Самоучитель

Ильдар Хабибуллин

Java

3-е издание



Принципы объектно-ориентированного программирования

Пакеты классов и интерфейсы Java SE 6

Библиотеки Swing и Java 2D

Апплеты, графика, звук и печать

Технологии Servlet, JSP, JSTL, JSF, XML

Около двухсот законченных программ

bhv

Ильдар Хабибуллин

Самоучитель Java

3-е издание

Санкт-Петербург
«БХВ-Петербург»
2008

УДК 681.3.06
ББК 32.973.26-018.2
Х12

Хабибуллин И. Ш.

Х12 Самоучитель Java. — 3-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2008. — 768 с.: ил.

ISBN 978-5-9775-0191-0

Самоучитель написан на основе материала лекций, читаемых автором для студентов первого и второго курсов. Изложены практические приемы использования как традиционных, так и новейших конструкций объектно-ориентированного языка Java, графической библиотеки классов Swing, расширенной библиотеки Java 2D, работа со звуком, печать, способы русификации программ. Третье издание содержит полное описание нововведений Java SE 6. Дано подробное изложение сервлетов, технологии JSP и библиотек тегов JSTL. Около двухсот законченных программ иллюстрируют приведенные приемы программирования. Подробные схемы и описания классов и методов Core Java API позволяют применять книгу и как настольный справочник по технологии Java.

Для широкого круга программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Владимир Красовский*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 28.03.08.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 61,92.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию № 77.99.60.953.Д.002108.02.07
от 28.02.2007 г. выдано Федеральной службой по надзору
в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0191-0

© Хабибуллин И. Ш., 2008
© Оформление, издательство "БХВ-Петербург", 2008

Оглавление

Введение **1**

Что такое Java? 2
Структура книги 3
Выполнение Java-программы 5
Что такое JDK? 7
Что такое JRE? 9
Как установить JDK? 9
Как использовать JDK? 10
Интегрированные среды Java 11
Особая позиция Microsoft 12
Java в Интернете 12
Литература по Java 13
Благодарности 14

ЧАСТЬ I. БАЗОВЫЕ КОНСТРУКЦИИ ЯЗЫКА JAVA **15**

Глава 1. Встроенные типы данных, операции над ними **17**

Первая программа на Java 17
Комментарии 20
Аннотации 22
Константы 22
Целые 22
Действительные 23
Символы 23
Строки 24
Имена 25
Примитивные типы данных и операции 26
Логический тип 27
Логические операции 27
Упражнения 28
Целые типы 28
Операции над целыми типами 29
Арифметические операции 29
Приведение типов 30

Операции сравнения ... 32
Побитовые операции ... 32
Сдвиги ... 33
Упражнения ... 34
Вещественные типы ... 34
Операции присваивания ... 35
Упражнения ... 36
Условная операция ... 36
Упражнения ... 36
Выражения ... 36
Приоритет операций ... 37
Операторы ... 38
Блок ... 39
Операторы присваивания ... 39
Условный оператор ... 39
Упражнения ... 41
Операторы цикла ... 42
Оператор *continue* и метки ... 44
Оператор *break* ... 45
Упражнения ... 45
Оператор варианта ... 46
Массивы ... 47
Многомерные массивы ... 49
Заключение ... 51
Вопросы для самопроверки ... 51

Глава 2. Объектно-ориентированное программирование в Java ... 52

Парадигмы программирования ... 52
Принципы объектно-ориентированного программирования ... 55
Абстракция ... 55
Иерархия ... 58
Ответственность ... 59
Модульность ... 60
Принцип KISS ... 62
Упражнения ... 63
Как описать класс и подкласс? ... 63
Передача аргументов в метод ... 64
Перегрузка методов ... 66
Переопределение методов ... 67
Реализация полиморфизма в Java ... 67
Упражнения ... 69
Абстрактные методы и классы ... 69
Окончательные члены и классы ... 70
Класс *Object* ... 70
Конструкторы класса ... 71

Операция *new* ... 72
Упражнение ... 73
Статические члены класса ... 73
Класс *Complex* ... 75
Метод *main()* ... 78
Методы с переменным числом аргументов ... 79
Где видны переменные ... 80
Вложенные классы ... 81
Отношения "быть частью" и "являться" ... 86
Заключение ... 86
Вопросы для самопроверки ... 87

Глава 3. Пакеты, интерфейсы и перечисления ... 88

Пакет и подпакет ... 89
Права доступа к членам класса ... 90
Размещение пакетов по файлам ... 92
Импорт классов и пакетов ... 94
Java-файлы ... 95
Интерфейсы ... 96
Перечисления ... 100
Объявление аннотаций ... 103
Design patterns ... 104
Схема проектирования MVC ... 105
Шаблон Singleton ... 106
Заключение ... 107
Вопросы для самопроверки ... 107

ЧАСТЬ II. ИСПОЛЬЗОВАНИЕ КЛАССОВ ИЗ JAVA DEVELOPMENT KIT ... 109

Глава 4. Классы-оболочки и generics ... 111

Числовые классы ... 112
Автоматическая упаковка и распаковка типов ... 114
Настраиваемые типы (generics) ... 115
Шаблон типа (wildcard type) ... 117
Настраиваемые методы ... 119
Класс *Boolean* ... 119
Класс *Character* ... 120
Класс *BigInteger* ... 123
Класс *BigDecimal* ... 125
Класс *Class* ... 129
Вопросы для самопроверки ... 131

Глава 5. Работа со строками 132

Класс *String* 133
Как создать строку 133
Упражнение 138
Сцепление строк 138
Манипуляции строками 138
Как узнать длину строки 139
Как выбрать символы из строки 139
Как выбрать подстроку 140
Как разбить строку на подстроки 140
Как сравнить строки 141
Как найти символ в строке 142
Как найти подстроку 143
Как изменить регистр букв 143
Как заменить отдельный символ 144
Как заменить подстроку 144
Как убрать пробелы в начале и конце строки 144
Как преобразовать в строку данные другого типа 144
Упражнения 145
Класс *StringBuilder* 145
Конструкторы 145
Как добавить подстроку 146
Как вставить подстроку 146
Как удалить подстроку 147
Как удалить символ 147
Как заменить подстроку 147
Как перевернуть строку 148
Синтаксический разбор строки 148
Класс *StringTokenizer* 148
Заключение 149
Вопросы для самопроверки 149

Глава 6. Классы-коллекции 150

Класс *Vector* 150
Как создать вектор 151
Как добавить элемент в вектор 151
Как заменить элемент 152
Как узнать размер вектора 152
Как обратиться к элементу вектора 152
Как узнать, есть ли элемент в векторе 152
Как узнать индекс элемента 152
Как удалить элементы 153
Класс *Stack* 154

Класс *Hashtable* 155
Как создать таблицу Hashtable 155
Как заполнить таблицу Hashtable 156
Как получить значение по ключу 156
Как узнать наличие ключа или значения 156
Как получить все элементы таблицы Hashtable 156
Как удалить элементы 157
Класс *Properties* 158
Интерфейс *Collection* 160
Интерфейс *List* 161
Интерфейс *Set* 162
Интерфейс *SortedSet* 162
Интерфейс *NavigableSet* 162
Интерфейс *Queue* 163
Интерфейс *BlockingQueue* 164
Интерфейс *Deque* 164
Интерфейс *BlockingDeque* 165
Интерфейс *Map* 165
Вложенный интерфейс *Map.Entry* 166
Интерфейс *SortedMap* 166
Интерфейс *NavigableMap* 167
Абстрактные классы-коллекции 168
Интерфейс *Iterator* 168
Интерфейс *ListIterator* 170
Классы, создающие списки 171
Двунаправленный список 171
Дек 172
Упражнение 172
Классы, создающие отображения 172
Связанные отображения 172
Упорядоченные отображения 173
Сравнение элементов коллекций 173
Упражнение 174
Классы, создающие множества 174
Связанные множества 175
Упорядоченные множества 175
Действия с коллекциями 176
Методы класса *Collections* 176
Упражнение 177
Заключение 177
Вопросы для самопроверки 177

Глава 7. Классы-утилиты 178

Работа с массивами 178
Сортировка массива 178

Бинарный поиск в массиве ... 178
Заполнение массива ... 179
Копирование массива ... 179
Сравнение массивов ... 180
Представление массива строкой ... 180
Получение хеш-кода массива ... 181
Локальные установки ... 181
Работа с датами и временем ... 182
Часовой пояс и летнее время ... 183
Класс *Calendar* ... 183
Подкласс *GregorianCalendar* ... 184
Представление даты и времени ... 185
Получение случайных чисел ... 186
Копирование массивов ... 186
Взаимодействие с системой ... 187

ЧАСТЬ III. СОЗДАНИЕ ГРАФИЧЕСКОГО ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ И АППЛЕТОВ ... 189

Глава 8. Принципы построения графического интерфейса ... 191

Компонент и контейнер ... 193
Иерархия классов AWT ... 196
Окно библиотеки Swing ... 197
Использование системных приложений ... 198
System Tray ... 199
Splash Screen ... 199
Заключение ... 200
Вопросы для самопроверки ... 200

Глава 9. Графические примитивы ... 201

Методы класса *Graphics* ... 201
Как задать цвет ... 201
Упражнение ... 203
Как нарисовать чертеж ... 203
Класс *Polygon* ... 204
Упражнение ... 205
Прочие методы класса *Graphics* ... 205
Как вывести текст ... 205
Как установить шрифт ... 206
Как задать шрифт ... 206
Класс *FontMetrics* ... 210
Упражнение ... 213
Возможности Java 2D ... 213

Преобразование координат 214
Класс *AffineTransform* 215
Упражнение 218
Рисование фигур средствами Java 2D 218
Класс *BasicStroke* 218
Класс *GeneralPath* 221
Классы *GradientPaint* и *TexturePaint* 222
Классы *LinearGradientPaint* и *RadialGradientPaint* 224
Вывод текста средствами Java 2D 225
Методы улучшения визуализации 230
Упражнение 231
Заключение 231
Вопросы для самопроверки 232

Глава 10. Основные компоненты AWT 233

Класс *Component* 233
Класс *Cursor* 235
Как создать свой курсор 235
Упражнение 236
События 237
Класс *Container* 237
События 238
Текстовая метка *Label* 238
События 238
Кнопка *Button* 238
События 239
Кнопка выбора *Checkbox* 239
События 239
Класс *CheckboxGroup* 239
Как создать группу радиокнопок 240
Раскрывающийся список *Choice* 241
События 242
Список *List* 242
События 243
Компоненты для ввода текста 244
Класс *TextComponent* 244
События 245
Строка ввода *TextField* 245
События 246
Поле ввода *TextArea* 246
События 246
Линейка прокрутки *Scrollbar* 248
События 249
Контейнер *Panel* 251
Контейнер *ScrollPane* 252

Контейнер *Window* 252
События 253
Контейнер *Frame* 253
События 254
Контейнер *Dialog* 255
События 257
Контейнер *FileDialog* 259
События 259
Создание собственных компонентов 260
Компонент *Canvas* 260
Создание "легкого" компонента 262
Упражнение 264
Создание меню 264
Всплывающее меню 270
Вопросы для самопроверки 273

Глава 11. Оформление ГИП компонентами Swing 274

Состав библиотеки Swing 275
Основные компоненты Swing 277
Компонент *JComponent* 277
Схема MVC в компонентах Swing 278
Надпись *JLabel* 280
Кнопки 282
Кнопка *JButton* 284
Кнопка выбора *JToggleButton* 285
Кнопка выбора *JCheckBox* 286
Радиокнопка *JRadioButton* 286
Упражнение 288
Раскрывающийся список *JComboBox* 288
Список выбора *JList* 289
Визуализация элементов списков 291
Упражнение 293
Счетчик *JSpinner* 293
Полосы прокрутки *JScrollBar* 294
Ползунок *JSlider* 295
Упражнение 296
Индикатор *JProgressBar* 296
Дерево объектов *JTree* 297
Построение меню средствами Swing 301
Строка меню *JMenuBar* 301
Меню *JMenu* 301
Пункт меню *JMenuItem* 302
Пункт меню *JCheckBoxMenuItem* 303
Пункт меню *JRadioButtonMenuItem* 304
Всплывающее меню *JPopupMenu* 304

Панель выбора цвета *JColorChooser* 305
Упражнение 307
Окно выбора файла *JFileChooser* 307
Фильтр файлов *FileFilter* 307
Как получить выбранный файл 309
Дополнительный компонент 310
Замена изображений 311
Русификация Swing 312
Вопросы для самопроверки 313

Глава 12. Текстовые компоненты 314

Компонент *JTextComponent* 314
Модель данных — документ 314
Строка символов *Segment* 315
Запись текста в документ 316
Атрибуты текста 316
Удаление текста из документа 317
Фильтрация документа 317
Внесение структуры в документ 317
События в документе 318
Реализации документа 318
Установка модели данных 319
Вид 319
Контроллер — редактор текста 321
Курсор 321
Ограничение перемещения курсора 322
Реализации редактора 323
Раскладка клавиатуры 323
Печать текста документа 324
Поле ввода *JTextField* 325
Поле ввода пароля *JPasswordField* 327
Редактор объектов *JFormattedTextField* 327
Область ввода *JTextArea* 328
Текстовый редактор *JEditorPane* 329
Редактор *JTextPane* 330
Вопросы для самопроверки 331

Глава 13. Таблицы 332

Класс *JTable* 332
Модель данных таблицы 334
Модель ячеек таблицы 334
Свойства столбца таблицы *TableColumn* 339
Модель столбцов таблицы 339
Заголовки столбцов таблицы *JTableHeader* 340

Модель выделения ячеек ... 341
Визуализация ячеек таблицы ... 342
Редактор ячеек таблицы ... 346
Сортировка строк таблицы ... 349
Фильтрация строк таблицы ... 350
Печать таблицы ... 352
Вопросы для самопроверки ... 352

Глава 14. Размещение компонентов и контейнеры Swing ... 353

Менеджер *FlowLayout* ... 353
Менеджер *BorderLayout* ... 355
Менеджер *GridLayout* ... 357
Менеджер *CardLayout* ... 359
Менеджер *GridBagLayout* ... 360
Контейнеры Swing ... 362
Панель *JPanel* ... 362
Панель прокрутки *JScrollPane* ... 363
Двойная панель *JSplitPane* ... 365
Панель с вкладками *JTabbedPane* ... 367
Линейная панель *Box* ... 369
Менеджер размещения *BoxLayout* ... 369
Компоненты-заполнители ... 369
Менеджер размещения *SpringLayout* ... 371
Размеры *Spring* ... 372
Промежутки *Constraints* ... 373
Размещение компонентов ... 373
Панель инструментальных кнопок *JToolBar* ... 375
Интерфейс *Action* ... 377
Слоеная панель *JLayeredPane* ... 378
Корневая панель *JRootPane* ... 380
Окно *JWindow* ... 381
Диалоговое окно *JDialog* ... 382
Окно верхнего уровня *JFrame* ... 383
Внутреннее окно *JInternalFrame* ... 384
Рабочий стол *JDesktopPane* ... 386
Стандартные диалоги *JOptionPane* ... 387
Окно с индикатором *ProgressMonitor* ... 392
Заключение ... 393
Вопросы для самопроверки ... 393

Глава 15. Обработка событий ... 395

Самообработка событий ... 399
Обработка вложенным классом ... 400
Упражнение ... 401

Событие *ActionEvent* 402
Обработка действий мыши 402
Упражнение 405
Классы-адаптеры 406
Управление колесиком мыши 406
Обработка действий клавиатуры 408
Упражнение 409
Событие *TextEvent* 409
Событие изменения *ChangeEvent* 409
Обработка действий с окном 410
Событие *ComponentEvent* 411
Событие *ContainerEvent* 411
Событие *FocusEvent* 412
Событие *ItemEvent* 412
Событие *AdjustmentEvent* 412
Несколько слушателей одного источника 415
Диспетчеризация событий 416
Создание собственного события 418
Вопросы для самопроверки 419

Глава 16. Оформление рамок 420

Пустая рамка *EmptyBorder* 422
Прямолинейная рамка *LineBorder* 422
Объемная рамка *BevelBorder* 423
Закругленная объемная рамка *SoftBevelBorder* 424
Врезанная рамка *EtchedBorder* 424
Рамка с изображением *MatteBorder* 424
Рамки с надписями *TitledBorder* 426
Сдвоенные рамки *CompoundBorder* 428
Создание собственных рамок 430
Вопросы для самопроверки 435

Глава 17. Изменение внешнего вида компонента 436

Получение свойств L&F 438
Задание стандартного L&F 440
Дополнительные L&F 442
Смена всего L&F 442
Замена отдельных свойств L&F 445
Темы Java L&F 447
Вопросы для самопроверки 450

Глава 18. Апплеты 451

Упражнения 457
Передача параметров в апплет 457

Параметры тега <*applet*> 460
Сведения об окружении апплета 461
Упражнение 461
Изображение и звук в апплетах 461
Слежение за процессом загрузки 462
Класс *MediaTracker* 463
Упражнения 465
Защита от апплета 465
Апплеты в библиотеке Swing 466
Апплет *JApplet* 467
Упражнение 468
Заключение 469
Вопросы для самопроверки 469

Глава 19. Прочие свойства Swing 470

Свойства экземпляра компонента 470
Прокрутка содержимого компонента 471
Передача фокуса ввода 471
Перенос данных Drag and Drop 476
Временная задержка *Timer* 478

Глава 20. Изображения и звук 480

Модель "поставщик-потребитель" 480
Классы-фильтры 483
Как выделить фрагмент изображения 484
Как изменить цвет изображения 486
Как переставить пикселы изображения 486
Упражнения 488
Модель обработки прямым доступом 488
Преобразование изображения в Java 2D 491
Аффинное преобразование изображения 491
Изменение интенсивности изображения 494
Изменение составляющих цвета 495
Создание различных эффектов 496
Упражнения 497
Анимация 498
Улучшение изображения двойной буферизацией 500
Упражнения 504
Звук 504
Проигрывание звука в Java 505
Синтез и запись звука в Java 2 510
Упражнение 513
Вопросы для самопроверки 513

ЧАСТЬ IV. НЕОБХОДИМЫЕ КОНСТРУКЦИИ JAVA 515

Глава 21. Обработка исключительных ситуаций 517

Блоки перехвата исключения 518
Упражнения 521
Часть заголовка метода *throws* 522
Оператор *throw* 524
Иерархия классов-исключений 525
Порядок обработки исключений 526
Упражнение 527
Создание собственных исключений 527
Заключение 528
Вопросы для самопроверки 528

Глава 22. Подпроцессы 530

Класс *Thread* 532
Синхронизация подпроцессов 536
Согласование работы нескольких подпроцессов 539
Приоритеты подпроцессов 543
Подпроцессы-демоны 543
Группы подпроцессов 544
Заключение 544
Вопросы для самопроверки 545

Глава 23. Потоки ввода/вывода и печать 546

Консольный ввод/вывод 551
Форматированный вывод 554
Спецификации вывода целых чисел 555
Спецификации вывода вещественных чисел 556
Спецификация вывода символов 556
Спецификации вывода строк 556
Спецификации вывода логических значений 556
Спецификации вывода хеш-кода объекта 556
Спецификации вывода даты и времени 556
Класс *Console* 557
Упражнения 558
Файловый ввод/вывод 558
Получение свойств файла 560
Буферизованный ввод/вывод 562
Каналы буферизованного ввода/вывода 563
Упражнения 564
Поток простых типов Java 564
Кодировка UTF-8 565
Класс *DataOutputStream* 565

Прямой доступ к файлу 566
Упражнение 567
Каналы обмена информацией 567
Сериализация объектов 569
Печать в Java 572
Печать средствами Java 2D 574
Печать файла 578
Печать страниц с разными параметрами 580
Вопросы для самопроверки 582

Глава 24. Сетевые средства Java 583

Работа в WWW 586
Упражнения 589
Работа по протоколу TCP 590
Работа с proxy-сервером 594
Упражнения 594
Работа по протоколу UDP 594
Упражнение 597
Вопросы для самопроверки 597

Часть V. Web-технологии Java 599

Глава 25. Web-инструменты Java 601

Архиватор *jar* 601
Создание архива 602
Файл описания MANIFEST.MF 604
Файл INDEX.LIST 605
Компоненты JavaBeans 606
Связь с базами данных через JDBC 607
Вопросы для самопроверки 612

Глава 26. Сервлеты 613

Web-приложение 614
Интерфейс *Servlet* 615
Конфигурационный файл 615
Интерфейс *ServletConfig* 618
Контекст сервлета 620
Метод *Service* 621
Интерфейс *ServletRequest* 621
Интерфейс *ServletResponse* 623
Цикл работы сервлета 623
Класс *GenericServlet* 624

Работа по протоколу HTTP 625
Интерфейс *HttpServletRequest* 625
Интерфейс *HttpServletResponse* 626
Класс *HttpServlet* 627
Сеанс связи с сервлетом 634
Фильтры 637
Обращение к другим ресурсам 641
Вопросы для самопроверки 642

Глава 27. Страницы JSP 643

Стандартные действия (теги) JSP 646
Язык записи выражений EL 649
Встроенные объекты JSP 650
Обращение к компоненту JavaBean 652
Выполнение апплета в браузере клиента 654
Передача управления 655
Пользовательские теги 655
Класс-обработчик пользовательского тега 657
Пользовательский тег с атрибутами 660
Пользовательский тег с телом 660
Обработка тела пользовательского тега 662
Обработка взаимодействующих тегов 664
Обработка исключений в пользовательских тегах 668
Обработка тегов средствами JSP 669
Стандартные библиотеки тегов JSTL 671
Библиотека core 671
Библиотека xml 675
Библиотека fmt 675
Библиотека sql 675
Библиотека fn 676
Frameworks 676
JavaServer Faces 677
Вопросы для самопроверки 681

Глава 28. Связь Java с технологией XML 683

Описание DTD 688
Пространства имен XML 690
Схема XML 692
Встроенные простые типы XSD 693
Вещественные числа 693
Целые числа 693
Строки символов 693
Дата и время 694

Двоичные типы ... 694
Прочие встроенные простые типы ... 694
Определение простых типов ... 694
Сужение ... 695
Список ... 696
Объединение ... 697
Описание элементов и их атрибутов ... 698
Определение сложных типов ... 698
Определение типа пустого элемента ... 699
Определение типа элемента с простым телом ... 699
Определение типа вложенных элементов ... 700
Определение типа со сложным телом ... 702
Пример: схема адресной книги ... 703
Безымянные типы ... 705
Пространства имен языка XSD ... 707
Включение файлов схемы в другую схему ... 709
Связь документа XML со своей схемой ... 711
Другие языки описания схем ... 711
Инструкции по обработке ... 712
Анализ документа XML ... 712
Анализ документов XML с помощью SAX2 ... 713
Связывание данных XML с объектами Java ... 720
Объекты данных JDO ... 721
Анализ документов XML с помощью DOM API ... 722
Интерфейс *Node* ... 723
Интерфейс *Document* ... 725
Интерфейс *Element* ... 726
Другие DOM-парсеры ... 729
Преобразование дерева объектов в XML ... 729
Таблицы стилей XSL ... 731
Преобразование документа XML в HTML ... 734
Вопросы для самопроверки ... 735

Список литературы ... 737

Предметный указатель ... 739

Введение

Книга, которую вы держите в руках, возникла из курса лекций, читаемых автором в течение последних лет студентам младших курсов. Подобные книги рождаются после того, как студенты в очередной раз зададут 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, последних "компьютерщики" называют *FAQ* (Frequently 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 — выделяет особенности языка Java среди других языков;
- предлагает обсуждение вопросов русификации Java.

Прочитав эту книгу, вы вступите в ряды программистов на Java — разработчиков передовой 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.

Прошло несколько лет с того дня, когда были написаны эти строки. Все случилось так, как я и написал. Разошлось два издания книги. Я видел много ее экземпляров в самом разном состоянии. Читатели высказали мне множество нелицеприятных соображений по поводу содержания книги, обнаруженных ошибок и опечаток. Студенты на зачетах и экзаменах пересказывали мне целые куски книги, что тоже наводило на размышления по поводу ее содержания и стиля изложения. У меня накопилось много дополнительного материала, который так и просился в книгу.

Технология Java развивается очень быстро. Со времени выхода первого издания Java прочно утвердилась на Web-серверах, проникла в сотовые телефоны и другие мобильные устройства. Теперь Java — обязательная часть Web-программирования.

Развивается и сам язык. В него вводятся новые конструкции, появляются новые библиотеки классов. Графическая библиотека Swing стала частью стандартной поставки Java. В стандартную поставку теперь включены и средства работы с документами XML.

Все это привело к необходимости сделать третье издание, дополнив книгу новым материалом.

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 сотовые телефоны и другие мобильные устройства.

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* (servlets), страницы *JSP* (JavaServer Pages) и *EJB* (Enterprise JavaBeans). Серверы должны взаимодействовать с базами данных — появились драйверы *JDBC*. 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.

Язык Java часто используется для описания различных приемов объектно-ориентированного программирования, так же как для записи 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 и по исходным кодам свободно распространяемых программных продуктов.

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 пяти частей.

Часть I 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.

В *главе 3* определяются пакеты классов и интерфейсы, ограничения доступа к классам и методам, на примерах подробно разбираются правила их использования. Объясняется структура встроенной библиотеки классов Java API.

В *части II* 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 всё в Java, являются объектами, приводятся примеры синтаксического анализа текстов, обсуждаются вопросы русификации.

В *главе 6* показано, как в языке Java реализованы коллекции, 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 *части III* объясняется создание графического интерфейса пользователя (ГИП) с помощью стандартной библиотеки классов AWT (Abstract Window Toolkit) с компонентами Swing 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 с помощью Swing 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 компонентов Swing, ставшая 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 графического компонента, рассматриваются готовые компоненты AWT и их применение, а также создание собственных компонентов AWT.

В *главе 11* рассматриваются графические компоненты общего назначения, относящиеся к библиотеке Swing.

В *главе 12* рассматриваются текстовые графические компоненты библиотеки Swing.

В *главе 13* подробно обсуждаются возможности создания таблиц средствами Swing.

В *главе 14* показано, какие способы размещения компонентов в графическом контейнере имеются в AWT и Swing и как их применять в разных ситуациях.

В *главе 15* вводятся способы реагирования компонентов на сигналы от клавиатуры и мыши, а именно модель делегирования, принятая в Java.

В *главе 16* описывается создание рамок, окружающих графические компоненты Swing.

В *главе 17* обсуждается интересная способность Swing изменять свой внешний вид, сливаясь с окружающей графической средой или, наоборот, выделяясь из нее.

В *главе 18*, наконец-то, появляются апплеты — Java-программы, предназначенные для выполнения в окне браузера, и обсуждаются особенности их создания.

В *главе 19* собраны сведения о библиотеке Swing, не вошедшие в предыдущие главы.

В *главе 20* рассматривается работа с изображениями и звуком средствами AWT.

В *части IV* 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 21 описывает встроенные в Java средства обработки исключительных ситуаций, возникающих во время выполнения готовой программы.

Глава 22 рассказывает об интересном свойстве языка Java — способности создавать подпроцессы (threads) и управлять их взаимодействием прямо из программы.

В *главе 23* обсуждается концепция потока данных и ее реализация в Java для организации ввода/вывода на внешние устройства.

Глава 24 рассматривает сетевые средства языка Java, позволяющие скрыть все сложности протоколов Интернета и максимально облегчить написание клиент-серверных и распределенных приложений.

Часть V книги посвящена Web-технологии Java, точнее, тех ее разделов, которые касаются программирования серверов.

В *главе 25* описываются те аспекты технологии Java, которые необходимы для Web-программирования: архиватор JAR, компоненты JavaBeans, драйверы соединения с базами данных JDBC.

Глава 26 посвящена основному средству программирования серверов — сервлетам.

В *главе 27* разбираются страницы JSP, значительно облегчающие оформление ответов на запросы Web-клиентов.

Наконец, в *главе 28* рассматривается вездесущая технология XML и инструменты Java для обработки документов XML.

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 скомпилировать, т. е. перевести в последовательность 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 (Java Virtual Machine, JVM). *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* (Virtual Machine Specification, VMS). Ознакомьтесь с этой спецификацией, если вы хотите в точности узнать, как работает виртуальная машина Java.

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 скомпилированной 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 конкретного компьютера. Она может быть выполнена один раз сразу же после написания программы, программу не надо перекомпилировать под разные платформы. 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. Корпорация Sun Microsystems разработала целую технологию HotSpot и включает ее в свою виртуальную машину Java. Но, конечно, наибольшую скорость может дать только специализированный процессор.

Компания 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.

Программы, приведенные в этой книге, выполнялись в операционных средах программирования MS Windows 2000/XP/Server 2003, Red Hat Linux, Fedora Core Linux, SUSE Linux без

перекомпиляции. Это видно по рисункам, приведенным во многих главах книги. Они "сняты" с экранов графических оболочек разных операционных систем.

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, написанные чаще всего на языке C/C++ и скомпилированные под определенную платформу, называют *"родными" методами* (native methods). Применение "родных" методов ускоряет выполнение программы.

Корпорация 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. Называется этот набор JDK (Java Development Kit). Есть наборы инструментальных программ и других фирм. Например, большой популярностью пользуется JDK корпорации IBM.

Что такое JDK?

Набор программ и классов JDK содержит:

- компилятор из исходного текста в байт-коды `javac`;
- 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 и генерирования документации `javadoc`;
- программу генерации заголовочных файлов языка C для создания "родных" методов `javah`;
- программу добавления электронной подписи `javakey`;
- 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 на диске около 8 Мбайт.

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 — в распакованном.

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 было 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 Мбайт 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 в литературе утвердилось название Java 2. Кроме 57 пакетов классов, обязательных на любой платформе и получивших название *Core API*, в Java 2 JDK 1.2 входят еще дополнительные пакеты классов, называемые Standard Extension API.

В версии J2SE JDK 1.5.0, вышедшей в конце 2004 г., было уже под сотню пакетов, составляющих Core API (Application Programming Interface). В упакованном виде — это файл размером около 46 Мбайт и необязательный файл с упакованной документацией такого же размера. J2SE JDK создается для каждой платформы: MS Windows, Solaris, Linux, отдельно, а документация написана на языке HTML и одинакова на всех платформах. Поэтому она записана в отдельном файле. Например, для MS Windows файл с J2SE JDK 1.5.0 называется jdk-1_5_0-windows-i586.exe, а файл с документацией называется jdk-1_5_0-doc.zip.

Все это распаковывается в несколько сотен мегабайт дискового пространства. Данная версия требует процессор Pentium III и выше и не менее 64 Мбайт оперативной памяти.

В это же время произошло очередное переименование технологии Java: из версии убрали первую цифру и стали писать *Java 2 Platform, Standard Edition 5.0*, сокращенно J2SE 5.0 и JDK 5.0, хотя во внутрифирменной документации сохраняется название JDK 1.5.0.

На время выхода этого издания книги версия J2SE 5.0 продолжала развиваться, вышло обновление JDK 1.5.0_13, но уже появилась новая, шестая версия.

В шестой версии, вышедшей в начале 2007 г., из названия технологии убрали цифру 2. Теперь пишут *Java Platform, Standard Edition 6,* сокращенно — Java SE 6 и JDK 6. Впрочем, во внутрифирменной документации остается прежнее обозначение, например третье обновление обозначается JDK 1.6.0_03.

Для создания Web-программ в *части V* книги вам потребуется еще набор пакетов *Java Platform, Enterprise Edition* (Java EE). Так же как Java SE, он поставляется одним самораспаковывающимся архивом, в который входит SDK (Software Development Kit), Java EE API и

сервер приложений. Архив можно скопировать с того же сайта. Набор Java EE SDK устанавливается после Java SE JDK. Впрочем, на том же сайте есть полная версия архива, содержащая в себе Java SE JDK.

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 — облегченный интерпретатор `jre`) и библиотеку классов. Это часть JDK, не содержащая компиляторы, отладчики и другие средства разработки. Именно Sun JRE или его аналог, созданный другими фирмами, присутствует в тех браузерах, которые умеют выполнять программы на Java, в операционных системах и системах управления базами данных.

Хотя JRE входит в состав JDK, корпорация Sun распространяет этот набор и отдельным файлом.

Как установить JDK?

Набор JDK упаковывается в самораспаковывающийся архив. Раздобыв каким-либо образом этот архив: скачав из Интернета, с сайта **http://java.sun.com/javase/downloads/** 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 каталог (directory) установки, например, /usr/java/jdk1.6.0_03. Каталог и его название можно поменять, место и название установки не имеют значения.

После установки вы получите каталог с названием, например, jdk1.6.0_03, а в нем подкаталоги:

- bin с исполнимыми файлами;
- db с небольшой базой данных;
- demo с примерами программ;
- docs с документацией, если вы ее установили в этот каталог;
- include с заголовочными файлами "родных" методов;
- jre с набором JRE;
- lib с библиотеками классов и файлами свойств;
- sample с примерами программ;
- src с исходными текстами программ JDK, получаемый после распаковки файла src.zip.

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.

Предупреждение

Не следует распаковывать zip- и jar-архивы, кроме архива исходных текстов src.zip.

После установки надо дополнить значение системной переменной `PATH`, добавив в нее путь к каталогу bin, например `/usr/java/jdk1.6.0_03/bin`. Некоторые программы, использующие Java, требуют определить и специальную переменную окружения `JAVA_HOME`, содержащую путь к каталогу установки JDK, например `/usr/java/jdk1.6.0_03`.

Проверить правильность установки Java, а заодно и посмотреть ее версию можно, набрав в командной строке

```
java -version
```

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/Server 2003/XP. В системах UNIX, Linux, BSD 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, а не графическом 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.

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 появится вывод результатов работы программы или сообщения об ошибках времени выполнения.

Работая в графических оболочках операционных систем, мы привыкли вызывать программу на исполнение двойным щелчком мыши по имени исполнимого файла (в MS Windows у имени исполнимого файла стандартное расширение exe) или щелчком по его ярлыку. В тех-

нологии Java тоже есть такая возможность. Надо только упаковать class-файлы с байт-кодами в архив специального вида JAR. Как это сделать, рассказано в *главе 25*. При установке JDK на MS Windows для файлов с расширением jar автоматически создается ассоциация с интерпретатором java, который будет вызван при двойном щелчке мыши на jar-архиве.

Кроме того, можно написать командный файл (файл с расширением bat в MS Windows или Shell-файл командной оболочки в UNIX), записав в нем строку вызова интерпретатора java со всеми нужными параметрами.

Еще один способ запустить Java-программу средствами операционной системы — написать загрузчик (launcher) виртуальной машины Java. Так и сделано в стандартной поставке JDK: исполнимый файл java.exe содержит программу, написанную на языке C, которая запускает виртуальную машину Java и передает ей на исполнение класс Java с методом `main()`. Исходный текст этой программы есть среди исходных текстов Java в каталоге src/launcher. Им можно воспользоваться для написания своего загрузчика. Есть много программ, облегчающих написание загрузчика, например программа Java Launcher фирмы SyncEdit, **http://www.syncedit.com/software/javalauncher/**, или Advanced Installer фирмы Caphyon, **http://www.advancedinstaller.com/**.

