будет снята. В нашем примере первый подпроцесс должен вначале заблокировать счет `myDep`, затем полностью выполнить всю транзакцию и снять блокировку. Второй подпроцесс приостановится и станет ждать, пока блокировка не будет снята, после чего начнет работать с объектом `myDep`.

Все это делается одним оператором `synchronized(){}`, как показано ниже:

```
Deposit myDep = getDeposit(numDep);
synchronized(myDep){
   int rest = myDep.getRest();
   Deposit newDep = myDep.operate(rest, sum);
   myDep.setDeposit(newDep);
}
```

В заголовке оператора `synchronized` в скобках указывается ссылка на объект, который будет заблокирован перед выполнением блока. Объект будет недоступен для других подпроцессов, пока выполняется блок. После выполнения блока блокировка снимается.

Если при написании какого-нибудь метода оказалось, что в блок `synchronized` входят все операторы этого метода, то можно просто пометить метод словом `synchronized`, сделав его *синхронизированным* (synchronized):

```
synchronized int getRest(){
    // Тело метода
}
synchronized Deposit operate(int rest, int sum){
   // Тело метода
}
synchronized void setDeposit(Deposit dep){
   // Тело метода
}
```

В этом случае блокируется объект, выполняющий метод, т. е. `this`. Если все методы, к которым не должны одновременно обращаться несколько подпроцессов, помечены `synchronized`, то оператор `synchronized(){}` уже не нужен. Теперь, если один подпроцесс выполняет синхронизированный метод объекта, то другие подпроцессы уже не могут обратиться ни к одному синхронизированному методу того же самого объекта.

Приведем простейший пример. Метод `run()` в листинге 17.5 выводит строку "Hello, World!" с задержкой в 1 секунду между словами. Этот метод выполняется двумя подпроцессами, работающими с одним объектом `th`. Программа выполняется два раза. Первый раз метод `run()` не синхронизирован, второй раз синхронизирован, его заголовок показан в листинге 17.4 как комментарий. Результат выполнения программы представлен на рис. 17.3.

Листинг 17.5. Синхронизация метода

```
class TwoThreads4 implements Runnable{
    public void run(){
```

```
//    synchronized public void run(){
       System.out.print("Hello, ");
       try{
            Thread.sleep(1000);
       }catch(InterruptedException ie){}
       System.out.println("World!");
  }
  public static void main(String[] args){
     TwoThreads4 th = new TwoThreads4();
     new Thread(th).start();
     new Thread(th).start();
  }
}
```

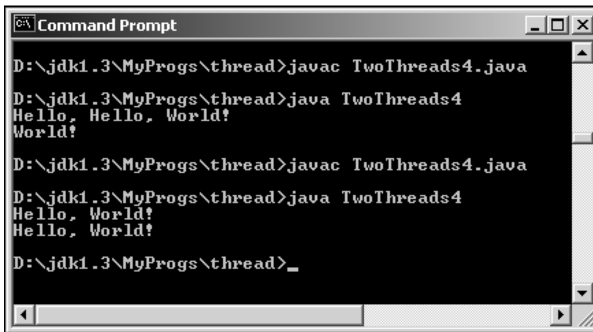


Рис. 17.3. Синхронизация метода

Действия, входящие в синхронизированный блок или метод образуют *критический участок* (critical section) программы. Несколько подпроцессов, собирающихся выполнять критический участок, встают в очередь. Это замедляет работу программы, поэтому для быстроты ее выполнения критических участков должно быть как можно меньше, и они должны быть как можно короче.

Многие методы Java 2 SDK синхронизированы. Обратите внимание, что на рис. 17.1 слова выводятся вперемешку, но каждое слово выводится полностью. Это происходит потому, что метод `print()` класса `PrintStream` синхронизирован, при его выполнении выходной поток `System.out` блокируется до тех пор, пока метод `print()` не закончит свою работу.

Итак, мы можем легко организовать последовательный доступ нескольких подпроцессов к полям одного объекта с помощью оператора `synchronized(){}`. Синхронизация обеспечивает *взаимно исключающее* (mutually exclusive) выполнение подпроцессов. Но что делать, если нужен совместный доступ нескольких подпроцессов к общим объектам? Для этого в Java существует механизм ожидания и уведомления (wait-notify).

Согласование работы нескольких подпроцессов

Возможность создания многопоточных программ заложена в язык Java с самого его создания. В каждом объекте есть три метода `wait()` и один метод `notify()`, позволяющие приостановить работу подпроцесса с этим объектом, позволить другому подпроцессу поработать с объектом, а затем уведомить (notify) первый подпроцесс о возможности продолжения работы. Эти методы определены прямо в классе `Object` и наследуются всеми классами.

С каждым объектом связано множество подпроцессов, ожидающих доступа к объекту (wait set). Вначале этот "зал ожидания" пуст.

Основной метод `wait(long millisec)` приостанавливает текущий подпроцесс `this`, работающий с объектом, на `millisec` миллисекунд и переводит его в "зал ожидания", в множество ожидающих подпроцессов. Обращение к этому методу допускается только в синхронизированном блоке или методе, чтобы быть уверенными в том, что с объектом работает только один подпроцесс. По истечении `millisec` или после того, как объект получит уведомление методом `notify()`, подпроцесс готов возобновить работу. Если аргумент `millisec` равен `0`, то время ожидания не определено и возобновление работы подпроцесса возможно только после того, как объект получит уведомление методом `notify()`.

Отличие данного метода от метода `sleep()` в том, что метод `wait()` снимает блокировку с объекта. С объектом может работать один из подпроцессов из "зала ожидания", обычно тот, который ждал дольше всех, хотя это не гарантируется спецификацией JLS.

Второй метод `wait()` эквивалентен `wait(0)`. Третий метод `wait(long millisec, int nanosec)` уточняет задержку на `nanosec` наносекунд, если их сумеет отсчитать операционная система.

Метод `notify()` выводит из "зала ожидания" только один, произвольно выбранный подпроцесс. Метод `notifyAll()` выводит из состояния ожидания все подпроцессы. Эти методы тоже должны выполняться в синхронизированном блоке или методе.

Как же применить все это для согласованного доступа к объекту? Как всегда, лучше всего объяснить это на примере.

Обратимся снова к схеме "поставщик-потребитель", уже использованной в *главе 15*. Один подпроцесс, поставщик, производит вычисления, другой, потребитель, ожидает результаты этих вычислений и использует их по мере поступления. Подпроцессы передают информацию через общий экземпляр `st` класса `Store`.

Работа этих подпроцессов должна быть согласована. Потребитель обязан ждать, пока поставщик не занесет результат вычислений в объект `st`, а поставщик должен ждать, пока потребитель не возьмет этот результат.

Для простоты поставщик просто заносит в общий объект класса `Store` целые числа, а потребитель лишь забирает их.

В листинге 17.6 класс `Store` не обеспечивает согласования получения и выдачи информации. Результат работы показан на рис. 17.4.

Листинг 17.6. Несогласованные подпроцессы

```
class Store{
   private int inform;
   synchronized public int getInform(){ return inform; }
   synchronized public void setInform(int n){ inform = n; }
}
class Producer implements Runnable{
   private Store st;
   private Thread go;
   Producer(Store st){
      this.st = st;
      go = new Thread(this);
      go.start();
   }
   public void run(){
      int n = 0;
      Thread th = Thread.currentThread();
      while(go == th){
         st.setInform(n);
         System.out.print("Put: " + n + "  ");
         n++;
      }
   }
   public void stop(){ go = null; }
}
class Consumer implements Runnable{
   private Store st;
   private Thread go;
   Consumer(Store st){
      this.st = st;
      go = new Thread(this);
      go.start();
   }
   public void run(){
      Thread th = Thread.currentThread();
```

```
      while(go == th)  System.out.println("Got: " + st.getInform());
   }
   public void stop(){ go = null; }
}
class ProdCons{
  public static void main(String[] args){
     Store st = new Store();
     Producer p = new Producer(st);
     Consumer c = new Consumer(st);
     try{
        Thread.sleep(30);
     }catch(InterruptedException ie){}
     p.stop(); c.stop();
  }
}
```

```
Command Prompt
D:\jdk1.3\MyProgs\thread>java ProdCons
Put: 0  Put: 1  Put: 2  Put: 3  Put: 4  Put: 5  Put: 6  Put: 7  Put: 8  Put: 9
Put: 10  Put: 11  Put: 12  Put: 13  Put: 14  Put: 15  Put: 16  Put: 17  Put: 18
 Put: 19  Put: 20  Put: 21  Put: 22  Put: 23  Put: 24  Put: 25  Put: 26  Put: 27
  Put: 28  Put: 29  Put: 30  Put: 31  Put: 32  Put: 33  Put: 34  Put: 35  Put: 3
6  Put: 37  Put: 38  Put: 39  Put: 40  Put: 41  Put: 42  Put: 43  Put: 44  Put:
45  Put: 46  Put: 47  Put: 48  Got: 48
Got: 48
Got: 48
Got: 48
Got: 48
Got: 48
Got: 48
Got: 48

D:\jdk1.3\MyProgs\thread>_
```

Рис. 17.4. Несогласованная работа двух подпроцессов

В листинге 17.7 в класс `Store` внесено логическое поле `ready`, отмечающее процесс получения и выдачи информации. Когда новая порция информации получена от поставщика `Producer`, в поле `ready` заносится значение `true`, получатель `Consumer` может забирать эту порцию информации. После выдачи информации переменная `ready` становится равной `false`.

Но этого мало. То, что получатель может забрать продукт, не означает, что он действительно заберет его. Поэтому в конце метода `setInform()` получатель уведомляется о поступлении продукта методом `notify()`. Пока поле `ready` не примет нужное значение, подпроцесс переводится в "зал ожидания" методом `wait()`. Результат работы программы с обновленным классом `Store` показан на рис. 17.5.

Листинг 17.7. Согласование получения и выдачи информации

```
class Store{
   private int inform = -1;
```

```
   private boolean ready;
   synchronized public int getInform(){
      try{
         if (!ready) wait();
         ready = false;
         return inform;
      }catch(InterruptedException ie){
      }finally{
         notify();
      }
      return -1;
   }
   synchronized public void setInform(int n){
      if (ready)
         try{
            wait();
         }catch(InterruptedException ie){}
      inform = n;
      ready = true;
      notify();
  }
}
```

Поскольку уведомление поставщика в методе `getInform()` должно происходить уже после отправки информации оператором `return inform`, оно включено в блок `finally{}`.

```
Command Prompt
D:\jdk1.3\MyProgs\thread>javac ProdCons1.java

D:\jdk1.3\MyProgs\thread>java ProdCons1
Put: 0   Put: 1   Got: 0
Put: 2   Got: 1
Put: 3   Got: 2
Put: 4   Got: 3
Put: 5   Got: 4
Put: 6   Got: 5
Put: 7   Got: 6

D:\jdk1.3\MyProgs\thread>_
```

Рис. 17.5. Согласованная работа подпроцессов

Обратите внимание: сообщение "Got: 0" отстает на один шаг от действительного получения информации.

Приоритеты подпроцессов

Планировщик подпроцессов виртуальной машины Java назначает каждому подпроцессу одинаковое время выполнения процессором, переключаясь с подпроцесса на подпроцесс по истечении этого времени. Иногда необходи-

мо выделить какому-то подпроцессу больше или меньше времени по сравнению с другим подпроцессом. В таком случае можно задать подпроцессу больший или меньший приоритет.

В классе `Thread` есть три целые статические константы, задающие приоритеты:

- `NORM_PRIORITY` — обычный приоритет, который получает каждый подпроцесс при запуске, его числовое значение 5;
- `MIN_PRIORITY` — наименьший приоритет, его значение 1;
- `MAX_PRIORITY` — наивысший приоритет, его значение 10.

Кроме этих значений можно задать любое промежуточное значение от 1 до 10, но надо помнить о том, что процессор будет переключаться между подпроцессами с одинаковым высшим приоритетом, а подпроцессы с меньшим приоритетом не станут выполняться, если только не приостановлены все подпроцессы с высшим приоритетом. Поэтому для повышения общей производительности следует приостанавливать время от времени методом `sleep()` подпроцессы с высоким приоритетом.

Установить тот или иной приоритет можно в любое время методом `setPriority(int newPriority)`, если подпроцесс имеет право изменить свой приоритет. Проверить наличие такого права можно методом `checkAccess()`. Этот метод выбрасывает исключение класса `SecurityException`, если подпроцесс не может изменить свой приоритет.

Порожденные подпроцессы будут иметь тот же приоритет, что и подпроцесс-родитель.

Итак, подпроцессы, как правило, должны работать с приоритетом `NORM_PRIORITY`. Подпроцессы, большую часть времени ожидающие наступления какого-нибудь события, например, нажатия пользователем кнопки **Выход**, могут получить более высокий приоритет `MAX_PRIORITY`. Подпроцессы, выполняющие длительную работу, например, установку сетевого соединения или отрисовку изображения в памяти при двойной буферизации, могут работать с низшим приоритетом `MIN_PRIORITY`.

Подпроцессы-демоны

Работа программы начинается с выполнения метода `main()` главным подпроцессом. Этот подпроцесс может породить другие подпроцессы, они, в свою очередь, способны породить свои подпроцессы. После этого главный подпроцесс ничем не будет отличаться от остальных подпроцессов. Он не следит за порожденными им подпроцессами, не ждет от них никаких сигналов. Главный подпроцесс может завершиться, а программа будет продолжать работу, пока не закончит работу последний подпроцесс.

Это правило не всегда удобно. Например, какой-то из подпроцессов может приостановиться, ожидая сетевого соединения, которое никак не может наступить. Пользователь, не дождавшись соединения, прекращает работу главного подпроцесса, но программа продолжает работать.

Такие случаи можно учесть, объявив некоторые подпроцессы *демонами* (daemons). Это понятие не совпадает с понятием демона в UNIX. Просто программа завершается по окончании работы последнего *пользовательского* (user) подпроцесса, не дожидаясь окончания работы демонов. Демоны будут принудительно завершены исполняющей системой Java.

Объявить подпроцесс демоном можно сразу после его создания, перед запуском. Это делается методом `setDaemon(true)`. Данный метод обращается к методу `checkAccess()` и может выбросить `SecurityException`.

Изменить статус демона после запуска процесса уже нельзя.

Все подпроцессы, порожденные демоном, тоже будут демонами. Для изменения их статуса необходимо обратиться к методу `setDaemon(false)`.

Группы подпроцессов

Подпроцессы объединяются в группы. В начале работы программы исполняющая система Java создает группу подпроцессов с именем `main`. Все подпроцессы по умолчанию попадают в эту группу.

В любое время программа может создать новые группы подпроцессов и подпроцессы, входящие в эти группы. Вначале создается группа — экземпляр класса `ThreadGroup`, конструктором

```
ThreadGroup(String name)
```

При этом группа получает имя, заданное аргументом `name`. Затем этот экземпляр указывается при создании подпроцессов в конструкторах класса `Thread`. Все подпроцессы попадут в группу с именем, заданным при создании группы.

Группы подпроцессов могут образовать иерархию. Одна группа порождается от другой конструктором

```
ThreadGroup(ThreadGroup parent, String name)
```

Группы подпроцессов используются главным образом для задания приоритетов подпроцессам внутри группы. Изменение приоритетов внутри группы не будет влиять на приоритеты подпроцессов вне иерархии этой группы. Каждая группа имеет максимальный приоритет, устанавливаемый методом `setMaxPriority(int maxPri)` класса `ThreadGroup`. Ни один подпроцесс из этой группы не может превысить значения `maxPri`, но приоритеты подпроцессов, заданные до установки `maxPri`, не меняются.

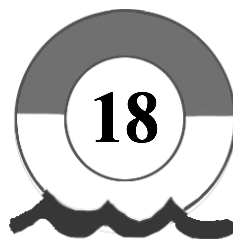
Заключение

Технология Java по своей сути — многозадачная технология, основанная на threads. Это одна из причин, по которым технология Java так и не может разумным образом реализоваться в MS-DOS и Windows 3.1, несмотря на многие попытки.

Поэтому, конструируя программу для Java, следует все время помнить, что она будет выполняться в многозадачной среде. Надо ясно представлять себе, что будет, если программа начнет выполняться одновременно несколькими подпроцессами, выделять критические участки и синхронизировать их.

С другой стороны, если программа осуществляет несколько действий, следует подумать, не сделать ли их выполнение одновременным, создав дополнительные подпроцессы и распределив их приоритеты.

ГЛАВА 18



Потоки ввода/вывода

Программы, написанные нами в предыдущих главах, воспринимали информацию только из параметров командной строки и графических компонентов, а результаты выводили на консоль или в графические компоненты. Однако во многих случаях требуется выводить результаты на принтер, в файл, базу данных или передавать по сети. Исходные данные тоже часто приходится загружать из файла, базы данных или из сети.

Для того чтобы отвлечься от особенностей конкретных устройств ввода/вывода, в Java употребляется понятие *потока* (stream). Считается, что в программу идет *входной поток* (input stream) символов Unicode или просто байтов, воспринимаемый в программе методами `read()`. Из программы методами `write()` или `print()`, `println()` выводится *выходной поток* (output stream) символов или байтов. При этом неважно, куда направлен поток: на консоль, на принтер, в файл или в сеть, методы `write()` и `print()` ничего об этом не знают.

Можно представить себе поток как трубу, по которой в одном направлении последовательно "текут" символы или байты, один за другим. Методы `read()`, `write()`, `print()`, `println()` взаимодействуют с одним концом трубы, другой конец соединяется с источником или приемником данных конструкторами классов, в которых реализованы эти методы.

Конечно, полное игнорирование особенностей устройств ввода/вывода сильно замедляет передачу информации. Поэтому в Java все-таки выделяется файловый ввод/вывод, вывод на печать, сетевой поток.

Три потока определены в классе `System` статическими полями `in`, `out` и `err`. Их можно использовать без всяких дополнительных определений, что мы и делали на протяжении всей книги. Они называются соответственно *стандартным вводом* (stdin), *стандартным выводом* (stdout) и *стандартным выводом сообщений* (stderr). Эти стандартные потоки могут быть соединены с разными конкретными устройствами ввода и вывода.

Потоки out и err — это экземпляры класса PrintStream, организующего выходной поток байтов. Эти экземпляры выводят информацию на консоль методами print(), println() и write(), которых в классе PrintStream имеется около двадцати для разных типов аргументов.

Поток err предназначен для вывода системных сообщений программы: трассировки, сообщений об ошибках или, просто, о выполнении каких-то этапов программы. Такие сведения обычно заносятся в специальные журналы, log-файлы, а не выводятся на консоль. В Java есть средства переназначения потока, например, с консоли в файл.

Поток in — это экземпляр класса InputStream. Он назначен на клавиатурный ввод с консоли методами read(). Класс InputStream абстрактный, поэтому реально используется какой-то из его подклассов.

Понятие потока оказалось настолько удобным и облегчающим программирование ввода/вывода, что в Java предусмотрена возможность создания потоков, направляющих символы или байты не на внешнее устройство, а в массив или из массива, т. е. связывающих программу с областью оперативной памяти. Более того, можно создать поток, связанный со строкой типа String, находящейся, опять-таки, в оперативной памяти. Кроме того, можно создать *канал* (pipe) обмена информацией между подпроцессами.

Еще один вид потока — поток байтов, составляющих объект Java. Его можно направить в файл или передать по сети, а потом восстановить в оперативной памяти. Эта операция называется *сериализацией* (serialization) объектов.

Методы организации потоков собраны в классы пакета java.io.

Кроме классов, организующих поток, в пакет java.io входят классы с методами преобразования потока, например, можно преобразовать поток байтов, образующих целые числа, в поток этих чисел.

Еще одна возможность, предоставляемая классами пакета java.io, — слить несколько потоков в один поток.

Итак, в Java есть целых четыре иерархии классов для создания, преобразования и слияния потоков. Во главе иерархии четыре класса, непосредственно расширяющих класс Object:

- Reader — абстрактный класс, в котором собраны самые общие методы символьного ввода;
- Writer — абстрактный класс, в котором собраны самые общие методы символьного вывода;
- InputStream — абстрактный класс с общими методами байтового ввода;
- OutputStream — абстрактный класс с общими методами байтового вывода.