Наконец, существуют компиляторы исходного текста, написанного на языке Java, непосредственно в исполнимый файл операционной системы, с которой вы работаете. Их общее название AOT (Ahead-Of-Time) compiler. Например, у знаменитого компилятора GCC (GNU Compiler Collection) есть вход с именем GCJ, с помощью которого можно сделать компиляцию как в байт-коды, так и в исполнимый файл, а также перекомпиляцию байт-кодов в исполнимый файл.

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 IDE (Integrated Development Environment) для Java, и их число все время возрастает. Некоторые из них, такие как Eclipse, IntelliJ IDEA, NetBeans, 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 JBuilder фирмы CodeGear или IBM Rational Application Developer. 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.

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 корпорации. Дело в том, что, имея свою операционную систему, огромное число приложений к ней и богатейшую библиотеку классов, 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, входящее в Visual Studio. Данный 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 корпорации Microsoft. В результате судебных разбирательств с Sun Microsystems компания Microsoft назвала свой продукт Visual J++. Виртуальная машина Java корпорации Microsoft умеет выполнять байт-коды, созданные "чистым" компилятором, но не всякий интерпретатор выполнит байт-коды, написанные с помощью Visual J++. Сейчас этот продукт вошел в состав Visual Studio .NET под названием J# (J sharp).

Чтобы прекратить появление несовместимых версий Java, корпорация 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, компания Sun распространяет пакет программ Java Plug-in, который можно подключить к браузеру, заменив тем самым встроенный в браузер JRE на "родной".

Java в Интернете

Разработанная для применения в сетях, Java просто не могла не найти отражения на сайтах Интернета. Действительно, масса сайтов полностью посвящена технологии Java или содержит информацию о ней. Одна только компания Sun имеет 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://developers.sun.com/** — здесь есть масса полезных вещей для разработчика;
- **http://forum.java.sun.com/** — форумы для разработчиков Java;
- **http://www.java.net/** — сайт для разработчиков, знакомящихся с технологией Java.

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

Корпорация Microsoft содержит информацию о Java на сайте **http://www.microsoft.com/mscorp/java/default.mspx**.

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/forums/** — форумы для разработчиков, в том числе Java;
- **http://www.blackdown.com/** — Java для Linux;
- **http://www.developer.com/java/** — большой сборник статей по Java;
- **http://www.freewarejava.com/** — советы разработчикам Java и готовые программы;
- **http://www.jars.com/** — Java Review Service;
- **http://www.javable.com/** — новостной сайт с русскими статьями, посвященный Java;
- **http://javaboutique.internet.com/** — еще один новостной сайт;
- **http://www.javalobby.com/** — новости, статьи и советы по Java;
- **http://www.javaranch.com/** — дружественный сайт и форум для разработчиков Java;
- **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://java.sys-con.com/** — новинки технологии Java;
- **http://www.theserverside.com/** — вопросы создания серверных Java-приложений;
- **http://www.codeguru.com/Java/** — большой сборник статей, апплетов и других программ;
- **http://securingjava.com/** — здесь обсуждаются вопросы безопасности;
- **http://www.servlets.com/** — здесь обсуждаются вопросы написания сервлетов;
- **http://www.javacats.com/** — общая информация о Java и не только о Java.

Персональные сайты:

- **http://www.bruceeckel.com/** — сайт Брюса Эккеля, автора популярных книг и статей;
- **http://www.davidreilly.com/** — сайт Девида Рейли, автора многих статей и книг о 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 приведено в конце книги.

Полное и строгое описание языка изложено в книге *James Gosling, Bill Joy, Guy Steele, Gilad Bracha, "The Java Language Specification, Third Edition"*. В электронном виде она находится по адресу **http://java.sun.com/docs/books/jls/**, занимает в упакованном виде около 400 Кбайт.

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

Здесь же необходимо отметить книгу "отца" технологии Java Джеймса Гослинга, написанную вместе с Кеном Арнольдом и Девидом Холмсом. Имеется русский перевод: Арнольд К., Гослинг Дж., Холмс Д. Язык программирования Java. 3-е изд.: Пер. с англ. — М.: Издательский дом "Вильямс", 2001. — 624 с.: ил.

Официальным учебником хорошего стиля программирования на языке Java стала книга Блоха Д., Java. Эффективное программирование. Пер. с англ. — М.: Лори, 2002. — 224 с.

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

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

Благодарности

Я 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; Игорю Шишигину, настоявшему на третьем издании книги, моим студентам с их бесконечными вопросами; своим "сплюснутым" друзьям, 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

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 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", не можем от этого уйти и мы.

Все правила языка Java исчерпывающе изложены в его спецификации, сокращенно называемой JLS (Java Language Specification), местоположение которой указано во *введении*. 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 приведена подобная программа. Она написана в самом простом виде, какой только возможен 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, только пять строчек! Но даже на этом простом примере можно заметить целый ряд существенных особенностей языка Java.

- Всякая программа, написанная на языке Java, 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 это имя `HelloWorld`. Все, что содержится в классе, записывается в фигурных скобках и составляет *тело класса* (class body).
- Все действия в программе производятся с помощью методов обработки информации, коротко говорят просто *метод* (method). Методы используются в объектно-ориентированных языках вместо функций, применяемых в процедурных языках.
- Методы различаются по именам и параметрам. Один из методов обязательно должен называться `main`, с него начинается выполнение программы. В нашей простейшей программе только один метод, а значит, имя его `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. Так и есть 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 *параметры* (parameters) метода. Для каждого параметра указывается его тип и, через пробел, имя. У метода `main()` только один параметр, его тип — массив, состоящий из строк символов. Строка символов — это встроенный в Java API тип `String`, а квадратные скобки — признак массива. Имя параметра 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 говоря, необязательны, но для метода `main()` они необходимы.

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

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

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

Составное имя `System.out.println` означает, что в классе `System`, входящем в Java API, определяется переменная с именем `out`, содержащая экземпляр 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 заданного ему аргумента в выходной поток, связанный обычно с выводом на экран текстового терминала, в окно **MS-DOS**

Prompt, **Command Prompt** или **Xterm** в зависимости от вашей системы. После вывода курсор переходит на начало следующей строки экрана, на что указывает окончание `ln`, само 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.

Сильное влияние языка C привело к появлению в Java SE 5 (Java Standard Edition) метода `System.out.printf()`, очень похожего на одноименную функцию языка C. Мы подробно опишем этот метод в *главе 23*, но желающие могут ознакомиться с ним прямо сейчас.

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` — со строчной. Но внутри текстовой константы неважно, писать `Century` или `century`, компилятор вообще не "смотрит" на текст в кавычках, разница будет видна только на экране.

> ***Замечание***
>
> Язык Java различает прописные и строчные буквы.

В именах нельзя оставлять пробелы. 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/**. Вот несколько пунктов этого соглашения:

- имена классов начинаются с прописной (заглавной) 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 с прописной 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 пустячный 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 в Блокноте (Notepad), emacs или vi. Теперь ее надо сохранить в файле в текстовом, но не в графическом формате. Имя файла должно в точности совпадать с именем класса, содержащего метод `main()`. Данное правило очень желательно выполнять. При этом система исполнения Java будет быстро находить метод `main()` для начала работы, просто отыскивая класс, совпадающий с именем файла. Расширение имени файла должно быть java.

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 байт-кодов, передав ему в качестве аргумента имя класса (а не файла!):

```
java HelloWorld
```

На экране появится строка:

```
Hello, XXI 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 2003.

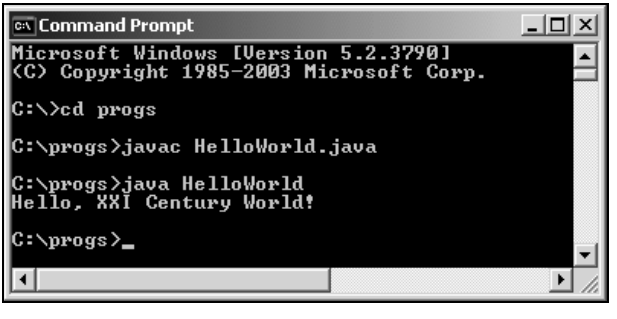


Рис. 1.1. Окно **Command Prompt**

При работе в какой-либо интегрированной среде, например Eclipse или NetBeans, 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 гово-

рят программисты, "закомментировав" их. Кроме того, некоторые программы, работающие с Java, извлекают из комментариев полезные для себя сведения.

Комментарии вводятся таким образом:

- за двумя наклонными чертами, написанными 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);
- за наклонной чертой и двумя звездочками `/**` начинается комментарий, который может занимать несколько строк, до звездочки и наклонной черты `*/`. Из таких комментариев формируется документация.

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 и введены комментарии третьего типа, а в состав JDK включена утилита — программа `javadoc`, извлекающая эти комментарии в отдельные файлы формата HTML и создающая гиперссылки между ними. В такой комментарий кроме собственно комментария 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, XXI Century World!");
    // Следующий вызов закомментирован,
    // метод не будет выполняться
    // System.out.println("Farewell, XX 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.

Аннотации

Обратите внимание на комментарий, приведенный в начале листинга 1.2. В него вставлены указания-теги `@author` и `@version` утилите `javadoc`. Просматривая текст этого комментария и встретив какой-либо из тегов, утилита `javadoc` выполнит предписанные тегом действия. Например, тег `@see` предписывает сформировать гиперссылку на другой документ HTML, а тег `@deprecated`, записанный в комментарий перед методом, вызовет пометку этого метода в документации как устаревшего.

Идея давать утилите предписания с помощью тегов оказалась весьма плодотворной. Кроме `javadoc` были написаны другие утилиты и целые программные продукты, которые вводят новые теги и используют их для своих целей. Например, программа XDoclet может автоматически создавать различные конфигурационные файлы, необходимые для работы сложных приложений. Разработчику достаточно вставить в свою программу комментарии вида `/**...*/` с тегами специального вида и запустить утилиту Xdoclet, которая сгенерирует все необходимые файлы.

Использование таких утилит стало общепризнанной практикой, и, начиная с пятой версии Java SE, было решено ввести прямо в компилятор возможность обрабатывать теги, которые получили название *аннотаций*. Аннотации записываются не внутри комментариев вида `/**...*/`, а непосредственно в том месте, где они нужны. Например, после того как мы запишем непосредственно перед заголовком какого-либо метода аннотацию `@Deprecated`, компилятор будет выводить на консоль предупреждение о том, что этот метод устарел и его следует заменить.

Несколько аннотаций, количество которых увеличивается с каждой новой версией JDK, объявлено прямо в компиляторе. Ими можно пользоваться без дополнительных усилий. Мы будем вводить их по мере надобности. Кроме них разработчик может объявить и использовать в своем приложении свои аннотации. Как это делается, рассказано в *главе 3*.

Константы

В языке Java можно записывать константы различных типов в разных видах. Форма записи констант почти полностью заимствована из языка C. Перечислим все разновидности констант.

Целые

Целые константы можно записывать в трех системах счисления:

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

  Замечание

  Целое 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 оперативной памяти в формате типа `int` (см. далее).

В конце целой константы можно записать латинскую букву "Эл" (прописную `L` или строчную `l`), тогда константа будет сохраняться в длинном формате типа `long` (см. далее): `+25L`, `-037l`, `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 оперативной памяти в формате типа `float` (см. далее): `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

Одиночные символы записываются в апострофах, чтобы отличить их от имен переменных. Для записи символов используются следующие формы:

- печатные символы, записанные на клавиатуре, просто записываются в апострофах (одинарных кавычках): `'a'`, `'N'`, `'?'`;
- управляющие и специальные символы записываются в апострофах с обратной наклонной чертой, чтобы отличить их от обычных символов:
  - `'\n'` — символ перевода строки LF (Line Feed) с кодом ASCII 10;
  - `'\r'` — символ возврата каретки CR (Carriage Return) с кодом 13;
  - `'\f'` — символ перевода страницы FF (Form Feed) с кодом 12;
  - `'\b'` — символ возврата на шаг BS (Backspace) с кодом 8;
  - `'\t'` — символ горизонтальной табуляции HT (Horizontal Tabulation) 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 счисления в апострофах после обратной наклонной черты: `'\123'` — буква `S`, `'\346'` — буква `Ж` в кодировке CP1251. Нет

смысла использовать эту форму записи для печатных и управляющих символов, перечисленных в предыдущем пункте, поскольку компилятор сразу же переведет восьмеричную запись в указанную ранее форму. Наибольший восьмеричный код `'\377'` — десятичное число 255;

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

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

> ***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\nc переносом"
"\"Зубило\" — Чемпион!"
```

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

Для строковых констант определена операция сцепления, обозначаемая плюсом. Запись

```
"Сцепление " + "строк"
```

дает в результате строку `"Сцепление строк"`. Обратите внимание на то, что между сцепляемыми строками не вставлены никакие дополнительные символы. Пробел между ними принадлежал первой строке.

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 98/ME вообще не работает с Unicode, а Windows 2000/XP/Server 2003/Vista использует для вывода в окно **Command Prompt** шрифт Terminal, в котором буквы кириллицы 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* (identifiers)) 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 набор букв Java обязательно входят прописные и строчные латинские буквы, знак доллара (`$`) и знак подчеркивания (_), а также 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.

Придумывая имена, 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: метод `isJavaIdentifierStart()`, проверяющий, является ли символ буквой Java, и метод `isJavaIdentifierPart()`, выясняющий, является ли символ буквой, цифрой, знаком подчеркивания (_) или знаком доллара (`$`).

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 включают *массивы* (arrays), *классы* (classes) и *интерфейсы* (interfaces). В Java SE 5 появился *перечислимый тип* (enum).

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

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

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

Символы можно применять везде, где используется тип `int`, поэтому JLS причисляет тип `char` к целым типам. Например, символы 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"`, где буква Ж записана, как строка, в кавычках, плюс понимается как сцепление строк, двойка будет преобразована в строку, в результате получится строка `"2Ж"`.

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

На рис. 1.2 показана иерархия типов данных Java.

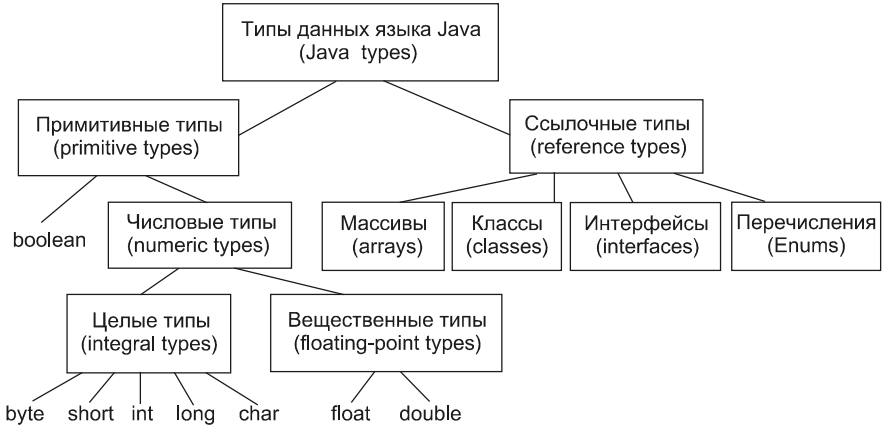


Рис. 1.2. Типы данных языка Java

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, относящихся к этому типу. Имена в списке разделяются запятой. Для всех или некоторых переменных можно ука-

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

зать начальные значения после знака равенства, которыми 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.

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 данного 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

В языке Java четыре логические операции:

- отрицание (NOT) — `!` (обозначается восклицательным знаком);
- конъюнкция (AND) — `&` (амперсанд);
- дизъюнкция (OR) — `|` (вертикальная черта);
- исключающее ИЛИ (XOR) — `^` (каре).

Они выполняются над логическими данными типа `boolean`, их результатом будет тоже логическое значение — `true` или `false`. Про эти операции 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 очень ловко используется программистами, 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.

Упражнения

1. Для переменных `b` и `bb`, определенных в *разд. "Логический тип" данной главы*, найдите значение выражения `b & bb && !bb | b`.
2. При тех же определениях вычислите выражение `(!b || bb) && (bb ^ b)`.

Целые типы

Спецификация языка Java, JLS, определяет разрядность (количество байтов, выделяемых для хранения значений типа в оперативной памяти) каждого типа. Для целых типов она приведена в табл. 1.2. В таблице указан также диапазон значений каждого типа, получаемый на процессорах архитектуры Pentium.

Таблица 1.2. *Целые типы*

Тип	Разрядность (байт)	Диапазон
`byte`	1	От –128 до 127
`short`	2	От –32 768 до 32 767
`int`	4	От –2 147 483 648 до 2 147 483 647
`long`	8	От –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
`char`	2	От `'\u0000'` до `'\uFFFF'`, в десятичной форме от 0 до 65 535

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

Вот примеры определения переменных целых типов:

```
byte b1 = 50, b2 = -99, b3;
short det = 0, ind = 1, sh = 'd';
int i = -100, j = 100, k = 9999;
long big = 50, veryBig = 2147483648L;
char c1 = 'A', c2 = '?', c3 = 36, newLine = '\n';
```

Целые типы, кроме `char`, 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 код получается так: в двоичном представлении числа 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 ранее, будет храниться в одном байте с содержимым 00110010, а значение –99 переменной `b2` — в байте с содержимым, которое вычисляется так: число 99 переводится в двоичную форму, получая 01100011, меняются единицы и нули, получая 10011100, и прибавляется единица, получая окончательно байт с содержимым 10011101.

Смысл всех этих преобразований в том, что сложение числа с его дополнительным кодом в двоичной арифметике даст в результате нуль; старший бит, равный 1, просто теряется, поскольку выходит за разрядную сетку. 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, слеш);
- 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 *целочисленное 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 вещественных 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 ранее 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 ранее 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 скомпилировать простую программу, представленную в листинге 1.3, приведет к сообщениям компилятора, показанным на рис. 1.3.

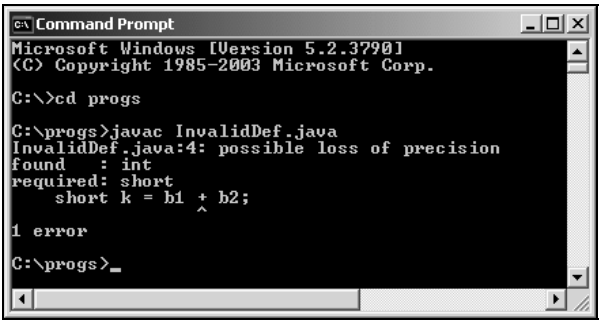


Рис. 1.3. Сообщения компилятора об ошибке

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).

В подобных ситуациях 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 и никаких сообщений об этом не появляется.

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`, например в результате 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, как говорят, "наложением маски".

В языке Java 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

Таблица 1.3 *(окончание)*

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

В нашем примере число `b1 == 50`, его двоичное представление `00110010`, число `b2 == -99`, а его двоичное представление равно `10011101`. Перед операцией происходит повышение типа `byte` до типа `int`. Получаем представления из 32 разрядов для `b1` — `0...00110010`, а 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 разности `(-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 число — 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.

Упражнения

3. Каково значение выражения `'D' + 5`?

При определениях, сделанных ранее, вычислите выражения:

4. `(b1 + c1) % (++b2 / b1++)`.
5. `(b1 < c1) && (b2 == -99) || (ind >= 0)`.
6. `(b1 | c1) & (big ^ b1)`.
7. `(b1<<3 + c1<<2) % (b2>>5 / b1>>>2)`.

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:

- положительная бесконечность, выражаемая константой `POSITIVE_INFINITY` и возникающая при переполнении положительного значения, например в результате операции умножения `3.0*6e307` или при делении на нуль;
- отрицательная бесконечность `NEGATIVE_INFINITY`, возникающая при переполнении отрицательного значения, например в результате операции умножения `-3.0*6e307` или при делении на нуль отрицательного числа;
- "не число", записываемое константой `NaN` (Not a Number) и возникающее, например, при 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 в результате истину, `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 байта	$3{,}4\times10^{-38} < \lvert x\rvert < 3{,}4\times10^{38}$	7—8 цифр в дробной части
`double`	8 байтов	$1{,}7\times10^{-308} < \lvert x\rvert < 1{,}7\times10^{308}$	17 цифр в дробной части

Примеры определения вещественных типов:

```
float x = 0.001f, y = -34.789F;
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`;
- иначе, 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` *операция* `= a`

эквивалентно

`x = (`*тип*`x)(x` *операция* `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 + 10` 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`.

Упражнения

8. Чему равно выражение `x = y = z = 1`?
9. Что получится в результате присваиваний `x += y -= z /= x + 2`?

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.

Упражнения

10. Каков смысл операции `x > 0 ? x : -x`?
11. Что дает в результате операция `x > y ? x : y`?
12. Что получится в результате операции `x > y ? y : x`?

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.

- 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)`.
- Левый операнд вычисляется раньше правого.
- Операнды полностью вычисляются перед выполнением операции.
- 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 выражений.

> ***Знатокам C/C++***
>
> Большинство компиляторов языка C++ во всех этих случаях вычислят значение 8.

Четвертое правило можно продемонстрировать так. При тех же определениях переменных `a` и `b` в результате вычисления выражения

```
b += a += b += 7
```

получим 20. Хотя операции присваивания выполняются справа налево и после первой, самой 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, т. е. 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.

Всякий язык программирования должен иметь средства записи алгоритмов. Они называются *операторами* (statements) языка. Минимальный набор операторов 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 (были рассмотрены ранее);
- операторы-выражения;
- операторы присваивания;
- условный оператор if;
- три оператора цикла while, do-while, for;
- оператор варианта switch;
- операторы перехода break, continue и return;
- блок, выделяемый фигурными скобками {};
- 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 это имеет смысл 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 применяются 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 как операция.

Условный оператор

Условный оператор (if-then-else statement) предназначен для организации разветвлений в программе. На языке Java он записывается так:

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

и действует следующим образом. Сначала вычисляется логическое выражение *`логВыр`*. Если результат вычисления `true`, то действует *`оператор1`* и на этом работа условного оператора завершается. *`оператор2`* не действует, далее будет выполняться оператор, следующий за оператором `if`. Это так называемая "ветвь `then`" условного оператора. Если результат логического выражения `false`, то действует *`оператор2`*, при этом *`оператор1`* вообще не выполняется ("ветвь `else`").

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 опять-таки 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 предыдущий условный оператор лучше записать так:

```
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 вычисления квадратного корня `sqrt()` из вещественного числа, взятые из входящего в Java API класса `Math`. Поскольку все вычисления с вещественными числами производятся приближенно, не следует ожидать, что вещественное число будет точно равно нулю. Поэтому 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, записанное в условном операторе.

"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 он 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? Загляните в *главу 23*. Но это еще не все, нужно обработать исключительные ситуации, которые могут возникнуть при вводе *(см. главу 21)*.

Упражнения

13. Вычислите с помощью условного оператора значение `y`, равное `x + 1`, если `x < 0`, равное `x + 2`, если `0<=x < 1`, и равное `x + 10` в остальных случаях.
14. Запишите условный оператор, дающий логической переменной `z` значение `true`, если точка `M(x, y)` лежит в единичном круге с центром в начале координат, и значение `false` в противном случае.

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 условия выполнения цикла 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 C. 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 сделано 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 клавиш <Ctrl>+<C> в UNIX или через окно **Task Manager** в Windows NT/2000/XP/Server 2003/Vista.

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*. Да-

лее снова проверяется *логВыр*. Если оно истинно, 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 переменной обязательно с начальным значением. Такие переменные известны лишь 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 даже 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 количество 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.

В языке Java есть еще одна форма оператора `for`, так называемый оператор "for-each", который используется для перебора элементов массивов и коллекций. Мы познакомимся с ним в соответствующих разделах книги.

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 данную 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. Это затрудняет чтение программы, поэтому не стоит увлекаться оператором `break` с меткой.

Упражнения

15. Напишите цикл, вычисляющий факториал заданного натурального числа.
16. Напишите цикл, определяющий, какая наибольшая степень числа 2 содержится среди делителей заданного натурального числа.

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 один из вариантов, а далее выполняются все оставшиеся варианты 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`.

С другой стороны, 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("Рабочий день"); break;
  case 6: case 7:
      System.out.println("Выходной день"); break;
  default:
      System.out.println("Неправильно задан день недели");
}
```

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 ссылочного типа, 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, если массив объявляется 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];
```

При этом все элементы массива получают нулевые значения.

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. Интерпретатор Java в таком случае прекратит выполнение программы и выведет на консоль сообщение о выходе индекса массива за границы его определения.

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` будут указывать на один и тот же массив из 100 вещественных переменных типа `double` и содержать 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`. Ее значение равно 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.

С помощью константы `length` 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 = 0; 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. Заметьте, что цикл можно было бы начать с 1.

Ситуация, когда надо перебрать все элементы массива в порядке возрастания их индексов, как в предыдущем примере, встречается очень часто. Начиная с версии Java SE 5 для таких случаев в язык Java введена упрощенная форма оператора цикла `for`, так называемый оператор "for-each", уже упоминавшийся ранее. Вот как можно записать предыдущий пример в Java SE 5 и следующих версиях:

```
double aMin = a[0], aMax = aMin;
for (double x : a){
  if (x < aMin) aMin = x;
  if (x > aMax) aMax = x;
}
double range = aMax - aMin;
```

Обратите внимание на то, что в цикле `for` сразу определяется переменная `x` того же типа, что и элементы массива. Эта переменная принимает последовательно значения всех элементов массива от первого элемента до последнего.

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 массивы. Можно объявить:

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

что эквивалентно

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

или

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

Затем определяем внешний массив и его размерность:

```
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. Рисунок 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. Но сначала проверьте свои знания и ответьте, пожалуйста, на контрольные вопросы.

Вопросы для самопроверки

1. Из чего состоит программа на языке Java?
2. Как оформляется метод обработки информации в Java?
3. Каков заголовок у метода main()?
4. Как записать комментарии к программе?
5. Что такое аннотация?
6. В каких системах счисления можно записывать целые константы?
7. Какое количество выражено числом 032?
8. Какое количество выражено числом 0x2C?
9. Как записать символ "наклонная черта"?
10. Как записать символ "обратная наклонная черта"?
11. Каков результат операции 3.45 % 2.4?
12. Что получится в результате операций 12 | 14 & 10?
13. Что даст операция 3 << 4?
14. Можно ли записать циклы внутри условного оператора?
15. Можно ли использовать оператор continue в операторе варианта?
16. Можно ли использовать оператор break с меткой в операторе варианта?
17. Можно ли определить массив нулевой длины?
18. Как можно перебрать все элементы массива в порядке возрастания индексов?
19. Как перебрать все элементы массива в порядке убывания индексов?
20. Что случится, если индекс массива превысит его длину?



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 поставленных задач, тут же возникают новые, еще более сложные задачи. Программисты придумывают новые методики, создают новые языки. За полвека появилось несколько сотен языков, предложено множество методов и стилей программирования. 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. Языки ассемблера облегчили чтение программ, но не упростили их. 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 громоздкими и 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 процедуры 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*. Струк-

турное 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. Появилось множество исследований различных структур данных и рекомендаций по их применению. Никлаус Вирт, создатель языка Pascal, даже назвал одну из своих книг "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 других программ.

Для того чтобы обеспечить максимальную независимость модулей друг от друга, надо четко отделить процедуры, которые будут использоваться другими модулями, — *открытые* (public) процедуры, от вспомогательных, которые обрабатывают данные, заключенные в этот модуль, — *закрытых* (private) процедур. Для этого модуль делится на две части. Открытые процедуры перечисляются в первой части модуля — *интерфейсе* (interface), вторые участвуют только во второй его части — *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 различных языках программирования это деление производится по-разному. В языке Turbo Pascal модуль специально делится на интерфейс и реализацию, в языке C интерфейс выносится в отдельные "головные" (header) файлы. В языке C++, 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 так называемый "приборный" стиль, то вдруг все куда-то пропадает, а на панели управления остаются только кнопка включения телевизора и ручка громкости. Любознательный телезритель, привыкший самостоятельно настраивать свой телевизор, берется за отвертку и лезет внутрь.

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 уничтожить 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 искусствен-

ных и природных объектов: роботов, станков с программным управлением, беспилотных самолетов; моделирования поведения 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 способен выполнять какие-то действия: перемещаться в пространстве, поворачиваться, поднимать груз, копать котлован или траншею, 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 начале работы программы создается один или несколько объектов. Объекты активны. Они выполняют свои методы обработки информации, сохраняя результаты обработки в своих полях, файлах, базах данных или в каких-то других хранилищах. По мере необходимости объекты обращаются к методам других объектов, передавая им нужные сведения. В процессе выполнения программы могут создаваться новые объекты и уничтожаться старые, ненужные объекты. Работа программы завершится, когда один из объектов выполнит метод завершения программы. Этот метод обычно сохраняет обработанную информацию в указанных ему хранилищах данных и удаляет все объекты, освобождая оперативную память и другие ресурсы, занятые объектами.

В виде объектов можно представить совсем неожиданные понятия. Например, окно на экране дисплея — это объект, имеющий ширину `width` и высоту `height`, определенное 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, полосы прокрутки и другие характеристики. Окно может перемещаться по экрану методом, описанным в какой-нибудь процедуре, скажем, `move()`, увеличиваться или уменьшаться в размерах каким-нибудь 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 характеристиками и действиями: 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 манипуляцией мышью и нажатием клавиш, 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 (object-oriented analysis, OOA). Решили проектировать сложные системы в виде объектов — появилось *объектно-ориентированное проектирование*, ООП (object-oriented design, OOD).

В начале разработки объектно-ориентированной программы сразу встает множество вопросов. Сколько объектов понадобится для правильной работы программы? Каким образом и в какое время создавать объекты? Как распределить работу между объектами? Как организовать взаимодействие объектов? В объектно-ориентированном программировании выработа-

но несколько общепризнанных принципов, более или менее полно отвечающих на эти вопросы. Познакомимся с ними.

Принципы объектно-ориентированного программирования

Объектно-ориентированное программирование развивается уже несколько десятков лет. Имеется несколько школ, каждая из которых предлагает свой набор принципов работы с объектами и по-своему излагает эти принципы. Бурные обсуждения и дискуссии, проходившие между представителями этих школ, позволили выработать несколько общепринятых принципов, признанных всеми школами и внедренных во все объектно-ориентированные языки программирования. Перечислим эти принципы.

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. Скажем, 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.

Для характеристики спортсмена обязательно надо указать его вес, рост, скорость реакции, спортивные достижения, а для ученого все эти качества несущественны, зато важны его квалификация, ученая степень, количество опубликованных научных работ.

Мы должны *абстрагироваться* от некоторых конкретных деталей объекта, отбросить их. Очень важно выбрать правильную степень абстракции. Слишком высокая степень даст только приблизительное описание объекта, не позволит правильно моделировать его поведение. Можно охарактеризовать человека как "Двуногое без перьев", но что это даст для его понимания? С другой стороны, 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.

Итак, прежде всего нам надо выбрать уровень абстракции, необходимый для правильного описания реального информационного процесса. Затем следует выделить объекты, участвующие в этом процессе, и установить связи между этими объектами. Как это сделать? Опишите процесс словами и проанализируйте получившиеся фразы. "Завод выпускает автомобили". Здесь два объекта — завод и автомобиль. Производственно-технические характеристики завода составят набор полей объекта "Завод", а процесс выпуска автомобиля будет описан в виде набора методов объекта "Завод".

Пример из другой области: "Преподаватель читает учебный курс". Полями объекта "Преподаватель" будут его фамилия, имя и отчество, научно-педагогический стаж, квалификация, ученая степень, выпущенные им учебники и методические пособия. Методами "Преподавателя"

будут такие действия, как "читать", "писать", "повышать квалификацию", "проводить консультацию", "принимать зачет". Полями объекта "Учебный курс" будут его название, программа, количество часов, перечень учебных пособий. Будет ли объект "Учебный курс" обладать какими-то методами или в этом объекте будут только поля? Какие действия выполняет "Учебный курс"? По-видимому, единственным действием объекта "Учебный курс" будет предоставление своих полей другим объектам, значит, нужны методы доступа к полям объекта.

Таким образом, если в словесном описании процесса вам потребовалось сформулировать какое-то понятие, то оно и будет кандидатом на оформление его в виде объекта. Существительные, описывающие это понятие, будут полями объекта, а глаголы — методами будущего объекта.

В объектно-ориентированных языках модель информационного процесса записывается в виде одного или нескольких *классов* (classes). Каждый класс описывает свойства одного объекта. Класс можно считать проектом, слепком, чертежом, по которому затем будут создаваться конкретные объекты. При описании класса применяются знакомые нам конструкции программирования.

Поля, в которых объект будет хранить необходимую ему информацию, описываются массивами, переменными и константами. Количество переменных и их типы выбираются так, чтобы в наибольшей степени охарактеризовать объект. Они называются *полями класса* (class fields). Полями класса могут быть не только простые переменные, константы или массивы, но и другие объекты и массивы объектов. Кроме полей в классе можно определить *локальные переменные*, хранящие промежуточные результаты работы методов класса.

Методы обработки информации, используемые объектом, описываются процедурами и функциями. Они называются *методами класса* (class methods). Сложные объекты могут содержать несколько десятков методов, а значит, несколько десятков процедур и функций. Методы класса активно используют поля класса, но кроме них могут создавать и свои локальные переменные, необходимые для работы метода.

Кроме полей и методов в классе можно описать и *вложенные классы* (nested classes) и *вложенные интерфейсы*, в которые, в свою очередь, можно вложить классы и интерфейсы. Мы можем создать сложную "матрешку" вложенных классов. Поля, методы и вложенные классы первого уровня называются *членами класса* (class members). Разные школы объектно-ориентированного программирования предлагают разные термины для описания структуры класса, 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 в точку (x, y).
     // Параметры метода x и y — уже не поля, а локальные переменные.
     int a = 1;    // a - локальная переменная, а не поле.
```

```
    // Тело метода. Здесь описывается способ перемещения автомобиля в точку
(x, y)
  }
  // Прочие методы класса...
}
```

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, называемые *экземплярами* (instances) описанного класса. Создание экземпляров производится в три этапа, подобно описанию массивов *(см. главу 1)*. Сначала объявляются ссылки на объекты: записывается имя класса и после пробела через запятую 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.

Имена полей, методов и вложенных классов у всех объектов одного класса одинаковы, они заданы в описании класса. Поэтому имена 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 количество символов в строке. В результате получаем значение `strlen`, равное 24. Подобная странная запись встречается в программах, написанных на языке 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, с меньшей степенью абстракции, модель будет использовать уже имеющиеся методы менее точной модели.

Иерархия

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 построить иерархию классов.

Иерархия объектов для их классификации используется давно. 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 пр. 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 `Pet`:

```
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 всех домашних животных, 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`. Например, создаем объекты:

```
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 можно 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", которое сообщает, 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 можно сказать, что класс `Dog` наследует класс `Pet`.

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

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

Ответственность

Хозяин и его домашние животные постоянно соприкасаются в жизни. Их взаимодействие выражается глаголами "гулять", "кормить", "охранять", "чистить", "ласкаться", "проситься" и пр. Для описания взаимодействия объектов применяется третий принцип объектно-ориентированного программирования — обязанность или 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 переведено на русский язык напрямую 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"? Дело в том, 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`. Объекты одного и того же класса отличаются друг от друга. 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. В нашем примере этот объект — хозяин животного. Один хозяин поставит миску с Chappi, другой бросит кость, третий выгонит собаку на улицу. Это интересное свойство называется *полиморфизмом* (polymorphism) и будет обсуждаться далее.
- 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 другой объект — текущее значение экземпляра `person` — возлагает на себя *ответственность* применить метод кормления `getFood()`, каким бы он ни был.

Модульность

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

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 объектами других классов 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 C++***
>
> 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 на железнодорожном переезде: "Нормальное положение шлагбаума — закрытое".

Если же надо обратиться к закрытому полю класса, то рекомендуется включить в класс специальные *методы доступа* (access methods), отдельно для чтения этого поля (get method) и отдельно для записи в это поле (set method). Имена методов доступа рекомендуется начинать со слов `get` и `set`, добавляя к этим словам имя поля. Для классов Java, используемых как компоненты большого приложения (такие классы-компоненты в технологии Java названы 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. В нашем примере можно ограничить список имен только членами клуба собаководства и сделать в методе проверку на наличие имени в этом списке.

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;
}
```

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 создания других программ.

Напротив, члены класса должны активно взаимодействовать друг с другом, как говорят, иметь тесную функциональную *связность* (high cohesion). 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 *закона Деметра*. Закон гласит: "В методе `m()` класса `A` следует использовать только методы класса `A`, методы классов, к которым принадлежат параметры 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 мы 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 их дополнительные возможности.

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

Принцип KISS

Основной, базовый и самый великий принцип программирования на любом языке и при любой парадигме — принцип KISS — не нуждается в переводе. Он расшифровывается так:

“Keep It Simple, Stupid!”

В самом деле, чем проще написана программа, тем легче ее отладить, тем лучше она будет работать. Кроме того, создатели языка проверяют его работу стандартными конструкциями, которые входят потом в руководства по языку. Эти-то проверенные конструкции и следует использовать в своих программах. Вычурные и запутанные программы показывают глубину мышления и остроумие их создателей, но очень неудобны для сопровождения и модификации и больше подвержены ошибкам.

Вообще говоря, сложность программы не может быть меньше сложности задачи, которую она решает. Надо только постараться, чтобы сложность программы не превысила сложность задачи. Стандартную задачу надо решать стандартными методами, используя стандартные конструкции языка. Нестандартную задачу тоже лучше решить стандартными методами и только если это не получается, вводить сложные конструкции.

Каждый язык программирования устанавливает свой стиль. Стиль всякого объектно-ориентированного языка программирования — следование принципам ООП и использование стандартных библиотек классов. Язык Java очень хорошо приспособлен для выражения этого стиля. На нем легко писать программы, следующие стилю ООП.

При изучении языка программирования важно усвоить его стиль и следовать этому стилю в своих программах. Удивительно, но правильный стиль программирования очень помогает в написании правильных эффективных программ.

Упражнения

1. Опишите в виде объекта строительный подъемный кран.
2. Опишите в виде объекта игровой автомат.
3. Смоделируйте в виде объекта сотовый телефон.

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 также модификаторы `protected`, `private`, `static`. Модификаторы класса мы будем вводить по мере изучения языка.

Тело класса, в котором в любом порядке перечисляются поля, методы, конструкторы, 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 допустимо записать константным выражением. Все это уже обсуждалось 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*. Конкретные значения параметров, переданные методу при обращении к нему, называются *аргументами* метода. Типы аргументов метода должны быть согласованы с типами соответствующих параметров метода.

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 класса.
  private double a = 0.0, b = 1.5, root;    // Закрытые поля экземпляра.
  public  double getRoot(){return root;}    // Метод доступа к полю root.
  private double f(double x){
    return x*x*x - 3*x*x + 3;   // Можно вернуть и что-нибудь другое.
  }
  private void bisect(){     // Параметров у метода 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, и мы переносим точку b в точку root.
        // В противном случае:
      else a = root;
        // переносим точку a в точку root
        // Продолжаем до тех пор, 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, скопированный в параметр `x`, 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).

Передача аргументов в метод

При обращении к методу создаются локальные переменные для хранения параметров метода на время его работы. Под них выделяются ячейки оперативной памяти, в которые копируются аргументы метода, заданные при обращении к нему. Локальные переменные-

параметры существуют только во время выполнения метода, по окончании его работы переменные уничтожаются и память освобождается.

Теория программирования знает несколько способов передачи аргументов в метод. Чаще всего применяются два способа: передача по значению и передача по ссылке.

Если аргумент передается *по значению*, то он копируется в локальную переменную-параметр метода, созданную во время выполнения метода и существующую только на время его работы. Сам аргумент при этом остается недоступным для метода и, следовательно, не изменяется им.

При передаче *по ссылке* в метод поступает не сам аргумент, а его адрес. Метод работает не с копией, а непосредственно с аргументом, обращаясь к нему по адресу, переданному в локальную переменную, и изменяет сам аргумент.

В языке Java, как и в языке C, реализован только один способ — передача аргументов по значению. Например, выполнив следующую программу:

```
class Dummy1{

  private static void f(int a){
     a = 5;
     }

  public static void main(String[] args){
     int x = 7;
     System.out.println("До: " + x);
     f(x);
     System.out.println("После: " + x);
  }
}
```

вы увидите значение 7 и до и после выполнения метода `f()`, потому что он менял локальную переменную `a`, а не переменную-аргумент `x`.

Очень часто у метода встречаются параметры ссылочного типа. В этом случае в локальную переменную-параметр копируется значение аргумента-ссылки. После этого метод будет работать непосредственно с объектом, на который ссылается аргумент, при помощи локальной копии аргумента-ссылки. Следующий пример поясняет это.

```
class Dummy2{

  private static void f(int[] a){
     a[0] = 5;
     }

  public static void main(String[] args){
     int[] x = {7};
     System.out.println("До: " + x[0]);
     f(x);
     System.out.println("После: " + x[0]);
  }
}
```

Теперь переменная x — это ссылка на массив, которая копируется в локальную переменную, созданную для параметра a. Ссылка a направляется на тот же массив, что и ссылка x. Она меняет нулевой элемент массива, и мы получаем "До: 7", "После: 5". По-прежнему сделана передача аргумента по значению, но теперь аргумент — это ссылка, и в метод f() передается ссылка, а не объект, на который она направлена.

Передача ссылок по значению приводит иногда к неожиданным результатам. В следующем примере:

```
 class Dummy3{

  private static void f(int[] a){
     a = new int[]{5};
     }

  public static void main(String[] args){
     int[] x = {7};
     System.out.println("До: " + x[0]);
     f(x);
     System.out.println("После: " + x[0]);
  }
}
```

мы опять оба раза увидим на экране число 7. Хотя теперь в методе f() изменилась ссылка на массив — параметр этого метода, а не сам массив, но изменилась копия a ссылки x, а не она сама. Копия a получила новое значение, она направлена на новый массив {5}, но сама ссылка x осталась прежней, она по-прежнему направлена на массив {7}.

> ***Знатокам Pascal и C++***
>
> В языке Java применяется только передача аргументов по значению.

Перегрузка методов

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 их *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().

Переопределение методов

Если же записать метод в подклассе с тем же именем и параметрами, что и в суперклассе, например:

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

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 метода его сигнатура должна полностью сохраняться. Если в подклассе мы изменим тип, количество или порядок следования параметров, то получим новый метод, не переопределяющий метод суперкласса. Проверку соответствия сигнатуры переопределяемого метода можно возложить на компилятор, записав перед методом подкласса аннотацию @Override, как это сделано в листинге 2.2. В этом случае компилятор пошлет на консоль сообщение об ошибке, если сигнатура помеченного метода не будет соответствовать сигнатуре ни одного метода суперкласса с тем же именем.

При переопределении метода права доступа к нему можно только расширить, но не сузить. Открытый метод public должен остаться открытым, защищенный protected может стать открытым, но не может стать закрытым.

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".

Реализация полиморфизма в Java

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;
  @Override
  void voice(){
    System.out.println("Gav-gav!");
  }
}
class Cat extends Pet{
  @Override
  void voice(){
    System.out.println("Miaou!");
  }
}
class Cow extends Pet{
  @Override
  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.

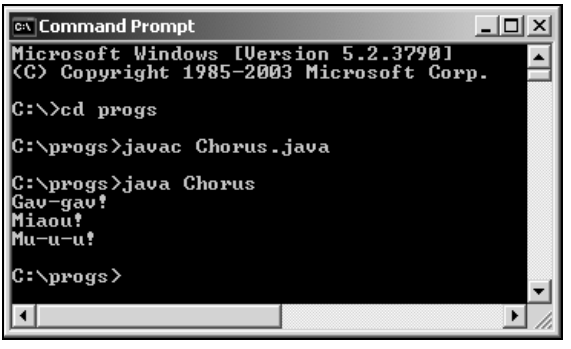


Рис. 2.1. Результат выполнения программы `Chorus`

Упражнения

4. Опишите в виде класса строительный подъемный кран.
5. Опишите в виде класса игровой автомат.
6. Смоделируйте в виде класса сотовый телефон.

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 необходимые 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 найти такое поле. 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, хотя он ничего не делает. 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 пр. 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.

Класс *Object*

На самой вершине иерархии классов Java стоит 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 обычно переопределяют в подклассах; более того, правильно спроектированные, "хорошо воспитанные" классы должны переопределить методы класса `Object`, если их не устраивает стандартная реализация. Например, в классе `String` метод `equals()` сравнивает не адреса размещения строк в оперативной памяти, а символы, из которых состоит строка, как мы увидим в *главе 5*.

Второй метод класса `Object`, часто требующий переопределения, — метод `hashCode()` — возвращает целое число, уникальное для каждого объекта данного класса, его идентификатор. Это число позволяет однозначно определить объект. Оно используется многими стандартными классами Java. Реализация метода `hashCode()`, сделанная в классе `Object`, может оказаться недостаточной для какого-то подкласса. В таком случае метод `hashCode()` следует переопределить.

Третий метод класса `Object`, который следует переопределять в подклассах, — метод `toString()`. Это метод без параметров, который выражает содержимое объекта строкой символов и возвращает объект класса `String`. В классе `Object` метод `toString()` реализован очень скудно — он выдает имя класса и идентификатор объекта, возвращаемый методом `hashCode()`. Метод `toString()` важен потому, что исполняющая система Java обращается к нему каждый раз, когда требуется представить объект в виде строки, например в методе `println()`. Обычно метод `toString()` переопределяют так, чтобы он возвращал информацию о классе объекта и текущие значения его полей, записанные в виде строк символов.

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 задача — определение полей создаваемого объекта начальными значениями. Своеобразие конструктора 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 аналогичного конструктора по умолчанию суперкласса.
- Конструктор выполняется автоматически при создании экземпляра класса, после распределения памяти и инициализации 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 конструктора суперкласса 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 обращение к конструктору суперкласса `super()` в начале конструктора не написано, то сначала 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.

Если вы не написали в своем классе ни одного конструктора, то компилятор, как уже сказано ранее, создаст конструктор по умолчанию. Но если вы написали хоть один конструктор, то компилятор ничего делать не будет и вам самим надо написать конструктор по умолчанию, без аргументов. Это следует сделать обязательно, если вы будете расширять свой класс, т. к. конструкторы подкласса вызывают конструктор по умолчанию суперкласса.

В наших примерах мы пока 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];
```

Элементы массива обнуляются.

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. Она возможна потому, что приоритет операции `new` выше, чем приоритет операции обращения к методу, обозначаемой точкой.

Упражнение

7. Введите конструкторы в классы, определенные в упражнениях 4—6.

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();
  }
}
```

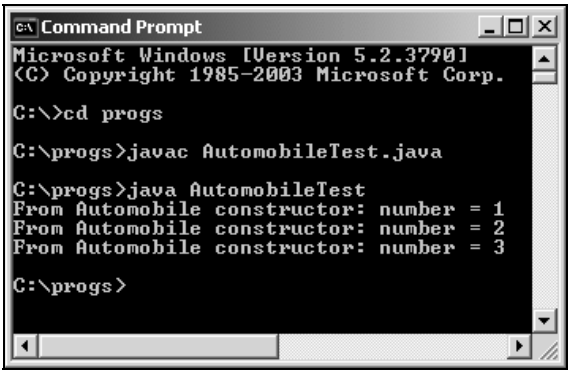


Рис. 2.2. Изменение статической переменной

Получаем результат, показанный на рис. 2.2.

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. Дело в том, что поля класса определяются при загрузке файла с классом в оперативную память, еще до создания экземпляров класса.

Для работы с такими *статическими полями* обычно создаются *статические методы*, помеченные модификатором `static`. Для методов слово `static` имеет совсем другой смысл, чем для полей, потому что 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 работают напрямую только со статическими полями и методами класса. При попытке обращения статического метода к нестатическим полям и методам класса на консоль посылается сообщение об ошибке. Это позволяет повысить безопасность программы, предотвратить случайное изменение полей и методов отдельных экземпляров класса.

Такая особенность статических методов приводит к интересному побочному эффекту. 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`;
- статические методы не могут быть абстрактными;
- статические методы переопределяются в подклассах только как статические;
- при переопределении статических методов полиморфизм не действует, ссылки всегда направляются на методы класса, а не объекта.

Именно по этим причинам 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 при загрузке класса, до конструктора:

```
static int[] a = new a[10];
static{
  for (int k: a)  a[k] = k * k;
}
```

Операторы, заключенные в такой блок, выполняются только один раз, при 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*

Компле́ксные 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 части.
    // 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.

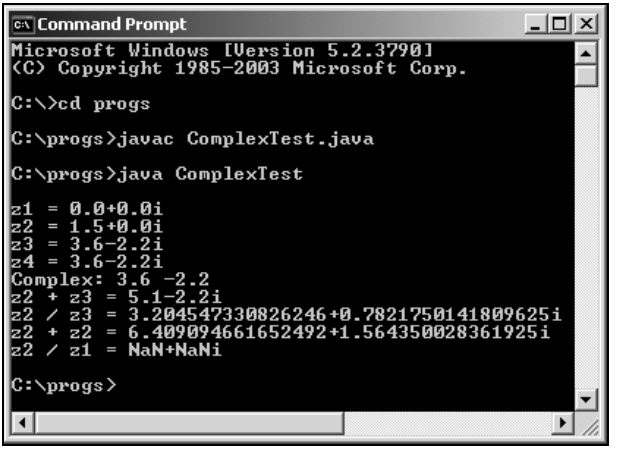


Рис. 2.3. Вывод программы `ComplexTest`

Метод *main()*

Всякая программа, оформленная как *приложение* (application), должна содержать метод с именем `main`. Он может быть один на все приложение или присутствовать 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`). У него один параметр, которым 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, когда еще не создан ни один объект. При вызове интерпретатора `java` указывается класс, где записан метод `main()`, с которого надо начать выполнение. Поскольку классов с методом `main()` может быть несколько, допустимо 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 аргументов, которые интерпретатор заносит в массив строк. Эти аргументы перечисляются в строке вызова `java` через пробел сразу после имени класса. Если же аргумент содержит пробелы, надо заключить его в кавычки. Кавычки не будут включены в аргумент, это только ограничители.

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

Листинг 2.5. Передача аргументов в метод `main()`

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

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

Как видите, имя класса не входит в число аргументов. 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 параметра `argc` используется переменная `args.length`, имеющаяся в каждом массиве. Доступ к переменным среды разрешен не всегда и осуществляется другим способом. Некоторые значения переменных среды можно просмотреть так:
>
> ```
> System.getProperties().list(System.out);
> ```

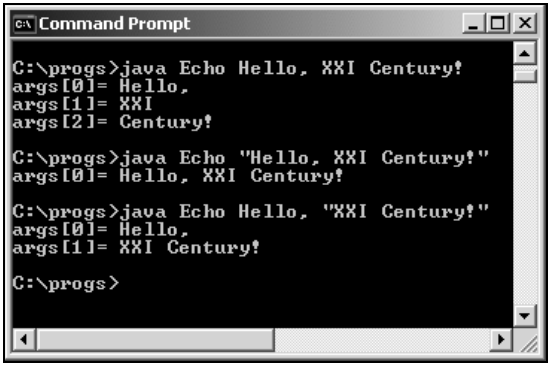


Рис. 2.4. Вывод параметров командной строки

Методы с переменным числом аргументов

Как видно из рис. 2.4, при вызове программы из командной строки мы можем задавать ей разное число аргументов. Исполняющая система Java создает массив этих аргументов и передает его методу `main()`. Такую же конструкцию можно сделать в своей программе:

```
class VarArgs{
   int[] args1 = {1, 2, 3, 4, 5, 6};
   int[] args2 = {100, 90, 80, 70};
   public int sum(int[] args){
      int result = 0;
      for (int k: args) result += k;
      return result;
   }

   public static void main(String[] args){
      System.out.println("Sum1 = " + sum(args1));
      System.out.println("Sum2 = " + sum(args2));
   }
}
```

Начиная с пятой версии Java эту конструкцию упростили. Теперь нет необходимости заранее формировать массив аргументов, это сделает компилятор. Программисту достаточно записать в списке параметров метода три точки подряд после имени параметра вместо квадратных скобок, обозначающих массив.

```
   public int sum(int... args){
      int result = 0;
      for (int k: args) result += k;
      return result;
   }
```

При обращении к методу мы просто записываем нужное число аргументов через запятую.

```
   public static void main(String[] args){
      System.out.println("Sum1 = " + sum(1, 2, 3, 4, 5, 6));
      System.out.println("Sum2 = " + sum(100, 90, 80, 70));
   }
```

Где видны переменные

В языке Java нестатические переменные разрешено объявлять в любом месте кода между операторами. Статические переменные могут быть только полями класса, а значит, не должны 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.
                           // Поле y получает значение 0.
  static{                  // Блок инициализации статических переменных.
                           // Выполняется один раз при первой загрузке класса по-
сле
                           // инициализаций в объявлениях переменных.
    x = 99;    // Этот оператор выполняется в блоке 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 име-
нем.
                           // Поле p получает значение 0.
  {                        // Блок инициализации экземпляра.
                           // Выполняется при создании каждого экземпляра
                           // после инициализаций при объявлениях переменных.
    p = 999;     // Этот оператор выполняется в блоке вне всякого метода!
  }
  static void f(int b){    // Параметр метода b — локальная переменная,
                           // известная 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 переменные класса и экземпляра неявно присваивают нулевые значения. Символы неявно получают значение `'\u0000'`, логические переменные — значение `false`, ссылки получают неявно значение `null`.

Локальные же переменные неявно не инициализируются. Они должны либо явно присваивать значения, либо 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 `static`-блоков и 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? Он применяется 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 статическими с помощью модификатора `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
                           // нельзя использовать в локальном классе D.
      final int j = 99;

      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 мат-
решки.
    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 можете перейти к следующему пункту. Если ничего не поняли, значит, вы — нормальный человек. Помните принцип KISS и используйте вложенные классы как можно реже.

Теперь 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 добавлены вынужденно, для того чтобы увязать концы с концами.
- Язык Java позволяет использовать одни и те же имена в разных областях видимости — поэтому 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, — поэтому пришлось использовать имя суперкласса, в примере это класс `Object`. Вместо конструктора в безымянном классе используется блок инициализации экземпляра, о котором говорилось в предыдущем пункте.
- Нельзя создать экземпляр вложенного класса, не создав предварительно экземпляр внешнего класса, — поэтому 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?.. Помните принцип 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 модификатором `private`, все равно они выходят на самый внешний уровень.

Все эти вопросы можно не брать в голову. Вложенные классы в Java используются, как правило, только в самом простом виде, главным образом при обработке событий, возникающих при действиях с мышью и клавиатурой.

В примере с домашними животными мы сделали объект `person` класса `Master` — владелец животного — полем класса `Pet`. Если класс `Master` больше нигде не используется, то можно определить его прямо внутри класса `Pet`, сделав класс `Master` вложенным (inner) классом. Это выглядит следующим образом:

```
class Pet{
  // В этом классе описываем общие свойства всех домашних любимцев.
    class Master{
        // Хозяин животного.
        string name; // Фамилия, имя.
        // Другие сведения…
        void getFood(int food, int drink); // Кормление.
        // Прочее…
     }
  int weight;     // Вес животного.
  int age;        // Возраст животного.
  Date eatTime[]; // Массив, содержащий время кормления.
  int eat(int food, int drink, Date time){ // Процесс кормления.
    // Начальные действия.
    if (time == eatTime[i]) person.getFood(food, drink);
    // Метод потребления пищи…
  }
  void voice(); // Звуки, издаваемые животным.
  // Прочее.
}
```

Вложение класса удобно тем, что методы внешнего класса могут напрямую обращаться к полям и методам вложенного в него класса. Но ведь того же самого можно было добиться по-другому. Может, следовало расширить класс `Master`, сделав класс `Pet` его наследником?

В каких же случаях следует создавать вложенные классы, а когда лучше создать иерархию классов? 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 объекты классов `Master` и `Pet` — в отношении "класс `Master` является экземпляром класса `Pet`" или в отношении "класс `Master` является частью класса `Pet`". Скажем, "собака является животным" или "собака является частью животного"? Другой пример: "мотор является автомобилем" или "мотор является частью автомобиля"? Ясно, что собака — животное и в этой ситуации надо выбрать наследование, но мотор — часть автомобиля и здесь надо выбрать вложение.

Отношение "класс A является экземпляром класса B" по-английски записывается как “a class A is a class B”, поэтому в теории ООП называется отношением “is-a”. Отношение же "класс A является частью класса B" по-английски “a class A has a class B”, и такое отношение называется отношением “has-a”.

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

Отношение “has-a” — это отношение "целое-часть" и ему соответствует вложение классов.

Вернемся к нашим животным и их хозяевам и постараемся ответить на вопрос: "класс `Master` является экземпляром класса `Pet`" или "класс `Master` является частью класса `Pet`"? Ясно, что не верно ни то, ни другое. Классы `Master` и `Pet` не связаны ни тем, ни другим образом. Поэтому мы и сделали объект класса `Master` полем класса `Pet`.

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. *Части II* и *III* книги как раз и дадут вам эту практику. Но сначала необходимо ознакомиться с важными понятиями языка Java — пакетами и интерфейсами.

Вопросы для самопроверки

1. Какие парадигмы возникали в программировании по мере его развития?
2. Какова современная парадигма программирования?
3. Что такое объектно-ориентированное программирование?
4. Что понимается под объектом в ООП?
5. Каковы основные принципы ООП?
6. Что такое класс в ООП?
7. Какая разница между объектом и экземпляром класса?
8. Что входит в класс Java?
9. Что такое конструктор класса?
10. Какая операция выделяет оперативную память для объекта?
11. Что такое суперкласс и подкласс?
12. Как реализуется полиморфизм в Java?
13. Для чего нужны статические поля и методы класса?
14. Какую роль играют абстрактные методы и классы?
15. Можно ли записать конструктор в абстрактном классе?
16. Почему метод main() должен быть статическим?



Глава 3

Пакеты, интерфейсы и перечисления

В стандартную библиотеку Java API входят сотни классов. Каждый программист в ходе работы добавляет к ним десятки своих классов. Множество классов растет и становится необозримым. Уже давно принято отдельные классы, решающие какую-то одну определенную задачу, объединять в библиотеки классов. Но библиотеки классов, кроме стандартной библиотеки, не являются частью языка.

Разработчики Java включили в язык дополнительную конструкцию — *пакеты* (packages). Все классы Java распределяются по пакетам. Кроме классов пакеты могут содержать 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 один пакет, хранятся в одном каталоге файловой системы. Подпакеты образуют подкаталоги этого каталога.

Каждый пакет создает 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 в одном месте программы 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, только в файловых системах для разделения имен каталогов в пути к файлу обычно используется наклонная черта или двоеточие, а не точка.

Еще одно отличие от файловой системы — подпакет не является частью пакета, и классы, находящиеся в нем, не относятся к пакету, а только к подпакету. Поэтому, для того чтобы создать подпакет, не надо предварительно создавать пакет. С другой стороны, включение в программу пакета не означает включение его подпакетов.

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), т. е. 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.

Следует обратить внимание на то, что члены с пакетным доступом не видны в подпакетах данного пакета.

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`.

Корпорация Sun Microsystems рекомендует записывать имена пакетов строчными буквами. Тогда они не будут совпадать с именами классов, которые, по соглашению, начинаются с прописной буквы. Кроме того, компания Sun советует использовать в качестве имени пакета или подпакета доменное имя своего сайта, записанное в обратном порядке, например:

```
com.sun.developer
```

Это обеспечит уникальность имени пакета во всем Интернете.

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 по-

этому у нас class-файл всегда оказывался в том же каталоге, что и соответствующий исходный java-файл.

Безымянный пакет служит обычно хранилищем небольших пробных или промежуточных классов. Большие проекты лучше хранить в пакетах. Более того, некоторые программные продукты Java вообще не работают с безымянным пакетом. Поэтому в технологии Java рекомендуется все классы помещать в пакеты.

Например, библиотека классов Java SE 6 API хранится в пакетах `java`, `javax`, `org`. Пакет `java` содержит только подпакеты `applet`, `awt`, `beans`, `io`, `lang`, `math`, `net`, `nio`, `rmi`, `security`, `sql`, `text`, `util` и ни одного класса. Эти пакеты имеют свои подпакеты, например пакет создания ГИП (Графический интерфейс пользователя) и графики `java.awt` содержит классы, интерфейсы и подпакеты `color`, `datatransfer`, `dnd`, `event`, `font`, `geom`, `im`, `image`, `print`.

Конечно, количество и состав пакетов Java SE API меняется с каждой новой версией.

Права доступа к членам класса

Пришло время подробно рассмотреть 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.

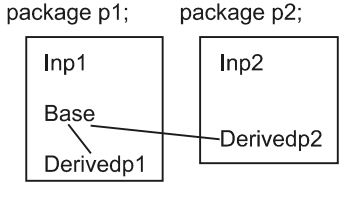


Рис. 3.1. Размещение наших классов по пакетам

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 них 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 в табл. 3.1 все сказанное.

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 уже рассмотренному 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. Рисунок 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.

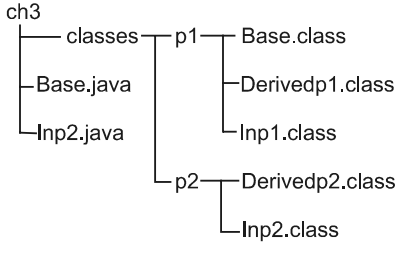


Рис. 3.2. Структура каталогов

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`.

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 какой-нибудь IDE, вроде Eclipse или NetBeans, то все эти действия будут сделаны без вашего участия.

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

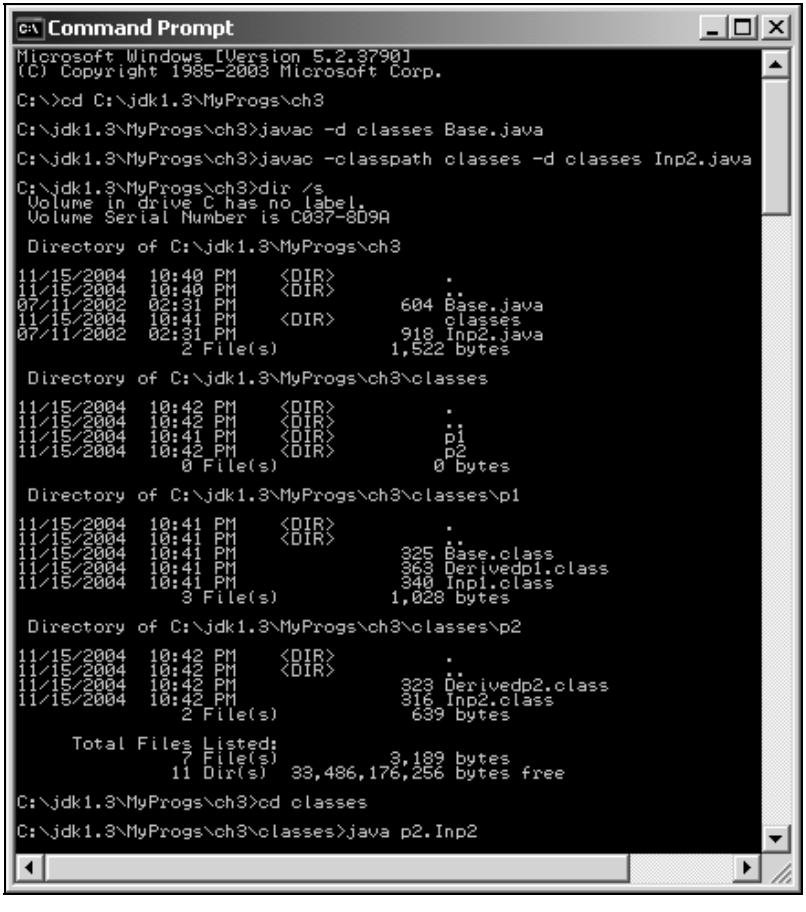


Рис. 3.3. Протокол компиляции и запуска программы

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 двух пакетах: в том, что указан в первой строке файла, и в пакете стандартных классов `java.lang`. 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 разрешается 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.

Начиная с версии Java SE 5 в язык введена еще одна форма оператора `import`, предназначенная для поиска статических полей и методов класса — оператор `import static`. Например, можно написать оператор

```
import static java.lang.Math.*;
```

После этого все статические поля и методы класса `Math` можно использовать без указания имени класса. Вместо записи

```
double r = Math.cos(Math.PI * alpha);
```

как мы делали раньше, можно записать просто

```
double r = cos(PI * alpha);
```

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.

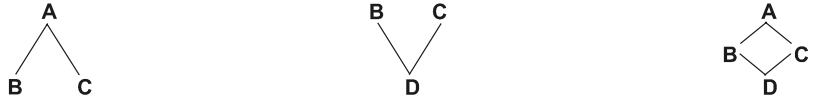
Отсюда следует, что если в проекте есть несколько открытых классов, то они должны находиться в разных файлах.

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

Для технологии Java характерно записывать исходный текст каждого класса в отдельном файле. В конце концов, компилятор всегда создает class-файл для каждого класса.

Интерфейсы

Вы уже заметили, что сделать 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.



а) *б)* *в)*

Рис. 3.4. Разные варианты наследования

В самом деле, пусть в классе `A` определен метод `f()`, к которому мы обращаемся из некоторого 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, допустимо определить экземпляры классов и обращаться к методам этих экземпляров, но это совсем другая ситуация.

В различных языках программирования этот вопрос решается по-разному, главным образом уточнением имени метода `f()`. Но при этом всегда нарушается 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 классов 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 тоже размещаются в пакетах и подпакетах, часто в тех же самых, что и классы, и тоже компилируются 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, одни интерфейсы могут порождаться от других 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 эти модификаторы принимаются по умолчанию.

Все константы и методы в интерфейсах всегда открыты, не обязательно 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 могут быть 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 методов можно лишь расширять доступ к ним, 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("Walk!"); 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 21*.

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.

Перечисления

Просматривая листинг 3.4, вы, наверное, заметили, что создавать интерфейс только для записи констант не совсем удобно. Начиная с версии Java SE 5 для этой цели в язык введены *перечисления* (enumerations). Создавая перечисление, мы сразу же указываем константы, входящие в него. Вместо интерфейса `Lights`, описанного в листинге 3.4, можно воспользоваться перечислением, сделав такую запись:

```
enum Lights{ RED, YELLOW, GREEN, ERROR }
```

Как видите, запись сильно упростилась. Мы записываем только константы, не указывая их характеристики. Каков же, в таком случае, их тип? У них тип перечисления `Lights`.

Перечисления в языке Java образуют самостоятельные типы, что указывается словом `enum` в описании перечисления, но все они неявно наследуют абстрактный класс `java.lang.Enum`. Это наследование не надо указывать словом `extends`, как мы обычно делаем, определяя классы. Оно введено только для того, чтобы включить перечисления в иерархию классов Java API. Тем не менее мы можем воспользоваться методами класса `Enum` для получения некоторых характеристик перечисления, как показано в листинге 3.5.

Листинг 3.5. Общие свойства перечислений

```
enum Lights { RED, YELLOW, GREEN, ERROR }
public class EnumMethods{
  public static void main(String[] args){
```

```
    for (Lights light: Lights.values()){
      System.out.println("Тип: " + light.getDeclaringClass());
      System.out.println("Числовое значение: " + light.ordinal());
    }
  }
}
```

Обратите внимание, во-первых, на то, как задается цикл для перебора всех значений перечисления `Lights`. В заголовке цикла определяется переменная `light` типа перечисления `Lights`. Метод `values()`, имеющийся в каждом перечислении, дает ссылку на его значения. Эти значения получает последовательно, одно за другим, переменная `light`.

Во-вторых, посмотрите, как можно узнать тип значений перечисления. Его возвращает метод `getDeclaringClass()` класса `Enum`. В случае листинга 3.5 мы получим тип `Lights`.

В-третьих, у каждой константы, входящей в перечисление, есть свой порядковый номер 0, 1, 2 и т. д. Его можно узнать методом `ordinal()` класса `Enum`.

Перечисление — это не только собрание констант. Это полноценный класс, в котором можно определить поля, методы и конструкторы. Мы уже видели, что в каждом перечислении есть метод `values()`, возвращающий массив значений перечисления.

Расширим определение перечисления `Lights`. Для использования его в классе `TrafficRegulator` нам надо сделать так, чтобы числовое значение константы `ERROR` было равно –1 и чтобы методом `shift()` можно было бы получить следующую константу. Этого можно добиться следующим определением.