Классы входных потоков Reader и InputStream определяют по три метода ввода:

- `read()` — возвращает один символ или байт, взятый из входного потока, в виде целого значения типа `int`; если поток уже закончился, возвращает −1;
- `read(char[] buf)` — заполняет заранее определенный массив `buf` символами из входного потока; в классе `InputStream` массив типа `byte[]` и заполняется он байтами; метод возвращает фактическое число взятых из потока элементов или −1, если поток уже закончился;
- `read(char[] buf, int offset, int len)` — заполняет часть символьного или байтового массива `buf`, начиная с индекса `offset`, число взятых из потока элементов равно `len`; метод возвращает фактическое число взятых из потока элементов или −1.

Эти методы выбрасывают `IOException`, если произошла ошибка ввода/вывода.

Четвертый метод `skip(long n)` "проматывает" поток с текущей позиции на `n` символов или байтов вперед. Эти элементы потока не вводятся методами `read()`. Метод возвращает реальное число пропущенных элементов, которое может отличаться от `n`, например поток может закончиться.

Текущий элемент потока можно пометить методом `mark(int n)`, а затем вернуться к помеченному элементу методом `reset()`, но не более чем через `n` элементов. Не все подклассы реализуют эти методы, поэтому перед расстановкой пометок следует обратиться к логическому методу `markSupported()`, который возвращает `true`, если реализованы методы расстановки и возврата к пометкам.

Классы выходных потоков `Writer` и `OutputStream` определяют по три почти одинаковых метода вывода:

- `write(char[] buf)` — выводит массив в выходной поток, в классе `OutputStream` массив имеет тип `byte[]`;
- `write(char[] buf, int offset, int len)` — выводит `len` элементов массива `buf`, начиная с элемента с индексом `offset`;
- `write(int elem)` в классе `Writer` — выводит 16, а в классе `OutputStream` 8 младших битов аргумента `elem` в выходной поток.

В классе `Writer` есть еще два метода:

- `write(String s)` — выводит строку `s` в выходной поток;
- `write(String s, int offset, int len)` — выводит `len` символов строки `s`, начиная с символа с номером `offset`.

Многие подклассы классов `Writer` и `OutputStream` осуществляют буферизованный вывод. При этом элементы сначала накапливаются в буфере, в оперативной памяти, и выводятся в выходной поток только после того, как буфер заполнится. Это удобно для выравнивания скоростей вывода из программы и вывода потока, но часто надо вывести информацию в поток еще

до заполнения буфера. Для этого предусмотрен метод `flush()`. Данный метод сразу же выводит все содержимое буфера в поток.

Наконец, по окончании работы с потоком его необходимо закрыть методом `close()`.

Классы, входящие в иерархии потоков ввода/вывода, показаны на рис. 18.1 и 18.2.

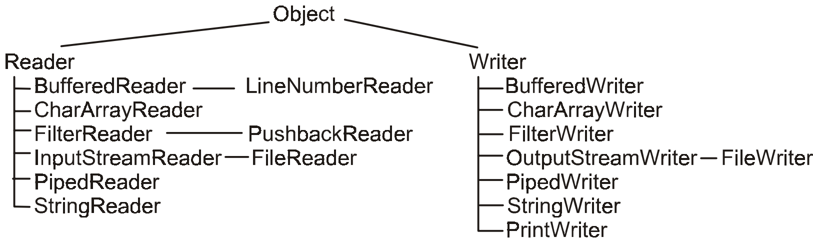


Рис. 18.1. Иерархия символьных потоков

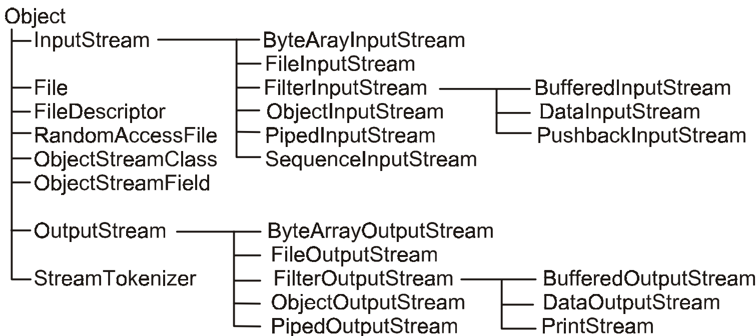


Рис. 18.2. Классы байтовых потоков

Все классы пакета `java.io` можно разделить на две группы: классы, создающие поток (data sink), и классы, управляющие потоком (data processing).

Классы, создающие потоки, в свою очередь, можно разделить на пять групп:

- классы, создающие потоки, связанные с файлами:

`FileReader`	`FileInputStream`
`FileWriterFile`	`OutputStream`
	`RandomAccessFile`

- классы, создающие потоки, связанные с массивами:

`CharArrayReader`	`ByteArrayInputStream`
`CharArrayWriter`	`ByteArrayOutputStream`

- классы, создающие каналы обмена информацией между подпроцессами:

PipedReader	PipedInputStream
PipedWriter	PipedOutputStream

- классы, создающие символьные потоки, связанные со строкой:

StringReader	
StringWriter	

- классы, создающие байтовые потоки из объектов Java:

	ObjectInputStream
	ObjectOutputStream

Слева перечислены классы символьных потоков, справа — классы байтовых потоков.

Классы, управляющие потоком, получают в своих конструкторах уже имеющийся поток и создают новый, преобразованный поток. Можно представлять их себе как "переходное кольцо", после которого идет труба другого диаметра.

Четыре класса созданы специально для преобразования потоков:

FilterReader	FilterInputStream
FilterWriter	FilterOutputStream

Сами по себе эти классы бесполезны — они выполняют тождественное преобразование. Их следует расширять, переопределяя методы ввода/вывода. Но для байтовых фильтров есть полезные расширения, которым соответствуют некоторые символьные классы. Перечислим их.

Четыре класса выполняют буферизованный ввод/вывод:

BufferedReader	BufferedInputStream
BufferedWriter	BufferedOutputStream

Два класса преобразуют поток байтов, образующих восемь простых типов Java, в эти самые типы:

	DataInputStream
	DataOutputStream

Два класса содержат методы, позволяющие вернуть несколько символов или байтов во входной поток:

PushbackReader	PushbackInputStream

Два класса связаны с выводом на строчные устройства — экран дисплея, принтер:

PrintWriter	PrintStream

Два класса связывают байтовый и символьный потоки:

- `InputStreamReader` — преобразует входной байтовый поток в символьный поток;
- `OutputStreamWriter` — преобразует выходной символьный поток в байтовый поток.

Класс `StreamTokenizer` позволяет разобрать входной символьный поток на отдельные элементы (tokens) подобно тому, как класс `StringTokenizer`, рассмотренный нами в *главе 5*, разбирал строку.

Из управляющих классов выделяется класс `SequenceInputStream`, сливающий несколько потоков, заданных в конструкторе, в один поток, и класс `LineNumberReader`, "умеющий" читать входной символьный поток построчно. Строки в потоке разделяются символами `'\n'` и/или `'\r'`.

Этот обзор классов ввода/вывода немного проясняет положение, но не объясняет, как их использовать. Перейдем к рассмотрению реальных ситуаций.

Консольный ввод/вывод

Для вывода на консоль мы всегда использовали метод `println()` класса `PrintStream`, никогда не определяя экземпляры этого класса. Мы просто использовали статическое поле `out` класса `System`, которое является объектом класса `PrintStream`. Исполняющая система Java связывает это поле с консолью.

Кстати говоря, если вам надоело писать `System.out.println()`, то вы можете определить новую ссылку на `System.out`, например:

```
PrintStream pr = System.out;
```

и писать просто `pr.println()`.

Консоль является байтовым устройством, и символы Unicode перед выводом на консоль должны быть преобразованы в байты. Для символов Latin 1 с кодами `'\u0000'—'\u00FF'` при этом просто откидывается нулевой старший байт и выводятся байты `'0x00'—'0xFF'`. Для кодов кириллицы, которые лежат в диапазоне `'\u0400'—'\u04FF'` кодировки Unicode, и других национальных алфавитов производится преобразование по кодовой таблице, соответствующей установленной на компьютере локали. Мы обсуждали это в *главе 5*.

Трудности с отображением кириллицы возникают, если вывод на консоль производится в кодировке, отличной от локали. Именно так происходит в русифицированных версиях MS Windows NT/2000. Обычно в них устанавливается локаль с кодовой страницей CP1251, а вывод на консоль происходит в кодировке CP866.

В этом случае надо заменить PrintStream, который не может работать с символьным потоком, на PrintWriter и "вставить переходное кольцо" между потоком символов Unicode и потоком байтов System.out, выводимых на консоль, в виде объекта класса OutputStreamWriter. В конструкторе этого объекта следует указать нужную кодировку, в данном случае, CP866. Все это можно сделать одним оператором:

```
PrintWriter pw = new PrintWriter(
        new OutputStreamWriter(System.out, "Cp866"), true);
```

Класс PrintStream буферизует выходной поток. Второй аргумент true его конструктора вызывает принудительный сброс содержимого буфера в выходной поток после каждого выполнения метода println(). Но после print() буфер не сбрасывается! Для сброса буфера после каждого print() надо писать flush(), как это сделано в листинге 18.2.

Замечание

Методы класса PrintWriter по умолчанию не очищают буфер, а метод print() не очищает его в любом случае. Для очистки буфера используйте метод flush().

После этого можно выводить любой текст методами класса PrintWriter, которые просто дублируют методы класса PrintStream, и писать, например,

```
pw.println("Это русский текст");
```

как показано в листинге 18.1 и на рис. 18.3.

Следует заметить, что конструктор класса PrintWriter, в котором задан байтовый поток, всегда неявно создает объект класса OutputStreamWriter с локальной кодировкой для преобразования байтового потока в символьный поток.

Ввод с консоли производится методами read() класса InputStream с помощью статического поля in класса System. С консоли идет поток байтов, полученных из scan-кодов клавиатуры. Эти байты должны быть преобразованы в символы Unicode такими же кодовыми таблицами, как и при выводе на консоль. Преобразование идет по той же схеме — для правильного ввода кириллицы удобнее всего определить экземпляр класса BufferedReader, используя в качестве "переходного кольца" объект класса InputStreamReader:

```
BufferedReader br = new BufferedReader(
        new InputStreamReader(System.in, "Cp866"));
```

Класс BufferedReader переопределяет три метода read() своего суперкласса Reader. Кроме того, он содержит метод readLine().

Метод readLine() возвращает строку типа String, содержащую символы входного потока, начиная с текущего, и заканчивая символом '\n' и/или

`'\r'`. Эти символы-разделители не входят в возвращаемую строку. Если во входном потоке нет символов, то возвращается `null`.

В листинге 18.1 приведена программа, иллюстрирующая перечисленные методы консольного ввода/вывода. На рис. 18.3 показан вывод этой программы.

Листинг 18.1. Консольный ввод/вывод

```
import java.io.*;

class PrWr{
  public static void main(String[] args){
    try{
       BufferedReader br =
          new BufferedReader(new InputStreamReader(System.in, "Cp866"));
       PrintWriter pw = new PrintWriter(
          new OutputStreamWriter(System.out, "Cp866"), true);
       String s = "Это строка с русским текстом";
       System.out.println("System.out puts:  " + s);
       pw.println("PrintWriter puts: " + s);
       int c = 0;
       pw.println("Посимвольный ввод:");
       while((c = br.read()) != -1)
          pw.println((char)c);
       pw.println("Построчный ввод:");
       do{
          s = br.readLine();
          pw.println(s);
       }while(!s.equals("q"));
   }catch(Exception e){
       System.out.println(e);
   }
  }
}
```

Поясним рис. 18.3. Первая строка выводится потоком `System.out`. Как видите, кириллица выводится неправильно. Следующая строка предварительно преобразована в поток байтов, записанных в кодировке CP866.

Затем, после текста "Посимвольный ввод:" с консоли вводятся символы "Россия" и нажимается клавиша <Enter>. Каждый вводимый символ отображается на экране — операционная система работает в режиме так называемого "эха". Фактический ввод с консоли начинается только после нажатия клавиши <Enter>, потому что клавиатурный ввод буферизуется операционной системой. Символы сразу после ввода отображаются по одному на строке. Обратите внимание на две пустые строки после буквы `я`. Это выве-

дены символы '\n' и '\r', которые попали во входной поток при нажатии клавиши <Enter>. У них нет никакого графического начертания (glyph).

Потом нажата комбинация клавиш <Ctrl>+<Z>. Она отображается на консоль как "^Z" и означает окончание клавиатурного ввода, завершая цикл ввода символов. Коды этих клавиш уже не попадают во входной поток.

Далее, после текста "Построчный ввод:" с клавиатуры набирается строка "Это строка" и, вслед за нажатием клавиши <Enter>, заносится в строку s. Затем строка s выводится обратно на консоль.

Для окончания работы набираем q и нажимаем клавишу <Enter>.

```
Command Prompt
D:\jdk1.3\MyProgs\io>javac PrWr.java

D:\jdk1.3\MyProgs\io>java PrWr
System.out puts:  Єю ёЄЁюър ё Ёєёёъшь ЄхъёЄюь
PrintWriter puts: Это строка с русским текстом
Посимвольный ввод:
Россия
Р
о
с
с
и
я



^Z
Построчный ввод:
Это строка
Это строка
q
q

D:\jdk1.3\MyProgs\io>
```

Рис. 18.3. Консольный ввод/вывод

Файловый ввод/вывод

Поскольку файлы в большинстве современных операционных систем понимаются как последовательность байтов, для файлового ввода/вывода создаются байтовые потоки с помощью классов FileInputStream и FileOutputStream. Это особенно удобно для бинарных файлов, хранящих байт-коды, архивы, изображения, звук.

Но очень много файлов содержат тексты, составленные из символов. Несмотря на то, что символы могут храниться в кодировке Unicode, эти тексты чаще всего записаны в байтовых кодировках. Поэтому и для текстовых файлов можно использовать байтовые потоки. В таком случае со стороны программы придется организовать преобразование байтов в символы и обратно.

Чтобы облегчить это преобразование, в пакет java.io введены классы FileReader и FileWriter. Они организуют преобразование потока: со стороны программы потоки символьные, со стороны файла — байтовые. Это

происходит потому, что данные классы расширяют классы `InputStreamReader` и `OutputStreamWriter`, соответственно, значит, содержат "переходное кольцо" внутри себя.

Несмотря на различие потоков, использование классов файлового ввода/вывода очень похоже.

В конструкторах всех четырех файловых потоков задается имя файла в виде строки типа `String` или ссылка на объект класса `File`. Конструкторы не только создают объект, но и отыскивают файл и открывают его. Например:

```
FileInputStream fis = new FileInputStream("PrWr.java");
FileReader fr = new FileReader("D:\\jdk1.3\\src\\PrWr.java");
```

При неудаче выбрасывается исключение класса `FileNotFoundException`, но конструктор класса `FileWriter` выбрасывает более общее исключение `IOException`.

После открытия выходного потока типа `FileWriter` или `FileOutputStream` содержимое файла, если он был не пуст, стирается. Для того чтобы можно было делать запись в конец файла, и в том и в другом классе предусмотрен конструктор с двумя аргументами. Если второй аргумент равен `true`, то происходит дозапись в конец файла, если `false`, то файл заполняется новой информацией. Например:

```
FileWriter fw = new FileWriter("ch18.txt", true);
FileOutputStream fos = new FileOutputStream("D:\\samples\\newfile.txt");
```

Внимание

Содержимое файла, открытого на запись конструктором с одним аргументом, стирается.

Сразу после выполнения конструктора можно читать файл:

```
fis.read(); fr.read();
```

или записывать в него:

```
fos.write((char)c); fw.write((char)c);
```

По окончании работы с файлом поток следует закрыть методом `close()`.

Преобразование потоков в классах `FileReader` и `FileWriter` выполняется по кодовым таблицам установленной на компьютере локали. Для правильного ввода кириллицы надо применять `FileReader`, а не `FileInputStream`. Если файл содержит текст в кодировке, отличной от локальной кодировки, то придется вставлять "переходное кольцо" вручную, как это делалось для консоли, например:

```
InputStreamReader isr = new InputStreamReader(fis, "KOI8_R"));
```

Байтовый поток `fis` определен выше.

Получение свойств файла

В конструкторах классов файлового ввода/вывода, описанных в предыдущем разделе, указывалось имя файла в виде строки. При этом оставалось неизвестным, существует ли файл, разрешен ли к нему доступ, какова длина файла.

Получить такие сведения можно от предварительно созданного экземпляра класса `File`, содержащего сведения о файле. В конструкторе этого класса

```
File(String filename)
```

указывается путь к файлу или каталогу, записанный по правилам операционной системы. В UNIX имена каталогов разделяются наклонной чертой `/`, в MS Windows — обратной наклонной чертой `\`, в Apple Macintosh — двоеточием `:`. Этот символ содержится в системном свойстве `file.separator` (см. рис. 6.2). Путь к файлу предваряется префиксом. В UNIX это наклонная черта, в MS Windows — буква раздела диска, двоеточие и обратная наклонная черта. Если префикса нет, то путь считается относительным и к нему прибавляется путь к текущему каталогу, который хранится в системном свойстве `user.dir`.

Конструктор не проверяет, существует ли файл с таким именем, поэтому после создания объекта следует это проверить логическим методом `exists()`.

Класс `File` содержит около сорока методов, позволяющих узнать различные свойства файла или каталога.

Прежде всего, логическими методами `isFile()`, `isDirectory()` можно выяснить, является ли путь, указанный в конструкторе, путем к файлу или каталогу.

Для каталога можно получить его содержимое — список имен файлов и подкаталогов — методом `list()`, возвращающим массив строк `String[]`. Можно получить такой же список в виде массива объектов класса `File[]` методом `listFiles()`. Можно выбрать из списка только некоторые файлы, реализовав интерфейс `FileNameFilter` и обратившись к методу `list(FileNameFilter filter)`.

Если каталог с указанным в конструкторе путем не существует, его можно создать логическим методом `mkdir()`. Этот метод возвращает `true`, если каталог удалось создать. Логический метод `mkdirs()` создает еще и все несуществующие каталоги, указанные в пути.

Пустой каталог удаляется методом `delete()`.

Для файла можно получить его длину в байтах методом `length()`, время последней модификации в секундах с 1 января 1970 г. методом `lastModified()`. Если файл не существует, эти методы возвращают нуль.

Логические методы `canRead()`, `canWrite()` показывают права доступа к файлу.

Файл можно переименовать логическим методом `renameTo(File newName)` или удалить логическим методом `delete()`. Эти методы возвращают `true`, если операция прошла удачно.

Если файл с указанным в конструкторе путем не существует, его можно создать логическим методом `createNewFile()`, возвращающим `true`, если файл не существовал, и его удалось создать, и `false`, если файл уже существовал.

Статическими методами

```
createTempFile(String prefix, String suffix, File tmpDir)
createTempFile(String prefix, String suffix)
```

можно создать временный файл с именем `prefix` и расширением `suffix` в каталоге `tmpDir` или каталоге, указанном в системном свойстве `java.io.tmpdir` (см. рис. 6.2). Имя `prefix` должно содержать не менее трех символов. Если `suffix == null`, то файл получит суффикс .tmp.

Перечисленные методы возвращают ссылку типа `File` на созданный файл. Если обратиться к методу `deleteOnExit()`, то по завершении работы JVM временный файл будет уничтожен.

Несколько методов `get`*`Xxx`*`()` возвращают имя файла, имя каталога и другие сведения о пути к файлу. Эти методы полезны в тех случаях, когда ссылка на объект класса `File` возвращается другими методами и нужны сведения о файле.

Наконец, метод `toURL()` возвращает путь к файлу в форме URL.

В листинге 18.2 показан пример использования класса `File`, а на рис. 18.4 — начало вывода этой программы.

Листинг 18.2. Определение свойств файла и каталога

```
import java.io.*;

class FileTest{
  public static void main(String[] args) throws IOException{
     PrintWriter pw = new PrintWriter(
         new OutputStreamWriter(System.out, "Cp866"), true);
     File f = new File("FileTest.java");
     pw.println();
     pw.println("Файл \"" + f.getName() + "\" " +
           (f.exists()?"":"не ") + "существует");
     pw.println("Вы " + (f.canRead()?"":"не ") + "можете читать файл");
     pw.println("Вы " + (f.canWrite()?"":"не ") +
           "можете записывать в файл");
     pw.println("Длина файла " + f.length() + " б");
```

```
      pw.println();
      File d = new File("D:\\jdk1.3\\MyProgs");
      pw.println("Содержимое каталога:");
      if (d.exists() && d.isDirectory()){
         String[] s = d.list();
         for (int i = 0; i < s.length; i++)
            pw.println(s[i]);
      }
   }
}
```

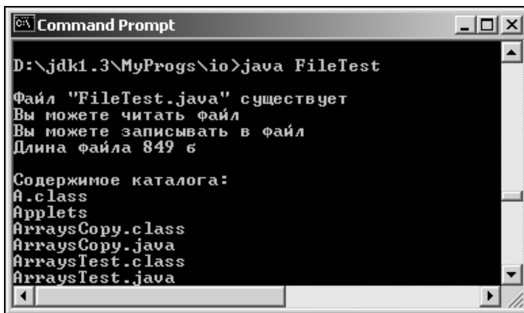


Рис. 18.4. Свойства файла и начало вывода каталога

Буферизованный ввод/вывод

Операции ввода/вывода по сравнению с операциями в оперативной памяти выполняются очень медленно. Для компенсации в оперативной памяти выделяется некоторая промежуточная область — буфер, в которой постепенно накапливается информация. Когда буфер заполнен, его содержимое быстро переносится процессором, буфер очищается и снова заполняется информацией.

Житейский пример буфера — почтовый ящик, в котором накапливаются письма. Мы бросаем в него письмо и уходим по своим делам, не дожидаясь приезда почтовой машины. Почтовая машина периодически очищает почтовый ящик, перенося сразу большое число писем. Представьте себе город, в котором нет почтовых ящиков, и толпа людей с письмами в руках дожидается приезда почтовой машины.

Классы файлового ввода/вывода не занимаются буферизацией. Для этой цели есть четыре специальных класса `Buffered`*`Xxx`*, перечисленных выше. Они присоединяются к потокам ввода/вывода как "переходное кольцо", например:

```
BufferedReader br = new BufferedReader(isr);
BufferedWriter bw = new BufferedWriter(fw);
```

Потоки `isr` и `fw` определены выше.

Программа листинга 18.3 читает текстовый файл, написанный в кодировке CP866, и записывает его содержимое в файл в кодировке KOI8_R. При чтении и записи применяется буферизация. Имя исходного файла задается в командной строке параметром `args[0]`, имя копии — параметром `args[1]`.

Листинг 18.3. Буферизованный файловый ввод/вывод

```
import java.io.*;

class DOStoUNIX{
  public static void main(String[] args) throws IOException{
      if (args.length != 2){
         System.err.println("Usage: DOStoUNIX Cp866file KOI8_Rfile");
         System.exit(0);
      }
      BufferedReader br = new BufferedReader(
            new InputStreamReader(
                  new FileInputStream(args[0]), "Cp866"));
      BufferedWriter bw = new BufferedWriter(
            new OutputStreamWriter(
                  new FileOutputStream(args[1]), "KOI8_R"));
      int c = 0;
      while ((c = br.read()) != -1)
         bw.write((char)c);
      br.close(); bw.close();
      System.out.println("The job's finished.");
  }
}
```

Поток простых типов Java

Класс `DataOutputStream` позволяет записать данные простых типов Java в выходной поток байтов методами `writeBoolean(boolean b)`, `writeByte(int b)`, `writeShort(int h)`, `writeChar(int c)`, `writeInt(int n)`, `writeLong(long l)`, `writeFloat(float f)`, `writeDouble(double d)`.

Кроме того, метод `writeBytes(String s)` записывает каждый символ строки `s` в один байт, отбрасывая старший байт кодировки каждого символа Unicode, а метод `writeChars(String s)` записывает каждый символ строки `s` в два байта, первый байт — старший байт кодировки Unicode, так же, как это делает метод `writeChar()`.

Еще один метод `writeUTF(String s)` записывает строку `s` в выходной поток в кодировке UTF-8. Надо пояснить эту кодировку.

Кодировка UTF-8

Запись потока в байтовой кодировке вызывает трудности с использованием национальных символов, запись потока в Unicode увеличивает длину потока в два раза. Кодировка UTF-8 (Universal Transfer Format) является компромиссом. Символ в этой кодировке записывается одним, двумя или тремя байтами.

Символы Unicode из диапазона `'\u0000'`—`'\u007F'`, в котором лежит английский алфавит, записываются одним байтом, старший байт просто отбрасывается.

Символы Unicode из диапазона `'\u0080'`—`'\u07FF'`, в котором лежат наиболее распространенные символы национальных алфавитов, записываются двумя байтами следующим образом: символ Unicode с кодировкой 00000xxxxxyyyyyy записывается как 110xxxxx10yyyyyy.

Остальные символы Unicode из диапазона `'\u0800'`—`'\uFFFF'` записываются тремя байтами по следующему правилу: символ Unicode с кодировкой xxxxyyyyyyzzzzzz записывается как 1110xxxx10yyyyyy10zzzzzz.

Такой странный способ распределения битов позволяет по первым битам кода узнать, сколько байтов составляет код символа, и правильно отсчитывать символы в потоке.

Так вот, метод `writeUTF(String s)` сначала записывает в поток в первые два байта потока длину строки `s` в кодировке UTF-8, а затем символы строки в этой кодировке. Читать эту запись потом следует парным методом `readUTF()` класса `DataInputStream`.

Класс `DataInputStream` преобразует входной поток байтов типа `InputStream`, составляющих данные простых типов Java, в данные этого типа. Такой поток, как правило, создается методами класса `DataOutputStream`. Данные из этого потока можно прочитать методами `readBoolean()`, `readByte()`, `readShort()`, `readChar()`, `readInt()`, `readLong()`, `readFloat()`, `readDouble()`, возвращающими данные соответствующего типа.

Кроме того, методы `readUnsignedByte()` и `readUnsignedShort()` возвращают целое типа `int`, в котором старшие три или два байта нулевые, а младшие один или два байта заполнены байтами из входного потока.

Метод `readUTF()`, двойственный методу `writeUTF()`, возвращает строку типа `String`, полученную из потока, записанного методом `writeUTF()`.

Еще один, статический, метод `readUTF(DataInput in)` делает то же самое со входным потоком `in`, записанным в кодировке UTF-8. Этот метод можно применять, не создавая объект класса `DataInputStream`.

Программа в листинге 18.4 записывает в файл fib.txt числа Фибоначчи, а затем читает этот файл и выводит его содержимое на консоль. Для контроля

записываемые в файл числа тоже выводятся на консоль. На рис. 18.5 показан вывод этой программы.

Листинг 18.4. Ввод/вывод данных

```
import java.io.*;

class DataPrWr{
  public static void main(String[] args) throws IOException{
      DataOutputStream dos = new DataOutputStream(
           new FileOutputStream("fib.txt"));
      int a = 1, b = 1, c = 1;
      for (int k = 0; k < 40; k++){
         System.out.print(b + " ");
         dos.writeInt(b);
         a = b; b = c; c = a + b;
      }
      dos.close();
      System.out.println("\n");
      DataInputStream dis = new DataInputStream(
           new FileInputStream("fib.txt"));
      while(true)
        try{
           a = dis.readInt();
           System.out.print(a + " ");
        }catch(IOException e){
           dis.close();
           System.out.println("End of file");
           System.exit(0);
        }
  }
}
```

Обратите внимание на то, что попытка чтения за концом файла выбрасывает исключение класса `IOException`, его обработка заключается в закрытии файла и окончании программы.

```
Command Prompt
D:\jdk1.3\MyProgs\io>javac DataPrWr.java

D:\jdk1.3\MyProgs\io>java DataPrWr
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 2
8657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702
887 9227465 14930352 24157817 39088169 63245986 102334155

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 2
8657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702
887 9227465 14930352 24157817 39088169 63245986 102334155 End of file

D:\jdk1.3\MyProgs\io>
```

Рис. 18.5. Ввод и вывод данных

Прямой доступ к файлу

Если необходимо интенсивно работать с файлом, записывая в него данные разных типов Java, изменяя их, отыскивая и читая нужную информацию, то лучше всего воспользоваться методами класса `RandomAccessFile`.

В конструкторах этого класса

```
RandomAccessFile(File file, String mode)
RandomAccessFile(String fileName, String mode)
```

вторым аргументом `mode` задается режим открытия файла. Это может быть строка `"r"` — открытие файла только для чтения, или `"rw"` — открытие файла для чтения и записи.

Этот класс собрал все полезные методы работы с файлом. Он содержит все методы классов `DataInputStream` и `DataOutputStream`, кроме того, позволяет прочитать сразу целую строку методом `readLine()` и отыскать нужные данные в файле.

Байты файла нумеруются, начиная с 0, подобно элементам массива. Файл снабжен неявным указателем (file pointer) текущей позиции. Чтение и запись производится, начиная с текущей позиции файла. При открытии файла конструктором указатель стоит на начале файла, в позиции 0. Текущую позицию можно узнать методом `getFilePointer()`. Каждое чтение или запись перемещает указатель на длину прочитанного или записанного данного. Всегда можно переместить указатель в новую позицию `pos` методом `seek(long pos)`. Метод `seek(0)` перемещает указатель на начало файла.

В классе нет методов преобразования символов в байты и обратно по кодовым таблицам, поэтому он не приспособлен для работы с кириллицей.

Каналы обмена информацией

В предыдущей главе мы видели, каких трудов стоит организовать правильный обмен информацией между подпроцессами. В пакете `java.io` есть четыре класса `Piped`*`Xxx`*, облегчающие эту задачу.

В одном подпроцессе — источнике информации — создается объект класса `PipedWriter+` или `PipedOutputstream`, в который записывается информация методами `write()` этих классов.

В другом подпроцессе — приемнике информации — формируется объект класса `PipedReader` или `PipedInputStream`. Он связывается с объектом-источником с помощью конструктора или специальным методом `connect()`, и читает информацию методами `read()`.

Источник и приемник можно создать и связать в обратном порядке.

Так создается однонаправленный *канал* (pipe) информации. На самом деле это некоторая область оперативной памяти, к которой организован совместный доступ двух или более подпроцессов. Доступ синхронизируется, записывающие процессы не могут помешать чтению.

Если надо организовать двусторонний обмен информацией, то создаются два канала.

В листинге 18.5 метод `run()` класса `Source` генерирует информацию, для простоты просто целые числа `k`, и передает ее в канал методом `pw.write(k)`. Метод `run()` класса `Target` читает информацию из канала методом `pr.read()`. Концы канала связываются с помощью конструктора класса `Target`. На рис. 18.6 видна последовательность записи и чтения информации.

Листинг 18.5. Канал обмена информацией

```
import java.io.*;

class Target extends Thread{
   private PipedReader pr;
   Target(PipedWriter pw){
      try{
         pr = new PipedReader(pw);
      }catch(IOException e){
         System.err.println("From Target(): " + e);
      }
   }
   PipedReader getStream(){ return pr;}
   public void run(){
      while(true)
         try{
            System.out.println("Reading: " + pr.read());
         }catch(IOException e){
            System.out.println("The job's finished.");
            System.exit(0);
         }
   }
}
class Source extends Thread{
   private PipedWriter pw;
   Source(){
      pw = new PipedWriter();
   }
   PipedWriter getStream(){ return pw;}
   public void run(){
      for (int k = 0; k < 10; k++)
```

```
         try{
            pw.write(k);
            System.out.println("Writing: " + k);
         }catch(Exception e){
            System.err.println("From Source.run(): " + e);
         }
   }
}
class PipedPrWr{
  public static void main(String[] args){
     Source s = new Source();
     Target t = new Target(s.getStream());
     s.start(); t.start();
  }
}
```

```
Command Prompt
D:\jdk1.3\MyProgs\io>java PipedPrWr
Writing: 0
Writing: 1
Writing: 2
Writing: 3
Writing: 4
Writing: 5
Writing: 6
Writing: 7
Writing: 8
Writing: 9
Reading: 0
Reading: 1
Reading: 2
Reading: 3
Reading: 4
Reading: 5
Reading: 6
Reading: 7
Reading: 8
Reading: 9
The job's finished.
```

Рис. 18.6. Данные, передаваемые между подпроцессами

Сериализация объектов

Методы классов `ObjectInputStream` и `ObjectOutputStream` позволяют прочитать из входного байтового потока или записать в выходной байтовый поток данные сложных типов — объекты, массивы, строки — подобно тому, как методы классов `DataInputStream` и `DataOutputStream` читают и записывают данные простых типов.

Сходство усиливается тем, что классы `Object`*`Xxx`* содержат методы как для чтения, так и записи простых типов. Впрочем, эти методы предназначены не для использования в программах, а для записи/чтения полей объектов и элементов массивов.

Процесс записи объекта в выходной поток получил название *сериализации* (serialization), а чтения объекта из входного потока и восстановления его в оперативной памяти — *десериализации* (deserialization).

Сериализация объекта нарушает его безопасность, поскольку зловредный процесс может сериализовать объект в массив, переписать некоторые элементы массива, представляющие private-поля объекта, обеспечив себе, например, доступ к секретному файлу, а затем десериализовать объект с измененными полями и совершить с ним недопустимые действия.

Поэтому сериализации можно подвергнуть не каждый объект, а только тот, который реализует интерфейс `Serializable`. Этот интерфейс не содержит ни полей, ни методов. Реализовать в нем нечего. По сути дела запись

```
class A implements Serializable{...}
```

это только пометка, разрешающая сериализацию класса `A`.

Как всегда в Java, процесс сериализации максимально автоматизирован. Достаточно создать объект класса `ObjectOutputStream`, связав его с выходным потоком, и выводить в этот поток объекты методом `writeObject()`:

```
MyClass mc = new MyClass("abc", -12, 5.67e-5);
int[] arr = {10, 20, 30};
ObjectOutputStream oos = new ObjectOutputStream(
  new FileOutputStream("myobjects.ser"));
oos.writeObject(mc);
oos.writeObject(arr);
oos.writeObject("Some string");
oos.writeObject(new Date());
oos.flush();
```

В выходной поток выводятся все нестатические поля объекта, независимо от прав доступа к ним, а также сведения о классе этого объекта, необходимые для его правильного восстановления при десериализации. Байт-коды методов класса не сериализуются.

Если в объекте присутствуют ссылки на другие объекты, то они тоже сериализуются, а в них могут быть ссылки на другие объекты, которые опять-таки сериализуются, и получается целое множество причудливо связанных между собой сериализуемых объектов. Метод `writeObject()` распознает две ссылки на один объект и выводит его в выходной поток только один раз. К тому же, он распознает ссылки, замкнутые в кольцо, и избегает зацикливания.

Все классы объектов, входящих в такое сериализуемое множество, а также все их внутренние классы, должны реализовать интерфейс `Serializable`, в противном случае будет выброшено исключение класса `NotSerializableException` и процесс сериализации прервется. Многие классы J2SDK реализуют этот

интерфейс. Учтите также, что все потомки таких классов наследуют реализацию. Например, класс `java.awt.Component` реализует интерфейс `Serializable`, значит, все графические компоненты можно сериализовать.

Не реализуют этот интерфейс обычно классы, тесно связанные с выполнением программ, например, `java.awt.Toolkit`. Состояние экземпляров таких классов нет смысла сохранять или передавать по сети. Не реализуют интерфейс `Serializable` и классы, содержащие внутренние сведения Java "для служебного пользования".

Десериализация происходит так же просто, как и сериализация:

```
ObjectInputStream ois = new ObjectInputStream(
  new FileInputStream("myobjects.ser"));
MyClass mc1 = (MyClass)ois.readObject();
int[] a = (int[])ois.readObject();
String s = (String)ois.readObject();
Date d = (Date)ois.readObject();
```

Нужно только соблюдать порядок чтения элементов потока.

В листинге 18.6 мы создаем объект класса `GregorianCalendar` с текущей датой и временем, сериализуем его в файл date.ser, через три секунды десериализуем и сравниваем с текущим временем. Результат показан на рис. 18.7.

Листинг 18.6. Сериализация объекта

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

class SerDate{
  public static void main(String[] args) throws Exception{
    GregorianCalendar d = new GregorianCalendar();
    ObjectOutputStream oos = new ObjectOutputStream(
        new FileOutputStream("date.ser"));
    oos.writeObject(d);
    oos.flush();
    oos.close();

    Thread.sleep(3000);

    ObjectInputStream ois = new ObjectInputStream(
        new FileInputStream("date.ser"));
    GregorianCalendar oldDate = (GregorianCalendar)ois.readObject();
    ois.close();

    GregorianCalendar newDate = new GregorianCalendar();

    System.out.println("Old time = " +
            oldDate.get(Calendar.HOUR) + ":" +
```

```
            oldDate.get(Calendar.MINUTE) +":" +
            oldDate.get(Calendar.SECOND) +"\nNew time = " +
            newDate.get(Calendar.HOUR) +":" +
            newDate.get(Calendar.MINUTE) +":" +
            newDate.get(Calendar.SECOND));
  }
}
```

Рис. 18.7. Сериализация объекта

Если не нужно сериализовать какое-то поле, то достаточно пометить его служебным словом `transient`, например:

```
transient MyClass mc = new MyClass("abc", -12, 5.67e-5);
```

Метод `writeObject()` не записывает в выходной поток поля, помеченные `static` и `transient`. Впрочем, это положение можно изменить, переопределив метод `writeObject()` или задав список сериализуемых полей.

Вообще процесс сериализации можно полностью настроить под свои нужды, переопределив методы ввода/вывода и воспользовавшись вспомогательными классами. Можно даже взять весь процесс на себя, реализовав не интерфейс `Serializable`, а интерфейс `Externalizable`, но тогда придется реализовать методы `readExternal()` и `writeExternal()`, выполняющие ввод/вывод.

Эти действия выходят за рамки книги. Если вам необходимо полностью освоить процесс сериализации, то обратитесь к спецификации Java Object Serialization Specification, расположенной среди документации J2SDK в каталоге docs\guide\serialization\spec\. Там же есть и примеры программ, реализующих эту спецификацию.

Печать в Java

Поскольку принтер — устройство графическое, вывод на печать очень похож на вывод графических объектов на экран. Поэтому в Java средства печати входят в графическую библиотеку AWT и в систему Java 2D.

В графическом компоненте кроме графического контекста — объекта класса `Graphics`, создается еще "печатный контекст". Это тоже объект класса `Graphics`, но реализующий интерфейс `PrintGraphics` и полученный из другого источника — объекта класса `PrintJob`, входящего в пакет `java.awt`. Сам

же этот объект создается с помощью класса `Toolkit` пакета `java.awt`. На практике это выглядит так:

```
PrintJob pj = getToolkit().getPrintJob(this, "Job Title", null);
Graphics pg = pj.getGraphics();
```

Метод `getPrintJob()` сначала выводит на экран стандартное окно **Печать** (Print) операционной системы. Когда пользователь выберет в этом окне параметры печати и начнет печать кнопкой **OK**, создается объект `pj`. Если пользователь отказывается от печати при помощи кнопки **Отмена** (Cancel), то метод возвращает `null`.

В классе `Toolkit` два метода `getPrintJob()`:

```
getPrintJob(Frame frame, String jobTitle, JobAttributes jobAttr,
     PageAttributes pageAttr)
getPrintJob(Frame frame, String jobTitle, Properties prop)
```

Аргумент `frame` указывает на окно верхнего уровня, управляющее печатью. Этот аргумент не может быть `null`. Строка `jobTitle` задает заголовок задания, который не печатается, и может быть равна `null`. Аргумент `prop` зависит от реализации системы печати, часто это просто `null`, в данном случае задаются стандартные параметры печати.

Аргумент `jobAttr` задает параметры печати. Класс `JobAttributes`, экземпляром которого является этот аргумент, устроен сложно. В нем пять подклассов, содержащих статические константы — параметры печати, которые используются в конструкторе класса. Впрочем, есть конструктор по умолчанию, задающий стандартные параметры печати.

Аргумент `pageAttr` задает параметры страницы. Класс `PageProperties` тоже содержит пять подклассов со статическими константами, которые и задают параметры страницы и используются в конструкторе класса. Если для печати достаточно стандартных параметров, то можно воспользоваться конструктором по умолчанию.

Мы не будем рассматривать эти десять подклассов с десятками констант, чтобы не загромождать книгу мелкими подробностями. К тому же система Java 2D предлагает более удобный набор классов для печати, который мы рассмотрим в следующем пункте.

После того как "печатный контекст" — объект `pg` класса `Graphics` — определен, можно вызывать метод `print(pg)` или `printAll(pg)` класса `Component`. Этот метод устанавливает связь с принтером по умолчанию и вызывает метод `paint(pg)`. На печать выводится все то, что задано этим методом.

Например, чтобы распечатать текстовый файл, надо в процессе ввода разбить его текст на строки и в методе `paint(pg)` вывести строки методом `pg.drawString()` так же, как мы выводили их на экран в *главе 9*. При этом

следует учесть, что в "печатном контексте" нет шрифта по умолчанию, всегда надо устанавливать шрифт методом `pg.setFont()`.

После выполнения всех методов `print()` применяется метод `pg.dispose()`, вызывающий прогон страницы, и метод `pj.end()`, заканчивающий печать.

В листинге 18.7 приведен простой пример печати текста и окружности, заданных в методе `paint()`. Этот метод работает два раза: первый раз вычерчивая текст и окружность на экране, второй раз, точно так же, на листе бумаги, вставленной в принтер. Все методы печати собраны в один метод `simplePrint()`.

Листинг 18.7. Печать средствами AWT

```
import java.awt.*;
import java.awt.event.*;

class PrintTest extends Frame{
   PrintTest(String s){
      super(s);
      setSize(400, 400);
      setVisible(true);
   }
   public void simplePrint(){
      PrintJob pj =
        getToolkit().getPrintJob(this, "Job Title", null);
      if (pj != null){
         Graphics pg = pj.getGraphics();
         if (pg != null){
            print(pg);
            pg.dispose();
         }else System.err.println("Graphics's null");
         pj.end();
      }else System.err.println("Job's null");
   }
   public void paint(Graphics g){
      g.setFont(new Font("Serif", Font.ITALIC, 30));
      g.setColor(Color.black);
      g.drawArc(100, 100, 200, 200, 0, 360);
      g.drawString("Страница 1", 100, 100);
   }
   public static void main(String[] args){
      PrintTest pt = new PrintTest("  Простой пример печати");
      pt.simplePrint();
      pt.addWindowListener(new WindowAdapter(){
```

```
            public void windowClosing(WindowEvent ev){
              System.exit(0);
            }
         });
      }
   }
```

Печать средствами Java 2D

Расширенная графическая система Java 2D предлагает новые интерфейсы и классы для печати, собранные в пакет `java.awt.print`. Эти классы полностью перекрывают все стандартные возможности печати библиотеки AWT. Более того, они удобнее в работе и предлагают дополнительные возможности. Если этот пакет установлен в вашей вычислительной системе, то, безусловно, нужно применять его, а не стандартные средства печати AWT.

Как и стандартные средства AWT, методы классов Java 2D выводят на печать содержимое графического контекста, заполненного методами класса `Graphics` или класса `Graphics2D`.

Всякий класс Java 2D, собирающийся печатать хотя бы одну страницу текста, графики или изображения называется классом, *рисующим страницы* (page painter). Такой класс должен реализовать интерфейс `Printable`. В этом интерфейсе описаны две константы и только один метод `print()`. Класс, рисующий страницы, должен реализовать этот метод. Метод `print()` возвращает целое типа `int` и имеет три аргумента:

```
print(Graphics g, PageFormat pf, int ind);
```

Первый аргумент `g` — это графический контекст, выводимый на лист бумаги, второй аргумент `pf` — экземпляр класса `PageFormat`, определяющий размер и ориентацию страницы, третий аргумент `ind` — порядковый номер страницы, начинающийся с нуля.

Метод `print()` класса, рисующего страницы, заменяет собой метод `paint()`, использовавшийся стандартными средствами печати AWT. Класс, рисующий страницы, не обязан расширять класс `Frame` и переопределять метод `paint()`. Все заполнение графического контекста методами класса `Graphics` или `Graphics2D` теперь выполняется в методе `print()`.

Когда печать страницы будет закончена, метод `print()` должен возвратить целое значение, заданное константой `PAGE_EXISTS`. Будет сделано повторное обращение к методу `print()` для печати следующей страницы. Аргумент `ind` при этом возрастет на 1. Когда `ind` превысит количество страниц, метод `print()` должен возвратить значение `NO_SUCH_PAGE`, что служит сигналом окончания печати.

Следует помнить, что система печати может несколько раз обратиться к методу `paint()` для печати одной и той же страницы. При этом аргумент `ind` не меняется, а метод `print()` должен создать тот же графический контекст.

Класс `PageFormat` определяет параметры страницы. На странице вводится система координат с единицей длины 1/72 дюйма, начало которой и направление осей определяется одной из трех констант:

- `PORTRAIT` — начало координат расположено в левом верхнем углу страницы, ось Ох направлена вправо, ось Оу — вниз;
- `LANDSCAPE` — начало координат в левом нижнем углу, ось Ох идет вверх, ось Оу — вправо;
- `REVERSE_LANDSCAPE` — начало координат в правом верхнем углу, ось Ох идет вниз, ось Оу — влево.

Большинство принтеров не может печатать без полей, на всей странице, а осуществляет вывод только в некоторой *области печати* (imageable area), координаты левого верхнего угла которой возвращаются методами `getImageableX()` и `getImageableY()`, а ширина и высота — методами `getImageableWidth()` и `getImageableHeight()`.

Эти значения надо учитывать при расположении элементов в графическом контексте, например, при размещении строк текста методом `drawString()`, как это сделано в листинге 18.9.

В классе только один конструктор по умолчанию `PageFormat()`, задающий стандартные параметры страницы, определенные для принтера по умолчанию вычислительной системы.

Читатель, добравшийся до этого места книги, уже настолько поднаторел в Java, что у него возникает вопрос: "Как же тогда задать параметры страницы?" Ответ простой: "С помощью стандартного окна операционной системы".

Метод `pageDialog(PageDialog pd)` открывает на экране стандартное окно **Параметры страницы** (Page Setup) операционной системы, в котором уже заданы параметры, определенные в объекте `pd`. Если пользователь выбрал в этом окне кнопку **Отмена**, то возвращается ссылка на объект `pd`, если кнопку **OK**, то создается и возвращается ссылка на новый объект. Объект `pd` в любом случае не меняется. Он обычно создается конструктором.

Можно задать параметры страницы и из программы, но тогда следует сначала определить объект класса `Paper` конструктором по умолчанию:

```
Paper p = new Paper()
```

Затем методами

```
p.setSize(double width, double height)
p.setImageableArea(double x, double y, double width, double height)
```

задать размер страницы и области печати.

Потом определить объект класса `PageFormat` с параметрами по умолчанию:

```
PageFormat pf = new PageFormat()
```

и задать новые параметры методом

```
pf.setPaper(p)
```

Теперь вызывать на экран окно **Параметры страницы** методом `pageDialog()` уже не обязательно, и мы получим *молчаливый* (silent) процесс печати. Так делается в тех случаях, когда печать выполняется на фоне отдельным подпроцессом.

Итак, параметры страницы определены, метод `print()` — тоже. Теперь надо дать *задание на печать* (print job) — указать количество страниц, их номера, порядок печати страниц, количество копий. Все эти сведения собираются в классе `PrinterJob`.

Система печати Java 2D различает два вида заданий. В более простых заданиях — Printable Job — есть только один класс, рисующий страницы, поэтому у всех страниц одни и те же параметры, страницы печатаются последовательно с первой по последнюю или с последней страницы по первую, это зависит от системы печати.

Второй, более сложный вид заданий — Pageable Job — определяет для печати каждой страницы свой класс, рисующий страницы, поэтому у каждой страницы могут быть собственные параметры. Кроме того, можно печатать не все, а только выбранные страницы, выводить их в обратном порядке, печатать на обеих сторонах листа. Для осуществления этих возможностей определяется экземпляр класса `Book` или создается класс, реализующий интерфейс `Pageable`.

В классе `Book`, опять-таки, один конструктор, создающий пустой объект:

```
Book b = new Book()
```

После создания в данный объект добавляются классы, рисующие страницы. Для этого в классе `Book` есть два метода:

- `append(Printable p, PageFormat pf)` — добавляет объект `p` в конец;
- `append(Printable p, PageFormat pf, int numPages)` — добавляет `numPages` экземпляров `p` в конец; если число страниц заранее неизвестно, то задается константа `UNKNOWN_NUMBER_OF_PAGES`.

При составлении задания на печать, т. е. после создания экземпляра класса `PrinterJob`, надо указать вид задания одним и только одним из трех методов этого класса `setPrintable(Printable pr)`, `setPrintable(Printable pr, PageFormat pf)` или `setPageble(Pageable pg)`. Заодно задаются один или несколько классов `pr`, рисующих страницы в этом задании.

Остальные параметры задания можно задать в стандартном диалоговом окне **Печать** (Print) операционной системы, которое открывается на экране при

выполнении логического метода `printDialog()`. Указанный метод не имеет аргументов. Он возвратит `true`, когда пользователь щелкнет по кнопке **OK**, и `false` после нажатия кнопки **Отмена**.

Остается задать число копий, если оно больше 1, методом `setCopies(int n)` и задание сформировано.

Еще один полезный метод `defaultPage()` класса `PrinterJob` возвращает объект класса `PageFormat` по умолчанию. Этот метод можно использовать вместо конструктора класса `PageFormat`.

Осталось сказать, как создается экземпляр класса `PrinterJob`. Поскольку этот класс тесно связан с системой печати компьютера, его объекты создаются не конструктором, а статическим методом `getPrinterJob()`, имеющимся в том же самом классе `PrinterJob`.

Начало печати задается методом `print()` класса `PrinterJob`. Этот метод не имеет аргументов. Он последовательно вызывает методы `print(g, pf, ind)` классов, рисующих страницы, для каждой страницы.

Соберем все это вместе в листинге 18.8. В нем средствами Java 2D печатается то же, что и в листинге 18.7. Обратите внимание на п. 6. После окончания печати программа не заканчивается автоматически, для ее завершения мы обращаемся к методу `System.exit(0)`.

Листинг 18.8. Простая печать методами Java 2D

```
import java.awt.*;
import java.awt.geom.*;
import java.awt.print.*;

class Print2Test implements Printable{
   public int print(Graphics g, PageFormat pf, int ind)
                                       throws PrinterException{
         // Печатаем не более 5 страниц
      if (ind > 4) return Printable.NO_SUCH_PAGE;
      Graphics2D g2 = (Graphics2D)g;
      g2.setFont(new Font("Serif", Font.ITALIC, 30));
      g2.setColor(Color.black);
      g2.drawString("Page " + (ind + 1), 100, 100);
      g2.draw(new Ellipse2D.Double(100, 100, 200, 200));
      return Printable.PAGE_EXISTS;
   }
   public static void main(String[] args){
         // 1. Создаем экземпляр задания
     PrinterJob pj = PrinterJob.getPrinterJob();
         // 2. Открываем диалоговое окно Параметры страницы
     PageFormat pf = pj.pageDialog(pj.defaultPage());
```

```
         // 3. Задаем вид задания, объект класса, рисующего страницу,
         //    и выбранные параметры страницы
     pj.setPrintable(new Print2Test(), pf);
         // 4. Если нужно напечатать несколько копий, то:
     pj.setCopies(2);              // По умолчанию печатается одна копия
         // 5. Открываем диалоговое окно Печать (необязательно)
     if (pj.printDialog()){        // Если OK...
        try{
           pj.print();             // Обращается к print(g, pf, ind)
        }catch(Exception e){
           System.err.println(e);
        }
     }
         // 6. Завершаем задание
     System.exit(0);
  }
}
```

Печать файла

Печать текстового файла заключается в размещении его строк в графическом контексте методом `drawString()`. При этом необходимо проследить за правильным размещением строк в области печати и разбиением файла на страницы.

В листинге 18.9 приведен упрощенный пример печати текстового файла, имя которого задается в командной строке. Из файла читаются готовые строки, программа не сравнивает их длину с шириной области печати, не выделяет абзацы. Вывод производится в локальной кодировке.

Листинг 18.9. Печать текстового файла

```
import java.awt.*;
import java.awt.print.*;
import java.io.*;

public class Print2File{
   public static void main(String[] args){
      if (args.length < 1){
         System.err.println("Usage: Print2File path");
         System.exit(0);
      }
      PrinterJob pj = PrinterJob.getPrinterJob();
      PageFormat pf = pj.pageDialog(pj.defaultPage());
      pj.setPrintable(new FilePagePainter(args[0]), pf);
```

```
      if (pj.printDialog()){
         try{
             pj.print();
         }catch(PrinterException e){}
      }
      System.exit(0);
   }
}
class FilePagePainter implements Printable{
   private BufferedReader br;
   private String file;
   private int page = -1;
   private boolean eof;
   private String[] line;
   private int numLines;

   public FilePagePainter(String file){
      this.file = file;
      try{
          br = new BufferedReader(new FileReader(file));
      }catch(IOException e){ eof = true; }
   }
   public int print(Graphics g, PageFormat pf, int ind)
                                    throws PrinterException{
      g.setColor(Color.black);
      g.setFont(new Font("Serif", Font.PLAIN, 10));
      int h = (int)pf.getImageableHeight();
      int x = (int)pf.getImageableX() + 10;
      int y = (int)pf.getImageableY() + 12;

      try{
             // Если система печати запросила эту страницу первый раз
          if (ind != page){
             if (eof) return Printable.NO_SUCH_PAGE;
             page = ind;
             line = new String[h/12];      // Массив строк на странице
             numLines = 0;                 // Число строк на странице
                 // Читаем строки из файла и формируем массив строк
             while (y + 48 < pf.getImageableY() + h){
                line[numLines] = br.readLine();
                if (line[numLines] == null){
                   eof = true;
                   break;
                }
                numLines++;
                y += 12;
```

```
            }
        }
            // Размещаем колонтитул
        y = (int)pf.getImageableY() + 12;
        g.drawString("Файл: " + file + ", страница " +
                    (ind + 1), x, y);
            // Оставляем две пустые строки
        y += 36;
            // Размещаем строки текста текущей страницы
        for (int i = 0; i < numLines; i++){
            g.drawString(line[i], x, y);
            y += 12;
        }
        return Printable.PAGE_EXISTS;
      }catch(IOException e){
        return Printable.NO_SUCH_PAGE;
    }
  }
}
```

Печать страниц с разными параметрами

Печать вида Printable Job не совсем удобна — у всех страниц должны быть одинаковые параметры, нельзя задать число страниц и порядок их печати, в окне **Параметры страницы** не видно число страниц, выводимых на печать.

Все эти возможности предоставляет печать вида Pageable Job с помощью класса `Book`.

Как уже говорилось выше, сначала создается пустой объект класса `Book`, затем к нему добавляются разные или одинаковые классы, рисующие страницы. При этом определяются объекты класса `PageFormat`, задающие параметры этих страниц, и число страниц. Если число страниц заранее неизвестно, то вместо него указывается константа `UNKNOWN_NUMBER_OF_PAGES`. В таком случае страницы будут печататься в порядке возрастания их номеров до тех пор, пока метод `print()` не возвратит `NO_SUCH_PAGE`.

Метод

```
setPage(int pageIndex, Printable p, PageFormat pf)
```

заменяет объект в позиции `pageIndex` на новый объект `p`.

В программе листинга 18.10 создаются два класса, рисующие страницы: `Cover` и `Content`. Эти классы очень просты — в них только реализован метод `print()`. Класс `Cover` рисует титульный лист крупным полужирным шрифтом. Текст печатается снизу вверх вдоль длинной стороны листа на его правой половине. Класс `Content` выводит обыкновенный текст обычным образом.

Параметры титульного листа определяются в классе `pf1`, параметры других страниц задаются в диалоговом окне **Параметры страницы** и содержатся в классе `pf2`.

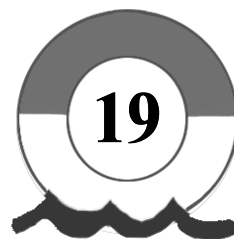
В объект `bk` класса `Book` занесены три страницы: первая страница — титульный лист, на двух других печатается один и тот же текст, записанный в методе `print()` класса `Content`.

Листинг 18.10. Печать страниц с разными параметрами

```
import java.awt.*;
import java.awt.print.*;

public class Print2Book{
   public static void main(String[] args){
      PrinterJob pj = PrinterJob.getPrinterJob();
         // Для титульного листа выбирается альбомная ориентация
      PageFormat pfl = pj.defaultPage();
      pfl.setOrientation(PageFormat.LANDSCAPE);
         // Параметры других страниц задаются в диалоговом окне
      PageFormat pf2 = pj.pageDialog(new PageFormat());

      Book bk = new Book();
         // Первая страница — титульный лист
      bk.append(new Cover(), pfl);
         // Две другие страницы
      bk.append(new Content(), pf2, 2);
         // Определяется вид печати — Pageable Job
      pj.setPageable(bk);

      if (pj.printDialog()){
         try{
            pj.print();
         }catch(Exception e){}
      }
      System.exit(0);
   }
}
class Cover implements Printable{
   public int print(Graphics g, PageFormat pf, int ind)
                                             throws PrinterException{
      g.setFont(new Font("Helvetica-Bold", Font.PLAIN, 40));
      g.setColor(Color.black);
      int y = (int)(pf.getImageableY() +
                        pf.getImageableHeight()/2);
      g.drawString("Это заголовок.", 72, y);
```