```
enum Lights{
     RED(0), YELLOW(1), GREEN(2), ERROR(-1);

  private int value;
     private int currentValue = 0;

  Lights(int value){
    this.value = value;
     }

     public int getValue(){ return value; }

  public Lights nextLight(){
    currentValue = (currentValue + 1) % 3;
    return Lights.values()[currentValue];
  }
}
```

Как видите, теперь константы создаются конструктором, определяющим для каждой константы поле `value`. А сейчас можно применить полученное перечисление `Lights` для регулирования дорожного движения. Это сделано в листинге 3.6.

Листинг 3.6. Система управления светофором с перечислением

```
enum Lights{
  RED(0), YELLOW(1), GREEN(2), ERROR(-1);
```

```
  private int value;
  private int currentValue = 0;
  Lights(int value){
    this.value = value;
  }
  public int getValue(){ return value; }
  public Lights nextLight(){
    currentValue = (currentValue + 1) % 3;
    return Lights.values()[currentValue];
  }
}
class Timer {
  private int delay;
  private static Lights light = Lights.RED;
  Timer(int sec){delay = 1000 * sec;}
  public int shift(){
   Lights count = light.nextLight();
   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 Lights.ERROR;}
   return count;
  }
}
class TrafficRegulator{
  public static void main(String[] args){
      Timer t = new Timer(1);

   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("Walk!"); break;
      case Lights.ERROR:  System.err.println("Time Error"); break;
      default:            System.err.println("Unknown light."); return;
    }
  }
}
```

Константы, входящие в перечисление, рассматриваются как константные вложенные классы. Поэтому в них можно определять *константно-зависимые* поля и методы. Одно такое закрытое поле есть в каждой константе любого перечисления. Оно хранит порядковый номер константы, возвращаемый методом `ordinal()`.

Программист может добавить в каждую константу свои поля и методы. В листинге 3.7 приведен известный из документации пример простейшего калькулятора, в котором абстрактный метод выполнения арифметической операции `eval()` переопределяется в зависимости от ее конкретного вида в каждой константе перечисления `Operation`.

Листинг 3.7. Простейший калькулятор

```
public enum Operation{
  PLUS  { double eval(double x, double y){ return x + y; }},
  MINUS { double eval(double x, double y){ return x - y; }},
  TIMES { double eval(double x, double y){ return x * y; }},
  DIVIDE{ double eval(double x, double y){ return x / y; }};
  abstract double eval(double x, double y);
  public static void main(String[] args){
      double x = -23.567, y = 0.235;
      for (Operation op: Operation.values())
                 System.out.println(op.eval(x, y));
  }
}
```

Объявление аннотаций

Аннотации, о которых уже шла речь в *главе 1*, объявляются интерфейсами специального вида, помеченными символом “at-sign”, на жаргоне называемом "собачкой". Например, аннотация `@Override`, использованная нами в листинге 2.2, может быть объявлена так:

```
public @interface Override{ }
```

Таково объявление самой простой аннотации — аннотации без элементов (marker annotation). У более сложной аннотации могут быть элементы, описываемые методами интерфейса-аннотации. У этих методов не может быть параметров, но можно задать значение по умолчанию, записываемое после слова `default` в кавычках и квадратных скобках. Например, следующий текст

```
public @interface MethodDescription{

    int id();

    String description() default "[Method]";

    String date();

}
```

объявляет аннотацию с тремя элементами `id`, `name` и `date`. У элемента `name` есть значение по умолчание, равное `Method`.

Объявление интерфейса-аннотации определяет новый тип — *тип аннотации* (annotation type).

Аннотация записывается в программе в тех местах, где можно записывать модификаторы. По соглашению аннотация записывается перед всеми модификаторами. Элементы аннота-

ции записываются как пары "имя-значение" через запятую. В каждой паре имя отделяется от значения знаком равенства:

```
@MethodDescription(
    id = 123456,
    description = "Calculation method",
    date = "04.01.2008"
 )
public int someMethod(){
    . . .
}
```

Если у аннотации только один элемент, то его лучше назвать `value()`, например

```
public @interface Copyright{
    String value();
}
```

потому что в этом случае можно записать значение этого элемента просто как строку в кавычках, а не как пару имя-значение:

```
@ Copyright("2008 My Company")
public class MyClass{
    . . .
}
```

Разумеется, интерфейс-аннотация должен быть реализован классом Java, в котором надо записать действия, выполняемые аннотацией. Это можно сделать разными способами, но все они выходят за рамки нашей книги.

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.

Такая информационная система очень часто проектируется по схеме MVC.

Схема проектирования MVC

Естественно спроектировать в нашей автоматизированной системе три части.

- Первая часть, назовем ее *Контроллером* (Controller), принимает сведения от датчиков и преобразует их в некоторую 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 таблицу чисел, график, диаграмму или пакет для отправки по сети. Для каждого устройства вывода 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 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 объектно-ориентированном программировании каждая из трех частей схемы MVC реализуется одним или несколькими классами. *Модель* обладает методами `set`*`Xxx`*`()`, которые использует *Контроллер* для передачи информации в *Модель*. Одна *Модель* может получать информацию от нескольких *Контроллеров*. *Модель* предоставляет *Виду* для получения информации методы `get`*`Xxx`*`()` и `is`*`Xxx`*`()`.

Вид и *Контроллер* обычно не взаимодействуют. *Контроллер*, реагируя на события, обращается к методам `set`*`Xxx`*`()` *Модели*, которые меняют хранящуюся в ней информацию. *Модель*, изменив информацию, сообщает об этом тем *Видам*, которые зарегистрировались у нее. Этот способ взаимодействия *Модели* и *Вида* получил название "подписка-рассылка" (subscribe-publish). *Виды* подписываются у *Модели*, и та рассылает им сообщения о всяком изменении состояния объекта методами `fire`*`Xxx`*`()`, после чего *Виды* забирают измененную информацию, обращаясь к методам `get`*`Xxx`*`()` и `is`*`Xxx`*`()` *Модели*.

По схеме MVC построены компоненты графической библиотеки Swing, которые мы рассмотрим в *главе 11*.

К середине 90-х годов XX века накопилось много схем, подобных MVC. В них сконцентрирован многолетний опыт тысяч программистов, выражены наилучшие решения типовых задач.

Шаблон Singleton

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 поставленной задачи предложена схема Singleton, представленная в листинге 3.8.

Листинг 3.8. Схема Singleton

```
final class Singleton{
  private static Singleton s = new Singleton(0);
  private int k;
  private Singleton(int i){  // Закрытый конструктор.
      k = i;
  }
  public static Singleton getReference(){  // Открытый статический метод.
      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 [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:

- шаблоны создания объектов: Factory, Abstract Factory, Singleton, Builder, Prototype;
- шаблоны структуры объектов: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy;
- 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. В ней описаны и дополнительные шаблоны. Той же теме посвящено электронное издание [9]. В книге [10] подробно обсуждаются вопросы разработки систем на основе design patterns.

К сожалению, нет возможности разобрать подробно design patterns в этой книге. Но каждый разработчик, программирующий на объектно-ориентированном языке, 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 *части IV*. *Части II* и *III* 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.

Вопросы для самопроверки

1. Что такое пакет в Java?
2. Могут ли классы и интерфейсы, входящие в один пакет, располагаться в нескольких каталогах файловой системы?
3. Обеспечивает ли "пакетный" доступ возможность обращения к полям и методам классов, расположенных в подпакете?
4. Можно ли в аналогичной ситуации обратиться из подпакета к полям и методам классов, расположенных в объемлющем пакете?
5. Могут ли два экземпляра одного класса пользоваться закрытыми полями друг друга?
6. Почему метод main() должен быть открытым (public)?
7. Обеспечивает ли импорт пакета поиск классов, расположенных в его подпакетах?
8. Зачем в Java есть и абстрактные классы, и интерфейсы? Нельзя ли было обойтись одной из этих конструкций?
9. Зачем в Java введены перечисления? Нельзя ли обойтись интерфейсами?



ЧАСТЬ II

Использование классов из Java Development Kit



Глава 4

Классы-оболочки и generics

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 не только из-за многолетней привычки к числам и символам. Арифметические действия удобнее и быстрее производить с обычными числами, а не с объектами классов, которые требуют много ресурсов от компьютера.

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 типов 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 в пакете `java.lang` 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 `Byte`, `Short`, `Integer`, `Long`, `Float` и `Double`. Имена классов-оболочек, за исключением класса `Integer`, совпадают с именами соответствующих примитивных типов, но начинаются с заглавной буквы.

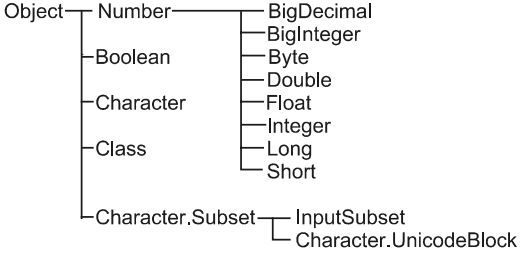


Рис. 4.1. Классы примитивных типов

Помимо метода сравнения объектов `equals()`, переопределенного из класса `Object`, все описанные в этой главе числовые классы и класс `Boolean` имеют метод `compareTo()`,

сравнивающий числовое значение, содержащееся в данном объекте, с числовым значением объекта — аргументом метода `compareTo()`. В результате работы метода получается целое значение:

- 0, если сравниваемые значения равны;
- отрицательное число (–1), если числовое значение в данном объекте меньше, чем в объекте-аргументе или, для класса `Boolean`, в данном объекте `false`, а в аргументе — `true`;
- положительное число (+1), если числовое значение в данном объекте больше числового значения, содержащегося в аргументе или в данном объекте `true`, а в аргументе — `false`.

Что полезного можно найти 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, служит параметром метода. Эти методы полезны при вводе данных в поля ввода, обработке аргументов командной строки, т. е. всюду, где числа представляются строками символов, состоящими из 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()`, представляющими последовательность битов, из которых состоит двоичное представление вещественного числа, в виде целого числа типа `int` или `long` соответственно. Исходное вещественное число задается как аргумент метода. Получив целочисленное представление, вы можете изменить отдельные биты получившегося целого числа побитовыми 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.

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));
  }
}
```

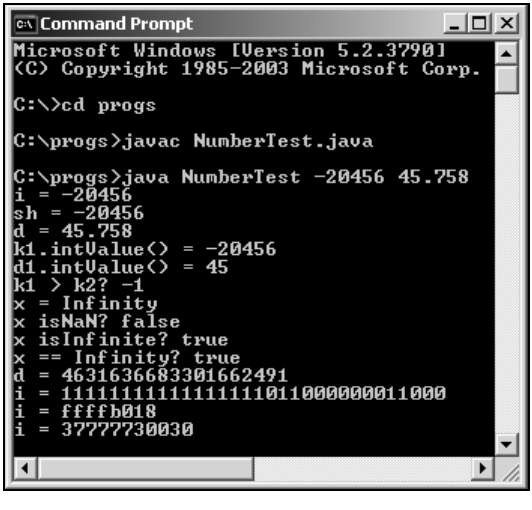


Рис. 4.2. Методы числовых классов

Методы `parseInt()` и конструкторы классов требуют обработки исключений, поэтому в листинг 4.1 вставлен блок `try{}catch(){}`. Обработку исключительных ситуаций мы подробно разберем в *главе 21*.

Начиная с версии Java SE 5 в JDK входит пакет `java.util.concurrent.atomic`, в котором, в частности, есть классы `AtomicInteger` и `AtomicLong`, обеспечивающие изменение числового значения этих классов на уровне машинных команд. Начальное значение задается конструкторами этих классов. Затем методами `addAndGet()`, `getAndAdd()`, `incrementAndGet()`, `getAndnIncrement()`, `decrementAndGet()`, `getAndDecrement`, `getAndSet()`, `set()` можно изменять это значение.

Автоматическая упаковка и распаковка типов

В листинге 4.1 объекты числовых классов создавались конструктором, в котором указывалось числовое значение объекта:

```
Integer k1 = new Integer(55);
```

Это правильно с точки зрения объектно-ориентированного программирования, но утомительно для программиста. Начиная с пятой версии Java было решено упростить такую запись. Теперь можно писать

```
Integer k1 = 55;
```

как будто `k1` — простая числовая переменная примитивного типа. Ничего нового в язык Java такая запись не вносит: компилятор, увидев ее, тут же запишет операцию `new` и восстановит конструктор. Но она облегчает работу программиста, предоставляя ему привычную форму определения переменной. Как говорят, компилятор делает *автоматическую упаковку* (auto boxing) числового значения в объект. Компилятор может сделать и *автоматическую распаковку*. После приведенных ранее определений объекта `k1` можно написать, например,

```
int n = k1;
```

и компилятор извлечет из объекта `k1` класса `Integer` числовое значение 55. Конечно, для этого компилятор впишет в предыдущее определение переменной `n` метод `intValue()` класса `Integer`, но это незаметно для программиста.

Автоматическая упаковка и распаковка возможна и в методах классов. Рассмотрим простой класс.

```
class AutoBox{
   static int f(Integer value){
       return value; // Распаковка.
   }
   public static void main(String[] args){
       Integer n = f(55);
   }
}
```

В методе `main()` этого примера сначала число 55 приводится к типу параметра метода `f()` с помощью упаковки. Затем результат работы метода `f()` упаковывается в объект `n` класса `Integer`.

Автоматическую упаковку и распаковку можно использовать в выражениях, написав `k1++` или даже `(k1 + k2 / k1)`, но это уже слишком! Представьте себе, сколько упаковок и распаковок вставит компилятор и насколько это замедлит работу программы!

Настраиваемые типы (generics)

Введение в язык Java автоматической упаковки типов позволило определить еще одну новую конструкцию — *настраиваемые типы* (generics), позволяющие создавать шаблоны классов, интерфейсов и методов. Например, можно записать обобщенный *настраиваемый* (generic) класс

```
class MyGenericClass<T>{
    private T data;
    public MyGenericClass(){}
    public MyGenericClass(T data){
      this.data = data;
    }
    public T getData(){
       return data;
    }
    public void setData(T data){
       this.data = data;
    }
}
```

в котором есть поле `data` неопределенного пока типа, обозначенного буквой `T`. Разумеется, можно написать другую букву или даже идентификатор. Буква `T` появилась просто как первая буква слова `Type`.

Перед использованием такого класса-шаблона его надо настроить, задав при обращении к его конструктору определенный тип в угловых скобках. Например

```
class MyGenericClassDemo{
    public static void main(String[] args){
        MyGenericClass<Integer> iMyGen = new MyGenericClass<Integer>(55);
        Integer n = iMyGen.getData();
        MyGenericClass<Double> dMyGen = new MyGenericClass<Double>(-37.3456);
        Double x = dMyGen.getData();
    }
}
```

Рассмотрим более содержательный пример. Пусть нам надо вычислять среднее арифметическое значение нескольких чисел, причем в одном случае это целые числа, в другом — вещественные, в третьем — короткие или, наоборот, длинные целые числа. У среднего значения в любом случае будет тип `double`. В листинге 4.2 написан один общий класс-шаблон для всех этих случаев.

Листинг 4.2. Настраиваемый класс

```
class Average<T extends Number>{
    T[] data;
     public Average(T[] data) {  this.data = data; }
     public double average(){
          double result = 0.0;
          for (T t: data) result += t.doubleValue();
          return result / data.length;
     }
     public static void main(String[] args){
           Integer[] iArray = {1, 2, 3, 4};
           Double[]  dArray = {3.4, 5.6, 2.3, 1.24};
           Average<Integer> iAver = new Average<Integer>(iArray);
           System.out.println("int average = " + iAver.average());
           Average<Double> dAver = new Average<Double>(dArray);
           System.out.println("double average = " + dAver.average());
     }
}
```

Обратите внимание на то, что в заголовке класса в угловых скобках указано, что тип `T` — подкласс класса `Number`. Это сделано потому, что здесь тип `T` не может быть произвольным. Действительно, в методе `average()` использован метод `doubleValue()` класса `Number`, а это означает, что тип `T` ограничен классом `Number` и его подклассами. Кроме того, операции сложения и деления тоже допустимы только для чисел.

Конструкция `<T extends SomeClass>` ограничивает сверху множество типов, пригодных для настройки параметра `T`. Таким же образом, написав `<T super SomeClass>`, можно ограничить снизу тип `T` только типом `SomeClass` и его супертипами.

У настраиваемого типа может быть более одного параметра. Они перечисляются в угловых скобках через запятую:

```
class MyGenericClass2<S, T>{
    private S id;
    private T data;
    public MyGenericClass2(){}
    public MyGenericClass2(S id, T data){
      this.id = id;
      this.data = data;
    }
    public S getId(){
       return id;
    }
    public void setId(S data){
       this.id = id;
    }
```

```
    public T getData(){
       return data;
    }
    public void setData(T data){
       this.data = data;
    }
}
```

Из этих примеров видно, что неопределенные типы `S`, `T` могут быть типами параметров конструкторов и типами возвращаемых методами значений. Разумеется, они могут быть типами параметров не только конструкторов, но и любых методов. Более того, типами параметров и типами возвращаемых значений методов могут быть настраиваемые типы. Можно написать метод в такой форме:

```
public MyGenericClass2<S, T> makeClass2(S id,
             MyGenericClass<T> data){
    return new MyGenericClass2(id, data.getData());
}
```

и обратиться к нему так, как показано в листинге 4.3.

Листинг 4.3. Настраиваемые классы — параметры методов

```
public class MyGenericClass2Demo<S, T>{
  public MyGenericClass2<S, T> makeClass2(S id, MyGenericClass<T> data){

    return new MyGenericClass2(id, data.getData());
  }
  public static void main(String[] args){
     MyGenericClass<Double> dMyGen = new MyGenericClass(34.456);
     MyGenericClass2Demo<Long, Double> d =
              new MyGenericClass2Demo<Long, Double>();
     MyGenericClass2<Long, Double> ldMyGen2 = d.makeClass2(123456L, dMy-
Gen);
  }
}
```

Шаблон типа (wildcard type)

В предыдущих главах мы часто пользовались тем, что можно определить ссылку типа суперкласса, ссылающуюся на объект подкласса, например:

```
Number n = new Long(123456L);
Number d = new Double(27.346);
```

Более того, это свойство распространяется на массивы:

```
Number[] n = new Long[100];
```

Можно ли распространить эту возможность на настраиваемые типы? Например, можно ли написать последний оператор листинга 4.3 так:

```
MyGenericClass2<Number, Number> n =                          //  Ошибка!
                d.makeClass2(123456L, dMyGen);
```

Ответ отрицательный. Из того, что какой-то класс `B` является подклассом класса `A`, не следует, что класс `G<B>` будет подклассом класса `G<A>`.

Это непривычное обстоятельство вынудило ввести дополнительную конструкцию — *шаблон типа* (wildcard type), применяемую в процессе настройки типа. Шаблон типа обозначается вопросительным знаком и означает "неизвестный тип" или "произвольный тип". Предыдущий код не вызовет возражений у компилятора, если написать его в таком виде:

```
MyGenericClass2<? extends Number, ? extends Number> n =    //  Верно.
                d.makeClass2(123456L, dMyGen);
```

или

```
MyGenericClass2<Long, ? extends Number> n =                 //  Верно.
                d.makeClass2(123456L, dMyGen);
```

Можно написать даже неограниченный шаблон типа

```
MyGenericClass2<?, ?> n =
                d.makeClass2(123456L, dMyGen);
```

Такая запись будет почти эквивалентна записи

```
MyGenericClass2 n =
                d.makeClass2(123456L, dMyGen);
```

за тем исключением, что в первом случае компилятор сделает более строгие проверки.

Кроме записи `<? extends Type>`, означающей "произвольный подтип типа `Type`, включая сам тип `Type`", можно написать выражение `<? super Type>`, означающее "произвольный супертип типа `Type`, включая сам тип `Type`".

Шаблон типа можно использовать в тех местах кода, где настраивается тип, в том числе в параметрах метода:

```
public MyGenericClass2<S, T> makeClass2(S id,
              MyGenericClass<? extends Number> data){

    return new MyGenericClass2(id, data.getData());
}
```

но, поскольку шаблон типа не является типом, его нельзя применять для создания объектов и массивов. Следующие определения неверны:

```
Average<? extends Number> a =                               // Ошибка!
                new Average<? extends Number>(iArray);
Average<? extends Number>[] a =                             // Ошибка!
                new Average<? extends Number>[10];
```

Тем не менее при определении массива (но не объекта) можно записать неограниченный шаблон типа:

```
Average<? extends Number>[] a =                             // Верно.
                new Average<?>[10];
```

Настраиваемые методы

Настраиваемыми могут быть не только типы, но и методы. Параметры настраиваемого метода (type parameters) указываются в заголовке метода в угловых скобках перед типом возвращаемого значения. Это выглядит так, как показано в листинге 4.4.

Листинг 4.4. Настраиваемый метод

```
public class MyGenericClass2Demo{
  public <S, T> MyGenericClass2<S, T>
            makeClass2(S id, MyGenericClass<T> data){

    return new MyGenericClass2(id, data.getData());
  }
  public static void main(String[] args){
     MyGenericClass<Double> dMyGen = new MyGenericClass(34.456);
     MyGenericClass2Demo d =
              new MyGenericClass2Demo();
     MyGenericClass2<Long, Double> ldMyGen2 = d.makeClass2(123456L,
dMyGen);
  }
}
```

Метод `makeClass2()` описан в простом, ненастраиваемом, классе `MyGenericClass2Demo`, и его параметры задаются в угловых скобках `<S, T>`. Здесь можно записывать ограниченные параметры

```
public <S extends Number, T extends Number>
          MyGenericClass2<S,   T>   makeClass2(S   id,   MyGenericClass<T>
data){

    return new MyGenericClass2(id, data.getData());
}
```

Как видно из листинга 4.4, специально настраивать метод не нужно, конкретные типы его параметров и возвращаемого значения определяются компилятором по переданным в метод аргументам.

Как вы убедились из приведенных примеров, настраиваемые типы и методы допускают сложную структуру параметров, так же как и вложенные классы. Мы еще не касались вопросов наследования настраиваемых типов, реализации настраиваемых интерфейсов, создания массивов настраиваемых типов. Все эти вопросы подробно рассмотрены на сайте Анжелики Лангер (Angelika Langer), в ее Java Generics FAQ, **http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html**.

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 произвольном сочетании регистров букв, и значение `false` — для любой другой строки.

Статический метод `valueOf(boolean b)` позволяет получить объект класса `Boolean` из значения примитивного типа `boolean`.

Пользуясь автоматической упаковкой, можно определение

```
Boolean b = new Boolean("true");
```

или

```
Boolean b = Boolean.valueOf(true);
```

сократить до

```
Boolean b = true;
```

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);
```

выполняет 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 в документации Java API.

Листинг 4.5 демонстрирует использование этих методов, а на рис. 4.3 показан вывод этой программы.

Листинг 4.5. 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 `UnicodeBlock` и еще один класс, `InputSubset`, являются расширениями класса `Subset`, как это видно на рис. 4.1. Объекты этого класса содержат подмножества кодировки Unicode.

Вместе с классами-оболочками удобно рассмотреть два класса для работы со сколь угодно большими числами.

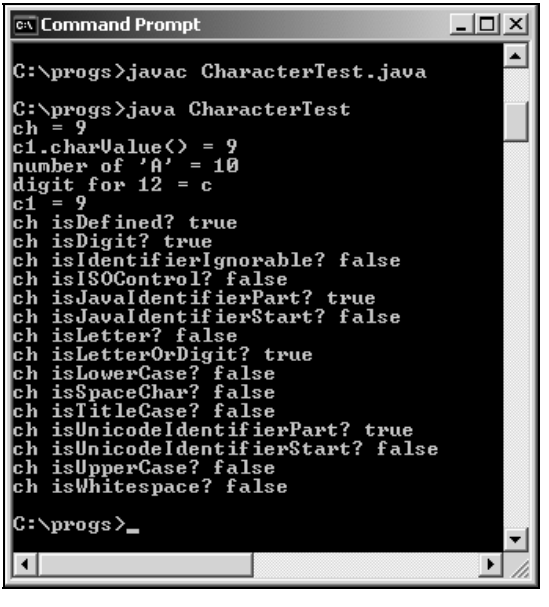


Рис. 4.3. Методы класса `Character` в программе `CharacterTest`

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 наращивается. Числа хранятся в двоичной форме с дополнительным кодом.

Перед выполнением операции числа выравниваются по длине распространением знакового разряда.

Шесть конструкторов класса создают объект класса `BigInteger` из строки символов (знака числа и цифр), массива байтов или задают случайное число. Чаще всего используются три конструктора:

- `BigInteger(String value)` — объект будет хранить большое целое число, заданное строкой цифр, перед которыми может стоять знак минус;
- `BigInteger(String value, int radix)` — задается строка цифр со знаком `value`, записанная в системе счисления с основанием `radix`;
- `BigInteger(byte[] value)` — объект будет хранить большое целое число, заданное массивом `value`, содержащим двоичное представление числа в дополнительном коде.

Три константы — `ZERO`, `ONE` и `TEN` — моделируют нуль, единицу и число десять 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 сложения `this + x`;
- `and(x)` — операция побитовой конъюнкции `this & x`;
- `andNot(x)` — операция побитовой дизъюнкции с дополнением `this & (~x)`;
- `divide(x)` — операция деления `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 умножения `this * x`;
- `negate()` — перемена знака числа, хранящегося в объекте;
- `not()` — операция отрицания `~this`;
- `or(x)` — операция побитовой дизъюнкции `this | x`;
- `pow(n)` — операция возведения числа, хранящегося в объекте, в степень `n`;
- `remainder(x)` — операция взятия остатка от деления `this % x`;
- `shiftLeft(n)` — операция сдвига влево `this << n`;
- `shiftRight(n)` — операция арифметического сдвига вправо `this >> n`;
- `signum()` — функция `sign(x)`;
- `subtract(x)` — операция вычитания `this - x`;
- `xor(x)` — операция "исключающее ИЛИ" `this ^ x`.

В листинге 4.6 приведены примеры использования данных методов, а рис. 4.4 показывает результаты выполнения этого листинга.

Листинг 4.6. 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("8888888888888888888");
    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));
  }
}
```

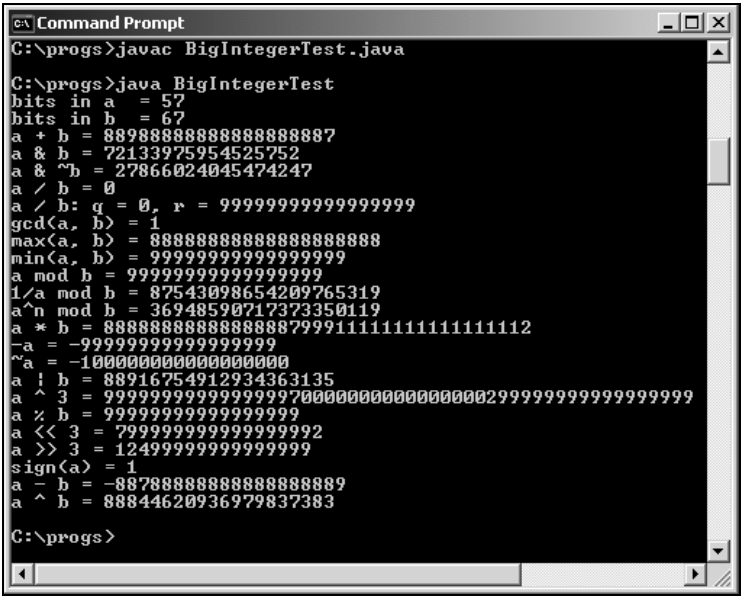


Рис. 4.4. Методы класса `BigInteger` в программе `BigIntegerTest`

Обратите внимание на то, что в программу листинга 4.6 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.

Три константы — ZERO, ONE и TEN — моделируют вещественные нуль, единицу и вещественное число десять в операциях с объектами класса BigDecimal.

В классе BigDecimal около двадцати конструкторов. Четыре из них были введены еще в Java 2.

- 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.

Остальные конструкторы определяют точность представления числового значения объекта и правила его округления с помощью объекта класса MathContext или непосредственно.

В классе переопределены методы doubleValue(), floatValue(), intValue(), longValue().

Большинство методов этого класса моделируют операции с вещественными числами. Они возвращают объект класса BigDecimal. Ниже в описании методов 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 сложения this + x;
- divide(x, r) — операция деления this / x с округлением по способу r;
- divide(x, n, r) — операция деления 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 умножения `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 вычитания `this - x`;
- `toBigInteger()` — округление числа, хранящегося в объекте;
- `unscaledValue()` — возвращает мантиссу числа;
- `upl()` — возвращает расстояние до следующего числа.

Листинг 4.7 показывает примеры использования этих методов, а рис. 4.5 — вывод результатов.

Начиная с версии Java SE 5 в класс `BigDecimal` введено еще много методов преобразования объекта и получения его характеристик.

Листинг 4.7. 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, BigDeci-
mal.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());
  }
}
```

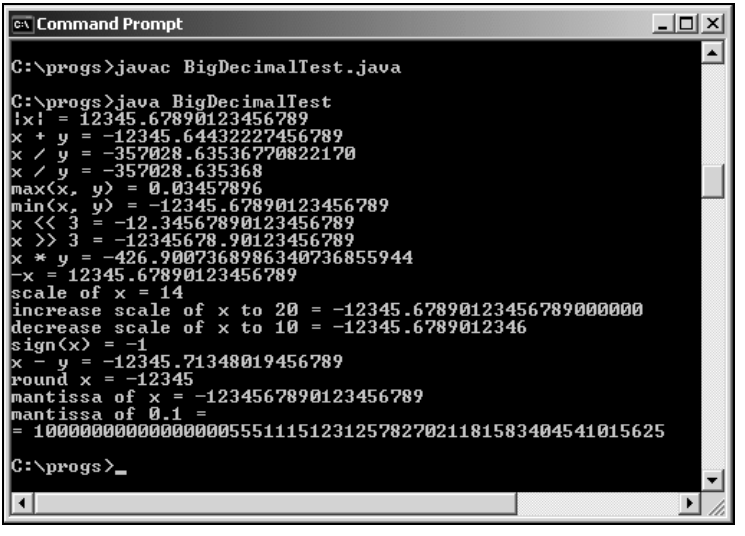


Рис. 4.5. Методы класса `BigDecimal` в программе `BigDecimalTest`

Приведем еще один пример. Напишем простенький калькулятор, выполняющий четыре арифметических действия с числами любой величины. Он работает из командной строки. Программа представлена в листинге 4.8, а примеры использования калькулятора — на рис. 4.6.

Листинг 4.8. 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");
    }
  }
}
```

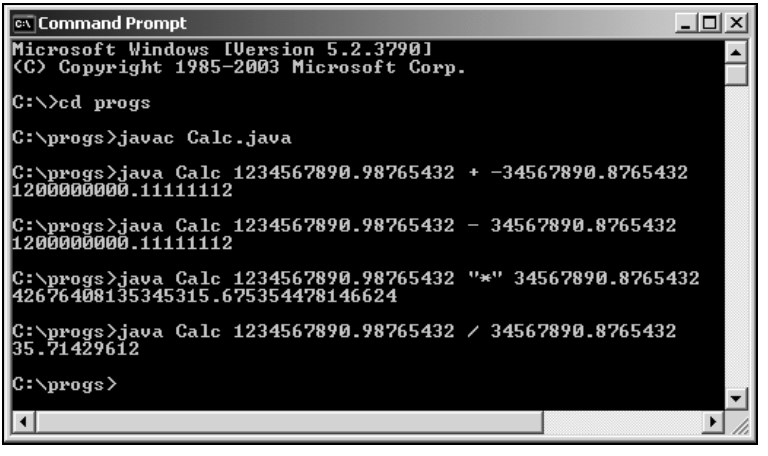


Рис. 4.6. Результаты работы калькулятора

Почему символ умножения — звездочка — заключен на рис. 4.6 в кавычки? Приверженцам ОС UNIX это понятно, а для других дадим краткое пояснение.

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, отделенные друг от друга пробелом.

Звездочка в кавычках понимается командной оболочкой как обычный символ. Командная оболочка снимает кавычки и передает интерпретатору Java звездочку, что нам и 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, перечислением 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();
```

Таким образом, у каждого действующего в программе объекта есть ссылка на экземпляр класса `Class`, содержащий описание класса этого объекта. Такое свойство объекта называется *рефлексией* (reflection). Кроме того, мы можем получить такую ссылку на классы по их имени.

Статический метод `forName(String class)` класса `Class` возвращает объект класса `Class` для класса, указанного в аргументе, например:

```
Class c1 = Class.forName("java.lang.String");
```

Третий способ получения экземпляра класса `Class` — к имени класса через точку добавить слово `class`:

```
Class c2 = java.lang.String.class;
```

Логические методы `isArray()`, `isInterface()`, `isEnum()`, `isPrimitive()` позволяют уточнить, не является ли объект массивом, интерфейсом, перечислением 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: `Class`, `Constructor`, `Method`, `Field` соответственно. 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 4.9 показывает применение этих методов, а рис. 4.7 — вывод результатов.

Листинг 4.9. 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 21*.



Рис. 4.7. Методы класса `Class` в программе `ClassTest`

Начиная с версии Java SE 5 класс `Class` сделан настраиваемым: `Class<String>` — это описание класса `String`, `Class<Long>` — описание класса `Long` и т. д. Это полезно, когда ссылка на класс `Class` передается в метод как параметр и надо определить, на какой же класс она направлена.

Вопросы для самопроверки

1. Зачем кроме примитивных типов в язык Java введены еще соответствующие классы-оболочки?
2. Можно ли использовать объекты числовых классов-оболочек в арифметических выражениях?
3. Какое наибольшее целое значение можно занести в объект класса BigInteger?
4. Какое наибольшее вещественное значение можно занести в объект класса BigDecimal?
5. Можно ли использовать в одном выражении значения примитивных типов и распакованные значения числовых классов-оболочек?
6. Для чего в язык Java введены настраиваемые типы?
7. Можно ли создавать настраиваемые интерфейсы или настраиваемыми могут быть только классы?
8. Должны ли методы настраиваемого класса быть настраиваемыми?
9. Можно ли создавать настраиваемые методы в обычных, не настраиваемых классах?



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 `StringBuilder`. В многопоточной среде вместо класса `StringBuilder`, не обеспечивающего синхронизацию, следует использовать класс `StringBuffer`, но эти вопросы мы отложим до *главы 23*.

Класс `StringBuilder` введен в стандартную библиотеку Java, начиная с версии Java SE 5, для ускорения работы с текстом в одном подпроцессе.

Все эти классы реализуют интерфейс `CharSequence`, в котором описаны общие методы работы со строками любого типа. Таких методов немного.

- Метод `length()` возвращает количество символов в строке.
- Метод `charAt(int pos)` возвращает символ, стоящий в позиции `pos` строки. Символы в строке нумеруются, начиная с нуля.
- Метод `subSequence(int start, int end)` возвращает подстроку, начинающуюся с позиции `start` и заканчивающуюся перед позицией `end` исходной строки.

Поначалу представление строк объектами необычно и кажется слишком громоздким, но, привыкнув, вы оцените удобство работы с классами, а не с массивами символов.

Конечно, можно 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 три 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 классов `StringBuilder` и `StringBuffer`, можно менять, вставляя и добавляя строки и символы, удаляя подстроки или сцепляя несколько строк в одну. Во многих случаях, когда надо изменить длину строки типа `String`, компилятор Java неявно преобразует ее к типу `StringBuilder` или `StringBuffer`, меняет длину, потом преобразует обратно в тип `String`. Например, следующее действие:

```
String s = "Это" + " одна " + "строка";
```

компилятор выполнит примерно так:

```
String s = new StringBuffer().append("Это").append(" одна ")
      .append("строка").toString();
```

Будет создан объект класса `StringBuffer` или класса `StringBuilder`, в него методом `append()` последовательно будут добавлены строки `"Это"`, `" одна "`, `"строка"`, и получившийся объект класса `StringBuffer` или `StringBuilder` 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, как и объект всякого класса. 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 о разнице 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 более десяти конструкторов:

- `String()` — создается объект с пустой строкой;
- `String(String str)` — конструктор копирования: из одного объекта создается его точная копия, поэтому данный конструктор используется редко;
- `String(StringBuffer str)` — преобразованная копия объекта класса `StringBuffer`;
- `String(StringBuilder str)` — преобразованная копия объекта класса `StringBuilder`;
- `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(int[] intArray, int offset, int count)` — то же, но массив состоит из символов Unicode, записанных в массив целого типа, что позволяет использовать символы Unicode, занимающие больше двух байтов;
- `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;
- `String(byte[] byteArray, Charset charset)` — символы, записанные в массиве байтов, задаются в Unicode-строке с учетом кодировки, заданной объектом `charset`;
- `String(byte[] byteArray, int offset, int count, Charset charset)` — 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, записанные кириллицей, 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 (Язык и стандарты) в окне **Control Panel** (Панель управления)), 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 применяются последние четыре конструктора, в которых параметром `encoding` или `charset` 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 = {'P', 'o', 'c', 'c', '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 без преобразования в Unicode выводятся массивы байтов `byteCp1251`, `byteCp866` и `byteKOI8R`. 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.

Далее 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.

Как уже упоминалось в *главе 1*, в консольное окно **Command Prompt** операционных систем 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`.

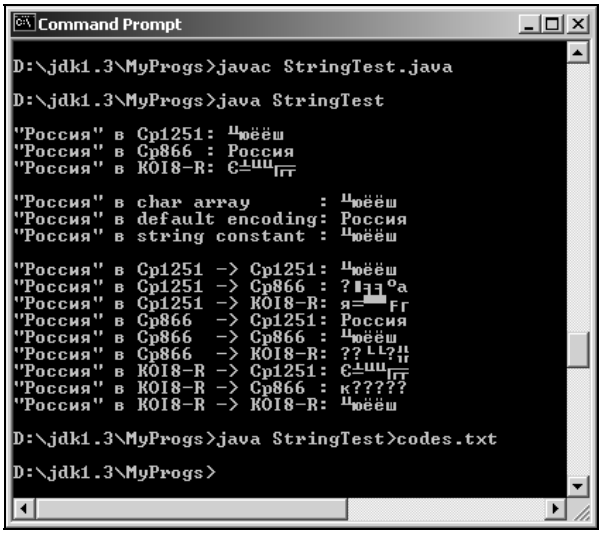


Рис. 5.1. Вывод кириллической строки на консоль MS Windows 2000

В предпоследней строке рис. 5.1 сделано перенаправление вывода программы в файл codes.txt. В MS Windows 2000/XP/Server 2003 вывод текста в файл происходит в кодировке CP1251. На рис. 5.2 показано содержимое файла codes.txt в окне программы Notepad (Блокнот).

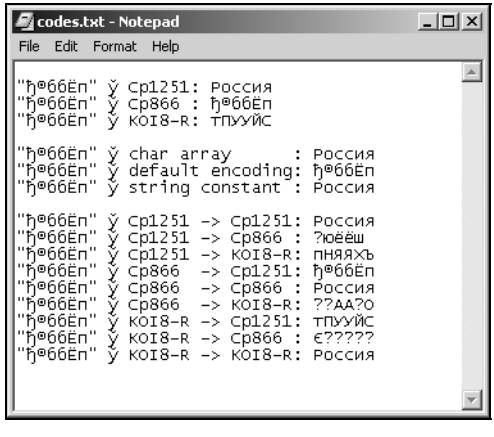


Рис. 5.2. Вывод кириллической строки в файл

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 *24*, 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);
copyValueOf(char[] charArray, int offset, int length);
```

Они формируют 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`".

Упражнение

1. Потренируйтесь в преобразованиях строки в массивы байтов с разной кириллической кодировкой.

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`".

Кроме операции сцепления соединить строки можно методом `concat()`, например:

```
String s = attention.concat("неизвестный символ");
```

Манипуляции строками

В классе `String` собрано очень много 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`.

Логический метод `isEmpty()`, появившийся в Java SE 6, возвращает `true`, если строка пуста, в ней нет ни одного символа.

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()`.

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)` больше `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, применяйте метод `getBytes(String encoding)` или метод `getBytes(Charset 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** операционных систем Windows 2000/XP/Server 2003.

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`. Индекс можно задать любым целым типом, кроме типа `long`.

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 следующего фрагмента

```
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 разбить строку на подстроки

Метод `split(String regExp)` разбивает строку на подстроки, используя в качестве разделителей символы, входящие в параметр `regExp`, записывает подстроки в массив строк и возвращает ссылку на этот массив. Сами разделители не входят ни в одну подстроку.

Например, после выполнения следующего фрагмента

```
String s = "Write:once,:run:anywhere.";
String[] sub = s.split(":");
```

получим в строке `sub[0]` значение `"Write"`, в строке `sub[1]` значение `"once,"`, в строке `sub[2]` значение `"run"`, а в `sub[3]` — значение `"anywhere."`.

Метод `split(String regExp, int n)` разбивает строку на `n` подстрок. Если параметр `n` меньше числа подстрок, то весь остаток строки заносится в последний элемент создаваемого массива строк. Применение метода

```
String[] sub = s.split(":", 2);
```

в предыдущем примере даст массив `sub` из двух элементов со значением `sub[0]`, равным `"Write"`, и значением `sub[1]`, равным `"once,:run:anywhere."`.

Разбить строку можно практически на любые подстроки, поскольку значением параметра `regExp` может быть любое регулярное выражение. *Регулярное выражение* (regular expres-

expression) — это шаблон для отбора строк, составляемый по сложным правилам, изложению которых посвящены целые книги. Регулярные выражения выходят за рамки нашей книги, но все-таки я приведу пример разбиения строки на слова, разделенные пробельными символами:

```
String[] word = s.split("\\s+");
```

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 `s3` так:

```
String s3 = "Какая-то строка";
```

то сравнение `s1 == s3` даст в результате `true`, потому что компилятор создаст только один экземпляр константы `"Какая-то строка"` и направит на него все ссылки — и ссылку `s1`, и ссылку `s3`.

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 параметр `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());
```

Эти методы не учитывают алфавитное расположение символов в локальной кодировке.

Русские буквы расположены в Unicode по алфавиту, за исключением одной буквы. Заглавная буква Ё находится 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('o')` выдаст в результате индекс 1.

Конечно, этот метод реализован так, что он 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('o', indexOf('o') + 1)` даст в результате индекс 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('o') даст в результате индекс 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);
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 новую строку `"Река в реке сеет хлеб"`.

Регистр букв при замене учитывается.

Как заменить подстроку

Метод `replaceAll(String old, String new)` возвращает новую строку, в которой все вхождения подстроки `old` заменены строкой `new`. Если подстроки `old` в исходной строке нет, то возвращается ссылка на исходную строку. Параметром `old` может служить регулярное выражение, пользуясь которым можно сделать очень сложную замену.

Метод `replaceFirst(String old, String new)` возвращает новую строку, в которой сделана только одна, первая, замена.

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 в строку данные другого типа

В языке Java принято соглашение — каждый класс отвечает за преобразование других типов в тип данного 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()`.

Упражнения

2. Подсчитайте количество появлений того или иного символа в заданной строке.
3. Подсчитайте количество слов в заданной строке.
4. Найдите число появлений заданного слова в заданной строке.

Класс *StringBuilder*

Объекты класса `StringBuilder` — 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 окажется обрезанной, 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 `StringBuilder` можно только конструкторами.

Конструкторы

В классе `StringBuilder` четыре конструктора:

- `StringBuilder()` — создает пустой объект с емкостью 16 символов;
- `StringBuilder(int capacity)` — создает пустой объект заданной емкости `capacity`;

- `StringBuilder(String str)` — создает объект емкостью `str.length() + 16`, содержащий строку `str`;
- `StringBuilder(CharSequence str)` — создает объект, содержащий строку `str`.

Как добавить подстроку

В классе `StringBuilder` есть более десяти 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"`.

Два аналогичных метода работают с параметром типа `StringBuffer` и `CharSequence`.

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);
append(char[] sub, int offset, int len);
```

Еще один метод, `append(CharSequence sub, int offset, int len)`, использует параметр типа `CharSequence`.

Тринадцатый метод, `append(Object obj)`, добавляет просто объект. Перед этим объект `obj` преобразуется в строку своим методом `toString()`.

Как вставить подстроку

Более десяти методов `insert()` предназначены для вставки строки, указанной вторым параметром метода, в данную строку. Место вставки задается первым параметром метода, индексом символа 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 самую, но 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 StringBuilder("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()`.

Еще два метода:

```
insert(int ind, CharSequence str);
insert(int ind, CharSequence sub, int start, int end);
```

работают с параметром типа `CharSequence`.

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 StringBuilder("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, а 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.

Например, после выполнения

```
String s = new StringBuilder("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. Большую помощь в разборе строки оказывает метод `split()`.

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 разбора очень проста (листинг 5.2).

Листинг 5.2. Разбиение строки на слова

```
import java.util.*;
class MyParser{
   public static void main(String[] args){
     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.

Исходные тексты хранятся в ZIP-архиве src.zip, лежащем в корневом каталоге JDK, например в каталоге D:\jdk1.6.0_02.

После распаковки в каталоге jdk1.6.0_02 появится подкаталог src, а в нем — подкаталоги, соответствующие пакетам и подпакетам JDK, с исходными файлами.

Вопросы для самопроверки

1. Зачем в язык Java введено несколько классов, обрабатывающих строки символов?
2. Какова разница между классами String и StringBuilder?
3. Какова разница между классами StringBuffer и StringBuilder?
4. Что лучше использовать для сцепления строк: операцию сцепления или метод append() класса StringBuilder?
5. Что лучше использовать для разбора строки: метод split() или класс StringTokenizer?

Глава 6



Классы-коллекции

В листинге 5.2 мы разобрали строку на слова. Как их сохранить для дальнейшей обработки?

До сих пор для таких целей мы пользовались массивами. Они удобны, если необходимо быстро обработать однотипные элементы, например просуммировать числа, найти наибольшее и наименьшее значение, отсортировать списки. Но уже для поиска нужных сведений в большом объеме информации массивы неудобны. Для этого лучше использовать другие разновидности хранения данных, например бинарные деревья поиска.

Кроме того, массивы всегда имеют постоянную, предварительно заданную длину, поэтому в массивы невозможно добавлять элементы без переопределения массивов. При удалении элемента из массива оставшиеся элементы следует перенумеровывать, чтобы сохранить их непрерывную нумерацию.

При решении задач, в которых количество элементов заранее неизвестно, а элементы надо часто удалять и добавлять, следует 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, ссылки любого типа. Количество элементов может быть произвольным и не определяться заранее. Элементы получают индексы 0, 1, 2 и т. д. 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 так, чтобы включить его в иерархию классов-коллекций. Для этого добавлено много новых методов, реализующих методы соответствующих интерфейсов-коллекций. Сейчас 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 с емкостью в 10 элементов;
- `Vector(int capacity)` — создает пустой объект указанной емкости `capacity`;
- `Vector(int capacity, int increment)` — формирует пустой объект указанной емкости `capacity` и задает число `increment`, на которое увеличивается емкость при необходимости;
- `Vector(Collection c)` — вектор создается по указанной коллекции.

Если число `capacity` отрицательно, то возникает исключительная ситуация.

После создания вектора его можно заполнять элементами. В векторе разрешено хранить объекты разных типов, поскольку на самом деле в нем хранятся не значения объектов, а ссылки на них. Класс `Vector` настраиваемый, и если предполагается заполнять вектор ссылками одного и того же типа, то этот тип можно задать при создании вектора в шаблоне (generic) коренного типа, в угловых скобках, по такой схеме:

```
Vector<String> v = new Vector<String>();
```

После этого определения компилятор будет следить за тем, чтобы у всех элементов вектора был тип `String`. Извлечение элементов из вектора не потребует приведения типа.

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`.

Вот пример создания и заполнения вектора:

```
Vector v = new Vector();
v.add(new Date());
v.add("Строка символов");
v.add(new Integer(10));
v.add(20);
```

Обратите внимание в этом примере на две последние строки. Первая из них записана по канонам работы с коллекцией: в вектор вместо числа 10 заносится ссылка на объект класса `Integer`, содержащий это число. В последней строке применена автоматическая упаковка типа: в методе `add()` записывается просто число 20, метод сам создает необходимую ссылку.

Как заменить элемент

Метод `set(int index, Object element)` заменяет элемент, стоявший в векторе в позиции `index`, на элемент `element` (то же самое 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, например:

```
String s = (String)v.get(1);
```

Если при создании вектора шаблоном был указан определенный тип элементов, то возвращается объект именно этого типа и явное приведение типа не требуется, можно написать просто

```
String s = v.get(1);
```

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 число –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()`, старым методом `removeAllElements()` или обнулив размер вектора методом `setSize(0)`.

Приведем пример работы с вектором. Листинг 6.1 расширяет листинг 5.2, обрабатывая выделенные из строки слова с помощью вектора.

Листинг 6.1. Работа с вектором

```
import java.util.*;
class MyParser{
  public static void main(String[] args){
     Vector<String> v = new Vector<String>();
     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();

        for (String s: v)          // Другой способ перебора элементов вектора.
            System.out.print(s + " ");
        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.

Класс *Stack*

Второй пример коллекции — класс `Stack` — расширяет класс `Vector`.

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, последним пришел — первым ушел).

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 в арифметическом выражении, записанном в строку.

Листинг 6.2. Проверка парности скобок

```
import java.util.*;

class StackTest{
```

```
  static boolean checkParity(String expression, String open, String close){
    Stack<String> stack = new Stack<String>();
    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 с неизвестным заранее числом элементов.

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". Их можно представить себе как таблицу из двух столбцов. Такие таблицы часто называют *словарями* или *ассоциативными массивами*.

Из таких пар "Фамилия И. О. — номер телефона" 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 емкость таблицы.

Как создать таблицу Hashtable

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)` — формирует 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.

Если предполагается хранить в таблице пары определенных типов, то эти типы можно указать заранее при создании таблицы. Для этого в угловых скобках в шаблоне (generics) коренного типа записываются конкретные типы по такой схеме:

```
Hashtable<Integer, String> h = new Hashtable<Integer, String>();
```

После такого определения компилятор будет следить за типами заносимых в таблицу элементов. Извлечение элементов из таблицы не потребует явного приведения типов.

Как заполнить таблицу Hashtable

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 аргумент равен `null`, возникает исключительная ситуация;
- `void putAll(Map f)` — добавляет все элементы отображения `f`.

В объектах-ключах `key` должны быть переопределены методы `hashCode()` и `equals()`, унаследованные от класса `Object`.

Как получить значение по ключу

Метод `get(Object key)` возвращает значение элемента с ключом `key` в виде объекта класса `Object` или того класса, с которым создана таблица. Если при создании таблицы класс не был указан, то для дальнейшей работы с полученным объектом 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 Hashtable

Метод `values()` представляет все значения `value` таблицы в виде объекта типа `Collection`. Все модификации в этом объекте изменяют таблицу, и наоборот.

Метод `keySet()` предоставляет все ключи `key` таблицы в виде объекта типа интерфейса `Set`. Все изменения в этом объекте типа `Set` корректируют таблицу, и наоборот.

Метод `entrySet()` представляет все пары "`key` — `value`" таблицы в виде объекта типа интерфейса `Set`. Все модификации в этом объекте типа `Set` изменяют таблицу, и наоборот.

Итак, таблицу типа Hashtable можно представить в трех формах: в виде коллекции значений, в виде множества ключей или в виде множества пар.

Метод `toString()` возвращает строку, содержащую все пары таблицы.

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.

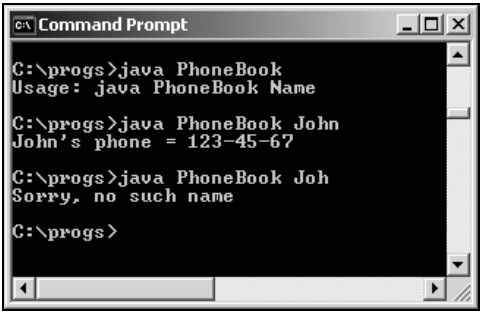


Рис. 6.1. Работа с телефонной книгой

Листинг 6.3. Телефонный справочник

```
import java.util.*;
class PhoneBook{
  public static void main(String[] args){
    Hashtable<String, String> yp = new Hashtable<String, String>();
    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");
  }
}
```

Класс *Properties*

Класс `Properties` расширяет класс `Hashtable` таким образом, что в нем хранятся пары ссылок не на произвольный тип, а на строки — пары типа `String`. Он предназначен в основном для работы с парами "свойства системы — их значения", записанными в файлах свойств.

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);
  }
}
```

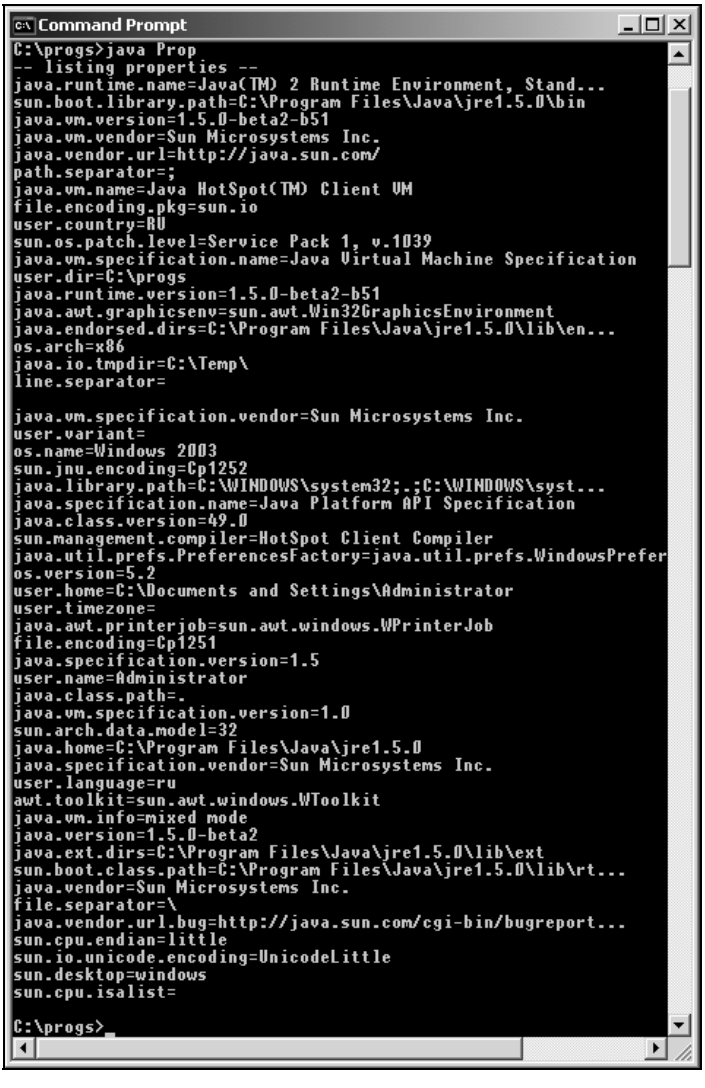


Рис. 6.2. Системные свойства

Примеры классов `Vector`, `Stack`, `Hashtable`, `Properties` показывают удобство классов-коллекций. Такое удобство и необходимость в коллекциях разных видов привели к тому, что для Java была разработана целая иерархия коллекций, получившая название Java Collections Framework. Она показана на рис. 6.3. Курсивом записаны имена интерфейсов. Пунктирные линии указывают классы, реализующие эти интерфейсы.

Все коллекции разбиты на четыре группы, описанные в интерфейсах `List`, `Set`, `Queue` и `Map`.

Примером реализации интерфейса `List` может служить описанный ранее класс `Vector`, примером реализации интерфейса `Map` — класс `Hashtable`.

Коллекции `List`, `Set` и `Queue` имеют много схожего, поэтому их общие методы вынесены в отдельный суперинтерфейс `Collection`.

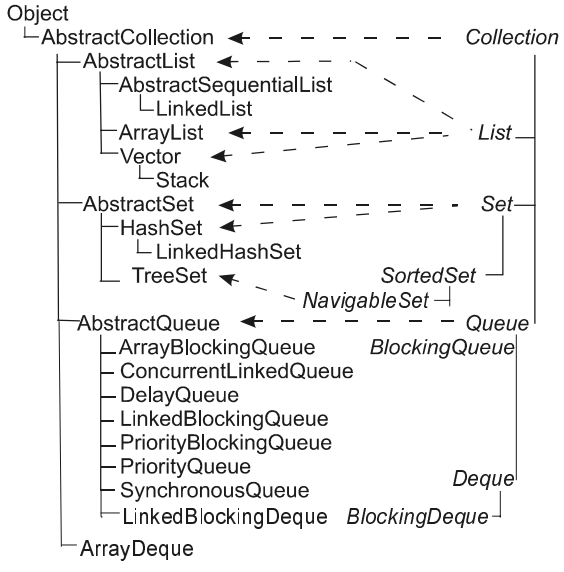


Рис. 6.3. Иерархия классов и интерфейсов-коллекций

Интерфейс `Map` не входит в эту иерархию — по мнению разработчиков Java Collections Framework, отображения типа `Map` не являются коллекциями. Они показаны на рис. 6.4.

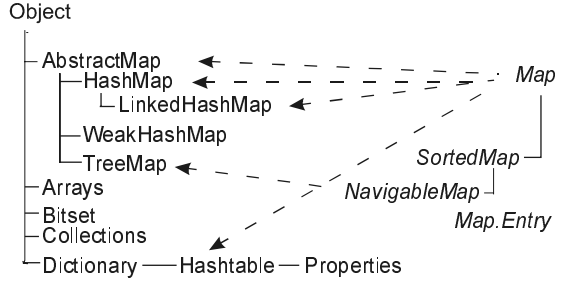


Рис. 6.4. Иерархия классов и интерфейсов-отображений

Все интерфейсы, входящие в Java Collections Framework, — настраиваемые *(см. главу 4)*, их можно использовать как шаблоны классов, хранящих ссылки на элементы одного и того же типа.

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`, `Set` и `Queue`. 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 какой-либо 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 *NavigableSet*

Интерфейс `NavigableSet` из пакета `java.util`, расширяющий интерфейс `SortedSet`, описывает отсортированное множество, в котором можно организовать бинарный поиск.

Чтобы осуществить это, в интерфейсе описаны методы, позволяющие для каждого данного элемента множества найти ближайший больший и ближайший меньший элементы того же множества.

Методы возвращают null, если элемент не удалось найти:

- Object lower(Object elem) — возвращает ссылку на наибольший элемент множества, меньший данного элемента elem;
- Object floor(Object elem) — возвращает ссылку на наибольший элемент множества, меньший или равный данному элементу elem;
- Object higher(Object elem) — возвращает ссылку на наименьший элемент множества, больший данного элемента elem;
- Object ceiling(Object elem) — возвращает ссылку на наименьший элемент множества, больший или равный данному элементу elem.

Следующие методы позволяют выделить отсортированное подмножество:

- NavigableSet subSet(Object fromElement, boolean fromInclusive, Object toElement, boolean toInclusive) — возвращает подмножество коллекции от элемента fromElement включительно, если fromInclusive == true, или исключительно, если fromInclusive == false, до элемента toElement включительно или исключительно в зависимости от истинности последнего параметра toInclusive;
- NavigableSet headSet(Object toElement, boolean inclusive) — возвращает начальные, меньшие элементы до элемента toElement включительно или исключительно в зависимости от истинности параметра inclusive;
- NavigableSet tailSet(Object fromElement, boolean inclusive) — возвращает последние, большие элементы коллекции от элемента fromElement включительно или исключительно в зависимости от истинности параметра inclusive.

Наконец, два метода удаляют наименьший и наибольший элементы множества:

- Object pollFirst() — возвращает ссылку на наименьший элемент множества и удаляет его;
- Object pollLast() — возвращает ссылку на наибольший элемент множества и удаляет его.

Интерфейс *Queue*

Интерфейс Queue из пакета java.util, расширяющий интерфейс Collection, описывает методы работы с очередями. *Очередью* называется коллекция, элементы в которую добавляются с одного конца, а удаляются с другого конца. Хороший пример такой коллекции — обычная житейская очередь в магазине или на автобусной остановке. Такой порядок обработки называется FIFO (First In — First Out, первым пришел — первым ушел).

Интерфейс Queue добавляет к методам интерфейса Collection методы, характерные для очередей:

- Object element() — возвращает первый элемент очереди, не удаляя его из очереди. Метод выбрасывает исключение, если очередь пуста;
- Object peek() — возвращает первый элемент очереди, не удаляя его. В отличие от метода element() не выбрасывает исключение;
- Object remove() — возвращает первый элемент очереди и удаляет его из очереди. Метод выбрасывает исключение, если очередь пуста;

- `Object poll()` — возвращает первый элемент очереди и удаляет его из очереди. В отличие от метода `remove()` не выбрасывает исключение;
- `boolean offer(Object obj)` — вставляет элемент в конец очереди и возвращает `true`, если вставка удалась.

Интерфейс *BlockingQueue*

Интерфейс `BlockingQueue` из пакета `java.util.concurrent`, расширяющий интерфейс `Queue`, описывает очередь, с которой работают одновременно несколько подпроцессов, вставляющих и удаляющих элементы. Их работа организуется таким образом, чтобы подпроцесс, пытающийся забрать элемент из пустой очереди, ждал, когда другой подпроцесс занесет в нее хотя бы один элемент. Подпроцесс, ставящий элемент в очередь, ждет, когда для него освободится место, если очередь уже переполнена.

Для организации такой совместной работы добавлены следующие методы:

- `Object take()` — возвращает и удаляет первый элемент, ожидая поступления элемента, если очередь пуста;
- `void put(Object element)` — ставит элемент `element` в очередь, ожидая уменьшения очереди, если она переполнена;
- `int drainTo(Collection coll, int num)` — удаляет по крайней мере `num` элементов из очереди, переписывая их в коллекцию `coll`, и возвращает их фактическое количество;
- `int drainTo(Collection coll)` — удаляет все доступные элементы из очереди, переписывая их в коллекцию `coll` и возвращая их количество.

Интерфейс *Deque*

Интерфейс `Deque` (double ended queue) из пакета `java.util`, расширяющий интерфейс `Queue`, описывает методы работы с разновидностью очередей, называемой *деком*, у которого элементы вставляются и удаляются с обоих концов.

Интерфейс `Deque` добавляет к методам интерфейса `Queue` методы, характерные для дека:

- `Object getFirst()` — возвращает первый элемент дека, не удаляя его из дека. Эквивалентен методу `element()` интерфейса `Queue`. Метод выбрасывает исключение, если дек пуст;
- `Object getLast()` — возвращает последний элемент дека, не удаляя его из дека. Метод выбрасывает исключение, если дек пуст;
- `Object peekFirst()` — возвращает первый элемент дека, не удаляя его. Эквивалентен методу `peek()` интерфейса `Queue`. Не выбрасывает исключение;
- `Object peekLast()` — возвращает последний элемент дека, не удаляя его. Не выбрасывает исключение;
- `void addFirst(Object obj)` — вставляет элемент в начало дека;
- `void addLast(Object obj)` — вставляет элемент в конец дека. Эквивалентен методу `add()` интерфейса `Collection`;
- `boolean offerFirst(Object obj)` — вставляет элемент в начало дека и возвращает `true`, если вставка удалась;

- `boolean offerLast(Object obj)` — вставляет элемент в конец дека и возвращает `true`, если вставка удалась. Эквивалентен методу `offer()` интерфейса `Queue`;
- `Object removeFirst()` — возвращает первый элемент дека и удаляет его из дека. Эквивалентен методу `remove()` интерфейса `Queue`. Метод выбрасывает исключение, если дек пуст;
- `Object removeLast()` — возвращает последний элемент дека и удаляет его из дека. Метод выбрасывает исключение, если дек пуст;
- `Object pollFirst()` — возвращает первый элемент дека и удаляет его из дека. Эквивалентен методу `poll()` интерфейса `Queue`. В отличие от метода `removeFirst()` не выбрасывает исключение;
- `Object pollLast()` — возвращает последний элемент дека и удаляет его из дека. В отличие от метода `removeLast()` не выбрасывает исключение;
- `boolean removeFirstOccurrence(Object obj)` — удаляет первый встретившийся элемент `obj` дека и возвращает `true`, если удалось это сделать. Метод выбрасывает исключение, если дек пуст;
- `boolean removeLastOccurrence(Object obj)` — удаляет последний встретившийся элемент `obj` из дека и возвращает `true`, если удалось это сделать. Метод выбрасывает исключение, если дек пуст.

Интерфейс *BlockingDeque*

Интерфейс `BlockingQueue` из пакета `java.util.concurrent`, расширяющий интерфейсы `Queue` и `Deque`, описывает дек, с которым работают одновременно несколько подпроцессов, вставляющих и удаляющих элементы. Их работа организуется таким образом, чтобы подпроцесс, пытающийся забрать элемент из пустого дека, ждал, когда другой подпроцесс занесет в него хотя бы один элемент. Подпроцесс, вставляющий элемент в дек, ждет, когда для него освободится место, если дек уже переполнен.

Для организации такой совместной работы добавлены следующие методы:

- `Object takeFirst()` — возвращает и удаляет первый элемент, ожидая поступления элемента, если дек пуст. Эквивалентен методу `take()` интерфейса `BlockingQueue`;
- `Object takeLast()` — возвращает и удаляет первый элемент, ожидая поступления элемента, если дек пуст;
- `void putFirst(Object element)` — вставляет элемент `element` в дек, ожидая уменьшения дека, если он переполнен. Эквивалентен методу `put()` интерфейса `BlockingQueue`;
- `void putLast(Object element)` — вставляет элемент `element` в дек, ожидая уменьшения дека, если он переполнен.

Интерфейс *Map*

Интерфейс `Map` из пакета `java.util` описывает своеобразную коллекцию, состоящую не из элементов, а 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 с элементами в виде пар из данного отображения, с которыми 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 отображения.

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, возвращающими:

- `Comparator comparator()` — способ упорядочения коллекции;
- `Object firstKey()` — первый, меньший элемент коллекции;
- `SortedMap headMap(Object toKey)` — начало коллекции до элемента с ключом `toKey` исключительно;
- `Object lastKey()` — последний, больший ключ коллекции;
- `SortedMap subMap(Object fromKey, Object toKey)` — часть коллекции от элемента с ключом `fromKey` включительно до элемента с ключом `toKey` исключительно;

- `SortedMap tailMap(Object fromKey)` — остаток коллекции, начинающийся от элемента `fromKey` включительно.

Интерфейс *NavigableMap*

Интерфейс `NavigableMap` из пакета `java.util`, расширяющий интерфейс `SortedMap`, описывает отсортированное по ключам отображение, в котором можно организовать бинарный поиск.

Чтобы осуществить это, в интерфейсе описаны методы, позволяющие для каждой данной пары из отображения получить ссылки на меньшую и большую ее пары или на ключ такой пары. Они возвращают `null`, если ключ не удалось найти:

- `Map lowerEntry(Object key)` — возвращает ссылку на наибольшую пару отображения с ключом, меньшим данного ключа `key`;
- `Object lowerKey(Object key)` — возвращает ссылку на наибольший ключ отображения, меньший данного ключа `key`;
- `Map floorEntry(Object key)` — возвращает ссылку на наибольшую пару отображения с ключом, меньшим или равным данному ключу `key`;
- `Object floorKey(Object key)` — возвращает ссылку на наибольший ключ отображения, меньший или равный данному ключу `key`;
- `Map higherEntry(Object key)` — возвращает ссылку на наименьшую пару отображения с ключом, большим данного ключа `key`;
- `Object higherKey(Object key)` — возвращает ссылку на наименьший ключ отображения, больший данного ключа `key`;
- `Map ceilingEntry(Object key)` — возвращает ссылку на наименьшую пару отображения с ключом, большим или равным данному ключу `key`;
- `Object ceilingKey(Object key)` — возвращает ссылку на наименьший ключ отображения, больший или равный данному ключу `key`.

Следующие методы позволяют выделить отсортированное подмножество пар:

- `NavigableMap subMap(Object fromKey, boolean fromInclusive, Object toKey, boolean toInclusive)` — возвращает подмножество отображения от пары с ключом `fromKey` включительно, если `fromInclusive == true`, или исключительно, если `fromInclusive == false`, до пары с ключом `toKey` включительно или исключительно в зависимости от истинности последнего параметра `toInclusive`;
- `NavigableMap headMap(Object toKey, boolean inclusive)` — возвращает начальные, меньшие пары до пары с ключом `toKey` включительно или исключительно в зависимости от истинности параметра `inclusive`;
- `NavigableMap tailMap(Object fromKey, boolean inclusive)` — возвращает последние, большие пары от пары с ключом `fromKey` включительно или исключительно в зависимости от истинности параметра `inclusive`.

Наконец, два метода удаляют наименьший и наибольший элементы множества:

- `Map pollFirstEntry()` — возвращает ссылку на наименьшую пару отображения и удаляет ее;
- `Map pollLastEntry()` — возвращает ссылку на наибольшую пару отображения и удаляет ее.

Абстрактные классы-коллекции

Вы можете создать свои коллекции, реализовав рассмотренные ранее интерфейсы. Это дело трудное, поскольку в интерфейсах много методов. Чтобы облегчить данную задачу, в Java Collections Framework введены частичные реализации интерфейсов — 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 `AbstractSequentialList` 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 `AbstractQueue` реализует интерфейс `Queue`, 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` и много других классов.

Для работы с указанными 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, хеширование.

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 Collections Framework 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`, описанном в конце *главы 5*.

В интерфейсе `Iterator` представлены 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 демонстрирует 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. Впрочем, для обхода коллекции типа `List` можно использовать оператор "`for-each`". Этот способ тоже показан в листинге 6.5.

Листинг 6.5. Использование итератора вектора

```
Vector<String> v = new Vector<String>();
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 элемент 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 элемент в позицию 3.
// Первый способ обхода коллекции типа List использует индексы ее элементов:
for (int i = 0; i < v.size(); i++)   // Перебираем весь вектор.
    System.out.print(v.get(i) + " ");
System.out.println();
// Второй способ обхода коллекции использует итератор:
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){}
```

```
// Третий способ обхода коллекции использует оператор for-each:
for (String s: v)            // Цикл по всем элементам вектора.
     System.out.println(s);  // Выводим текущий элемент вектора.
```

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 часть 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 находится за концом вектора.
                // Перейдем к началу вектора.
  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 полностью определенные, готовые к работе классы-коллекции Java.

Классы, создающие списки

Класс `ArrayList` полностью реализует интерфейс `List` и итератор типа `Iterator`. Класс `ArrayList` очень похож на класс `Vector`, у него тот же набор методов, он может использоваться в тех же ситуациях. Главное 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 22*.

В классе `ArrayList` три конструктора:

- `ArrayList()` — создает пустой объект;
- `ArrayList(Collection coll)` — формирует объект, содержащий все элементы коллекции `coll`;
- `ArrayList(int initCapacity)` — создает пустой объект емкости `initCapacity`.

В качестве примера использования класса `ArrayList` перепишем класс `Chorus` из листинга 3.3, используя вместо массива коллекцию.