```
      g.drawString("Он печатается вдоль длинной", 72, y+60);
      g.drawString("стороны листа бумаги.", 72, y+120);

      return Printable.PAGE_EXISTS;
  }
}
class Content implements Printable{
   public int print(Graphics g, PageFormat pf, int ind)
                                     throws PrinterException{
      Graphics2D g2 = (Graphics2D)g;
      g2.setFont(new Font("Serif", Font.PLAIN, 12));
      g2.setColor(Color.black);

      int x = (int)pf.getImageableX() + 30;
      int y = (int)pf.getImageableY();

      g2.drawString("Это строки обычного текста.",   x, y += 16);
      g2.drawString("Они печатаются с параметрами,", x, y += 16);
      g2.drawString("выбранными в диалоговом окне.", x, y += 16);

      return Printable.PAGE_EXISTS;
   }
}
```

ГЛАВА 19



Сетевые средства Java

Когда число компьютеров в учреждении переваливает за десяток и сотрудникам надоедает бегать с дискетами друг к другу для обмена файлами, тогда в компьютеры вставляются сетевые карты, протягиваются кабели и компьютеры объединяются в сеть. Сначала все компьютеры в сети равноправны, они делают одно и то же — это *одноранговая* (peer-to-peer) сеть. Потом покупается компьютер с большими и быстрыми жесткими дисками, и все файлы учреждения начинают храниться на данных дисках — этот компьютер становится файл-сервером, предоставляющим услуги хранения, поиска, архивирования файлов. Затем покупается дорогой и быстрый принтер. Компьютер, связанный с ним, становится принт-сервером, предоставляющим услуги печати. Потом появляются графический сервер, вычислительный сервер, сервер базы данных. Остальные компьютеры становятся клиентами этих серверов. Такая архитектура сети называется архитектурой *клиент-сервер* (client-server).

Сервер постоянно находится в состоянии ожидания, он *прослушивает* (listen) сеть, ожидая запросов от клиентов. Клиент связывается с сервером и посылает ему *запрос* (request) с описанием услуги, например, имя нужного файла. Сервер обрабатывает запрос и отправляет ответ (response), в нашем примере, файл, или сообщение о невозможности оказать услугу. После этого связь может быть разорвана или продолжиться, организуя сеанс связи, называемый *сессией* (session).

Запросы клиента и ответы сервера формируются по строгим правилам, совокупность которых образует *протокол* (protocol) связи. Всякий протокол должен, прежде всего, содержать правила соединения компьютеров. Клиент перед посылкой запроса должен удостовериться, что сервер в рабочем состоянии, прослушивает сеть, и услышал клиента. Послав запрос, клиент должен быть уверен, что запрос дошел до сервера, сервер понял запрос и готов ответить на него. Сервер обязан убедиться, что ответ дошел до клиен-

та. Окончание сессии должно быть четко зафиксировано, чтобы сервер мог освободить ресурсы, занятые обработкой запросов клиента.

Все правила, образующие протокол, должны быть понятными, однозначными и короткими, чтобы не загружать сеть. Поэтому сообщения, пересылаемые по сети, напоминают шифровки, в них имеет значение каждый бит.

Итак, все сетевые соединения основаны на трех основных понятиях: клиент, сервер и протокол. Клиент и сервер — понятия относительные. В одной сессии компьютер может быть сервером, а в другой — клиентом. Например, файл-сервер может послать принт-серверу файл на печать, становясь его клиентом.

Для обслуживания протокола: формирования запросов и ответов, проверок их соответствия протоколу, расшифровки сообщений, связи с сетевыми устройствами создается программа, состоящая из двух частей. Одна часть программы работает на сервере, другая — на клиенте. Эти части так и называются серверной частью программы и клиентской частью программы, или, короче, сервером и клиентом.

Очень часто клиентская и серверная части программы пишутся отдельно, разными фирмами, поскольку от этих программ требуется только, чтобы они соблюдали протокол. Более того, по каждому протоколу работают десятки клиентов и серверов, отличающихся разными удобствами.

Обычно на одном компьютере-сервере работают несколько программ-серверов. Одна программа занимается электронной почтой, другая — пересылкой файлов, третья предоставляет Web-страницы. Для того чтобы их различать, каждой программе-серверу придается *номер порта* (port). Это просто целое положительное число, которое указывает клиент, обращаясь к определенной программе-серверу. Число, вообще говоря, может быть любым, но наиболее распространенным протоколам даются стандартные номера, чтобы клиенты были твердо уверены, что обращаются к нужному серверу. Так, стандартный номер порта электронной почты 25, пересылки файлов — 21, Web-сервера — 80. Стандартные номера простираются от 0 до 1023. Числа, начиная с 1024 до 65 535, можно использовать для своих собственных номеров портов.

Все это похоже на телевизионные каналы. Клиент-телевизор обращается посредством антенны к серверу-телецентру и выбирает номер канала. Он уверен, что на первом канале ОРТ, на втором — РТР и т. д.

Чтобы равномерно распределить нагрузку на сервер, часто несколько портов прослушиваются программами-серверами одного типа. Web-сервер, кроме порта с номером 80, может прослушивать порт 8080, 8001 и еще какой-нибудь другой.

В процессе передачи сообщения используется несколько протоколов. Даже когда мы отправляем письмо, мы сначала пишем сообщение, начиная его:

"Глубокоуважаемый Иван Петрович!" и заканчивая: "Искренне преданный Вам". Это один протокол. Можно начать письмо словами: "Вася, привет!" и закончить: "Ну, пока". Это другой протокол. Потом мы помещаем письмо в конверт и пишем на нем адрес по протоколу, предложенному Министерством связи. Затем письмо попадает на почту, упаковывается в мешок, на котором пишется адрес по протоколу почтовой связи. Мешок загружается в самолет, который перемещается по своему протоколу. Заметьте, что каждый протокол только добавляет к сообщению свою информацию, не меняя его, ничего не зная о том, что сделано по предыдущему протоколу и что будет сделано по правилам следующего протокола. Это очень удобно — можно программировать один протокол, ничего не зная о других протоколах.

Прежде чем дойти до адресата, письмо проходит обратный путь: вынимается из самолета, затем из мешка, потом из конверта. Поэтому говорят о стеке (stack) протоколов: "Первым пришел, последним ушел".

В современных глобальных сетях принят стек из четырех протоколов, называемый стеком протоколов TCP/IP.

Сначала мы пишем сообщение, пользуясь программой, реализующей *прикладной* (application) протокол: HTTP (80), SMTP (25), TELNET (23), FTP (21), POP3 (100) или другой протокол. В скобках записан стандартный номер порта.

Затем сообщение обрабатывается по *транспортному* (transport) протоколу. К нему добавляются, в частности, номера портов отправителя и получателя, контрольная сумма и длина сообщения. Наиболее распространены транспортные протоколы TCP (Transmission Control Protocol) и UDP (User Datagram Protocol). В результате работы протокола TCP получается *TCP-пакет* (packet), а протокола UDP — *дейтаграмма* (datagram).

Дейтаграмма невелика — всего около килобайта, поэтому сообщение делится на прикладном уровне на части, из которых создаются отдельные дейтаграммы. Дейтаграммы посылаются одна за другой. Они могут идти к получателю разными маршрутами, прийти совсем в другом порядке, некоторые дейтаграммы могут потеряться. Прикладная программа получателя должна сама позаботиться о том, чтобы собрать из полученных дейтаграмм исходное сообщение. Для этого обычно перед посылкой части сообщения нумеруются, как страницы в книге. Таким образом, протокол UDP работает как почтовая служба. Посылая книгу, мы разрезаем ее на страницы, каждую страницу отправляем в своем конверте, и никогда не уверены, что все письма дойдут до адресата.

TCP-пакет тоже невелик, и пересылка также идет отдельными пакетами, но протокол TCP обеспечивает надежную связь. Сначала устанавливается соединение с получателем. Только после этого посылаются пакеты. Получение каждого пакета подтверждается получателем, при ошибке посылка пакета повторяется. Сообщение аккуратно собирается получателем. Для отправите-

ля и получателя создается впечатление, что пересылаются не пакеты, а сплошной поток байтов, поэтому передачу сообщений по протоколу TCP часто называют передачей потоком. Связь по протоколу TCP больше напоминает телефонный разговор, чем почтовую связь.

Далее сообщением занимается программа, реализующая *сетевой* (network) протокол. Чаще всего это протокол IP (Internet Protocol). Он добавляет к сообщению адрес отправителя и адрес получателя, и другие сведения. В результате получается *IP-пакет.*

Наконец, IP-пакет поступает к программе, работающей по *канальному* (link) протоколу ENET, SLIP, PPP, и сообщение принимает вид, пригодный для передачи по сети.

На стороне получателя сообщение проходит через эти четыре уровня протоколов в обратном порядке, освобождаясь от служебной информации, и доходит до программы, реализующей прикладной протокол.

Какой же адрес заносится в IP-пакет? Каждый компьютер или другое устройство, подключенное к объединению сетей Internet, так называемый *хост* (host), получает уникальный номер — четырехбайтовое целое число, называемое *IP-адресом* (IP-address). По традиции содержимое каждого байта записывается десятичным числом от 0 до 255, называемым *октетом* (octet), и эти числа пишутся через точку: 138.2.45.12 или 17.056.215.38.

IP-адрес удобен для машины, но неудобен для человека. Представьте себе рекламный призыв: "Заходите на наш сайт 154.223.145.26!" Поэтому IP-адрес хоста дублируется *доменным именем* (domain name).

В доменном имени присутствует краткое обозначение страны: ru — Россия, su — Советский Союз, ua — Украина, de — ФРГ и т. д., или обозначение типа учреждения: com — коммерческая структура, org — общественная организация, edu — образовательное учреждение. Далее указывается регион: msc.ru — Москва, spb.ru — Санкт-Петербург, kcn.ru — Казань, или учреждение: bhv.ru — "БХВ-Петербург", ksu.ru — Казанский госуниверситет, sun.com — SUN Microsystems. Потом подразделение: www.bhv.ru, java.sun.com. Такую цепочку кратких обозначений можно продолжать и дальше.

В Java IP-адрес и доменное имя объединяются в один класс `InetAddress` пакета `java.net`. Экземпляр этого класса создается статическим методом `getByName(String host)` данного же класса, в котором аргумент `host` — это доменное имя или IP-адрес.

Работа в WWW

Среди программного обеспечения Internet большое распространение получила информационная система WWW (World Wide Web), основанная на прикладном протоколе HTTP (Hypertext Transfer Protocol). В ней использу-

ется расширенная адресация, называемая URL (Uniform Resource Locator). Эта адресация имеет такие схемы:

protocol://authority@host:port/path/file#ref

protocol://authority@host:port/path/file/extra_path?info

Здесь необязательная часть authority — это пара имя:пароль для доступа к хосту, host — это IP-адрес или доменное имя хоста. Например:

http://www.bhv.ru/

http://132.192.5.10:8080/public/some.html

ftp://guest:password@lenta.ru/users/local/pub

file:///C:/text/html/index.htm

Если какой-то элемент URL отсутствует, то берется стандартное значение. Например, в первом примере номер порта port равен 80, а имя файла path — какой-то головной файл, определяемый хостом, чаще всего это файл с именем index.html. В третьем примере номер порта равен 21. В последнем примере в форме URL просто записано имя файла index.htm, расположенного на разделе C: жесткого диска той же самой машины.

В Java для работы с URL есть класс `URL` пакета `java.net`. Объект этого класса создается одним из шести конструкторов. В основном конструкторе

```
URL(String url)
```

задается расширенный адрес `url` в виде строки. Кроме методов доступа `get`*`Xxx`*`()`, позволяющих получить элементы URL, в этом классе есть два интересных метода:

- `openConnection()` — определяет связь с URL и возвращает объект класса `URLConnection`;
- `openStream()` — устанавливает связь с URL и открывает входной поток в виде возвращаемого объекта класса `InputStream`.

Листинг 19.1 показывает, как легко можно получить файл из Internet, пользуясь методом `openStream()`.

Листинг 19.1. Получение Web-страницы

```
import java.net.*;
import java.io.*;

class SimpleURL{
  public static void main(String[] args){
     try{
        URL bhv = new URL("http://www.bhv.ru/");
        BufferedReader br = new BufferedReader(
          new InputStreamReader(bhv.openStream()));
```

```
        String line;
        while ((line = br.readLine()) != null)
            System.out.println(line);
        br.close();
    }catch(MalformedURLException me){
        System.err.println("Unknown host: " + me);
        System.exit(0);
    }catch(IOException ioe){
        System.err.println("Input error: " + ioe);
    }
  }
}
```

Если вам надо не только получить информацию с хоста, но и узнать ее тип: текст, гипертекст, архивный файл, изображение, звук, или выяснить длину файла, или передать информацию на хост, то необходимо сначала методом `openConnection()` создать объект класса `URLConnection` или его подкласса `HttpURLConnection`.

После создания объекта соединение еще не установлено, и можно задать параметры связи. Это делается следующими методами:

- `setDoOutput(boolean out)` — если аргумент `out` равен `true`, то передача пойдет от клиента на хост; значение по умолчанию `false`;
- `setDoInput(boolean in)` — если аргумент `in` равен `true`, то передача пойдет с хоста к клиенту; значение по умолчанию `true`, но если уже выполнено `setDoOutput(true)`, то значение по умолчанию равно `false`;
- `setUseCaches(boolean cache)` — если аргумент `cache` равен `false`, то передача пойдет без кэширования, если `true`, то принимается режим по умолчанию;
- `setDefaultUseCaches(boolean default)` — если аргумент `default` равен `true`, то принимается режим кэширования, предусмотренный протоколом;
- `setRequestProperty(String name, String value)` — добавляет параметр `name` со значением `value` к заголовку посылаемого сообщения.

После задания параметров нужно установить соединение методом `connect()`. После соединения задание параметров уже невозможно. Следует учесть, что некоторые методы доступа `get`*`Xxx`*`()`, которым надо получить свои значения с хоста, автоматически устанавливают соединение, и обращение к методу `connect()` становится излишним.

Web-сервер возвращает информацию, запрошенную клиентом, вместе с заголовком, сведения из которого можно получить методами `get`*`Xxx`*`()`, например:

- `getContentType()` — возвращает строку типа `String`, показывающую тип пересланной информации, например, "text/html", или `null`, если сервер его не указал;

- `getContentLength()` — возвращает длину полученной информации в байтах или −1, если сервер ее не указал;
- `getContent()` — возвращает полученную информацию в виде объекта типа `Object`;
- `getContentEncoding()` — возвращает строку типа `String` с кодировкой полученной информации, или `null`, если сервер ее не указал.

Два метода возвращают потоки ввода/вывода, созданные для данного соединения:

- `getInputStream()` — возвращает входной поток типа `InputStream`;
- `getOutputStream()` — возвращает выходной поток типа `OutputStream`.

Прочие методы, а их около двадцати, возвращают различные параметры соединения.

Обращение к методу `bhv.openStream()`, записанное в листинге 19.1, — это, на самом деле, сокращение записи

```
bhv.openConnection().getInputStream()
```

В листинге 19.2 показано, как переслать строку текста по адресу URL.

Web-сервер, который получает эту строку, не знает, что делать с полученной информацией. Занести ее в файл? Но с каким именем, и есть ли у него право создавать файлы? Переслать на другую машину? Но куда?

Выход был найден в системе CGI (Common Gateway Interface), которая вкратце действует следующим образом. При посылке сообщения мы указываем URL исполнимого файла некоторой программы, размещенной на машине-сервере. Получив сообщение, Web-сервер запускает эту программу и передает сообщение на ее стандартный ввод. Вот программа-то и знает, что делать с полученным сообщением. Она обрабатывает сообщение и выводит результат обработки на свой стандартный вывод. Web-сервер подключается к стандартному выводу, принимает результат и отправляет его обратно клиенту.

CGI-программу можно написать на любом языке: C, C++, Pascal, Perl, PHP, лишь бы у нее был стандартный ввод и стандартный вывод. Можно написать ее и на Java, но в технологии Java есть более изящное решение этой задачи с помощью сервлетов (servlets). CGI-программы обычно лежат на сервере в каталоге cgi-bin.

Листинг 19.2. Посылка строки по адресу URL

```
import java.net.*;
import java.io.*;

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

```
    String req = "This text is posting to URL";
    try{
            // Указываем URL нужной CGI-программы
       URL url = new URL("http://www.bhv.ru/cgi-bin/some.pl");
            // Создаем объект uc
       URLConnection uc = url.openConnection();
            // Собираемся отправлять
       uc.setDoOutput(true);
            // и получать сообщения
       uc.setDoInput(true);
            // без кэширования
       uc.setUseCaches(false);
            // Задаем тип
       uc.setRequestProperty("content-type",
                  "application/octet-stream");
            // и длину сообщения
       uc.setRequestProperty("content-length", "" + req.length());
            // Устанавливаем соединение
       uc.connect();
            // Открываем выходной поток
       DataOutputStream dos = new DataOutputStream(
           uc.getOutputStream());
            // и выводим в него сообщение, посылая его на адрес URL
       dos.writeBytes(req);
            // Закрываем выходной поток
       dos.close();
            // Открываем входной поток для ответа сервера
       BufferedReader br = new BufferedReader(new InputStreamReader(
           uc.getInputStream()));
       String res = null;
            // Читаем ответ сервера и выводим его на консоль
       while ((res = br.readLine()) != null)
          System.out.println(res);
       br.close();
    }catch(MalformedURLException me){
       System.err.println(me);
    }catch(UnknownHostException he){
       System.err.println(he);
    }catch(UnknownServiceException se){
       System.err.println(se);
    }catch(IOException ioe){
       System.err.println(ioe);
    }
  }
}
```

Работа по протоколу TCP

Программы-серверы, прослушивающие свои порты, работают под управлением операционной системы. У машин-серверов могут быть самые разные операционные системы, особенности которых передаются программам-серверам.

Чтобы сгладить различия в реализациях разных серверов, между сервером и портом введен промежуточный программный слой, названный *сокетом* (socket). Английское слово socket переводится как электрический разъем, розетка. Так же как к розетке при помощи вилки можно подключить любой электрический прибор, лишь бы он был рассчитан на 220 В и 50 Гц, к сокету можно присоединить любой клиент, лишь бы он работал по тому же протоколу, что и сервер. Каждый сокет связан (bind) с одним портом, говорят, что сокет прослушивает (listen) порт. Соединение с помощью сокетов устанавливается так.

1. Сервер создает сокет, прослушивающий порт сервера.
2. Клиент тоже создает сокет, через который связывается с сервером, сервер начинает устанавливать (accept) связь с клиентом.
3. Устанавливая связь, сервер создает новый сокет, прослушивающий порт с другим, новым номером, и сообщает этот номер клиенту.
4. Клиент посылает запрос на сервер через порт с новым номером.

После этого соединение становится совершенно симметричным — два сокета обмениваются информацией, а сервер через старый сокет продолжает прослушивать прежний порт, ожидая следующего клиента.

В Java сокет — это объект класса `Socket` из пакета `java.io`. В классе шесть конструкторов, в которые разными способами заносится адрес хоста и номер порта. Чаще всего применяется конструктор

```
Socket(String host, int port)
```

Многочисленные методы доступа устанавливают и получают параметры сокета. Мы не будем углубляться в их изучение. Нам понадобятся только методы, создающие потоки ввода/вывода:

- `getInputStream()` — возвращает входной поток типа `InputStream`;
- `getOutputStream()` — возвращает выходной поток типа `OutputStream`.

Приведем пример получения файла с сервера по максимально упрощенному протоколу HTTP.

1. Клиент посылает серверу запрос на получение файла строкой "POST filename HTTP/1.1\n\n", где filename — строка с путем к файлу на сервере.
2. Сервер анализирует строку, отыскивает файл с именем filename и возвращает его клиенту. Если имя файла filename заканчивается наклонной

чертой /, то сервер понимает его как имя каталога и возвращает файл index.html, находящийся в этом каталоге.

3. Перед содержимым файла сервер посылает строку вида "HTTP/1.1 code OK\n\n", где code — это код ответа, одно из чисел: 200 — запрос удовлетворен, файл посылается; 400 — запрос не понят; 404 — файл не найден.
4. Сервер закрывает сокет и продолжает слушать порт, ожидая следующего запроса.
5. Клиент выводит содержимое полученного файла в стандартный вывод `System.out` или выводит код сообщения сервера в стандартный вывод сообщений `System.err`.
6. Клиент закрывает сокет, завершая связь.

Этот протокол реализуется в клиентской программе листинга 19.3 и серверной программе листинга 19.4.

Листинг 19.3. Упрощенный HTTP-клиент

```
import java.net.*;
import java.io.*;
import java.util.*;

class Client{
  public static void main(String[] args){
     if (args.length != 3){
       System.err.println("Usage: Client host port file");
       System.exit(0);
     }
     String host = args[0];
     int    port = Integer.parseInt(args[1]);
     String file = args[2];
     try{
        Socket sock = new Socket(host, port);
        PrintWriter pw = new PrintWriter(new OutputStreamWriter(
           sock.getOutputStream()), true);
        pw.println("POST " + file + " HTTP/1.1\n");
        BufferedReader br = new BufferedReader(new InputStreamReader(
           sock.getInputStream()));
        String line = null;
        line = br.readLine();
        StringTokenizer st = new StringTokenizer(line);
        String code = null;
        if ((st.countTokens() >= 2) && st.nextToken().equals("POST")){
           if ((code = st.nextToken()) != "200"){
              System.err.println("File not found, code = " + code);
```

```
                System.exit(0);
            }
        }
        while ((line = br.readLine()) != null)
            System.out.println(line);
        sock.close();
    }catch(Exception e){
        System.err.println(e);
    }
  }
}
```

Закрытие потоков ввода/вывода вызывает закрытие сокета. Обратно, закрытие сокета закрывает и потоки.

Для создания сервера в пакете `java.net` есть класс `ServerSocket`. В конструкторе этого класса указывается номер порта

```
ServerSocket(int port)
```

Основной метод этого класса `accept()` ожидает поступления запроса. Когда запрос получен, метод устанавливает соединение с клиентом и возвращает объект класса `Socket`, через который сервер будет обмениваться информацией с клиентом.

Листинг 19.4. Упрощенный HTTP-сервер

```
import java.net.*;
import java.io.*;
import java.util.*;

class Server{
  public static void main(String[] args){
     try{
        ServerSocket ss = new ServerSocket(Integer.parseInt(args[0]));
        while (true)
           new HttpConnect(ss.accept());
     }catch(ArrayIndexOutOfBoundsException ae){
        System.err.println("Usage: Server port");
        System.exit(0);
     }catch(IOException e){
        System.out.println(e);
     }
  }
}
class HttpConnect extends Thread{
    private Socket sock;
```

```
    HttpConnect(Socket s){
      sock = s;
      setPriority(NORM_PRIORITY - 1);
      start();
    }
    public void run(){
       try{
          PrintWriter pw = new PrintWriter(new OutputStreamWriter(
             sock.getOutputStream()), true);
          BufferedReader br = new BufferedReader(new InputStreamReader(
             sock.getInputStream()));
          String req = br.readLine();
          System.out.println("Request: " + req);
          StringTokenizer st = new StringTokenizer(req);
          if ((st.countTokens() >= 2) && st.nextToken().equals("POST")){
             if ((req = st.nextToken()).endsWith("/") || req.equals(""))
                req += "index.html";
             try{
                File f = new File(req);
                BufferedReader bfr =
                          new BufferedReader(new FileReader(f));
                char[] data = new char[(int)f.length()];
                bfr.read(data);
                pw.println("HTTP/1.1 200 OK\n");
                pw.write(data);
                pw.flush();
             }catch(FileNotFoundException fe){
                pw.println("HTTP/1.1 404 Not Found\n");
             }catch(IOException ioe){
                System.out.println(ioe);
             }
          }else pw.println("HTTP/1.1 400 Bad Request\n");
          sock.close();
       }catch(IOException e){
          System.out.println(e);
       }
   }
}
```

Вначале следует запустить сервер, указав номер порта, например:

```
java Server 8080
```

Затем надо запустить клиент, указав IP-адрес или доменное имя хоста, номер порта и имя файла:

```
java Client localhost 8080 Server.java
```

Сервер отыскивает файл Server.java в своем текущем каталоге и посылает его клиенту. Клиент выводит содержимое этого класса в стандартный вывод и завершает работу. Сервер продолжает работать, ожидая следующего запроса.

Замечание по отладке

Программы, реализующие стек протоколов TCP/IP, всегда создают так называемую "петлю" с адресом 127.0.0.1 и доменным именем `localhost`. Это адрес самого компьютера. Он используется для отладки приложений клиент-сервер. Вы можете запускать клиент и сервер на одной машине, пользуясь этим адресом.

Работа по протоколу UDP

Для посылки дейтаграмм отправитель и получатель создают сокеты дейтаграммного типа. В Java их представляет класс `DatagramSocket`. В классе три конструктора:

- `DatagramSocket()` — создаваемый сокет присоединяется к любому свободному порту на локальной машине;
- `DatagramSocket(int port)` — создаваемый сокет присоединяется к порту `port` на локальной машине;
- `DatagramSocket(int port, InetAddress addr)` — создаваемый сокет присоединяется к порту `port`; аргумент `addr` — один из адресов локальной машины.

Класс содержит массу методов доступа к параметрам сокета и, кроме того, методы отправки и приема дейтаграмм:

- `send(DatagramPacket pack)` — отправляет дейтаграмму, упакованную в пакет `pack`;
- `receive(DatagramPacket pack)` — дожидается получения дейтаграммы и заносит ее в пакет `pack`.

При обмене дейтаграммами соединение обычно не устанавливается, дейтаграммы посылаются наудачу, в расчете на то, что получатель ожидает их. Но можно установить соединение методом

```
connect(InetAddress addr, int port)
```

При этом устанавливается только одностороннее соединение с хостом по адресу `addr` и номером порта `port` — или на отправку или на прием дейтаграмм. Потом соединение можно разорвать методом

```
disconnect()
```

При посылке дейтаграммы по протоколу UDP сначала создается сообщение в виде массива байтов, например,

```
String mes = "This is the sending message.";
byte[] data = mes.getBytes();
```

Потом записывается адрес — объект класса `InetAddress`, например:

```
InetAddress addr = InetAddress.getByName(host);
```

Затем сообщение упаковывается в пакет — объект класса `DatagramPacket`. При этом указывается массив данных, его длина, адрес и номер порта:

```
DatagramPacket pack = new DatagramPacket(data, data.length, addr, port)
```

Далее создается дейтаграммный сокет

```
DatagramSocket ds = new DatagramSocket()
```

и дейтаграмма отправляется

```
ds.send(pack)
```

После посылки всех дейтаграмм сокет закрывается, не дожидаясь какой-либо реакции со стороны получателя:

```
ds.close()
```

Прием и распаковка дейтаграмм производится в обратном порядке, вместо метода `send()` применяется метод `receive(DatagramPacket pack)`.

В листинге 19.5 показан пример класса `Sender`, посылающего сообщения, набираемые в командной строке, на localhost, порт номер 1050. Класс `Recipient`, описанный в листинге 19.6, принимает эти сообщения и выводит их в свой стандартный вывод.

Листинг 19.5. Посылка дейтаграмм по протоколу UDP

```
import java.net.*;
import java.io.*;

class Sender{
   private String host;
   private int port;
   Sender(String host, int port){
      this.host = host;
      this.port = port;
   }
   private void sendMessage(String mes){
      try{
         byte[] data = mes.getBytes();
         InetAddress addr = InetAddress.getByName(host);
         DatagramPacket pack =
            new DatagramPacket(data, data.length, addr, port);
         DatagramSocket ds = new DatagramSocket();
         ds.send(pack);
         ds.close();
```

```
      }catch(IOException e){
         System.err.println(e);
      }
   }
   public static void main(String[] args){
     Sender sndr = new Sender("localhost", 1050);
     for (int k = 0; k < args.length; k++)
        sndr.sendMessage(args[k]);
   }
}
```

Листинг 19.6. Прием дейтаграмм по протоколу UDP

```
import java.net.*;
import java.io.*;

class Recipient{
  public static void main(String[] args){
     try{
        DatagramSocket ds = new DatagramSocket(1050);
        while (true){
           DatagramPacket pack =
                new DatagramPacket(new byte[1024], 1024);
           ds.receive(pack);
           System.out.println(new String(pack.getData()));
        }
     }catch(Exception e){
        System.out.println(e);
     }
  }
}
```

ПРИЛОЖЕНИЕ

Развитие Java

В приложении мы вкратце перечислим аспекты технологии Java, не освещенные в основном тексте книги.

Переход к Swing

В *части 3* мы подробно рассмотрели возможности графической библиотеки AWT. Там же мы заметили, что в состав Java 2 SDK входит еще одна графическая библиотека, Swing, с более широкими возможностями, чем AWT. Фирма SUN настоятельно рекомендует использовать Swing, а не AWT, но, во-первых, Swing требует больше ресурсов, что существенно для российского разработчика, во-вторых, большинство браузеров не имеет в своем составе Swing. В-третьих, удобнее сначала познакомиться с библиотекой AWT, а уже потом изучать Swing.

Все примеры графических программ, приведенные в книге, будут выполняться методами библиотеки Swing после небольшой переделки:

1. Добавьте в заголовок строку `import javax.swing.*;`.
2. Поменяйте `Frame` на `JFrame`, `Applet` на `JApplet`, `Component` на `JComponent`, `Panel` на `JPanel`. Не расширяйте свои классы от класса `Canvas`, используйте `JPanel` или другие контейнеры Swing.
3. Замените компоненты AWT на близкие к ним компоненты Swing. Чаще всего надо просто приписать букву `J`: `JButton`, `JCheckBox`, `JDialog`, `JList`, `JMenu` и т. д. Закомментируйте временно строку `import java.awt.*;` и попробуйте откомпилировать программу. Компилятор покажет, какие компоненты требуют замены.
4. Включите в конструктор класса, расширяющего `JFrame`, строку `Container c = getContentPane();` и располагайте все компоненты в контейнере `c`, т. е. пишите `c.add()`, `c.setLayout()`.

5. Класс `JFrame` содержит средства закрытия своего окна, надо только настроить их. Вы можете убрать `addWindowListener(...)` и включить в конструктор обращение к методу `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`.
6. В прямых подклассах класса `JPanel` замените метод `paint()` на `paintComponent()` и удалите метод `update()`. Класс `JPanel` автоматически производит двойную буферизацию и надобности в методе `update()` больше нет. Уберите весь код двойной буферизации. В начало метода `paintComponent()` включите обращение `super.paintComponent(g)`. Из подклассов классов `JFrame`, `JDialog`, `JApplet` метод `paintComponent()` надо переместить в другие компоненты, например, `JButton`, `JLabel`, `JPanel`.
7. Используйте вместо класса `Image` класс `ImageIcon`. Конструкторы этого класса выполнят необходимое преобразование. Класс `ImageIcon` автоматически применяет методы класса `MediaTracker` для ожидания окончания загрузки.
8. При создании апплетов расширением класса `JApplet` не забывайте, что в классе `Applet` менеджером размещения по умолчанию служит класс `FlowLayout`, а в классе `JApplet` менеджер размещения по умолчанию `BorderLayout`.

Пункты 4 и 6 требуют пояснения. Окно верхнего уровня в Swing, такое как `JFrame`, содержит *корневую панель* (root pane), на которой размещена *слоеная панель* (layered pane). Компоненты на слоеной панели можно размещать несколькими слоями, перекрывая друг друга. В одном из слоев находится *панель содержимого* (content pane) и *строка меню* (menu bar). Поверх самого верхнего слоя расположена *прозрачная панель* (glass pane). Поэтому нельзя просто поместить компонент в окно верхнего уровня. Его надо "положить" на какую-нибудь панель. Пункт 4 рекомендует размещать компоненты на панели содержимого.

Откомпилировав и запустив измененную программу, вы увидите, что ее внешний вид изменился, чаще всего не в лучшую сторону. Теперь надо настроить компоненты Swing. Библиотека Swing предоставляет для этого широчайшие возможности.

Архиватор *jar*

Для упаковки нескольких файлов в один архивный файл, со сжатием или без сжатия, в технологии Java разработан формат JAR. Имя архивного jar-файла может быть любым, но обычно оно получает расширение jar. Способ упаковки и сжатия основан на методе ZIP. Название JAR (Java ARchive) перекликается с названием известной утилиты TAR (Tape ARchive), разработанной в UNIX.

Отличие jar-файлов от zip-файлов только в том, что в первые автоматически включается каталог META-INF, содержащий несколько файлов с информацией об упакованных в архив файлах.

Архивные файлы очень удобно использовать в апплетах, поскольку весь архив загружается по сети сразу же, одним запросом. Все файлы апплета с байт-кодами, изображениями, звуковые файлы упаковываются в один или несколько архивов. Для их загрузки достаточно в теге `<applet>` указать имена архивов в параметре `archive`, например:

```
<applet code = "MillAnim.class" archive = "first.jar, second.jar"
      width = "100%" height = "100%"></applet>
```

Основной файл MillAnim.class должен находиться в каком-либо из архивных файлов first.jar или second.jar. Остальные файлы отыскиваются в архивных файлах, а если не найдены там, то на сервере, в том же каталоге, что и HTML-файл. Впрочем, файлы апплета можно упаковать и в zip-архив, со сжатием или без сжатия.

Архивные файлы удобно использовать и в приложениях (applications). Все файлы приложения упаковываются в архив, например, appl.jar. Приложение выполняется прямо из архива, интерпретатор запускается с параметром `-jar`, например:

```
java -jar appl.jar
```

Имя основного класса приложения, содержащего метод `main()`, указывается в файле MANIFEST.MF, речь о котором пойдет чуть ниже.

Архивные файлы удобны и просты для компактного хранения всей необходимой для работы программы информации. С файлами архива можно работать прямо из архива, не распаковывая их, с помощью классов пакета `java.util.jar`.

Создание архива

Jar-архивы создаются с помощью классов пакета `java.util.jar` или с помощью утилиты командной строки `jar`.

Правила применения утилиты `jar` очень похожи на правила применения утилиты `tar`. Набрав в командной строке слово `jar` и нажав клавишу <Enter>, вы получите краткое пояснение, показанное на рис. П.1.

В строке

```
jar {ctxu}[vfm0M] [jar-file] [manifest-file] [-C dir] files...
```

зашифрованы правила применения утилиты. Фигурные скобки показывают, что после слова `jar` и пробела надо написать одну из букв `c`, `t`, `x` или `u`. Эти буквы означают следующие операции:

- `c` (create) — создать новый архив;
- `t` (table of contents) — вывести в стандартный вывод список содержимого архива;
- `x` (extract) — извлечь из архива один или несколько файлов;
- `u` (update) — обновить архив, заменив или добавив один или несколько файлов.

```
Command Prompt
D:\jdk1.3\MyProgs>jar
Usage: jar {ctxu}[vfm0M] [jar-file] [manifest-file] [-C dir] files ...
Options:
    -c  create new archive
    -t  list table of contents for archive
    -x  extract named (or all) files from archive
    -u  update existing archive
    -v  generate verbose output on standard output
    -f  specify archive file name
    -m  include manifest information from specified manifest file
    -0  store only; use no ZIP compression
    -M  do not create a manifest file for the entries
    -i  generate index information for the specified jar files
    -C  change to the specified directory and include the following file
If any file is a directory then it is processed recursively.
The manifest file name and the archive file name needs to be specified
in the same order the 'm' and 'f' flags are specified.

Example 1: to archive two class files into an archive called classes.jar:
       jar cvf classes.jar Foo.class Bar.class
Example 2: use an existing manifest file 'mymanifest' and archive all the
           files in the foo/ directory into 'classes.jar':
       jar cvfm classes.jar mymanifest -C foo/ .

D:\jdk1.3\MyProgs>_
```

Рис. П.1. Правила употребления утилиты `jar`

После буквы, без пробела, можно написать одну или несколько букв, перечисленных в квадратных скобках. Они означают следующее:

- `v` (verbose) — выводить сообщения о процессе работы с архивом в стандартный вывод;
- `f` (file) — записанный далее параметр `jar-file` показывает имя архивного файла;
- `m` (manifest) — записанный далее параметр `manifest-file` показывает имя файла описания;
- `0` (нуль) — не сжимать файлы, записывая их в архив;
- `M` (manifest) — не создавать файл описания;

Параметр `-i` (index) предписывает создать в архиве файл INDEX.LIST. Он используется уже после формирования архивного файла.

После буквенных параметров-файлов через пробел записывается имя архивного файла `jar-file`, потом, через пробел, имя файла описания `manifest-file`, затем перечисляются имена файлов, которые надо занести в архив или

извлечь из архива. Если это имена каталогов, то операция выполняется рекурсивно со всеми файлами каталога.

Перед первым именем каталога может стоять параметр `-C`. Конструкция `-C dir` означает, что на время выполнения утилиты `jar` текущим каталогом станет каталог `dir`.

Необязательные параметры занесены в квадратные скобки.

Итак, в конце командной строки должно быть записано хотя бы одно имя файла или каталога. Если среди параметров есть буква `f`, то первый из этих файлов понимается как архивный jar-файл. Если среди параметров находится буква `m`, то первый файл понимается как файл описания (manifest-file). Если среди параметров присутствуют обе буквы, то имя архивного файла и имя файла описания должны идти в том же порядке, что и буквы `f` и `m`.

Если параметр `f` и имя архивного файла отсутствуют, то архивным файлом будет служить стандартный вывод.

```
Command Prompt
D:\jdk1.3\MyProgs\ch3>dir
 Volume in drive D has no label.
 Volume Serial Number is AC95-B8D2

 Directory of D:\jdk1.3\MyProgs\ch3

17.01.2001  17:12       <DIR>          .
17.01.2001  17:12       <DIR>          ..
17.01.2001  17:12                  329 Base.class
18.10.2000  16:44                  604 Base.java
18.10.2000  17:08       <DIR>          classes
17.01.2001  17:12                  348 Derivedp1.class
17.01.2001  17:12                  340 Inp1.class
18.10.2000  16:48                  911 Inp2.java
               5 File(s)          2 532 bytes
               3 Dir(s)   3 413 966 848 bytes free

D:\jdk1.3\MyProgs\ch3>jar cf Base.jar classes Base.class

D:\jdk1.3\MyProgs\ch3>dir
 Volume in drive D has no label.
 Volume Serial Number is AC95-B8D2

 Directory of D:\jdk1.3\MyProgs\ch3

17.01.2001  17:13       <DIR>          .
17.01.2001  17:13       <DIR>          ..
17.01.2001  17:12                  329 Base.class
17.01.2001  17:13                3 318 Base.jar
18.10.2000  16:44                  604 Base.java
18.10.2000  17:08       <DIR>          classes
17.01.2001  17:12                  348 Derivedp1.class
17.01.2001  17:12                  340 Inp1.class
18.10.2000  16:48                  911 Inp2.java
               6 File(s)          5 850 bytes
               3 Dir(s)   3 413 962 752 bytes free

D:\jdk1.3\MyProgs\ch3>jar tf Base.jar
META-INF/
META-INF/MANIFEST.MF
classes/
classes/p1/
classes/p1/Base.class
classes/p1/Derivedp1.class
classes/p1/Inp1.class
classes/p2/
classes/p2/Derivedp2.class
classes/p2/Inp2.class
classes/p2/Inp2.java
Base.class

D:\jdk1.3\MyProgs\ch3>_
```

Рис. П.2. Работа с утилитой `jar`

Если параметр m и имя файла описания отсутствуют, то по умолчанию файл MANIFEST.MF, лежащий в каталоге META-INF архивного файла, будет содержать только номер версии.

На рис. П.2 показан процесс создания архива Base.jar в каталоге ch3.

Сначала показано содержимое каталога ch3. Затем создается архив, в который включается файл Base.class и все содержимое подкаталога classes. Снова выводится содержимое каталога ch3. В нем появляется файл Base.jar. Потом выводится содержимое архива.

Как видите, в архиве создан каталог META-INF, а в нем файл MANIFEST.MF.

Файл описания MANIFEST.MF

Файл MANIFEST.MF, расположенный в каталоге META-INF архивного файла, предназначен для нескольких целей:

- перечисления файлов из архива, снабженных цифровой подписью;
- перечисления компонентов JavaBeans, расположенных в архиве;
- указания имени основного класса для выполнения приложения из архива;
- записи сведений о версии пакета.

Вся информация сначала записывается в обычном текстовом файле с любым именем, например, manif. Потом запускается утилита jar, в которой этот файл указывается как значение параметра m, например:

```
jar cmf manif Base.jar classes Base.class
```

Утилита проверяет правильность записей в файле manif и переносит их в файл MANIFEST.MF, добавляя свои записи.

Файл описания manif должен быть написан по строгим правилам, изложенным в спецификации JAR File Specification. Ее можно найти в документации Java 2 SDK, в файле docs\guide\jar\jar.html.

Например, если мы хотим выполнять приложение с главным файлом Base.class из архива Base.jar, то файл manif должен содержать как минимум две строки:

```
Main-Class: Base

```

Первая строка содержит относительный путь к главному классу, но не к файлу, т. е. без расширения class. В этой строке каждый символ имеет значение, даже пробел. Вторая строка пустая — файл обязательно должен заканчиваться пустой строкой, точнее говоря, символом перевода строки '\n'.

После того как создан архив Base.jar, можно выполнять приложение прямо из него:

```
java -jar Base.jar
```

Файл INDEX.LIST

Для ускорения поиска файлов и более быстрой их загрузки можно создать файл поиска INDEX.LIST. Это делается после создания архива. Утилита `jar` запускается еще раз с параметром `-i`, например:

```
jar -i Base.jar
```

После этого в каталоге META-INF архива появляется файл INDEX.LIST. На рис. П.3 представлено, как создается файл поиска и как выглядит содержимое архива после его создания.

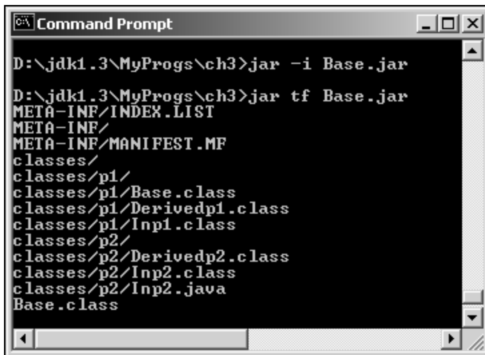


Рис. П.3. Создание файла поиска

Компоненты JavaBeans

Многие программисты предпочитают разрабатывать приложения с графическим интерфейсом пользователя с помощью визуальных средств разработки: JBuilder, Visual Age for Java, Visual Cafe и др. Эти средства позволяют помещать компоненты в контейнер графически, с помощью мыши. На рис. П.4 показано окно JBuilder 4.

Левый нижний угол окна занимает форма, на которой размещаются компоненты. Сами компоненты показаны ярлыками на панели компонентов, расположенной выше формы. На рис. П.4 на панели компонентов виден ярлык компонента `Button`, показанный прямоугольником с надписью **OK**, ярлык компонента `Checkbox`, показанный квадратиком с крестиком, ярлык компонента `CheckboxGroup`, обозначенный группой радиокнопок. Далее видны ярлыки компонентов `Choice`, `Label`, `List` и другие компоненты AWT.