```
public class Chorus{
  public static void main(String[] args){
     List<Voice> singer = new ArrayList<Voice>();
     singer.add(new Dog());
     singer.add(new Cat());
     singer.add(new Cow());
     for (Voice v: singer)
        v.voice();
  }
}
```

Двунаправленный список

Класс `LinkedList` полностью реализует интерфейсы `List`, `Queue` и `Deque`. Он реализует итераторы типа `Iterator` и `ListIterator`, что превращает его в двунаправленный список. Он удобен и для организации списков, стеков, очередей и деков. Класс `LinkedList` не синхронизован. Кроме того, он допускает хранение ссылок `null`.

В классе `LinkedList` два конструктора:

- `LinkedList()` — создает пустой объект;
- `LinkedList(Collection coll)` — создает объект, содержащий все элементы коллекции `coll`.
- В классе `LinkedList` реализованы только методы интерфейсов. Других методов в нем нет.

Дек

Класс `ArrayDeque` полностью реализует интерфейсы `Queue` и `Deque`. В отличие от класса `LinkedList` он синхронизован и допускает одновременную работу нескольких подпроцессов с его объектом. Кроме того, он не допускает хранение ссылок `null`. Он удобен для организации стеков, очередей и деков, тем более что он работает быстрее, чем классы `Stack` и `LinkedList`.

В классе `ArrayDeque` три конструктора:

- `ArrayDeque()` — создает пустой объект;
- `ArrayDeque(Collection coll)` — создает объект, содержащий все элементы коллекции `coll`;
- `ArrayDeque(int numElement)` — создает пустой объект емкости `numElement`.

Упражнение

1. Перепишите листинг 6.1 с использованием классов списков.

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)` — формирует 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.

Связанные отображения

Класс `LinkedHashMap` полностью реализует интерфейс `Map`. Реализация сделана в виде двунаправленного списка, а значит, его элементы хранятся в упорядоченном виде. Порядок элементов задается порядком их занесения в список.

В этом классе пять конструкторов:

- `linkedHashMap()` — создает пустой объект с емкостью в 16 элементов;
- `LinkedHashMap(int capacity)` — создает пустой объект с емкостью `capacity` элементов;

- `LinkedHashMap(int capacity, float loadFactor)` — формирует объект с емкостью `capacity` элементов и показателем загруженности `loadFactor`;
- `LinkedHashMap(int capacity, float loadFactor, boolean order)` — создает объект с емкостью `capacity` элементов, показателем загруженности `loadFactor` и порядком элементов `order`, прямым или обратным;
- `LinkedHashMap(Map sf)` — создает объект, содержащий все элементы отображения `sf`.

Упорядоченные отображения

Класс `TreeMap` полностью реализует интерфейс `SortedMap`. Класс реализован как бинарное дерево поиска, а 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 данном 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)` — формирует 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.

Хотя элементы отображения упорядочены, чтобы получить итератор для его обхода, надо преобразовать отображение во множество методом `entrySet()`, например так:

```
Iterator it = tm.entrySet().iterator();
```

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;
  }
}
```

Упражнение

2. Перепишите листинг 6.3 с использованием классов отображений.

Классы, создающие множества

Класс `HashSet` полностью реализует интерфейс `Set` и итератор типа `Iterator`. Класс `HashSet` применяется 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)` — формирует 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.

Связанные множества

Класс `LinkedHashSet` полностью реализует интерфейс `Set` и итератор типа `Iterator`. Класс реализован как двунаправленный список, значит, его элементы хранятся в упорядоченном виде. Порядок элементов задается последовательностью их занесения в объект.

В классе `LinkedHashSet` четыре конструктора, которые создают:

- `LinkedHashSet()` — пустой объект с емкостью 16 и показателем загруженности 0,75;
- `LinkedHashSet(int capacity)` — пустой объект с начальной емкостью `capacity` и показателем загруженности 0,75;
- `LinkedHashSet(int capacity, float loadFactor)` — пустой объект с начальной емкостью `capacity` и показателем загруженности `loadFactor`;
- `LinkedHashSet(Collection coll)` — объект, содержащий все элементы коллекции `coll`, с показателем загруженности 0,75.

Упорядоченные множества

Класс `TreeSet` полностью реализует интерфейс `SortedSet` и итератор типа `Iterator`. Класс `TreeSet` реализован как бинарное дерево поиска, а значит, его элементы хранятся в упорядоченном виде. Это существенно 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, создающих:

- `TreeSet()` — пустой объект с естественным порядком элементов;
- `TreeSet(Comparator c)` — пустой объект, в котором порядок задается объектом сравнения `c`;
- `TreeSet(Collection coll)` — объект, содержащий все элементы коллекции `coll`, с естественным порядком ее элементов;
- `TreeSet(SortedMap sf)` — 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`. Методов очень много, их количество увеличивается с каждой новой версией JDK, поэтому мы перечислим только основные методы.

Как обычно в статических методах, коллекция, с которой работает метод, задается его параметром.

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 класса `Collections` познакомимся по мере надобности.

Упражнение

3. Упорядочите коллекции, созданные в этой главе, и проделайте в них бинарный поиск.

Заключение

Итак, в данной главе мы выяснили, что язык Java предоставляет множество средств для работы с большими объемами информации. В массе 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.

Вопросы для самопроверки

1. Что называется коллекцией?
2. В чем отличие вектора от массива?
3. Что дает задание конкретного класса в шаблоне при определении коллекции?
4. В чем различие интерфейсов List и Set?
5. В чем различие интерфейсов List и Queue?
6. Что дополняет интерфейс Deque к интерфейсу Queue?
7. Зачем в Java введены интерфейсы NavigableSet и NavigableMap?
8. Что такое стек?
9. Что такое ассоциативный массив?
10. Что такое линейный список?
11. Что такое двунаправленный список?
12. Какие способы обхода коллекции вы знаете?

Глава 7



Классы-утилиты

В этой главе описаны средства, облегчающие работу с часто применяемыми конструкциями, в числе которых массивы, даты, случайные числа.

Работа с массивами

В классе `Arrays` из пакета `java.util` собрано множество методов для работы с массивами. Их можно разделить на несколько больших групп.

Сортировка массива

Восемнадцать статических методов класса `Arrays` 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);
```

Бинарный поиск в массиве

После сортировки массива можно организовать в нем бинарный поиск элемента `element` одним из восемнадцати статических методов поиска.

Восемь методов имеют вид:

```
static int binarySearch(type[] a, type element);
```

где `type` — один из семи примитивных типов: `byte`, `short`, `int`, `long`, `char`, `float`, `double` — или тип `Object`.

Восемь методов сортируют часть массива, начиная от элемента с индексом `from` включительно, до элемента с индексом `to` исключительно:

```
static int binarySearch(type[] a, int from, int to, type element);
```

Оставшиеся два метода поиска применяют настраиваемые типы и имеют более сложный вид:

```
static <T> int binarySearch(T[] a, T element, Comparator<? Super T> c);

static <T> int binarySearch(T[] a, int from,  int to, T element,
                            Comparator<? Super T> c);
```

Они отыскивает элемент `element` в массиве или его части, отсортированном в порядке, заданном объектом `c`.

Методы поиска возвращают индекс найденного элемента массива. Если элемент не найден, то возвращается отрицательное число, абсолютная величина которого означает индекс, с которым элемент был бы вставлен в массив в заданном порядке.

Заполнение массива

Восемнадцать статических методов класса `Arrays` 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`.

Копирование массива

Восемь методов класса `Arrays`, написанных для всех примитивных типов, обозначенных здесь словом `type`:

```
static type[] copyOf(type[] a, int newLength);
```

копируют массив `a`, возвращая ссылку на копию массива. Они обрезают массив `a` до длины `newLength`, если `newLength` меньше длины массива `a`, или дополняют массив нулями, если его длина меньше `newLength`.

Следующий метод использует настраиваемый тип:

```
static <T> T[] copyOf(T[] a, int newLength);
```

Еще один метод копирования позволяет изменить тип массива:

```
static <T, U> T[] copyOf(U[] a, int newLength, Class<? extends T[]> newType);
```

При несовместимости старого и нового типов возникает исключитеьная ситуация.

Часть массива, начиная от элемента с индексом `from` и заканчивая перед элементом с индексом `to`, можно скопировать одним из десяти методов `copyOfRange()`. Восемь из них написаны для примитивных типов. Они имеют вид:

```
static type[] copyOfRange(type[] a, int from, int to);
```

Индекс `to` может оказаться больше длины массива, в таком случае массив-копия будет дополнен нулями.

Девятый метод использует настраиваемый тип массива:

```
static <T> T[] copyOfRange(T[] a, int newLength, int from, int to);
```

Десятый метод может изменить тип копируемого массива:

```
static <T, U> T[] copyOfRange(U[] a, int from, int to, Class<? ex-
tends T[]> newType);
```

При несовместимости старого и нового типов возникает исключительная ситуация.

Сравнение массивов

Девять статических логических методов класса `Arrays` 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 все элементы массивов с одинаковыми индексами.

Еще один метод сравнения массивов:

```
static boolean deepEquals(Object[] a1, Object[] a2);
```

полезен для сравнения многомерных массивов, поскольку он сравнивает элементы массивов с любым количеством индексов.

Представление массива строкой

Девять статических методов класса `Arrays` преобразуют массив в строку:

```
static String toString(type[] a);
```

где `type` — один из восьми примитивных типов или тип `Object`.

В формируемой строке массив записывается в квадратных скобках, а его элементы перечисляются через запятую и пробел. Каждый элемент преобразуется в строку методом `String.valueOf(type)`.

Еще один метод преобразования массива в строку:

```
static String deepToString(Object[] a);
```

применяется для преобразования многомерных массивов. Он рекурсивно просматривает их подмассивы с любым количеством индексов.

Получение хеш-кода массива

Девять статических методов класса `Arrays` вычисляют хеш-код массива:

```
static int hashCode(type[] a);
```

где `type` — один из восьми примитивных типов или тип `Object`.

Сначала массив представляется списком типа `List`, а затем вычисляется хеш-код списка методом `List.hashCode()`.

Еще один метод:

```
static int deepHashCode(Object[] a);
```

применяется для вычисления хеш-кода многомерных массивов. Он просматривает их подмассивы с любым количеством индексов.

В листинге 7.1 приведен простой пример работы с некоторыми из методов класса `Arrays`.

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: a)
        System.out.print(a[i] + " ");
    System.out.println();
    Arrays.fill(a, Arrays.binarySearch(a, 12), a.length, 0);
    for (int i: a)
        System.out.print(a[i] + " ");
    System.out.println();
  }
}
```

Локальные установки

Некоторые данные — даты, время — традиционно представляются в различных местностях по-разному. Например, дата в России выводится в формате `число.месяц.год` (через точку): 27.06.08. В США принята запись `месяц/число/год` (через наклонную черту): 06/27/08.

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 следующие строки (см. также 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 сведения о 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 (Java Virtual Machine, виртуальная машина Java), 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 из пакета `java.util`: `Calendar` и `Date`.

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 параметра `when`; нуль, если времена совпадают; положительное число, если данное время больше параметра `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 (Daylight Savings Time, DST), 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 большинства 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.

Метод get(int field) возвращает элемент календаря, заданный параметром field. Для этого параметра 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

Метод set(int field, int value), использующий эти константы, устанавливает соответствующие значения даты и времени, оставляя остальные значения без изменения. Еще несколько методов set() устанавливают дату и время по дням, часам, минутам и другим элементам.

Метод setTime(Date d), наиболее часто применяемый для заполнения календаря, устанавливает в календарь все элементы даты d полностью.

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 на григорианский календарь 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()`.

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: 27.12.04 17:32; в локали США: 12/27/04 5:32 PM;
- стиль `MEDIUM` задает год четырьмя цифрами и показывает секунды: 27.12.2004 17:32:45; в локали США месяц представляется тремя буквами;
- стиль `LONG` представляет месяц словом и добавляет часовой пояс: 27 декабрь 2004 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: `27-12-2004 17.32`.

В шаблоне буква `d` означает цифру дня месяца, `M` — цифру месяца, `y` — цифру года, `h` — цифру часа, `m` — цифру минут. Полностью 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

Для копирования массивов, кроме новых методов `copyOf()` и `copyOfRange()` класса `Arrays()`, можно применить статический метод копирования массивов из класса `System` пакета `java.lang`, 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 полностью полагаться 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 параметрами.

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 здесь 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 текстовый редактор Notepad (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.



ЧАСТЬ III

Создание графического интерфейса пользователя и апплетов



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.

За десятилетия развития вычислительной техники создано 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 запутаны. Графические API, предназначенные для создания пользовательского интерфейса, содержат сотни функций.

Для облегчения создания окон и их компонентов написаны библиотеки функций и классов: MFC, Motif, OpenLook, Qt, Tk, Xview, OpenWindows, OpenGL, GTK+ 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 графической библиотеки Java реализуют эти интерфейсы для создания приложений. Приложения Java используют методы этих интерфейсов для размещения и перемещения графических объектов, изменения их размеров, взаимодействия объектов друг с другом.

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 самой 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). Для вывода на экран объекта, созданного в приложении Java и основанного на peer-интерфейсе, создается парный ему (peer-to-peer) объект графической подсистемы операционной системы, который и отображается на экране. 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 этой графической среды.

Пара объектов, реализующих один peer-интерфейс, тесно взаимодействуют во время работы приложения. Изменение объекта в приложении Java немедленно влечет изменение объекта графической оболочки и меняет его вид на экране.

Именно из-за такой реализации peer-интерфейсов и других "родных" (native) 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 их 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 еще какой-нибудь другой, и сменить этот вид в любой момент работы.

Эта интересная особенность "легких" компонентов получила название PL&F (Pluggable Look and Feel). Это сокращение иногда записывают в виде "plaf".

Тогда же была создана обширная библиотека "легких" компонентов Java, названная *Swing*. В ней были переписаны все компоненты библиотеки AWT, так что компоненты библиотеки Swing могут 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 следующие версии Java SE JDK она включена наряду с AWT как основная графическая библиотека классов, реализующая идею "100% Pure Java".

В Java SE библиотека AWT значительно расширена не только библиотекой Swing, но и 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 SE 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 перечисленные средства Java SE: 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 библиотек AWT и Swing.

Компонент и контейнер

Основное понятие графического интерфейса пользователя (ГИП) — *компонент* (component) графической системы. В русском языке слово "компонент" 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, поле ввода, кнопка, строка меню, полоса прокрутки, радиокнопка (переключатель). 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 поместить компоненты на экран. 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 контейнера `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 действия мыши и клавиатуры. В окне должны быть как минимум следующие стандартные компоненты:

- *строка заголовка* (title bar), с левой стороны которой необходимо поместить кнопку контекстного меню, а с правой — кнопки сворачивания и разворачивания окна и кнопку закрытия приложения;
- окно должно быть окружено *рамкой* (border), реагирующей на действия мыши.

Окно с этими компонентами в готовом виде описано в классе `Frame`. Чтобы создать окно в библиотеке AWT, 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 AWT

```
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 методом `setSize()` устанавливаются размеры окна на экране — 400×150 пикселов. Если не задать размер окна, то на экране появится окно минимального размера — будет видна только строка заголовка. Конечно, потом окно 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, 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 завершать 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");
  }
}
```

Для того чтобы показать разные варианты построения программы, в нее 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 15*, а пока просто добавляйте приведенные строчки во все ваши программы, использующие библиотеку AWT, для закрытия окна и завершения работы приложения.

Итак, окно готово. Но оно пока пусто. Выведем в него, по традиции, приветствие "Hello, XXI Century World!", правда, слегка измененное. В листинге 8.3 представлена 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` всегда наследует этот метод, но с пустой реализацией.

Метод `paint()` получает в качестве аргумента экземпляр `g` класса `Graphics`, умеющего, в частности, выводить на экран 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.

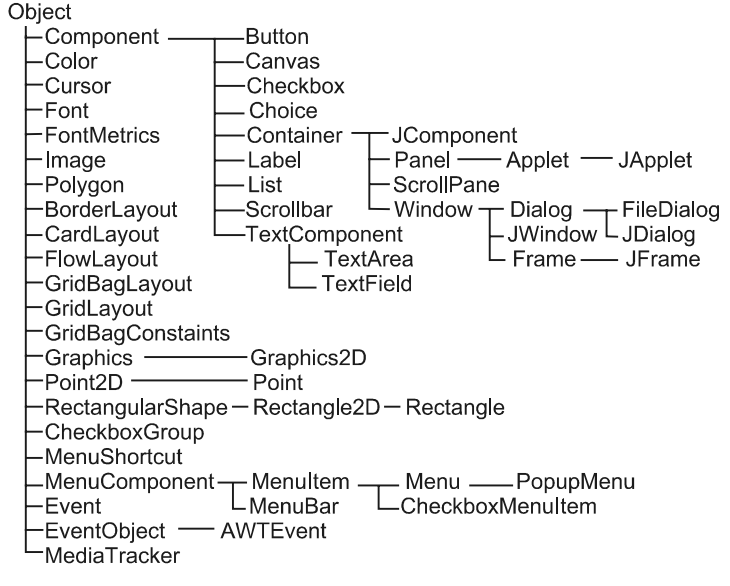


Рис. 8.2. Иерархия основных классов AWT

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`.

Окно библиотеки Swing

Для получения окна с помощью средств библиотеки Swing необходимо импортировать в свою программу пакет `javax.swing` и расширить класс `JFrame`, как показано в листинге 8.4. Вместо длинного метода закрытия окна можно обратиться к методу

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

указав в нем константу `EXIT_ON_CLOSE` класса `JFrame`, предписывающую завершить работу приложения при закрытии окна. Другие константы, определенные в интерфейсе `WindowConstants`, предписывают:

- `DISPOSE_ON_CLOSE` — закрыть окно и освободить память, занимаемую им, но не завершать приложение;
- `DO_NOTHING_ON_CLOSE` — игнорировать команду закрытия окна;
- `HIDE_ON_CLOSE` — только убрать окно с экрана. Это значение по умолчанию.

Фон окна Swing серый, поэтому в конструктор добавлен еще метод `setBackground(Color.WHITE)`, устанавливающий белый цвет фона.

Листинг 8.4. Простое окно приложения Swing

```
import java.awt.*;
import javax.swing.*;
class SimpleFrame extends JFrame{
  SimpleFrame(String s){
      super(s);
      getContentPane().setBackground(Color.WHITE);
      setSize(400, 150);
      setVisible(true);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
  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){
      new SimpleFrame("  Моя программа");
  }
}
```

Метод `paint()` принадлежит классу `Component`, он выглядит точно так же, как в листинге 8.3. Его можно без всяких изменений перенести в программу, использующую библиотеку Swing.

Использование системных приложений

В большинстве операционных систем пользователь устанавливает для себя браузер по умолчанию, который открывается, когда пользователь выбирает файл с расширением html. Файлы с другими расширениями тоже часто связываются с приложениями, обрабатывающими их. Например, файл с расширением txt часто открывается в текстовом редакторе. Кроме того, пользователь может выбрать для себя почтовый клиент по умолчанию.

Эти возможности использует класс `Desktop` из пакета `java.awt`. У него есть методы, позволяющие запустить некоторые приложения пользователя:

- `browse(URI file)` — открывает браузер по умолчанию, загружающий указанный файл `file`;
- `mail()` — открывает почтовый клиент по умолчанию;
- `mail(URI mailto)` — открывает почтовый клиент по умолчанию, заполняя поля “To”, “Cc”, “Subject”, “Body” значениями, взятыми из аргумента `mailto`;
- `edit(File file)` — открывает текстовый редактор, связанный с указанный файлом `file`, и загружает в него файл `file`;
- `open(File file)` — открывает указанный файл `file`;
- `print(File file)` — печатает указанный файл `file` на принтере, назначенном этому файлу.

Связь с приложениями пользователя устанавливается через операционную систему, поэтому экземпляр класса `Desktop` создается не конструктором, а статическим методом `getDesktop()`. Эту связь удается установить не во всех системах, поэтому предварительно следует сделать проверку статическим логическим методом `isDesktopSupported()`. Итак, создание экземпляра класса `Desktop` выглядит так:

```
Desktop d = null;
if (Desktop.isDesktopSupported())
    d = Desktop.getDesktop();
```

Вызвать каждое из перечисленных выше приложений тоже удается не всегда, поэтому перед обращением к какому-либо методу класса `Desktop` имеет смысл сделать проверку логическим методом

```
public boolean isSupported(Desktop.Action action);
```

Аргументом этого метода может служить одна из следующих констант вложенного перечисления `Action`:

- `BROWSE` — у пользователя есть браузер по умолчанию;
- `MAIL` — у пользователя есть почтовый клиент по умолчанию,
- `EDIT` — можно открыть текстовый редактор, связанный с файлами;
- `OPEN` — можно открыть файл;
- `PRINT` — можно напечатать файл.

С учетом этой проверки, обращение к методам класса `Desktop` будет выглядеть так:

```
if (d.isSupported(Desktop.Action.BROWSE))
       d.browse(new URI("http://www.bhv.ru"));
```

```
if (d.isSupported(Desktop.Action.MAIL))
       d.mail();
if (d.isSupported(Desktop.Action.EDIT))
       d.edit(new File("/home/user/java/src/MyDesktop.java");
if (d.isSupported(Desktop.Action.OPEN))
       d.open(new File("/home/user/java/src/MyDesktop.java");
if (d.isSupported(Desktop.Action.PRINT))
       d.print new File("/home/user/java/src/MyDesktop.java");
```

System Tray

Графическое приложение Java может установить ярлык в зоне уведомлений (notification area), называемой также системным лотком (system tray) или просто треем. Эта зона обычно размещается в правом нижнем углу экрана и содержит часы и ярлыки запущенных программ. Для работы с зоной уведомлений в пакет `java.awt` включен класс `SystemTray`. Его метод `add(TrayIcon icon)` помещает ярлык `icon` в зону уведомлений, а метод `remove(TrayIcon icon)` удаляет его из зоны. Каждое приложение может поместить в зону уведомлений несколько ярлыков.

Как видно из заголовков, параметр этих методов, ярлык — это объект класса `TrayIcon`. Он создается конструктором класса. Объект активен — он может реагировать на события, такие как действия мыши, открывая, например, всплывающее меню после щелчка на ярлыке правой кнопкой мыши или меняя изображение при наведении на него курсора мыши. Обработка таких событий описана в *главе 15*.

Как и при действиях с приложениями пользователя, не каждая система позволяет управлять зоной уведомлений. Поэтому работа с этой зоной связана с проверками и выглядит следующим образом:

```
TrayIcon icon = null;
if (SystemTray.isSupported()){
    SystemTray tray = SystemTray.getSystemTray();
    Image im = Toolkit.getDefaultToolkit.getImage("myicon.gif");
    icon = new TrayIcon(im);
    tray.add(icon);
 }
```

Более подробно работа с зоной уведомлений показана в документации к классу `SystemTray`. Она будет понятна после прочтения *главы 15*.

Splash Screen

Очень часто при загрузке приложения на экране вначале появляется небольшое окно-заставка (splash screen) с каким-нибудь изображением, сменяемое затем главным окном приложения. Такое окно можно открыть при запуске приложения из командной строки, указав ключ `-splash`. Например, если файл с изображением называется name.gif, то запустить приложение можно так:

```
java -splash:name.gif SimpleFrame
```

При запуске приложения из архива, например

```
java -jar SimpleFrame.jar
```

имя файла с изображением записывается в файл MANIFEST.MF, как показано в *главе 25*.

Некоторые возможности управления окном-заставкой предоставляет класс `SplashScreen` из пакета `java.awt`. Это возможность менять изображение методом `setImageURL(URL image)`, и возможность рисовать в окне, получив ссылку на объект класса `Graphics2D` методом `createGraphics()`. После заполнения окна-заставки оно выводится на экран методом `update()`. Все это делается по следующей схеме:

```
SplashScreen splash = SplashScreen.getSplashScreen();
if (splash != null){
    Graphics2D g = splash.createGraphics();
    g.setPaintMode();
    g.drawString("Loading...", 100, 200);
    // и.т.д., рисуем, как написано в главе 9.
    g.update();
 }
```

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, SWAT, SWT 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 библиотеки AWT и Swing для создания собственных приложений с графическим интерфейсом пользователя, изображениями, анимацией и звуком.

Вопросы для самопроверки

1. Что такое графический интерфейс пользователя?
2. Что такое графическая библиотека классов?
3. Что называется графическим компонентом?
4. Назовите известные вам графические компоненты.
5. Что такое контейнер в графическом интерфейсе?
6. Будет ли основное окно приложения контейнером?
7. Можно ли использовать библиотеку Swing без библиотеки AWT?
8. Какая разница между компонентами AWT и компонентами Swing?

Глава 9



Графические примитивы

При создании графического 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 — точка с координатами (0, 0) — расположено в верхнем левом углу области рисования, ось *Ox* направлена вправо, ось *Oy* — вниз. Точки координатной системы находятся между точками области рисования.

Управляет контекстом класс `Graphics` или более новый класс `Graphics2D`, созданный в рамках библиотеки Java 2D. 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 передает его 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 (отсут-

ствие составляющей цвета) до 255 (полная интенсивность этой составляющей цвета). Например, следующие строки:

```
Color pureRed = new Color(255, 0, 0);
Color pureGreen = new Color(0, 255, 0);
```

определяют чистый ярко-красный цвет `pureRed` и чистый ярко-зеленый цвет `pureGreen`.

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(0xFF8F48AF);
```

Здесь красная составляющая задана с интенсивностью `0x8F`, зеленая — `0x48`, синяя — `0xAF`.

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, нижний, цвет не просвечивает сквозь него. Если альфа равна 0 или 0.0, то цвет абсолютно прозрачен, — для каждого пиксела виден только предыдущий цвет. Промежуточные значения позволяют создать полупрозрачные изображения, сквозь которые просвечивает фон.

Последний из этих конструкторов учитывает альфа-составляющую, 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`.

Можно сказать, что 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, CIE XYZ, 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" в первых версиях JDK их записывали строчными буквами: `black`, `blue`, `cyan`, `darkGray`, `gray`, `green`, `lightGray`, `magenta`, `orange`, `pink`, `red`, `white`, `yellow`. В последние версии JDK добавили те же константы, записанные прописными буквами: `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE`, `YELLOW`. Сейчас можно использовать и ту и другую запись.

В классе `SystemColor`, расширяющем класс `Color`, собраны константы, выражающие системные цвета десктопа, окна, меню, текста, различных системных компонентов. С их помощью можно оформить приложение стандартными цветами графической оболочки операционной системы.

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 изображение активного компонента ярким цветом или, наоборот, показать неактивный компонент бледнее остальных компонентов.

Два статических метода возвращают цвет, преобразованный из цветовой модели RGB в модель 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.

Упражнение

1. Создайте чисто желтый цвет в разных цветовых моделях.

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 параметр `raised` равен `true`, или как будто вдавленный в плоскость, если параметр `raised` равен `false`;
- `drawOval(int x, int y, int width, int height)` — чертит овал, вписанный в прямоугольник, заданный параметрами 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 параметрами. Дуга имеет величину `arc` градусов и отсчитывается от угла `startAngle`. Угол отсчитывается в градусах от оси *Ox*. 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 рисуются 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 координаты вершин многоугольника `(xPoints[i], yPoints[i])` и их число `nPoints`. Несоответствие параметров вызывает исключительную ситуацию.

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 параметрами метода точка, отрезок прямой или целый прямоугольник со сторонами, параллельными сторонам экрана:

```
boolean contains(int x, int y);
boolean contains(double x, double y);
boolean contains(Point p);
boolean contains(Point2D p);
boolean contains(double x, double y, double width, double height);
boolean contains(Rectangle2D rectangle);
```

Два логических метода `intersects()` позволяют проверить, не пересекается ли с данным многоугольником отрезок прямой, заданный параметрами метода, или прямоугольник со сторонами, параллельными сторонам экрана:

```
boolean intersects(double x, double y, double width, double height);
boolean intersects(Rectangle2D rectangle);
```

Методы `getBounds()` и `getBounds2D()` возвращают прямоугольники класса `Rectangle` и класса `Rectangle2D` соответственно, целиком содержащие в себе данный многоугольник.

Упражнение

2. Сделайте рисунок по описанию: "Точка, точка, запятая, минус — рожица кривая".

Прочие методы класса *Graphics*

Вернемся к методам класса `Graphics`. Несколько методов вычерчивают фигуры, залитые текущим цветом: `fillRect()`, `fill3DRect()`, `fillArc()`, `fillOval()`, `fillPolygon()`, `fillRoundRect()`. У них такие же параметры, 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 указать 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);
```

Во всех этих методах 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, а именно объект класса `Font`. Посмотрим, какие возможности предоставляет данный класс.

Как задать шрифт

Объекты класса `Font` хранят *начертания* (glyphs) символов, образующие шрифт. Их можно создать двумя конструкторами:

- `Font(Map attributes)` — задает шрифт с указанным 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.

Именем шрифта `name` может быть строка 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[] fonts = Toolkit.getGraphicsEnvironment.getAllFonts();
for (Font f: fonts)
     System.out.println(f.getFontName());
```

В состав Sun Java SE входит семейство шрифтов Lucida. Установив JDK, вы можете быть уверены, что эти шрифты есть в вашей системе.

Таблицы сопоставления логических и физических имен шрифтов находятся в виртуальной машине Java или в файлах с именами:

- fontconfig.properties;
- fontconfig.Me.properties;
- fontconfig.2000.XP.properties;
- fontconfig.XP.properties;
- fontconfig.2003.properties;
- fontconfig.RedHat.properties

и т. д.

Эти файлы должны быть расположены в JDK в каталоге jdk1.6.0_02/jre/lib или каком-либо другом подкаталоге lib корневого каталога JDK той машины, на которой выполняется приложение.

Файлы хранятся в исходном виде, с расширением src, и в откомпилированном виде, с расширением bfc.

Нужный файл выбирается виртуальной машиной Java по названию операционной системы. Если такой файл не найден, то применяется файл fontconfig.properties, не соответствующий никакой конкретной операционной системе.

Поэтому можно оставить в системе только один файл fontconfig.properties, переписав в него содержимое нужного файла или создав файл заново. Для любой операционной системы будет использоваться именно он.

В листинге 9.1 показано сокращенное содержимое файла fontconfig.properties.src из JDK 5.0 для платформы MS Windows.

Листинг 9.1. Примерный файл fontconfig.properties.src

```
# @(#)fontconfig.properties  1.5 04/01/28
#
# Copyright 2003 Sun Microsystems, Inc. All rights reserved.
#
# Version
version=1
# Component Font Mappings
```

```
allfonts.chinese-ms936=SimSun
allfonts.chinese-gb18030=SimSun-18030
allfonts.chinese-hkscs=MingLiU_HKSCS
allfonts.devanagari=Mangal
allfonts.dingbats=Wingdings
allfonts.lucida=Lucida Sans Regular
allfonts.symbol=Symbol
allfonts.thai=Lucida Sans Regular
serif.plain.alphabetic=Times New Roman
serif.plain.chinese-ms950=MingLiU
serif.plain.hebrew=David
serif.plain.japanese=MS Mincho
serif.plain.korean=Batang
serif.bold.alphabetic=Times New Roman Bold
# И так далее
serif.italic.alphabetic=Times New Roman Italic
# И так далее
serif.bolditalic.alphabetic=Times New Roman Bold Italic
# И так далее
sansserif.plain.alphabetic=Arial
# И так далее
monospaced.plain.alphabetic=Courier New
# И так далее
dialog.plain.alphabetic=Arial
# И так далее
dialoginput.plain.alphabetic=Courier New
# И так далее
# Search Sequences
sequence.allfonts=alphabetic/default,dingbats,symbol
# И так далее
# Exclusion Ranges
exclusion.alphabetic=0700-1e9f,1f00-20ab,20ad-f8ff
exclusion.chinese-gb18030=0390-03d6,2200-22ef,2701-27be
exclusion.hebrew=0041-005a,0060-007a,007f-00ff,20ac-20ac
# Monospaced to Proportional width variant mapping
# (Experimental private syntax)
proportional.MS_Gothic=MS PGothic
proportional.MS_Mincho=MS PMincho
proportional.MingLiU=PMingLiU
# Font File Names
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.SimSun=SIMSUN.TTC
filename.SimSun-18030=SIMSUN18030.TTC
filename.MingLiU=MINGLIU.TTC
filename.PMingLiU=MINGLIU.TTC
filename.MingLiU_HKSCS=hkscsm3u.ttf
filename.David=DAVID.TTF
filename.David_Bold=DAVIDBD.TTF
filename.MS_Mincho=MSMINCHO.TTC
filename.MS_PMincho=MSMINCHO.TTC
filename.MS_Gothic=MSGOTHIC.TTC
filename.MS_PGothic=MSGOTHIC.TTC
filename.Gulim=gulim.TTC
filename.Batang=batang.TTC
filename.GulimChe=gulim.TTC
filename.Lucida_Sans_Regular=LucidaSansRegular.ttf
filename.Mangal=MANGAL.TTF
filename.Symbol=SYMBOL.TTF
filename.Wingdings=WINGDING.TTF
```

Большая часть этого файла занята сопоставлениями логических и физических имен. Вы видите, что:

- ❒ 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 другие строки листинга 9.1. Строка

```
exclusion.alphabetic=0700-1e9f,1f00-20ab,20ad-f8ff
```

означает, что в алфавитных шрифтах не станут отыскиваться начертания (glyphs) символов с кодами в диапазонах `'\u0700'—'\u1e9f'`, `'\u1f00'—'\u20ab'` и `'\u20ad'—'\uf8ff'`.

Они будут взяты из шрифта, следующего далее в строке

```
sequence.allfonts=alphabetic/default,dingbats,symbol
```

а именно из шрифта Wingdings.

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 fontconfig.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 соответствующего класса, например класса `CharToByteCp1251`.

Эти вопросы обсуждаются в документации Sun Java SE в файле docs/technotes/guides/intl/fontconfig.html.

Завершая рассмотрение логических и физических имен шрифтов, следует сказать, что в JDK 1.0 использовались логические имена `"Helvetica"`, `"TimesRoman"`, `"Courier"`, из лицензионных соображений замененные в JDK 1.1 на `"SansSerif"`, `"Serif"`, `"Monospaced"` соответственно.

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`, следует обратиться к методу `getFontMetrics(Font 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 `р`, `у`, 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 javax.swing.*;
class GraphTest extends JFrame{
  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.setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
}
```

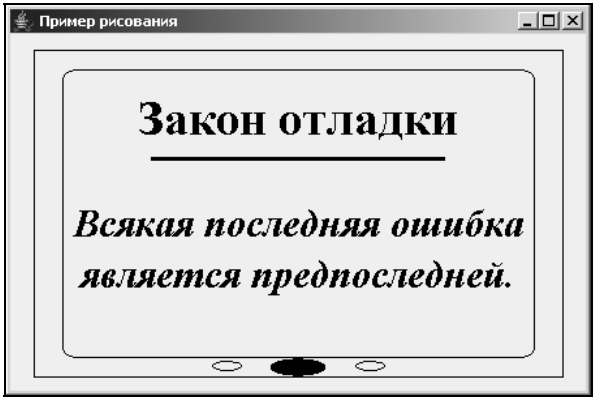


Рис. 9.2. Пример использования класса `Graphics`

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.

Класс `Graphics`, в рамках системы Java 2D, значительно расширен классом `Graphics2D`.

Упражнение

3. Поупражняйтесь в выводе текста различными шрифтами с разным расположением на экране.

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 объ-

ект, 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 — это статическая константа `CAP_ROUND`, прямым обрезом — константа `CAP_SQUARE` (по умолчанию) или не фиксировать определенный способ окончания — константа `CAP_BUTT`;
  - оно может сопрягать линии (line joins) закруглением — статическая константа `JOIN_ROUND`, отрезком прямой — константа `JOIN_BEVEL` или просто состыковывать — константа `JOIN_MITER` (по умолчанию);
  - оно может чертить линию различными пунктирами (dash) и штрихпунктирами, при этом длины штрихов и промежутков задаются в массиве, элементы которого с четными индексами задают длину штриха, а с нечетными индексами — длину промежутка между штрихами.
- Методы заполнения фигур описаны в интерфейсе `Paint`. Несколько классов 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). Класс `MultipleGradientPaint` организует градиентную заливку с несколькими градиентами, причем его подкласс `LinearGradientPaint` делает линейную заливку, а подкласс `RadialGradientPaint` — радиальную заливку.
- 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 фигур и их 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. Параметром 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 параметра массив из шести элементов-коэффициентов преобразования `{a, b, c, d, e, f}` или массив из четырех элементов `{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 копию другого, уже имеющегося, объекта:

```
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)`.
- `getRotateInstance(double vx, double vy)` — поворот, заданный вектором с координатами `(vx, vy)`. Эквивалентен методу `getRotateInstance(Math.atan2(vx, vy))`.
- `getRotateInstance(double vx, double vy, double x, double y)` — поворот вокруг точки с координатами `(x, y)`, заданный вектором с координатами `(vx, vy)`. Эквивалентен методу `getRotateInstance(Math.atan2(vx, vy), x, y)`.
- `getQuadrantRotateInstance(int n)` — поворот `n` раз на угол 90° вокруг начала координат. Эквивалентен методу `getRotateInstance(n * Math.PI / 2.0)`.
- `getQuadrantRotateInstance(int n, double x, double y)` — поворот `n` раз на угол 90° вокруг точки с координатами `(x, y)`. Эквивалентен методу `getRotateInstance(n * Math.PI / 2.0, 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);
setToRotation(double vx, double vy);
setToRotation(double vx, double vy, double x, double y);
setToQuadrantRotation(int n);
setToQuadrantRotation(int n, 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);
rotate(double vx, double vy);
rotate(double vx, double vy, double x, double y);
quadrantRotate(int n);
```

```
quadrantRotate(int n, 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()`, показанного в листинге 9.2, строку импорта

```
import java.awt.geom.*;
```

и 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
```

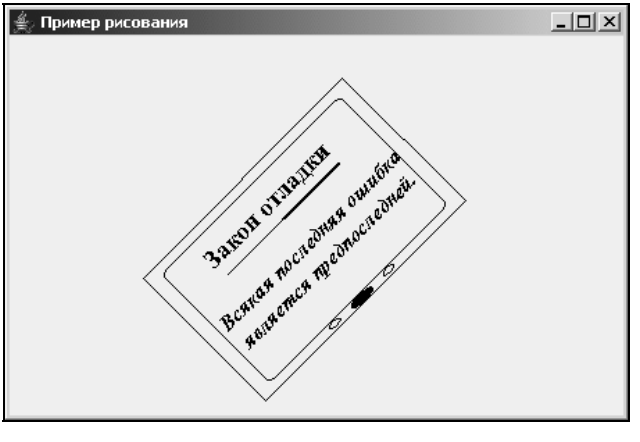


Рис. 9.3. Преобразование координат

Метод `paint()` начинается с получения экземпляра `g` класса `Graphics2D` простым приведением аргумента `gr` к типу `Graphics2D`. Затем, методом `getRotateInstance()` опреде-

ляется 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.

Упражнение

4. Напишите полную программу по листингу 9.3 и выполните ее несколько раз, меняя коэффициенты преобразований.

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 определены 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 javax.swing.*;

class StrokeTest extends JFrame{
  StrokeTest(String s){
    super(s);
    setSize(500, 400);
    setVisible(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
  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("  Различные перья");
  }
}
```

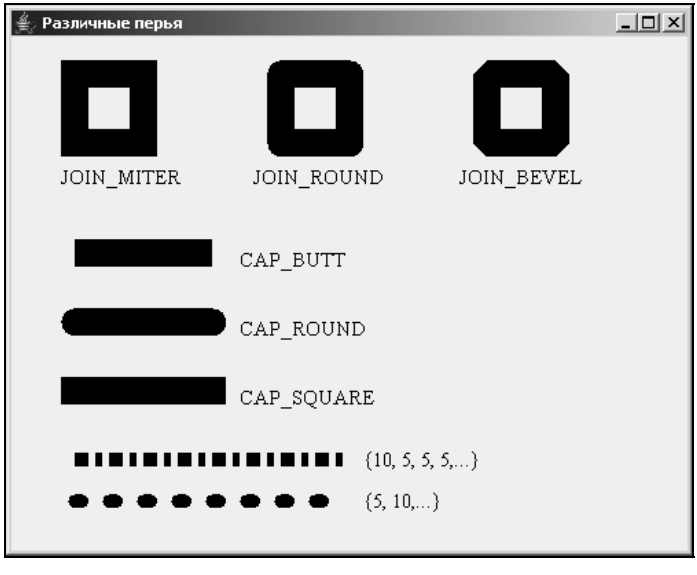


Рис. 9.4. Перья с различными характеристиками

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`. Данный 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.

Класс *GeneralPath*

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`.

Объект класса `GeneralPath` строится так. 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 несколько реализаций этого интерфейса — классы `Color`, `GradientPaint`,

TexturePaint, абстрактный класс MultipleGradientPaint и его расширения LinearGradientPaint и RadialGradientPaint. 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 20*, 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 являются 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 демонстрируют листинг 9.5 и рис. 9.5.

Листинг 9.5. Способы заливки

```
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;
```

```
import javax.swing.*;
class PaintTest extends JFrame{
  PaintTest(String s){
    super(s);
    setSize(300, 300);
    setVisible(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
  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");
  }
}
```

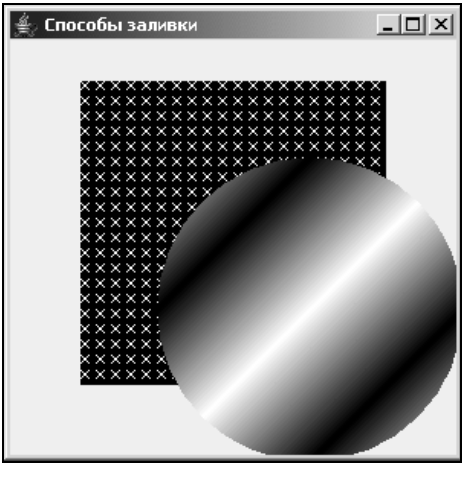


Рис. 9.5. Способы заливки

Классы *LinearGradientPaint* и *RadialGradientPaint*

Классы `LinearGradientPaint` и `RadialGradientPaint` позволяют сделать градиентную заливку несколькими цветами, повторяя их вдоль прямой или радиально, вдоль радиуса окружности. Общие свойства этих классов собраны в абстрактном суперклассе `MultipleGradientPaint`.

Так же, как и в классе `GradientPaint`, задается промежуток `MN`, но теперь промежуток делится на несколько частей точками с вещественными координатами от 0.0 — начало промежутка, до 1.0 — конец промежутка. Эти точки заносятся в массив, например:

```
float[] dist = {0.0f, 0.2f, 1.0f};
```

Здесь промежуток разделен на две неравные части: одна пятая часть и четыре пятых части.

В каждой точке деления промежутка задается свой цвет. Цвета тоже записываются в массиве:

```
Color[] color = {Color.RED, Color.WHITE, Color.BLUE};
```

В каждой части промежутка цвет плавно меняется от одного цвета к другому. В нашем примере на одной пятой части промежутка идет переход от красного к белому цвету, а на оставшихся четырех пятых — переход от белого к синему цвету.

После этого для линейной заливки создается экземпляр класса `LinearGradientPaint`:

```
LinearGradientPaint(float x1, float y1, float x2, float y2,
                    float[] dist, Color[] color);
```

а для радиальной заливки — экземпляр класса `RadialGradientPaint`.

```
RadialGradientPaint(float x, float y, float radius,
                    float[] dist, Color[] color);
```

Как вы поняли, для линейной заливки задаются координаты начальной `M(x1, y1)` и конечной `N(x2, y2)` точки промежутка, а для радиальной заливки — координаты центра круга `A(x, y)` и радиус окружности `radius`.

Способы задания цветов вне промежутка `MN` заданы константами вложенного класса `MultipleGradientPaint.CycleMethod`. В шестой версии Java SE есть три способа:

- `NO_CYCLE` — используются первый и последний цвет (по умолчанию);
- `REFLECT` — перед промежутком циклически перебираются цвета от первого до последнего, а после промежутка — от последнего до первого;
- `REFLECT` — циклически перебираются цвета от первого до последнего.

Эти константы указываются в следующих конструкторах:

```
LinearGradientPaint(float x1, float y1, float x2, float y2,
    float[] dist, Color[] color, MultipleGradientPaint.CycleMethod method);
RadialGradientPaint(float x, float y, float radius,
    float[] dist, Color[] color, MultipleGradientPaint.CycleMethod method);
```

Еще несколько конструкторов задают точки как объекты класса `Point2D`.

По умолчанию оба класса используют пространство цветов RGB, но соответствующими конструкторами можно задать и другое пространство цветов.

После создания заливки — объекта класса `LinearGradientPaint` или `RadialGradientPaint` — она устанавливается в графическом контексте методом `setPaint(Paint p)` и используется в дальнейшем методом `fill(Shape sh)`.

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`
`KERLING`	Керлинг — сдвиг букв в слове с целью уменьшения промежутков между ними, например в слове “AWAY”. Константа `KERLING_ON`.
`LIGATURES`	Лигатура — слияние букв, например в слове “float”. Константа `LIGATURES_ON`.
`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`

Таблица 9.1 *(окончание)*

Атрибут	Значение
SWAP_COLORS	Замена местами цвета текста и цвета фона. Константа SWAP_COLORS_ON, по умолчанию замены нет
TRAKING	Трекинг — пропорциональное изменение расстояний между буквами. Константа TRAKING_TIGHT увеличивает расстояния, а константа TRAKING_LOOSE уменьшает их.
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
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. Например:

```
Font f = new Font("Times New Roman", Font.BOLD, 12);
AttributedCharacterIterator.Attribute[] a = f.getAvailableAttributes();
for (int i = 0; i < a.length; i++)
    System.out.println(a[i]);
```

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 о 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 графической 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 javax.swing.*;
class StillText extends JFrame{
  StillText(String s){
    super(s);
    setSize(400, 200);
    setVisible(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
  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` применяется 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 вы-

вести измененный текст методом `drawGlyphVector()` класса `Graphics2D`. Все это показано в листинге 9.7 и на рис. 9.7 — выводе программы листинга 9.7.

Листинг 9.7. Вывод отдельных символов

```
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import javax.swing.*;
class GlyphTest extends JFrame{
  GlyphTest(String s){
    super(s);
    setSize(400, 150);
    setVisible(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
  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");
  }
}
```

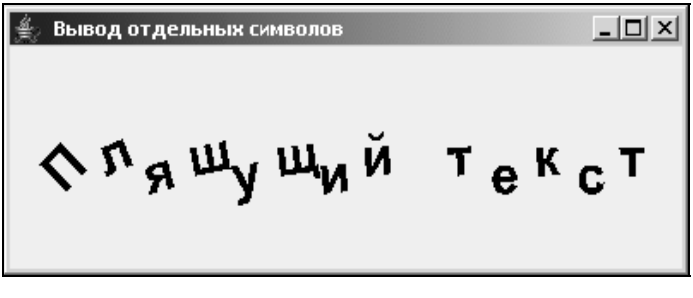


Рис. 9.7. Вывод отдельных символов

Методы улучшения визуализации

Визуализацию (rendering) созданной графики можно усовершенствовать, установив метод (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`, некоторые из них перечислены 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`. Для LCD-мониторов константы: `VALUE_TEXT_ANTIALIAS_GASP`, `VALUE_TEXT_ANTIALIAS_LCD_HRGB`, `VALUE_TEXT_ANTIALIAS_LCD_HBGR`, `VALUE_TEXT_ANTIALIAS_LCD_VRGB`, `VALUE_TEXT_ANTIALIAS_LCD_VBGR`
`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`
`KEY_DITHERING`	Замена близких цветов. Константы: `VALUE_DITHER_ENABLE`, `VALUE_DITHER_DISABLE`, `VALUE_DITHER_DEFAULT`
`KEY_ALPHA_INTERPOLATION`	Способ альфа-интерполяции. Константы: `VALUE_ALPHA_INTERPOLATION_DEFAULT`, `VALUE_ALPHA_INTERPOLATION_QUALITY`, `VALUE_ALPHA_INTERPOLATION_SPEED`

Таблица 9.2 *(окончание)*

Метод	Значение
KEY_STROKE_CONTROL	Способ рисования. Константы: VALUE_STROKE_DEFAULT, VALUE_STROKE_NORMALIZE, VALUE_STROKE_PURE

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
}
```

Для того чтобы как можно лучше связать визуализацию с настройками дисплея, определено свойство “awt.font.desktophints”, хранящее таблицу типа `Map` с методами улучшения визуализации, имеющимися в настройках дисплея. Воспользоваться этим свойством можно так:

```
public void paint(Graphics gr){
  Graphics2D g = (Graphics2D)gr;
  Toolkit tk = Toolkit.getDefaultToolkit();
   Map map = (Map)(tk.getDesktopProperty("awt.font.desktophints"));
   if (map != null) g.addRenderingHints(map);
  // Продолжение метода
}
```

Упражнение

5. Перепишите предыдущие упражнения средствами Java 2D.

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 20* будут рассмотрены средства Java 2D для работы с изображениями, в *главе 23* — средства печати.

В документации Sun Java SE, в каталоге docs/technotes/guides/2d/spec, есть руководство Programmer’s Guide to the Java 2D API с обзором всех возможностей Java 2D. Там помещены ссылки на руководства и пособия по Java 2D. В каталоге demo/jfc/Java2D/ приведена демонстрационная программа и исходные тексты программ, использующих Java 2D.

Вопросы для самопроверки

1. Что такое цвет в библиотеке AWT?
2. Что такое шрифт в библиотеке AWT?
3. Что такое графический контекст?
4. Как нарисовать треугольник?
5. Как нарисовать окружность?
6. Как преобразовать чертеж: повернуть его, уменьшить или увеличить?
7. Можно ли писать текст сверху вниз или справа налево?



Глава 10

Основные компоненты AWT

Казалось, что компоненты AWT постепенно выходят из употребления. Они не могут удовлетворить возрастающие потребности программистов и пользователей и заменяются компонентами Swing, SWAT, SWT и других графических библиотек. Тем не менее апплеты и некоторые приложения еще часто приходится создавать на основе компонентов AWT, а они используются в сотовых телефонах и других мобильных устройствах.

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`.

Забегая вперед, мы для каждого компонента перечислим события, которые в нем происходят. Обработку событий разберем в *главе 15*.

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. Размеры компонента 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`, а установить — методом `setPreferredSize(Dimension)`.

Компонент обладает минимальным и максимальным размерами. Их возвращают методы `getMinimumSize()` и `getMaximumSize()` в виде объекта `Dimension`. Установить эти размеры можно методами `setMinimumSize(Dimension)` и `setMaximumSize(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)`.

Положение курсора мыши над компонентом можно отследить методом `getMousePosition()`, возвращающим координаты курсора в виде объекта класса `Point`.

Остановимся подробнее на классе `Cursor`.

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 используются для задания аргумента `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 перечисленных предопределенных курсоров можно задать 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, рассказыва-

ется в *главе 20*. Аргумент `name` задает имя курсора, можно написать просто `null`. Аргумент `hotspot` определяет 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). Данный 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 некотором компоненте `someComp`.

Упражнение

1. Создайте курсор в форме правильного треугольника.

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.

Класс *Container*

Каждый компонент перед выводом на экран помещается в *контейнер* — подкласс класса `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 и обратно клавишами <Tab> и <Shift>+<Tab>. Компонент способен 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 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`.

Текстовая метка *Label*

Перейдем к рассмотрению конкретных компонентов. Самый простой компонент описывает класс `Label`.

Компонент `Label` — это 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()`, а также установить текст методами `setText(String)` и размещение `setAlignment(int)`.

События

В классе `Label` происходят события класса `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`.

Кнопка *Button*

Немногим сложнее класс `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 15*.

События

Кроме событий класса `Component` — `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent` — при воздействии на кнопку происходит событие `ActionEvent`.

Кнопка выбора *Checkbox*

Немного сложнее класса `Button` класс `Checkbox`, создающий кнопки выбора.

Компонент `Checkbox` — это кнопка с двумя состояниями. Графически она выглядит как 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` соответственно.

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`.

Класс *CheckboxGroup*

В библиотеке AWT радиокнопки не образуют отдельный компонент. Вместо этого несколько компонентов `Checkbox` объединяются в группу с помощью объекта класса `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 параметров.

Только одна радиокнопка в группе может иметь состояние `state == true`.

Пора привести пример. В листинге 10.1 представлена 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);
       }
    });
  }
}
```

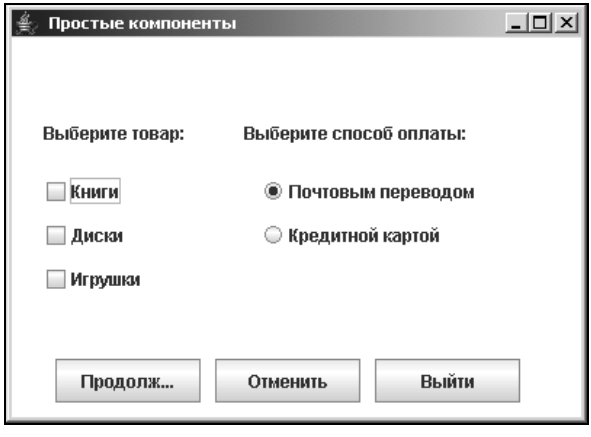


Рис. 10.1. Простые компоненты

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.

Раскрывающийся список *Choice*

Если нет необходимости отображать весь список на экране, то вместо группы радиокнопок можно создать раскрывающийся список — объект класса `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)`.

Удаление одного пункта из списка выполняется методом `remove(String text)` или `remove(int position)`, а всех пунктов 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`.

Список *List*

Если надо показать на экране несколько пунктов списка, то создайте объект класса `List`.

Компонент `List` — это список с линейкой 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()`, а 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 с помощью классов `Choice` и `List` задаются компоненты, аналогичные компонентам листинга 10.1, 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);
       }
    });
  }
}
```

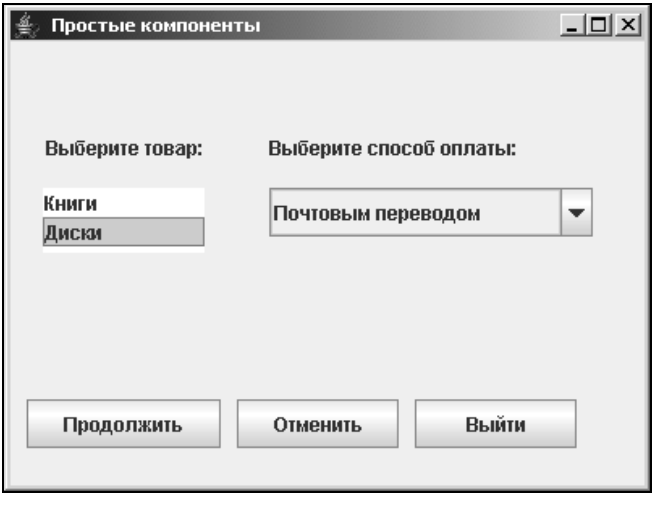


Рис. 10.2. Использование списков

Компоненты для ввода текста

В библиотеке AWT есть два компонента для ввода текста с клавиатуры: компонент `TextField`, позволяющий ввести только одну строку, и компонент `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 — `getText()` — возвращает в виде строки `String` текст, находящийся в поле ввода.

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`.

Строка ввода *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 методы `getColumns()` и `setColumns(int col)`.

Интересная разновидность строки ввода — строка для ввода пароля. В таком поле вместо вводимых символов появляется какой-нибудь особый эхо-символ, чаще всего звездочка, чтобы пароль никто не подсмотрел из-за плеча.

Строка ввода пароля получается из обычной строки ввода после выполнения 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 строку 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 клавиши <Enter> — событие `ActionEvent`.

Поле ввода *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:

- `append(String text)` — добавляет текст `text` в конец уже введенного текста;
- `insert(String text, int pos)` — вставляет текст в указанную позицию `pos`;
- `replaceRange(String text, int begin, int end)` — удаляет текст, начиная с индекса `begin` включительно по `end` исключительно, и помещает 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`.

Рассмотрим пример. В листинге 10.3 создаются три поля: `tf1`, `tf2`, `tf3` — для ввода имени пользователя, его пароля и заказа и не редактируемая область ввода, в которой накапливается заказ. В поле ввода пароля `tf2` появляется эхо-символ `*`. Результат показан на рис. 10.3.

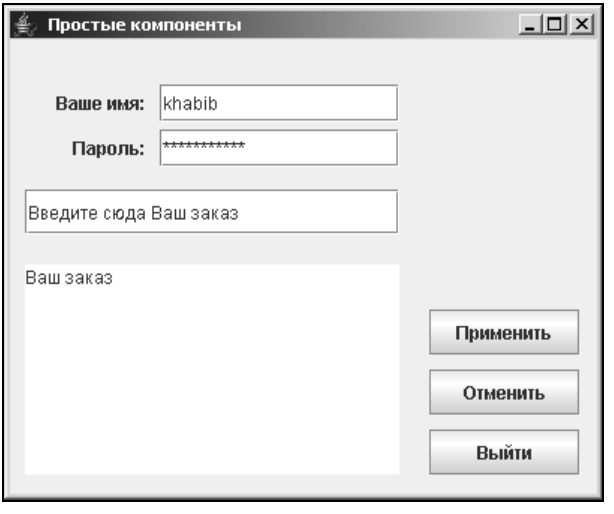


Рис. 10.3. Поля ввода

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);
       }
    });
  }
}
```

Линейка прокрутки *Scrollbar*

Компонент `Scrollbar` — это линейка 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 линейка 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 величиной в 10 единиц;

- `Scrollbar(int orientation)` — ориентация полосы прокрутки `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 15.6.

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. Линейки 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.



Рис. 10.4. Полосы прокрутки для выбора цвета

В листинге 15.6 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);
       }
    });
  }
}
```

Контейнер *Panel*

В листинге 10.4 использован контейнер `Panel`. Рассмотрим возможности этого класса.

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 передвинутся 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 линейки 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 линейки прокрутки не установлены, а 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 в виде объекта класса `Point` координаты `(x, y)` точки компонента, находящейся в левом верхнем углу панели `ScrollPane`;
- 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, линеек прокрутки. Это просто прямоугольная область на экране. Окно типа `Window` самостоятельно, оно 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 `setVisible(true)`. Убрать окно с экрана можно методом `setVisible(false)`, а проверить, видно ли окно на экране, — логическим методом `isShowing()`.

Методами

```
setIconImage(Image icon);
setIconImages(List<? extends Image> icons);
```

можно задать один или несколько ярлыков для окна, а посмотреть их можно методом

```
List<Image> getIconImages();
```

Окно типа `Window` возможно использовать для создания всплывающих окон предупреждения, сообщения, подсказки. Для создания диалоговых окон есть подкласс `Dialog`, всплывающих меню — класс `PopupMenu`.

Видимое на экране окно выводится на передний план методом `toFront()` или, наоборот, помещается на задний план методом `toBack()`. Методом `setAlwaysOnTop(true)` можно дать указание графическому менеджеру всегда держать окно на переднем плане. Не все графические менеджеры могут выполнить это указание, поэтому такую возможность следует предварительно проверить логическим методом `isAlwaysOnTopSupported()`. Выполнение этого указания можно проверить логическим методом `isAlwaysOnTop()`.

Уничтожить окно, освободив занимаемые им ресурсы, можно методом `dispose()`.

Менеджер размещения компонентов в окне по умолчанию — `BorderLayout`.

Окно создает свой экземпляр класса `Toolkit`, который можно 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 конце данной главы.

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 20*.

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 па-

нель 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;
- `Frame(GraphicsConfiguration gc)` — определяет конфигурацию окна параметром `gc`.
- `Frame(String title, GraphicsConfiguration gc)` — определяет строку заголовка `title` и конфигурацию окна параметром `gc`.

Методы класса `Frame` осуществляют доступ к элементам окна, но не забывайте о том, 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 показе 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();
       }
    });
  }
}
```

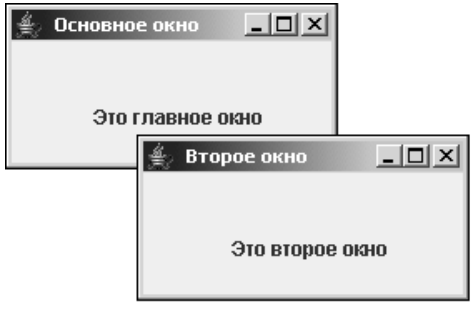


Рис. 10.5. Программа с двумя окнами

На рис. 10.5 представлен вывод этой программы. Взаимное положение окон определяется оконным менеджером операционной системы и может быть не таким, какое показано на рисунке.

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 родительское окно — 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. Модальное окно блокирует родительское окно и, возможно, еще несколько окон, образующих *область блокировки* (scope of blocking). Заблокированные окна не могут получить фокус ввода и находятся на экране позади модального окна.

Есть четыре области блокировки, определяемые *типом модальности* (modality type). Тип модальности задается следующими константами вложенного перечисления `Dialog.ModalityType`:

- `MODELESS` — отсутствие блокировки;
- `DOCUMENT_MODAL` — блокируются родительские окна, образующие один документ, причем под документом здесь понимаются все окна с общим предком, document root;
- `APPLICATION_MODAL` — блокируются родительские окна, относящиеся к одному приложению;
- `TOOLKIT_MODAL` — блокируются родительские окна, относящиеся к одному экземпляру класса `Toolkit`.

Более подробное и точное описание типов модальности приведено в документе The AWT Modality, хранящемся в файле docs/api/java/awt/doc-files/Modality.html.

В классе `Dialog` определена константа `DEFAULT_MODALITY_TYPE`, равная `APPLICATION_MODAL` в Java SE 6. Она неявно применяется в конструкторах класса и методе `setModal(true)`.

Графическая система, в которой открыто окно, может не отрабатывать все типы модальности, поэтому в класс `Toolkit` введен логический метод `isModalityTypeSupported(Dialog.ModalityType modalityType)`, которым можно проверить тот или иной тип `modalityType`.

Отдельные окна можно исключить из области блокировки. Для этого в класс `Window` введен метод `setModalExclusionType(Dialog.ModalExclusionType excType)`, аргументом которого служат константы из перечисления `Dialog.ModalExclusionType`:

- `NO_EXCLUDE` — отсутствие исключения;
- `APPLICATION_EXCLUDE` — модальные окна, имеющие тип модальности `APPLICATION_MODAL`, не могут блокировать это родительское окно;
- `TOOLKIT_EXCLUDE` — модальные окна, имеющие тип модальности `TOOLKIT_MODAL`, не могут блокировать это родительское окно.

Опять-таки, графическая система, в которой открыто окно, может не отрабатывать все эти исключения, поэтому в класс `Toolkit` введен еще один логический метод — `isModalExclusionTypeSupported(Dialog.ModalExclusionType excType)`.

В классе `Dialog` более десяти конструкторов. Из них:

- `Dialog(Dialog owner)` — создает немодальное диалоговое окно типа `MODELESS` с пустой строкой заголовка;
- `Dialog(Dialog owner, String title)` — создает немодальное диалоговое окно типа `MODELESS` со строкой заголовка `title`;

- `Dialog(Dialog owner, String title, boolean modal)` — создает диалоговое окно, которое будет модальным типа `DEFAULT_MODALITY_TYPE`, если параметр `modal == true`;
- `Dialog(Dialog owner, String title, boolean modal, GraphicConfiguration gc)` — создает диалоговое окно с конфигурацией, определяемой параметром `gc`.

Пять конструкторов аналогичны предыдущим, 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);
Dialog(Frame owner, String title, boolean modal, GraphicsConfiguration gc);
```

Еще у пяти конструкторов, аналогичных предыдущим, первый параметр — типа `Window`, а не `Frame`.

Среди методов класса `Dialog` интересны методы, связанные с модальностью: метод

```
Dialog.ModalityType getModalityType();
```

возвращает установленный для окна тип модальности, а метод

```
setModalityType(Dialog.ModalityType type);
```

меняет тип модальности. Если тип модальности `type` не отрабатывается графической системой, то устанавливается тип `MODELESS`. Во многих графических системах новый тип модальности вступит в силу только после того, как окно будет закрыто и вновь открыто.

В прежних версиях Java SE использовались методы `isModal()`, проверяющий состояние модальности, и `setModal(boolean modal)`, меняющий это состояние. Эти методы сейчас не используются, но оставлены в JDK для обратной совместимости со старыми программами.

События

Кроме событий класса `Componen` — `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent` — при изменении размеров окна, его перемещении или удалении с экрана, а также появлении на экране происходит событие `WindowEvent`.

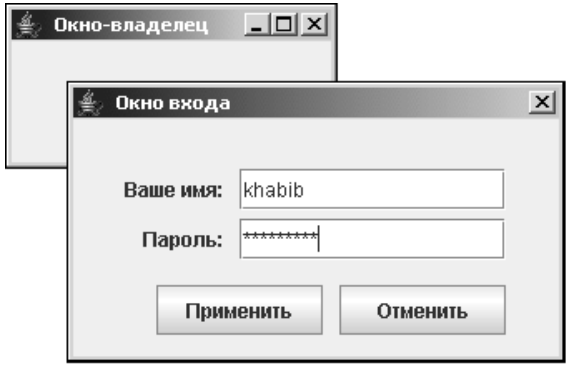


Рис. 10.6. Модальное окно доступа

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);
       }
    });
  }
}
```

Контейнер *FileDialog*

Контейнер `FileDialog` — это модальное окно с владельцем типа `Frame`, содержащее стандартное окно выбора файла для открытия (константа `LOAD`) или сохранения (константа `SAVE`). Окна операционной системы создаются и помещаются в объект класса `FileDialog` автоматически.

В классе шесть конструкторов:

- `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: окно открытия файла `FileDialog.LOAD` и окно сохранения файла `FileDialog.SAVE`.

Остальные три конструктора аналогичны первым трем, только первый параметр у них типа `Dialog`, а не `Frame`.

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 23*. 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 его 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 появлении на экране происходит событие `WindowEvent`.

Создание собственных компонентов

Создать собственный компонент, дополняющий свойства и методы уже существующих компонентов AWT, очень просто — надо лишь унаследовать свой класс от существующего класса `Button`, `TextField` или другого класса-компонента.

Если нужно 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` довольно прост — в нем чаще всего используется 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 — это 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 15*, 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. Рисунок 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);
       }
    });
  }
}
```

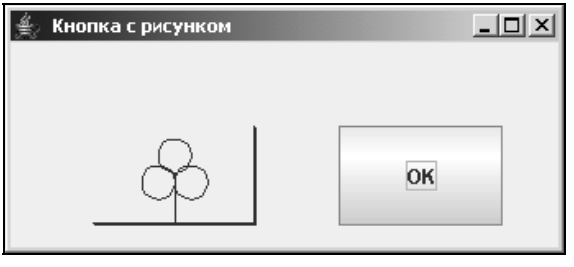


Рис. 10.7. Кнопка с рисунком

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();
  }
```

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 15*.

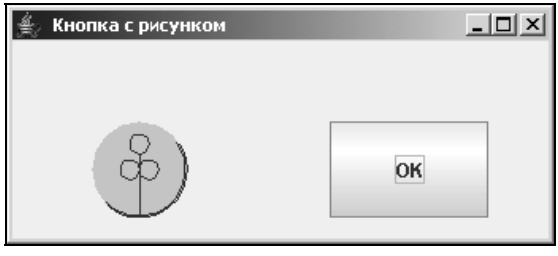


Рис. 10.8. Круглая кнопка

В документации есть хорошие примеры создания "легких" компонентов, посмотрите страницу docs/guide/awt/demos/lightweight/index.html.

Упражнение

2. Создайте треугольную кнопку.

Создание меню

В контейнер типа `Frame` заложена возможность установки стандартной *строки меню* (menu bar), располагаемой ниже строки заголовка, как показано на рис. 10.9. 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. 10.9. Во многих графических системах принято меню **Справка** (Help) прижимать к правому краю строки меню. Это достигается обращением к методу `setHelpMenu()`, но фактическое положение меню **Справка** определяется графической оболочкой.

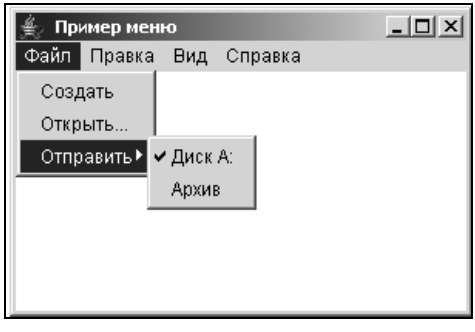


Рис. 10.9. Система меню

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. 10.9 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. 10.9.

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**. "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 комбинации клавиш <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`.

В листинге 10.9 приведена полная программа рисования с обработкой событий. Ее объяснение отложим до *главы 15*. Результат работы программы показан на рис. 10.10.

Листинг 10.9. 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");
  }
   }

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. 10.10. 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 15.2.

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 программу рисования из листинга 10.9, 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 10.10, а на рис. 10.11 — вид всплывающего меню в MS Windows.

Листинг 10.10. 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. 10.11. Программа рисования с всплывающим меню

Вопросы для самопроверки

1. Почему класс `Container` наследует от класса `Component`, а не наоборот?
2. Каковы общие свойства всех компонентов?
3. Почему класс `Component` сделан абстрактным?
4. Почему надпись в контейнере — это целый компонент класса `Label`, а не просто строка символов?
5. Почему для группы радиокнопок не создан отдельный компонент?
6. В чем разница между текстовыми компонентами `TextField` и `TextArea`?
7. Почему мы всегда используем окно типа `Frame`, а не типа `Window`?
8. Чем отличается создание "тяжелого" компонента от создания "легкого" компонента?



Глава 11

Оформление ГИП компонентами Swing

Стандартная поставка Java Platform, Standard Edition (Java SE), включает в себя богатейшую библиотеку классов, обеспечивающих создание графического интерфейса пользователя GUI (Graphical User Interface). Эта графическая библиотека получила название JFC (Java Foundation Classes). В библиотеке JFC можно выделить шесть основных частей:

- AWT (Abstract Window Toolkit) — базовая библиотека классов с несколько устарелым набором "тяжелых" (heavyweight) графических компонентов, расположенная в пакете `java.awt` и его подпакетах. Мы уже рассмотрели ее возможности в предыдущих главах;
- Swing — библиотека "легких" (lightweight) графических компонентов, дополняющая и во многом заменяющая библиотеку AWT. Занимает почти двадцать пакетов с префиксом `javax.swing`;
- Java 2D — часть библиотеки AWT, обеспечивающая рисование графики, выбор цвета, вывод изображений и фигурного текста, а также преобразование их перед выводом. Ее возможности уже показаны в *главе 9*;
- DnD (Drag and Drop) — библиотека классов, позволяющих перемещать объекты из одного компонента в другой с помощью *буфера обмена* (clipboard). Классы этой библиотеки помещены в пакеты `java.awt.datatransfer` и `java.awt.dnd`;
- Input Method Framework — классы для создания новых методов ввода/вывода. Они занимают пакеты `java.awt.im` и `java.awt.im.spi`;
- Accessibility — библиотека классов для взаимодействия с нестандартными устройствами ввода/вывода: клавиатурой Бройля, световым пером и др. Она расположена в пакете `javax.accessibility`.

Основное средство построения графического интерфейса пользователя в технологии Java — это библиотека Swing. Она может применяться везде, где установлен пакет Java Runtime Environment (JRE), в частности, в браузере Netscape Communicator. Для браузеров, в которые не встроен пакет JRE, корпорация Sun Microsystems выпускает модуль Java Plug-in, автоматически загружающийся с сайта **http://java.sun.com/products/plugin/** при загрузке апплета, использующего классы библиотеки Swing. Модуль Java Plug-in входит в состав JRE и автоматически подгружается в браузер, работающий там, где установлен пакет JRE. Нужно лишь, чтобы браузер распознавал тег `<OBJECT>` или `<EMBED>`, в котором указан класс апплета.

В состав Java SE в число демонстрационных программ входят апплет и приложение SwingSet2, расположенные в каталоге $JAVA_HOME/demo/jfc/SwingSet2/. Они показывают большинство возможностей Swing. Там же можно посмотреть исходные тексты соответствующих классов. Если в состав браузера входит библиотека Swing, то для просмотра апплета достаточно загрузить в браузер файл SwingSet2.html. Если Swing в браузере нет, то для загрузки Java Plug-in надо загрузить в браузер файл SwingSet2Plugin.html.

Состав библиотеки Swing

Библиотека Swing очень велика. Она содержит более пятисот классов и интерфейсов, предоставляющих богатейшие возможности оформления графического интерфейса. Полное описание ее возможностей занимает более тысячи страниц. Но большинство классов, входящих в Swing, предназначено для удовлетворения самых изысканных потребностей разработчика, окончательной "доводки" графического интерфейса, придания ему особого лоска.

Для построения же стандартного интерфейса достаточно возможностей, предоставляемых классами из пакета `javax.swing`. Очень легко создать стандартное окно приложения. В листинге 11.1 показан шаблон для приложения, использующего библиотеку Swing.

Листинг 11.1. Шаблон приложения, использующего Swing

```
import java.awt.*;          // Базовые классы AWT.
   import javax.swing.*;       // Основные классы Swing.

   public class SwingApplicationTemplate extends JFrame{

      public SwingApplicationTemplate(String title){

         // Создаем основное окно.
         super(title);

         // Получаем контейнер верхнего уровня.
         // Для JDK 5.0 и выше это необязательно.
         Container c = getContentPane();

         // Помещаем компонент в контейнер.
         c.add(xxxx);

            // Прочие установки...