Чтобы поместить компонент в форму, надо щелкнуть кнопкой мыши на ярлыке компонента, перенести курсор мыши в нужное место формы и щелкнуть кнопкой мыши еще раз.

Затем с помощью мыши необходимо установить нужные размеры компонента. При этом можно передвинуть компонент на другое место. На рис. П.4 в форму помещена кнопка типа `Button`.

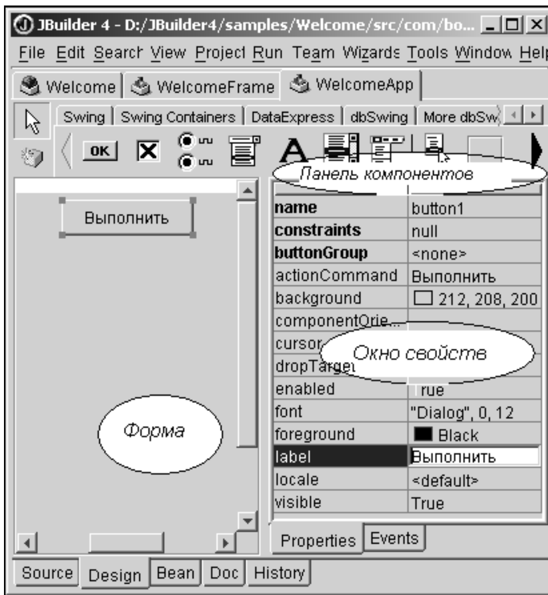


Рис. П.4. Окно JBuilder 4

Далее следует определить свойства (properties) компонента: текст, цвет текста и фона, вид курсора мыши, когда он появляется над компонентом. Свойства определяются в окне свойств, расположенном справа от формы. В левой колонке окна перечислены имена свойств, в правую колонку надо записать их значения. На рис. П.4 свойству **label** — надписи на кнопке — дано значение `Выполнить`. Это значение тут же появляется на кнопке.

Потом можно задать обработку событий, открыв вторую страницу окна свойств.

Для того чтобы компонент можно было применять в таком визуальном средстве разработки как JBuilder, он должен обладать дополнительными качествами. У него должен быть ярлык, помещаемый на панель компонентов. Среди полей компонента должны быть выделены свойства (properties), которые будут показаны в окне свойств. Следует определить методы доступа `get`*`Xxx`*`()`/`set`*`Xxx`*`()`/`is`*`Xxx`*`()` к каждому свойству.

Компонент, снабженный этими и другими необходимыми качествами, в технологии Java называется компонентом JavaBean. В него может входить один или несколько классов. Как правило, файлы этих классов упаковываются в jar-архив и отмечаются в файле MANIFEST.MF как `Java-Bean: True`.

Все компоненты AWT и Swing являются компонентами JavaBeans. Если вы создаете свой графический компонент по правилам, изложенным в *части 3*, то вы тоже получаете свой JavaBean. Но для того чтобы не упустить каких-либо важных качеств JavaBean, лучше использовать для их разработки специальные средства.

Фирма SUN поставляет набор необходимых утилит и классов BDK (Beans Development Kit) для разработки JavaBeans. Так же, как и SDK, этот набор хранится на сайте фирмы в виде самораспаковывающегося архива, например, bdk1_1-win.exe. Его содержимое распаковывается в один каталог, например, D:\bdk1.1. После перехода в каталог bdk1.1\beanbox\ и запуска файла run.bat (или run.sh в UNIX) открываются несколько окон утилиты BeanBox, работа с которой ведется по тому же принципу, что и в визуальных средствах разработки.

Правила оформления компонентов JavaBeans изложены в спецификации JavaBeans API Specification, которую можно найти по адресу **http://java.sun.com/products/javabeans/docs/spec.html**.

Визуальные средства разработки — это не основное применение JavaBeans. Главное достоинство компонентов, оформленных как JavaBeans, в том, что они без труда встраиваются в любое приложение. Более того, приложение можно собрать из готовых JavaBeans как из строительных блоков, остается только настроить их свойства.

Специалисты пророчат большое будущее компонентному программированию. Они считают, что скоро будут созданы тысячи компонентов JavaBeans на все случаи жизни и программирование сведется к поиску в Internet нужных компонентов и сборке из них приложения.

Связь с базами данных через JDBC

Большинство информации хранится не в файлах, а в базах данных. Приложение должно уметь связываться с базой данных для получения из нее информации или для помещения информации в базу данных. Дело здесь осложняется тем, что СУБД (системы управления базами данных) сильно отличаются друг от друга и совершенно по-разному управляют базами данных. Каждая СУБД предоставляет свой набор функций для доступа к базам данных, и приходится для каждой СУБД писать свое приложение. Но что делать при работе по сети, когда неизвестно, какая СУБД управляет базой на сервере?

Выход был найден корпорацией Microsoft, создавшей набор интерфейсов ODBC (Open Database Connectivity) для связи с базами данных, оформленных как прототипы функций языка С. Эти прототипы одинаковы для любой СУБД, они просто описывают набор действий с таблицами базы данных. В приложение, обращающееся к базе данных, записываются вызовы функций ODBC. Для каждой системы управления базами данных разрабатывается так называемый *драйвер ODBC*, реализующий эти функции для конкретной СУБД. Драйвер просматривает приложение, находит обращения к базе данных, передает их СУБД, получает от нее результаты и подставляет их в приложение. Идея оказалась очень удачной, и использование ODBC для работы с базами данных стало общепринятым.

Фирма SUN подхватила эту идею и разработала набор интерфейсов и классов, названный JDBC, предназначенный для работы с базами данных. Эти интерфейсы и классы составили пакет `java.sql`, входящий в J2SDK Standard Edition, и его расширение `javax.sql`, входящее в J2SDK Enterprise Edition.

Кроме классов с методами доступа к базам данных для каждой СУБД необходим драйвер JDBC — промежуточная программа, реализующая методы JDBC. Существуют четыре типа драйверов JDBC.

1. Драйвер, реализующий методы JDBC вызовами функций ODBC. Это так называемый *мост* (bridge) JDBC-ODBC. Непосредственную связь с базой при этом осуществляет драйвер ODBC.
2. Драйвер, реализующий методы JDBC вызовами функций API самой СУБД.
3. Драйвер, реализующий методы JDBC вызовами функций сетевого протокола, независимого от СУБД. Этот протокол должен быть, затем, реализован средствами СУБД.
4. Драйвер, реализующий методы JDBC вызовами функций сетевого протокола СУБД.

Перед обращением к базе данных следует установить нужный драйвер, например, мост JDBC-ODBC:

```
try{
   Class dr = sun.jdbc.odbc.JdbcOdbcDriver.class;
}catch(ClassNotFoundException e){
   System.err.println("JDBC-ODBC bridge not found " + e);
}
```

Объект `dr` не понадобится в программе, но таков синтаксис.

Другой способ установки драйвера показан в листинге П.1.

После того как драйвер установлен, надо связаться с базой данных. Методы связи описаны в интерфейсе `Connection`. Экземпляр класса, реализующего этот интерфейс, можно получить одним из статических методов `getConnection()` класса `DriverManager`, например:

```
String url = "jdbc:odbc:mydb";
String login = "habib";
String password = "lnF4vb";
Connection con = DriverManager.getConnection(url, login, password);
```

Обратите внимание на то, как формируется адрес базы данных `url`. Он начинается со строки `"jdbc:"`, потом записывается *подпротокол* (subprotocol), в данном примере используется мост JDBC-ODBC, поэтому записывается `"odbc:"`. Далее указывается адрес (subname) по правилам подпротокола,

здесь просто имя локальной базы "mydb". Второй и третий аргументы — это имя и пароль для соединения с базой данных.

Если в вашей вычислительной системе установлен пакет javax.sql, то вместо класса DriverManager лучше использовать интерфейс DataSource.

Связавшись с базой данных, можно посылать запросы. Запрос хранится в объекте, реализующем интерфейс Statement. Этот объект создается методом createStatement(), описанным в интерфейсе Connection. Например:

```
Statement st = con.createStatement();
```

Затем запрос (query) заносится в этот объект методом execute() и потом выполняется методом getResultSet(). В простых случаях это можно сделать одним методом executeQuery(), например:

```
ResultSet rs = st.executeQuery("SELECT name, code FROM tbl1");
```

Здесь из таблицы tbl1 извлекается содержимое двух столбцов name и code и заносится в объект rs класса, реализующего интерфейс ResultSet.

SQL-операторы INSERT, UPDATE, DELETE, CREATE TABLE и другие в простых случаях выполняются методом executeUpdate().

Остается методом next() перебрать элементы объекта rs — строки полученных столбцов — и извлечь данные многочисленными методами get*Xxx*() интерфейса ResultSet:

```
while (rs.next()){
   emp[i] = rs.getString("name");
   num[i] = rs.getInt("code");
   i++;
}
```

Методы интерфейса ResultSetMetaData позволяют узнать количество полученных столбцов, их имена и типы, название таблицы, имя ее владельца и прочие сведения о представленных в объекте rs сведениях.

Если объект st получен методом

```
Statement st = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
   ResultSet.CONCUR_UPDATABLE);
```

то можно перейти к предыдущему элементу методом previous(), к первому элементу — методом first(), к последнему — методом last(). Можно также изменять объект rs методами update*Xxx*() и даже изменять, удалять и добавлять соответствующие строки базы данных. Не все драйверы обеспечивают эти возможности, поэтому надо проверить реальный тип объекта rs методами rs.getType() и rs.getConcurrency().

Интерфейс Statement расширен интерфейсом PreparedStatement, тоже позволяющим изменять объект ResultSet методами set*Xxx*().

Интерфейс `PreparedStatement`, в свою очередь, расширен интерфейсом `CallableStatement`, в котором описаны методы выполнения хранимых процедур.

В листинге П.1 приведен типичный пример запроса к базе Oracle через драйвер Oracle Thin. Апплет выводит в окно браузера четыре поля ввода для адреса базы, имени и пароля пользователя, и запроса. По умолчанию формируется запрос к стартовой базе Oracle, расположенной на локальном компьютере. Результат запроса выводится в окно браузера.

Листинг П.1. Апплет, обращающийся к базе Oracle

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.util.*;
import java.sql.*;

public class JdbcApplet extends Applet
                          implements ActionListener, Runnable{
   private TextField tf1, tf2, tf3;
   private TextArea ta;
   private Button b1, b2;
   private String url = "jdbc:oracle:thin:@localhost:1521:ORCL",
                login = "scott",
             password = "tiger",
                query = "SELECT * FROM dept";
   private Thread th;
   private Vector results;
   public void init(){
     setBackground(Color.white);
     try{
        DriverManager.registerDriver(
                new oracle.jdbc.driver.OracleDriver());
     }catch(SQLException e){
        System.err.println(e);
     }
     setLayout(null);
     setFont(new Font("Serif", Font.PLAIN, 14));

     Label l1 = new Label("URL базы:", Label.RIGHT);
     l1.setBounds(20, 30, 70, 25); add(l1);

     Label l2 = new Label("Имя:", Label.RIGHT);
     l2.setBounds(20, 60, 70, 25); add(l2);

     Label l3 = new Label("Пароль:", Label.RIGHT);
     l3.setBounds(20, 90, 70, 25); add(l3);
```

```
    tf1 = new TextField(url, 30);
    tf1.setBounds(100, 30, 280, 25); add(tf1);

    tf2 = new TextField(login, 30);
    tf2.setBounds(100, 60, 280, 25); add(tf2);

    tf3 = new TextField(password, 30);
    tf3.setBounds(100, 90, 280, 25); add(tf3);
    tf3.setEchoChar('*');

    Label l4 = new Label("Запрос:", Label.LEFT);
    l4.setBounds(10, 120, 70, 25); add(l4);

    ta = new TextArea(query, 5, 50, TextArea.SCROLLBARS_NONE);
    ta.setBounds(10, 150, 370, 100); add(ta);

    Button b1 = new Button("Отправить");
    b1.setBounds(280, 260, 100, 30); add(b1);
    b1.addActionListener(this);
  }
  public void actionPerformed(ActionEvent ae){
    url      = tf1.getText();
    login    = tf2.getText();
    password = tf3.getText();
    query    = ta.getText();
    if (th == null){
      th = new Thread(this);
      th.start();
    }
  }
  public void run(){
    try{
      Connection con =
           DriverManager.getConnection(url, login, password);
      Statement st = con.createStatement();
      ResultSet rs = st.executeQuery(query);
      ResultSetMetaData rsmd = rs.getMetaData();
              // Узнаем число столбцов
      int n = rsmd.getColumnCount();
      results = new Vector();
      while (rs.next()){
        String s = " ";
               // Номера столбцов начинаются с 1!
        for (int i = 1; i <= n; i++)
          s += "  " + rs.getObject(i);
        results.addElement(s);
      }
```

```
            rs.close();
            st.close();
            con.close();
            repaint();
        }catch(Exception e){
            System.err.println(e);
        }
        repaint();
    }
    public void paint(Graphics g){
        if (results == null){
            g.drawString("Can't execute the query", 5, 30);
            return;
        }
        int y = 30, n = results.size();
        for (int i = 0; i < n; i++)
            g.drawString((String)results.elementAt(i), 5, y += 20);
    }
}
```

Замечание по отладке

В *главе 19* упоминалось, что для отладки сетевой программы удобно запустить и клиентскую, и серверную часть на одном компьютере, обращаясь к серверной части по адресу 127.0.0.1 или доменному имени localhost. Не забывайте, что апплет может связаться по сети только с тем хостом, откуда он загружен. Следовательно, на компьютере должен работать Web-сервер. Если Web-сервер прослушивает порт 8080, то, чтобы загрузить HTML-страницу с апплетом, надо в браузере указывать адрес URL вида **http://localhost:8080/public/JdbcApplet.html**. При этом учтите, что Web-сервер устанавливает свою иерархию каталогов, и каталог public на самом деле может быть каталогом usr/local/http/public или каким-нибудь другим.

Таким образом, JDBC позволяет проделать весь цикл работы с базой данных. Подробно со всеми возможностями JDBC можно познакомиться, прочитав спецификацию JDBC, имеющуюся в документации Java 2 SDK, в каталоге docs\guide\jdbc\spec\. Дополнения спецификации версии JDBC 2.0 изложены в каталоге docs\guide\jdbc\spec2\. В каталоге docs\guide\jdbc\getstart\ есть пособие по использованию JDBC.

Сервлеты

В *главе 19* была упомянута технология CGI. Ее суть в том, что сетевой клиент, обычно браузер, посылает Web-серверу информацию вместе с указанием программы, которая будет обрабатывать эту информацию. Web-сервер, получив информацию, запускает программу, передает информацию на ее

стандартный ввод и ждет окончания обработки. Результаты обработки программа отправляет на свой стандартный вывод, Web-сервер забирает их оттуда и отправляет клиенту. Написать программу можно на любом языке, лишь бы Web-сервер мог взаимодействовать с ней.

В технологии Java такие программы оформляются как *сервлеты* (servlets). Это название не случайно похоже на название "апплеты". Сервлет на Web-сервере играет такую же роль, что и апплет в браузере, расширяя его возможности.

Чтобы Web-сервер мог выполнять сервлеты, в его состав должна входить JVM и средства связи с сервлетами. Обычно все это поставляется в виде отдельного модуля, встраиваемого в Web-сервер. Существует много таких модулей: Resin, Tomcat, JRun, JServ. Они называются на жаргоне *сервлетными движками* (servlet engine).

Основные интерфейсы и классы, описывающие сервлеты, собраны в пакет `javax.servlet`. Расширения этих интерфейсов и классов, использующие конкретные особенности протокола HTTP, собраны в пакет `javax.servlet.http`. Еще два пакета — `javax.servlet.jsp` и `javax.servlet.jsp.tagext` — предназначены для связи сервлетов со скриптовым языком JSP (JavaServer Pages). Все эти пакеты входят в состав J2SDK Enterprise Edition. Они могут поставляться и отдельно как набор JSDK (Java Servlet Development Kit). Сервлетный движок должен реализовать эти интерфейсы.

Основу пакета `javax.servlet` составляет интерфейс `Servlet`, частично реализованный в абстрактном классе `GenericServlet` и его абстрактном подклассе `HttpServlet`. Основу этого интерфейса составляют три метода:

- `init(ServletConfig config)` — задает начальные значения сервлету, играет ту же роль, что и метод `init()` в апплетах;
- `service(ServletRequest req, ServletResponse resp)` — выполняет обработку поступившей сервлету информации `req` и формирует ответ `resp`;
- `destroy()` — завершает работу сервлета.

Опытный читатель уже понял, что вся работа сервлета сосредоточена в методе `service()`. Действительно, это единственный метод, не реализованный в классе `GenericServlet`. Достаточно расширить свой класс от класса `GenericServlet` и реализовать в нем метод `service()`, чтобы получить собственный сервлет. Сервлетный движок, встроенный в Web-сервер, реализует интерфейсы `ServletRequest` и `ServletResponse`. Он создает объекты `req` и `resp`, заносит всю поступившую информацию в объект `req` и передает этот объект сервлету вместе с пустым объектом `resp`. Сервлет принимает эти объекты как аргументы `req` и `resp` метода `service()`, обрабатывает информацию, заключенную в `req`, и оформляет ответ, заполняя объект `resp`. Движок забирает этот ответ и через Web-сервер отправляет его клиенту.

Основная информация, заключенная в объекте req, может быть получена методами read() и readLine() из байтового потока класса ServletInputStream, непосредственно расширяющего класс InputStream, или из символьного потока класса BufferedReader. Эти потоки открываются, соответственно, методом req.getInputStream() или методом req.getReader(). Дополнительные характеристики запроса можно получить многочисленными методами get*Xxx*() объекта req. Кроме того, класс GenericServlet реализует массу методов get*Xxx*(), позволяющих получить дополнительную информацию о конфигурации клиента.

Интерфейс ServletResponse описывает симметричные методы для формирования ответа. Метод getOutputStream() открывает байтовый поток класса ServletOutputStream, непосредственно расширяющего класс OutputStream. Метод getWriter() открывает символьный поток класса PrintWriter.

Итак, реализуя метод service(), надо получить информацию из входного потока объекта req, обработать ее и результат записать в выходной поток объекта resp.

Очень часто в объекте req содержится запрос к базе данных. В таком случае метод service() обращается через JDBC к базе данных и формирует ответ resp из полученного объекта ResultSet.

Протокол HTTP предлагает несколько методов передачи данных: GET, POST, PUT, DELETE. Для их использования класс GenericServlet расширен классом HttpServlet, находящимся в пакете javax.servlet.http. В этом классе есть методы для реализации каждого метода передачи данных:

```
doGet(HttpServletRequest req, HttpServletResponse resp)
doPost(HttpServletRequest req, HttpServletResponse resp)
doPut(HttpServletRequest req, HttpServletResponse resp)
doDelete(HttpServletRequest req, HttpServletResponse resp)
```

Для работы с конкретным HTTP-методом передачи данных достаточно расширить свой класс от класса HttpServlet и реализовать один из этих методов. Метод service() переопределять не надо, в классе HttpServlet он только определяет, каким HTTP-методом передан запрос клиента, и обращается к соответствующему методу do*Xxx*(). Аргументы перечисленных методов req и resp — это объекты, реализующие интерфейсы HttpServletRequest и HttpServletResponse, расширяющие интерфейсы ServletRequest и ServletResponse, соответственно.

Интерфейс HttpServletRequest к тому же описывает множество методов get*Xxx*(), позволяющих получить дополнительные свойства запроса req.

Интерфейс HttpServletResponse описывает методы add*Xxx*() и set*Xxx*(), дополняющие ответ resp, и статические константы с кодами ответа Web-сервера.

В листингах П.2 и П.3 те же действия, что выполняет программа листинга П.1, реализованы с помощью сервлета. Апплет теперь не нужен, в окно браузера выводится HTML-форма, описанная в листинге П.2.

Листинг П.2. HTML-форма запроса к базе данных

```
<html><head><title> JDBC Servlet</title></head>
  <body>
    <form method = "POST" action = "/servlet/JdbcServlet">
    <pre>
        URL базы: <input type = "text" size = "40" name = "url"
                   value = "jdbc:oracle:thin:@localhost:1521:ORCL">
             Имя: <input type = "text" size = "40" name = "login"
                   value = "scott">
          Пароль: <input type = "password" size = "40" name = "password"
                   value = "tiger">
    Запрос:
    <textarea rows = "5" cols "70" name = "query">
        SELECT * FROM dept
    </textarea>
                     <input type = "submit" value = "Послать">
    </pre>
    </form>
  </body>
</html>
```

Сервлет получает из этой формы адрес базы `url`, имя `login` и пароль `password` пользователя, а также запрос `query`. Он обращается к базе данных Oracle через драйвер JDBC Oracle Thin, формирует полученный ответ в виде HTML-страницы и отправляет браузеру.

Листинг П.3. Сервлет, выполняющий запрос к базе Oracle

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.sql.*;

public class JdbcServlet extends HttpServlet{
   static{
       try{
         DriverManager.registerDriver(
             new oracle.jdbc.driver.OracleDriver());
       }catch(SQLException e){
```

```
          System.err.println(e);
        }
    }
    private Vector results = new Vector();
    private int n;

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
                         throws IOException, ServletException{
      String url  = req.getParameter("url"),
         login    = req.getParameter("login"),
         password = req.getParameter("password"),
         query    = req.getParameter("query");
            // Задаем MIME-тип и кодировку для выходного потока pw
      resp.setContentType("text/html;charset=windows-1251");
      PrintWriter pw = resp.getWriter();
      try{
         Connection con =
            DriverManager.getConnection(url, login, password);
         Statement st = con.createStatement();
         ResultSet rs = st.executeQuery(query);
         ResultSetMetaData rsmd = rs.getMetaData();
         n = rsmd.getColumnCount();
         while (rs.next()){
            String s = " ";
            for (int i = 1; i <= n; i++)
                s += "  " + rs.getObject(i);
            results.addElement(s);
         }
         rs.close();
         st.close();
         con.close();
       }catch(SQLException e){
          pw.println("From doPost(): " + e);
       }
       pw.println("<html><head><title>Answers</title></head>");
       pw.println("<body><h2>Результаты запроса</h2>");
       n = results.size();
       for (int i = 0; i < n; i++)
           pw.println((String)results.elementAt(i) + "<br>");
       pw.println("</body></html>");
       pw.flush();
       pw.close();
   }
}
```

Применение сервлета позволило "облегчить" клиент — браузер не загружает апплет, а только отправляет запрос и получает ответ. Вся обработка запроса ведется в сервлете на сервере.

В системе J2SDKEE (Java 2 SDK Enterprise Edition) HTML-файл и сервлет образуют один Web-компонент. Они упаковываются в один файл с расширением war (web archive) и помещаются в так называемый Web-контейнер, управляемый Web-сервером, расширенным средствами J2SDKEE.

Java на сервере

Тенденция написания сетевых программ — побольше функций возложить на серверную часть программы и поменьше оставить клиентской части, сделав клиент "тонким", а сервер "толстым". Это позволяет, с одной стороны, использовать клиентскую часть программы на самых старых и маломощных компьютерах, а с другой стороны, облегчает модификацию программы — все изменения достаточно сделать только в одном месте, на сервере.

Сервер выполняет все больше функций, как говорят, *служб* или *сервисов* (services). Он и отправляет клиенту Web-страницы, и выполняет сервлеты, и связывается с базой данных, и обеспечивает транзакции. Такой многофункциональный сервер называется *сервером приложений* (application server). Большой популярностью сейчас пользуются серверы приложений WebLogic фирмы BEA Systems, IAS (Inprise Application Server) фирмы Borland, WebSphere фирмы IBM, OAS (Oracle Application Server). Важной характеристикой сервера приложений является способность расширять свои возможности путем включения новых модулей. Это удобно делать с помощью компонентов.

Фирма SUN Microsystems предложила свою систему компонентов EJB (Enterprise JavaBeans), включенную в Java 2 SDK Enterprise Edition. Подобно тому, как графические компоненты JavaBeans реализуют графический интерфейс пользователя, размещаясь в графических контейнерах, компоненты EJB реализуют различные службы на сервере, располагаясь в EJB-контейнерах. Этими контейнерами управляет EJB-сервер, включаемый в состав сервера приложений. В составе J2SDKEE EJB-сервер — это программа `j2ee`. Серверу приложений достаточно запустить эту программу, чтобы использовать компоненты EJB.

В отличие от JavaBeans у компонентов EJB не может быть графического интерфейса и ввода с клавиатуры. У них может отсутствовать даже консольный вывод. Контейнер EJB занимается не размещением компонентов, а созданием и удалением их объектов, связью с клиентами и другими компонентами, проверкой прав доступа и обеспечением транзакций.

Программы, оформляемые как EJB, могут быть двух типов: `EntityBean` и `SessionBean`. Они реализуют соответствующие интерфейсы из пакета

javax.ejb. Первый тип EJB-компонентов удобен для создания программ, обращающихся к базам данных и выполняющих сложную обработку полученной информации. Компоненты этого типа могут работать сразу с несколькими клиентами. Второй тип EJB-компонентов более удобен для организации взаимодействия с клиентом. Компоненты этого типа бывают двух видов: сохраняющие свое состояние от запроса к запросу (stateful) и теряющие это состояние (stateless). Методами интерфейсов EntityBean и SessionBean контейнер EJB управляет поведением экземпляров класса. Если достаточно стандартного управления, то можно сделать пустую реализацию этих методов.

Кроме класса, реализующего интерфейс EntityBean или SessionBean, для создания компонента EJB необходимо создать еще два интерфейса, расширяющие интерфейсы EJBHome и EJBObject. Первый интерфейс (home interface) служит для создания объекта EJB своими методами create(), для поиска и связи с этим объектом в процессе работы, и удаления его методом remove(). Второй интерфейс (remote interface) описывает методы компонента EJB. Интересная особенность технологии EJB — клиентская программа не образует объекты компонента EJB. Вместо этого она создает объекты home- и remote-интерфейсов и работает с этими объектами. Реализация home- и remote-интерфейсов, создание объектов компонента EJB и взаимодействие с ними остается на долю контейнера EJB.

Приведем простейший пример. Пусть мы решили обработать выборку из базы данных, занесенную в объект rs в сервлете листинга П.3, с помощью компонента EJB. Для простоты пусть обработка заключается в слиянии двух столбцов методом merge(). Напишем программу:

```
import java.rmi.RemoteException;
import javax.ejb.*;

public class MergeBean implements SessionBean{
   public String merge(String s1, String s2){
      String s = s1 + " " + s2;
      return s;
   }
        // Выполняется при обращении к методу create()
        // интерфейса MergeHome. Играет роль конструктора класса
   public void ejbCreate(){}
        // Пустая реализация методов интерфейса
   public void setSessionContext(SessionContext ctx){}
   public void ejbRemove(){}
   public void ejbActivate(){}
   public void ejbPassivate(){}
}
```

```
public interface MergeHome extends EJBHome{
        // Реализуется методом ejbCreate() класса MergeBean
   Merge create() throws CreateException, RemoteException;
}
public interface Merge extends EJBObject{
   public double merge(String s1, String s2)
       throws RemoteException;
}
```

В сервлете листинга П.3 создаем объекты типа `MergeHome` и `Merge` и обращаемся к их методам:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;

public class JDBCServlet extends HttpServlet {
   MergeHome mh;
   Merge m;
   // Следующие определения
   // . . . . . . .
   public void init(ServletConfig conf) throws ServletException{
      try{   // Поиск объекта merge, реализующего MergeHome
         InitialContext ic = new InitialContext();
         Object ref = ic.lookup("merge");
         mh = (MergeHome)PortableRemoteObject.narrow(
                 ref, MergeHome.class);
      }catch(Exception e){
         e.printStackTrace();
      }
   public void doPost(HttpServletRequest req, HttpServletResponse resp)
                     throws IOException, ServletException{
      // Начало метода
      // . . . . . . . .
      m = mh.create();
      String s = m.merge(s1, s2);
      // и т. д.
   }
}
```

После компиляции получаем EJB-приложение, состоящее из пяти файлов: JdbcServlet.html, JdbcServlet.class, MergeBean.class, MergeHome.class и Merge.class. Осталось правильно установить (deploy) его в контейнер EJB. Файлы `JdbcServlet.html` и `JdbcServlet.class` надо упаковать в один war-файл, ос-

тальные файлы — в один jar-файл, потом оба получившихся файла упаковать в один ear-файл (Enterprise ARchive). Кроме того, надо создать еще файл описания установки (deployment descriptor) в формате XML и занести его в архив. В этот файл, в частности, записывается имя "merge", по которому компонент отыскивается методом lookup().

Все это можно сделать утилитой deploytool, входящей в состав Java 2 SDK Enterprise Edition. Эта же утилита позволяет проверить работу приложения и установить его в контейнер EJB. Надо только предварительно запустить EJB-сервер командой j2ee.

Впрочем, все файлы EJB-приложения можно упаковать в один jar-файл.

Многие серверы приложений и средства разработки, такие как Borland JBuilder и IBM Visual Age for Java, имеют в своем составе утилиты для установки EJB-приложений.

EJB-приложение готово. Теперь достаточно вызвать в браузере HTML-файл и заполнить появившуюся в окне форму.

Заключение

Ну вот и все. Книга прочитана. Теперь вы можете уверенно чувствовать себя в мире Java, свободно разбираться в новых технологиях и создавать свои приложения на самом современном уровне. Конечно, в этой книге мы не смогли подробно разобрать все аспекты технологии Java, но, зная основные приемы, вы сможете легко освоить все ее новые методы. Успехов вам!

Список литературы

За недолгую историю технологии Java выпущено целое море учебников, справочников, монографий по отдельным вопросам. Ввиду необычайно быстрого развития технологии книги быстро устаревают, выходят их новые, обновленные издания. Книги, описывающие JDK 1.0, уже безнадежно устарели.

По технологии Java 2 на русском языке уже выпущены книги [1, 13, 14]. Группа волонтеров: Сергей Ковалев, Владислав Кравченко, Сергей Гордиенко, Шамиль Низамов, готовит электронный вариант перевода прекрасной книги [2] (**http://www.bruceeckel.com**). Его можно посмотреть по адресу **http://www.BruceEckel.by.ru**.

Большое количество книг по технологии Java готовится к выпуску и находится в планах издательств.

Поэтому попытка составить полную библиографию технологии Java обречена на неудачу. В этом списке литературы перечислены только те издания, которые связаны с текстом книги и не указаны во *введении*.

1. Морган М. Java 2. Руководство разработчика: Пер. с англ. Уч. пос. — М.: Издательский дом "Вильямс", 2000. — 720 с.: ил.
2. Bruce Eckel, Thinking in Java, 2nd ed, 2000.
3. Бадд Т. Объектно-ориентированное программирование в действии: Пер. с англ. — СПб.: Питер, 1997. — 464 с.: ил.
4. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на С++: Пер. с англ. — 2-е изд. — М.: Бином, СПб.: Невский диалект, 1998. — 560 с.: ил.
5. Коуд П., Норт Д., Мейфилд М. Объектные модели. Стратегии, шаблоны и приложения: Пер. с англ. — "Лори", 1999.
6. Страуструп Б. Язык программирования С++: Пер. с англ. — 3-е изд. — СПб.: Невский диалект, М.: Бином, 1999. — 991 с.: ил.

7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.

8. James W. Cooper, The Design Patterns. Java Companion, Addison-Wesley, 1998.

http://www.patterndepot.com/put/8/DesignJavaPDF.zip.

9. Bruce Eckel, Thinking in Patterns with Java, 2000.

http://www.bruceeckel.com/.

10. Bill Venners, Interface Design. Best Practices in Object-Oriented API Design in Java, 2000.

http://www.artima.com/preciseobject.

11. Вирт Н. Алгоритмы и структуры данных: Пер. с англ. — М.: Мир, 1989. — 360 с.: ил.

12. Ярмола Ю. А. Компьютерные шрифты. — СПб.: BHV — Санкт-Петербург, 1994. — 208 с.: ил.

13. Смирнов Н. И. JAVA 2: Учебное пособие. — М.: "Три Л", 2000. — 320 с.

14. Ноутон П., Шилдт Г. Java 2: Пер. с англ. — СПб.: БХВ-Петербург, 2000. — 1072 с.: ил.

Предметный указатель

A

abstract method 80
access methods 74
anonymous classes 93
applet 301
application 301
arguments 29
array 37
ascent 204
associative array 163

B

baseline 197
bitwise 44
bytecodes 17

C

callback 271
capacity 145
caret 221, 238
class 29, 37, 69
◊ body 29
◊ constructor 82
◊ fields 69
◊ methods 69, 85
◊ modifiers 76
◊ variables 84
Code Conventions 30
color model 194
component 186
compound assignment operators 47
concatenation 138
container 186
context menu 297
contract 73
convolution 335
Core API 20

D

daemon 382
dangling else 53
data processing 387
data sink 387
declaration 60
default access 99
default constructor 83
delegation 271
deprecated 14
descent 204
deserialization 403
design patterns 111
dictionary 163
double buffering 338
drop-down menu 292
dynamic binding 17

E

Epoch 177
event 269
◊ source 269
exception 354
expression 48
◊ statement 51
extension 71

F

fall through labels 59
FAQ 12
floating-point types 37
font:
◊ face name 198
◊ height 204
◊ name 198
fully qualified name 98

G

gap 259
get method 74
glyph 197
graphics context 192

H

heavy component 185
high cohestion 75
hint 223
HSB 194

I

idenifiers 36
immediate mode model 326
implementation 108
incapsulation 67
inheritance 72
initialization 60
inner class 92
input focus 230
instance 69
◊ initialization 92
◊ methods 85
◊ variables 84
instantation 60
integral types 37
interface 37, 107
item 234

J

J2SDK 20
Java 2 20
Java Plug-in 23
JDK 18
JFC 186
JIT-компилятор 17
JLS 28
JRE 20
JVM 17

L

layout manager 230, 258
leading 204
LIFO 155
lightweight component 185
listener 270
local class 93
locale 134
logical font names 198
low coupling 74

M

map 163
member classes 93
menu bar 292
message 72
method 29
modal window 249
modifiers 29
modularity 73
multiple inheritance 106
MVC 112

N

names 36
namespace 98
narrowing 43
native methods 18
nested class 92
nested top-level classes 93
numeric 37

O

object-oriented programming 67
OOD 68
OOP 67
overloading 78
overriding 78

P

package 98
parser 148
parsing 148
peer-интерфейс 185
pipe 385, 401
polymorphism 73
popup menu 297
primitive types 37
private 81
process 367
producer-consumer 319
promotion 42
Pure Java 23

Q

qualified name 37
qualified names 36

R

reference 60
◊ types 37
rendering 223
responsibility 73
RGB 193
runtime 354

S

sandbox 316
scope 90
sequence 162
serialization 385, 403
servlet 447
set 162
set method 74
signature 77
simple assignment operator 47
slider 241
socket 425
stack 155
statement 50
status bar 304
stderr 384
stdin 384
stdout 384
stream 384
subclass 72
subpackage 98
superclass 72

T

tag 303
tear-off menu 294
thread 368
throws 359
trusted applet 317

U

unnamed package 99

V

virtual key codes 281
VMS 17

W

widening 43
wrapper 116

А

Абстракция 68
Альфа 193
Апплет 301
◊ доверенный 317
Аргумент метода 29

Б

Байт-коды 17
Безымянный пакет 99
Блок:
◊ инициализации экземпляра 92
◊ помеченный 58
◊ статической инициализации 86
Буква Java 36

В

Вещественные типы 37, 46
Виртуальная машина Java 17
◊ спецификация 17
Виртуальные коды клавиш 281
Всплывающее меню 297
Выпадающее меню 292
Выражение 48

Г

Графический контекст 192

Д

Двойная буферизация 338
Действительная константа 34
Делегирование 271
Демон 382
Десериализация 403
Дизъюнкция 39
◊ сокращенная 39
Динамическая компоновка 17
Длина массива 60
Дополнение 44
Дополнительный код 40

Е

Емкость 145

З

Загрузочный модуль 17
Закон Деметра 75
Зацепление 74

И

Идентификатор 36
Иерархия 70
Имя 36
◊ семейства шрифтов 198
◊ составное 36, 37
Инкапсуляция 67
Интерфейс 37, 107
◊ реализация 108
Исключающее ИЛИ 39
Исключительная ситуация 354
Источник события 269
Исходный модуль 16

К

Канал 385, 401
Класс 29, 37, 69
◊ абстрактный 80
◊ безымянный 93
◊ вложенный 92
◊ вложенный верхнего уровня 93
◊ внутренний 92
◊ локальный 93
◊ полное имя 98
Класс-оболочка 116
Классы-члены 93
Комментарий 32
Компонент 186
◊ "легкий" 185
◊ "тяжелый" 185
Константа 33, 81
◊ целая 33
Конструктор:
◊ класса 82
◊ по умолчанию 83
Контейнер 186
Контекстное меню 297
Контракт 73
Конъюнкция 38
◊ сокращенная 39

Л

Логические имена шрифтов 198
Логический тип 37
Локаль 134, 176

М

Массив 37, 60
◊ ассоциативный 163
◊ инициализация 60
◊ объявление 60
◊ определение 60
Менеджер размещения 230, 258
◊ BorderLayout 260
◊ CardLayout 264
◊ FlowLayout 259
◊ GridBagLayout 266
◊ GridLayout 263
Метка 58
Метод 29
◊ main 89
◊ доступа 74
◊ класса 69, 85
◊ окончательный 81
◊ параметр 29
◊ перегрузка 78
◊ переопределение 78
◊ "родной" 18
◊ статический 85
◊ экземпляра 85
Множество 162
Модальное окно 249
Модель прямого доступа 326
Модификатор 29
Модификаторы класса 76
Модульность 73

Н

Надкласс 72
Наследование 72
◊ множественное 106

О

Область видимости 90
Обратный вызов 271
Объектно-ориентированное программирование 67
Объектно-ориентированное проектирование 68
Объектно-ориентированный анализ 68
Объектный модуль 17
ООА 68
ООП 67
Оператор 50
◊ break 58
◊ continue 57
◊ варианта 58
◊ помеченный 58
◊ присваивания 52
◊ условный 52
◊ цикла 54
Операция:
◊ присваивания 47
◊ условная 48
◊ логическая 38
Ответственность 73
Отношение "has-a" 97
Отношение "is-a" 97
Отображение 163
Отрицание 38
Отсоединяемое меню 294

П

Пакет 98
Пакетный доступ 98
Парсер 148
Парсинг 148
Переменные:
◊ класса 84
◊ экземпляра 84
◊ статические 85
Песочница 316
Побитовые операции 44
◊ дизъюнкция 44
◊ конъюнкция 44
◊ исключающее ИЛИ 44
Повышение типа 42
Подкласс 72
Подпакет 98
Подпроцесс 368
Поле класса 69
Полиморфизм 73, 108
Последовательность 162
Поставщик-потребитель 319
Поток 384
Примитивные типы 37
Принцип KISS 75

Программирование:
◊ модульное 66
◊ процедурное 66
◊ структурное 66
Пространство имен 98
Процесс 367

Р

Расширение:
◊ класса 71
◊ типа 43

С

Связность 75
Сдвиг:
◊ арифметический 45
◊ беззнаковый вправо 45
◊ влево 45
◊ вправо 45
◊ логический 46
Сервлет 447
Сериализация 385, 403
Сигнатура метода 77
Символ 34
Словарь 163
Слушатель 270
Событие 269
Сокет 425
Составные операции присваивания 47
Ссылка 60
Ссылочные типы 37
Стандартный ввод 384
Стандартный вывод 384
Стандартный вывод сообщений 384
Стек 155
Строка 35
◊ сцепление 138
Строка меню 292
Строка состояния 304
Сужение типа 43
Суперкласс 72

Т

Тег 303
Тело:
◊ класса 29
◊ метода 30

Ф

Физическое имя шрифта 198
Фокус ввода 230

Ц

Целое деление 41
Целые типы 37

Ч

Числовые типы 37
Член класса 69
◊ закрытый 73
◊ защищенный 75
◊ открытый 73

Э

Экземпляр класса 69