         // Задаем начальную ширину и высоту окна.
         setSize(500, 400);

         // Завершаем работу приложения при закрытии окна.
         setDefaultCloseOperation(EXIT_ON_CLOSE);

         // Выводим окно на экран.
      setVisible(true);
   }
```

```
    public static void main(String[] args){
        new SwingApplicationTemplate("Заголовок основного окна");
    }
}
```

Если создаваемое приложение должно предоставить пользователю нестандартное диалоговое окно выбора цвета (а стандартное окно — это экземпляр класса `JColorChooser`), то понадобится пакет `javax.swing.colorchooser`.

При создании диалогового окна выбора файла (окно **Открыть** или **Сохранить как**) методами класса `JFileChooser`, для отбора файлов по типу или другому признаку применяются классы из пакета `javax.swing.filechooser`.

Для обработки содержимого таблицы — экземпляра класса `JTable` — пригодится пакет `javax.swing.table`.

Работу с объектами, расположенными в виде дерева типа `JTree`, можно организовать с помощью классов пакета `javax.swing.tree`.

При создании текстового редактора большую помощь окажут классы из пакета `javax.swing.text`. Они помогут создать нужную форму курсора, отследить и изменить его позицию, выделить фрагмент текста, задать формат записи дат и чисел и многое другое.

Возможность отмены и повтора действий (undo/redo) в текстовом редакторе обеспечивают классы пакета `javax.swing.undo`.

Подпакеты `javax.swing.text.html` и `javax.swing.text.rtf` дадут возможность текстовому редактору работать с форматами HTML и RTF, а классы подпакета `javax.swing.text.html.parser` содержат средства синтаксического разбора HTML-файлов.

Для оформления рамок различного вида, ограничивающих группы компонентов, предназначены классы из пакета `javax.swing.border`.

Наконец, пять пакетов `javax.swing.plaf.*` задают внешний вид и поведение приложения (Look and Feel, L&F) в различных графических средах. Можно сделать вид и поведение независимым от графической оболочки операционной системы. Тогда приложение в любой графической оболочке будет выглядеть одинаково и в равной степени реагировать на действия мыши и клавиатуры. Можно, наоборот, сделать так, что в каждой графической среде: MS Windows, CDE/Motif, Macintosh, приложение будет выглядеть как "родное" для этой среды и реагировать на внешние воздействия по правилам данной графической среды. Можно заложить изменение внешнего вида и поведения в настройки приложения, сделав его изменяемым (Pluggable Look and Feel, PL&F, PLAF или plaf) по желанию пользователя. Эту возможность можно реализовать классами пакета `javax.swing.plaf.multi`.

Технология Java предлагает свой собственный стиль, называемый "Java Look and Feel", ранее называвшийся стилем "Metal". Этот стиль у нас иногда называется "приборным" стилем, потому что приложение, оформленное в этом стиле, выглядит как алюминиевая панель научного прибора. "Родной" стиль Java L&F реализуется классами пакета `javax.swing.plaf.metal` и принимается по умолчанию в технологии Java. На сайте компании Sun Microsystems есть подробнейшее руководство по созданию графического интерфейса пользователя в стиле Java L&F — "Java Look and Feel Design Guidelines". Его можно посмотреть по адресу **http://java.sun.com/products/jlf/**.

Разумеется, в этой книге мы не сможем полностью рассмотреть все возможности библиотеки Swing, но ее структуру и основные средства освоим в той мере, которая позволит создать удобное и красивое приложение, приятное для работы. Изложение библиотеки Swing в этой части книги рассчитано на то, что читатель знаком с постоянно обновляемым электронным учебником "The Java Tutorial. A practical guide for programmers", расположенным по адресу **http://java.sun.com/docs/books/tutorial/**.

Начнем с обзора готовых графических компонентов Swing.

Основные компоненты Swing

В библиотеку Swing входит около тридцати готовых графических компонентов: надписи, кнопки, поля ввода, линейки прокрутки, ползунки, меню и пункты меню, деревья, таблицы. Они собраны главным образом в пакет `javax.swing`. Рассмотрим их последовательно от самых простых до самых сложных компонентов. Но начнем с вершины иерархии компонентов — класса `JComponent`.

Компонент *JComponent*

Основные свойства всех компонентов Swing сосредоточены в их суперклассе `JComponent`. Класс `JComponent` расширяет класс `Container`, входящий в графическую библиотеку AWT. Поэтому компонент `JComponent` и все его расширения являются контейнерами и могут содержать в себе другие компоненты. Класс `Container`, в свою очередь, расширяет класс `Component`, содержащий около сотни методов работы с компонентами. Эти методы и методы класса `Container` наследуются классом `JComponent`, который добавляет к ним добрую сотню своих методов. Все компоненты Swing расширяют класс `JComponent`, наследуя его богатейшие свойства.

Класс `JComponent` — это абстрактный класс, поэтому нельзя создать его экземпляры. Кроме того, он реализован как "легкий" компонент и, несмотря на то что является контейнером, не может служить контейнером верхнего уровня. По этим причинам он не используется самостоятельно, а только как суперкласс для создания новых компонентов, перенося на них всю свою мощь.

В классе `JComponent` сосредоточена основная функциональность компонентов Swing. Перечислим некоторые возможности компонентов.

- Для компонента `JComponent` и его наследников можно задать рамку методом `setBorder(Border)`.
- Компонент можно сделать прозрачным или непрозрачным с помощью метода `setOpaque(boolean)`.
- Фон непрозрачного компонента можно закрасить определенным цветом методом `setBackground(Color)`.
- Для любого компонента можно установить шрифт методом `setFont(Font)` и его цвет методом `setForeground(Color)`.
- Для каждого компонента создается графический контекст класса `Graphics`, которым можно воспользоваться для рисования фигур и линий на компоненте, обратившись к методу `paint(Graphics)`.

- Можно задать определенную форму курсора мыши методом `setCursor(Cursor)`. Такую форму курсор мыши будет принимать, когда он проходит над компонентом.
- Каждый компонент разрешается снабдить всплывающей подсказкой, которая появится, если задержать на секунду курсор мыши над компонентом. Для этого достаточно задать текст всплывающей подсказки методом `setToolTipText(String)`.
- Для каждого компонента можно определить минимальный, максимальный и предпочтительный размер методами `setMinimumSize(Dimension)`, `setMaximumSize(Dimension)` и `setPreferredSize(Dimension)` соответственно, а также собственную локаль — методом `setLocale(Locale)`.
- Все компоненты отслеживают события клавиатуры `KeyEvent` и мыши `MouseEvent`, `MouseWheelEvent`, передачу фокуса `FocusEvent`, события изменения компонента `ComponentEvent` и контейнера `ContainerEvent`.

У компонентов Swing сложное строение — они построены по схеме MVC.

Схема MVC в компонентах Swing

Конструктивная схема *Модель-Вид-Контроллер* (MVC, Model-View-Controller) рассмотрена нами в *главе 3*. Повторим вкратце ее основные понятия.

Первую часть, Model, составляет один или несколько классов, в которых хранится или вырабатывается вся информация, обрабатываемая компонентом, и текущее состояние объектов, созданных этим компонентом. Эти классы обладают методами `set`*`XXX`*`()` ввода и изменения информации.

Вторая часть — один или несколько классов, составляющих View. Эта часть компонента описывает способ представления результатов, сгенерированных *Моделью*, на экране дисплея, принтере или другом устройстве в определенном виде: таблица, график, диаграмма. К одной *Модели* можно подключить несколько *Видов*, по-разному представляющих одни и те же результаты или отражающие разную информацию. *Виды* получают информацию методами `get`*`Xxx`*`()` и `is`*`Xxx`*`()` *Модели*.

Третья часть — классы, образующие Controller, — создают интерфейс для ввода информации и изменения состояния объекта. Они реагируют на события ввода с клавиатуры, действия мыши и прочие воздействия на объект и обращаются к методам `set`*`Xxx`*`()` *Модели*, изменяя ее поля или вызывая генерацию информации. Одна *Модель* может использоваться несколькими *Контроллерами*.

Вид и *Контроллер* не взаимодействуют. *Контроллер*, реагируя на события, обращается к методам `set`*`Xxx`*`()` *Модели*, которые меняют хранящуюся в ней информацию. *Модель*, изменив информацию, сообщает об этом тем *Видам*, которые зарегистрировались у нее. Этот способ взаимодействия *Модели* и *Вид*а получил название "подписка-рассылка" (subscribe-publish). *Виды* подписываются у *Модели*, и та рассылает им сообщения о всяком изменении состояния объекта методами `fire`*`Xxx`*`()`, после чего *Виды* забирают измененную информацию, обращаясь к методам `get`*`Xxx`*`()` и `is`*`Xxx`*`()` *Модели*.

В библиотеке Swing модели описаны интерфейсами, в которых перечислены необходимые методы. У каждого интерфейса есть хотя бы одна стандартная реализация, принимаемая компонентами Swing по умолчанию. Некоторые классы реализуют сразу несколько интер-

фейсов, некоторые интерфейсы реализованы несколькими классами. Эти интерфейсы и классы, реализующие их, приведены в табл. 11.1.

Таблица 11.1. *Интерфейсы моделей и классы, реализующие их*

Интерфейс	Класс
`BoundedRangeModel`	`DefaultBoundedRangeModel`
`ButtonModel`	`DefaultButtonModel`
	`JToggleButton.ToggleButtonModel`
`ComboBoxModel`	`DefaultComboBoxModel`
`MutableComboBoxModel`	
`ListModel`	`AbstractListModel`
	`DefaultListModel`
`ListSelectionModel`	`DefaultListSelectionModel`
`SingleSelectionModel`	`DefaultSingleSelectionModel`
`ColorSelectionModel`	`DefaultColorSelectionModel`
`SpinnerModel`	`AbstractSpinnerModel`
	`SpinnerDateModel`
	`SpinnerListModel`
	`SpinnerNumberModel`
`TableColumnModel`	`DefaultTableColumnModel`
`TableModel`	`DefaultTableModel`
`TreeModel`	`DefaultTreeModel`
`TreeSelectionModel`	`DefaultTreeSelectionModel`
	`JTree.EmptySelectionModel`

В графическом интерфейсе пользователя очень часто *Вид* и *Контроллер* работают с одними и теми же графическими компонентами. *Контроллер* связан с нажатием кнопок, протаскиванием мыши по линейкам прокрутки и движкам, вводом текста в поля ввода, а *Вид* меняет на экране эти графические компоненты, получив от *Модели* сообщение о происшедших изменениях.

Для реализации модели MVC библиотека Swing использует *делегирование* (delegation) полномочий, назначая в качестве модели данных *представителя* (delegate) — экземпляр класса с именем вида `xxxModel`. Класс, описывающий компонент, содержит защищенное или даже закрытое поле `model` — объект этого класса-модели, и метод `getModel()`, предоставляющий разработчику доступ к полю `model`. Сложные компоненты могут иметь несколько моделей, например в классе `JTable` есть три поля-представителя:

```
protected TableColumnModel columnModel;
protected TableModel dataModel;
protected ListSelectionModel selectionModel;
```

и, соответственно, три метода доступа

```
TableColumnModel getColumnModel();
TableModel getModel();
ListSelectionModel getSelectionModel();
```

Делегирование полномочий используется и для обеспечения PL&F. Класс `JComponent` содержит защищенное поле `ui` — экземпляр класса-представителя `ComponentUI` из пакета `javax.swing.plaf`, непосредственно отвечающего за вывод изображения на экран в нужном виде. Класс-представитель содержит методы `paint()` и `update()`, формирующие и обновляющие графические примитивы. Такие представители образуют целую иерархию с общим суперклассом `ComponentUI`. Они собраны в пакет `javax.swing.plaf` и его подпакеты. В их именах есть буквы UI (User Interface), например: `ButtonUI`, `BasicButtonUI`.

Представители класса тоже являются полями класса компонента, а доступ к ним осуществляется методами вида `getUI()`.

Класс, описывающий компонент, дублирует большинство методов модели, например в том же классе `JTable` есть множество методов доступа к информации `getXxx()`, большинство из них просто обращаются к соответствующим методам модели, например метод получения числа строк таблицы:

```
public int getRowCount(){
   return getModel().getRowCount();
}
```

Поэтому при построении графического интерфейса пользователя редко приходится обращаться к моделям и представителям компонента. В большинстве случаев достаточно обращаться к методам самого класса компонента. Если модель, принятая по умолчанию, в чем-то не устраивает разработчика, можно заменить ее другой моделью, реализовав подходящий интерфейс или расширив существующий класс `xxxModel`. Новая модель данных устанавливается методом `setModel(xxxModel)`. Если приложение не обращалось непосредственно к методам модели, то в нем ничего изменять не придется.

Надпись *JLabel*

Различные неизменяемые надписи и небольшие изображения в окне приложения представляются компонентом `JLabel`. Для создания экземпляра этого класса есть шесть конструкторов.

- Конструктор по умолчанию `JLabel()` выделяет прямоугольную область в контейнере без надписи и изображения, в которую потом можно поместить текст методом `setText(String)` и изображение методом `setIcon(Icon)`.
- Конструкторы `JLabel(String)` и `JLabel(Icon)` выделяют прямоугольную область и заносят в нее строку текста — экземпляр класса `String`, или изображение — экземпляр класса, реализующего интерфейс `Icon`, обычно это класс `ImageIcon`. Изображение размещается в центре области, а строка в центре по вертикали и слева.
- В конструкторах с двумя параметрами `JLabel(String, int)` и `JLabel(Icon, int)` второй параметр задает горизонтальное размещение текста или изображения константами `LEFT`, `CENTER`, `RIGHT`, `LEADING` или `TRAILING` интерфейса `SwingConstants`. Этот интерфейс реализован в классе `JLabel`. Понятия `LEADING` и `TRAILING` зависят от уста-

новленной локали. Для языков с написанием слева направо это левая и правая сторона области, для других, например арабского языка, наоборот.

Размещение можно потом изменить методом `setHorizontalAlignment(int)`. Можно изменить и размещение по вертикали методом `setVerticalAlignment(int)` с константами `TOP`, `CENTER` или `BOTTOM`.

- Последний конструктор, `JLabel(String, Icon, int)`, задает и строку, и изображение, и размещение, при этом строка располагается справа от изображения для языков с написанием слева направо. Изменить расположение текста относительно изображения по горизонтали и вертикали можно методами `setHorizontalTextPosition(int)` и `setVerticalTextPosition(int)` с такими же константами. По умолчанию текст от изображения отделяют 4 пиксела. Изменить это расстояние можно методом `setIconTextGap(int)`.

Например, создание надписи и размещение ее сверху и справа в выделенной для компонента области контейнера выглядит так:

```
JLabel l = new JLabel("Какая-то надпись", JLabel.RIGHT);
l.setVerticalAlignment(JLabel.TOP);
```

Если же мы хотим разместить в компоненте `JLabel` текст и изображение, причем текст расположить слева от изображения, оставив между ними 10 пикселов, то надо сделать примерно так:

```
JLabel l = new JLabel("Надпись",
     new ImageIcon("myimage.gif"), JLabel.CENTER);
l.setHorizontalTextPosition(JLabel.LEFT);
l.setIconTextGap(10);
```

Интересно, что Swing "понимает" разметку языка HTML и в строке можно с помощью тегов менять цвет, шрифт, создавать списки, размещать текст в нескольких строках:

```
l.setText("<html>Первая <font color=\"red\">строка<p>вторая");
```

> ***Внимание!***
>
> Тег `<html>` должен начинать строку, идти сразу же после открывающей кавычки, без пробелов.

При этом следует учитывать, что размер компонента может быть вычислен неправильно и проинтерпретированный текст не поместится в компоненте. Поэтому, например, вместо тега `<br>` лучше употреблять тег `<p>`.

Еще одно интересное свойство. Компонент `JLabel` можно связать с другим компонентом методом `setLabelFor(Component)`. Затем с какой-либо буквой надписи, например `A`, нужно связать командную клавишу методом

```
setDisplayedMnemonic('A');
```

или методом

```
setDisplayedMnemonic(KeyEvent.VK_A);
```

Второй из этих методов требует включения в программу пакета `java.awt.event`. Буква `A` в надписи будет подчеркнута. После этого нажатие комбинации клавиш <Alt>+<A> вызовет

передачу фокуса связанному с надписью компоненту (на компонент `JLabel` фокус никогда не передается). В тексте HTML буква, связанная с командной клавишей, не подчеркивается автоматически, для нее надо задать подчеркивание тегом `<u>`.

Если в надписи несколько одинаковых букв, то будет подчеркнута первая из них. Методом `setDisplayedMnemonicIndex(int)` можно подчеркнуть букву с указанным в качестве параметра индексом. У первой буквы надписи нулевой индекс.

При переводе компонента `JLabel` в недоступное состояние методом `setEnabled(false)` текст и изображение становятся бледными. Можно при этом заменить изображение, если оно уже было в компоненте, другим изображением с помощью метода `setDisabledIcon(Icon)`.

Остальные методы класса `JLabel` выполняют проверки и предоставляют сведения о компоненте, но не забывайте, что можно воспользоваться еще и методами классов `JComponent`, `Container` и `Component`. Достаточно просто установить цвет надписи методом `setForeground(Color)`, шрифт — методом `setFont(Font)`, цвет фона — методом `setBackground(Color)`. При этом учтите, что по умолчанию компонент `JLabel` прозрачен, и перед закрашиванием фона надо сделать его непрозрачным, обратившись к методу `setOpaque(true)`. Можно обрамить компонент методом `setBorder(Border)`. Можно задать всплывающую подсказку методом `setToolTipText(String)`. Можно даже задать реакцию на внешние воздействия, но для этого лучше применять кнопки.

Кнопки

Библиотека Swing предлагает целую иерархию кнопок, показанную на рис. 11.1. В нее включены и пункты меню `JMenuItem`, и кнопки выбора `JCheckBox`, и радиокнопки `JRadioButton`.

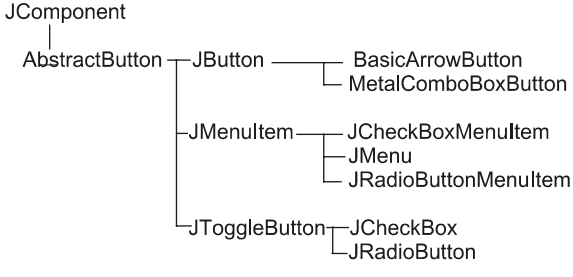


Рис. 11.1. Иерархия классов кнопок

Во главе иерархии стоит абстрактный класс `AbstractButton`, содержащий методы, общие для всех типов кнопок. В нем также собраны константы, определяющие общее поведение всех кнопок.

Все кнопки типа `AbstractButton` реагируют на событие `ActionEvent`, происходящее при щелчке кнопкой мыши, событие класса `ChangeEvent` из пакета `javax.swing.event`, возникающее при всех действиях мыши: наведении курсора мыши на компонент, удаление его с компонента, нажатии кнопки мыши и т. д., и на событие `ItemEvent`, возникающее при смене состояния кнопки.

На любую кнопку всегда можно поместить новый текст методом `setText(String)` и сменить существующее изображение методом `setIcon(Icon)`.

Как и в компоненте класса `JLabel`, можно методом `setDisabledIcon(Icon)` сменить изображение на кнопке, сделанной недоступной, т. е. на кнопке, к которой применен метод `setEnabled(false)`. При попытке выделения недоступной кнопки можно установить на ней новое изображение методом `setDisabledSelectedIcon(Icon)`.

Кроме того, можно сменить изображение при наведении курсора мыши на кнопку методом `setRolloverIcon(Icon)`, но только если предварительно эта возможность включена методом `setRolloverEnabled(true)`. По умолчанию она отключена.

Аналогично можно сменить изображение при выделении кнопки методом `setSelectedIcon(Icon)`, при наведении курсора мыши на выделенную кнопку методом `setRolloverSelectedIcon(Icon)`, при нажатии кнопки мыши `setPressedIcon(Icon)`. Эти возможности всегда включены.

Итак, с одной кнопкой допустимо связать семь изображений и заменять их при наведении курсора мыши на кнопку, нажатии кнопки мыши, выделении кнопки, наведении курсора мыши на выделенную кнопку, при переводе кнопки в недоступное состояние и при выделении недоступной кнопки. Следует заметить, что не все графические системы реализуют перечисленные возможности.

Командную клавишу можно назначить кнопке методом `setMnemonic(int)` с указанием в качестве параметра этого метода константы из класса `java.awt.event.KeyEvent`. Будет подчеркнута первая буква надписи, связанная с командной клавишей. Как и в классе `JLabel`, можно подчеркнуть не только первое появление этой буквы, но и какое-нибудь из следующих появлений с помощью метода `setDisplayedMnemonicIndex(int)`.

Всплывающая подсказка для кнопки задается методом `setToolTipText(String)`.

Кнопки типа `AbstractButton` используют по умолчанию модель класса `DefaultButtonModel`, реализующего интерфейс `ButtonModel`. Эта модель отслеживает пять состояний кнопки.

- Кнопка находится в состоянии "наведенная" (rollover), когда над ней располагается курсор мыши. Контроллер отмечает это состояние методом `setRollover(boolean)` модели, а вид курсора определяется методом `isRollover()`.
- В состояние "наготове" (armed) кнопка переходит при нажатии на ней кнопки мыши. Это состояние устанавливается в модели методом `setArmed(boolean)`, а отслеживается логическим методом `isArmed()`.
- В состояние "нажатая" (pressed) кнопка переходит из состояния "наготове" после отпускания кнопки мыши. За этим состоянием следят методы `setPressed(boolean)` и `isPressed()`.
- После щелчка кнопка "выделяется" (selected), что отмечается методами `setSelected(boolean)` и `isSelected()`.
- Наконец, кнопку можно сделать "доступной" (enabled) или "недоступной" (disabled) методом `setEnabled(boolean)` и отследить ее состояние методом `isEnabled()`.

Класс `AbstractButton` дублирует только методы `setEnabled(boolean)` и `isEnabled()`, остальные состояния надо отслеживать методами модели `DefaultButtonModel`, получив предварительно ее экземпляр методом `getModel()`.

На практике, разумеется, применяются не объекты класса `AbstractButton`, а расширения этого класса, которые мы рассмотрим подробнее. Самое небольшое расширение, реализующее все свойства кнопки `AbstractButton`, — это компонент `JButton`.

Кнопка *JButton*

Обычная прямоугольная кнопка — экземпляр класса `JButton` — может, так же как и `Jlabel`, содержать текст и/или изображение. Их размещение и взаимное положение не задается конструктором, а устанавливается методами

```
setHorizontalAlignment(int);
setVerticalAlignment(int);
setHorisontalTextPosition(int);
setVerticalTextPosition(int);
setIconTextGap(int);
```

точно так же, как и в компонентах `JLabel` и с теми же константами в качестве параметра этих методов. По умолчанию и текст, и изображение располагаются по центру кнопки, а изображение слева от текста.

Для создания объектов класса `JButton` есть пять конструкторов: конструктор по умолчанию `JButton()`, конструкторы кнопок с текстом `JButton(String)` и изображением `Jbutton(Icon)`, конструктор с двумя параметрами `JButton(String, Icon)`. Пятый конструктор, `JButton(Action)`, использует объект класса, реализующего интерфейс `Action`.

Как прямое расширение класса `AbstractButton`, класс `JButton` наследует все его свойства и методы и к нему относится все сказанное в предыдущем разделе. С учетом этого определение кнопки может выглядеть так:

```
ImageIcon def = new ImageIcon("default.gif");
JButton b = new JButton("<html><u>Д</u>алее", def);
b.setBackground(new Color(183, 220, 65));
b.setFont(new Font("Lucida", Font.ITALIC, 12));
b.setPreferredSize(new Dimension(100, 30));
b.setMnemonic(KeyEvent.VK_L);
b.setToolTipText("Переход к следующей странице");
b.setRolloverEnabled(true);
b.setRolloverIcon(new ImageIcon("rollover.gif"));
b.setSelectedIcon(new ImageIcon("select.gif"));
b.setRolloverSelectedIcon(new ImageIcon("rollselect.gif"));
b.setPressedIcon(new ImageIcon("press.gif"));
b.setDisabledIcon(new ImageIcon("disable.gif"));
b.setDisabledSelectedIcon(new ImageIcon("disselect.gif"));
b.setActionCommand("next");
b.addActionListener(this);
b.addChangeListener(this);
b.addItemListener(this);
```

Кнопка JButton реагирует на событие ActionEvent, возникающее при щелчке кнопкой мыши на компоненте, событие ChangeEvent, происходящее при всех действиях мышью на компоненте, и событие ItemEvent. Кроме того, кнопка наследует события ComponentEvent и ContainerEvent, а также события мыши и клавиатуры.

Кнопка выбора *JToggleButton*

Компонент JToggleButton представляет прямоугольную кнопку стандартного вида, имеющую два состояния, отмечаемые как булево значение true/false, и меняющую одно состояние на другое при щелчке кнопкой мыши на компоненте или нажатии "горячей" клавиши. Изменение L&F при смене состояния обычно заключается в том, что кнопка на экране становится "нажатой" и остается в этом состоянии до следующего щелчка кнопкой мыши. Отследить текущее состояние кнопки можно логическим методом isSelected(), установить то или другое состояние программно — методом setSelected(boolean).

Экземпляры класса JToggleButton создаются восемью конструкторами. Основной конструктор,

```
JToggleButton(String, Icon, boolean);
```

создает кнопку с надписью, изображением и выбранным значением true или false. В других конструкторах отсутствуют какие-то из этих параметров, причем отсутствующий третий параметр считается равным false. В строке можно сделать разметку HTML. Конструктор JToggleButton(Action) использует в качестве параметра экземпляр класса, реализующего интерфейс Action.

В листинге 11.2 показан простейший пример кнопки с двумя состояниями.

Листинг 11.2. Простейшая кнопка с двумя состояниями

```
import java.awt.*;
import java.awt.event.*;
   import javax.swing.*;

   class DummyToggleButton extends JFrame{

     private JToggleButton tb;

  public DummyToggleButton(){
    tb = new JToggleButton("<html><u>Д</u>а?<p>Нет?");
    tb.setMnemonic(KeyEvent.VK_L);
    tb.setToolTipText("Сделайте выбор");
       add(tb);

    // Для JDK версии ранее 5.0 уберите комментарий
       // getContentPane().add(tb);

    setSize(300,300);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
     }
```

```
  public static void main(String[] args){
    new DummyToggleButton();
  }
}
```

Класс `JToggleButton` применяется обычно как суперкласс для создания кнопок выбора нестандартного вида. Для получения стандартных кнопок выбора используются его подклассы `JCheckBox` и `JRadioButton`.

Кнопка выбора *JCheckBox*

Компонент `JCheckBox` — это стандартная кнопка выбора с меткой или изображением слева или справа от надписи, в которой показывается состояние кнопки: `true` или `false`. Форма метки зависит от установки L&F.

Создать экземпляр класса `JCheckBox` можно одним из восьми конструкторов. Основной конструктор — `JCheckBox(String, Icon, boolean)`, в других конструкторах те же параметры, что и у класса `JToggleButton`.

Класс `JCheckBox` не добавляет функциональности своему суперклассу и используется точно так же, как класс `JToggleButton`.

Радиокнопка *JRadioButton*

Стандартная радиокнопка создается одним из восьми конструкторов, основной из них — `JRadioButton(String, Icon, boolean)`, остальные имеют те же параметры, что и конструкторы класса `JToggleButton`.

Класс `JRadioButton` не добавляет функциональности своему суперклассу и используется точно так же, как класс `JToggleButton`.

Радиокнопки имеет смысл использовать только в составе группы, обеспечивая выбор одного из нескольких значений. Для получения группы сначала создается пустая группа как экземпляр класса `ButtonGroup`, затем она заполняется радиокнопками методом `add(AbstractButton)`.

Группа радиокнопок никак не выделяется на экране, это только логическое объединение элементов. Чтобы выделить кнопки, входящие в группу, их обычно размещают на отдельной панели, окружая эту панель рамкой. В листинге 11.3 показано обычное размещение группы радиокнопок. Кнопки в листинге устанавливают цвет фона окна приложения. На рис. 11.2 показан вид этой группы радиокнопок.

Листинг 11.3. Группа радиокнопок

```
import java.awt.*;
import javax.swing.*;
   import javax.swing.border.*;

   class RadioButtonTest extends JFrame{

  public RadioButtonTest(){
    Container c = getContentPane(); // Излишне для JDK 5.0 и выше.
```

```
    c.setBackground(Color.white);
       c.setLayout(new FlowLayout());

    JPanel p = new JPanel();
    p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
       p.setBorder(BorderFactory.createEtchedBorder());

    JRadioButton rb1 =
        new JRadioButton("<html><u>К</u>расный<p>фон");
    rb1.setMnemonic(KeyEvent.VK_R);
    rb1.setToolTipText("<html>Вы выбираете<p>красный фон");
    rb1.addActionListener(this);
       rb1.setActionCommand("red");

    JRadioButton rb2 =
        new JRadioButton("<html><u>З</u>еленый<p>фон");
    rb2.setMnemonic(KeyEvent.VK_P);
    rb2.setToolTipText("<html>Вы выбираете<p>зеленый фон");
    rb2.addActionListener(this);
       rb2.setActionCommand("green");

    JRadioButton rb3 =
        new JRadioButton("<html><u>С</u>иний<p>фон");
    rb3.setMnemonic(KeyEvent.VK_C);
    rb3.setToolTipText("<html>Вы выбираете<p>синий фон");
    rb3.addActionListener(this);
       rb3.setActionCommand("blue");

    ButtonGroup bg = new ButtonGroup();
       bg.add(rb1); bg.add(rb2); bg.add(rb3);

    p.add(rb1); p.add(rb2); p.add(rb3);
       c.add(p);

    setSize(300, 150);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
     }

  public static void main(String[] args){
    new RadioButtonTest();
  }
}
```

Группу радиокнопок составляют несколько объектов класса `JRadioButton`, что при большом числе вариантов приводит к неоправданному расходу ресурсов. Кроме того, такая группа занимает много места в окне приложения. Для выбора из большого числа вариантов библиотека Swing предлагает создать объекты одного из двух классов: `JList` и `JComboBox`.

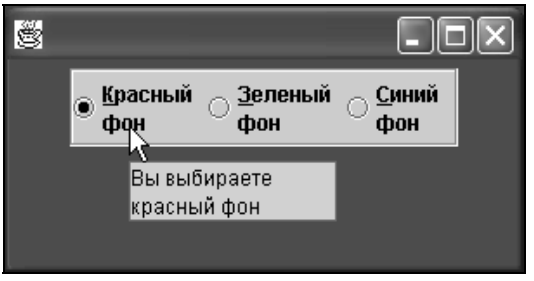


Рис. 11.2. Группа радиокнопок и всплывающая подсказка

Упражнение

1. Перепишите листинг 10.1 с использованием компонентов Swing.

Раскрывающийся список *JComboBox*

Выбор одного варианта из большого числа возможностей удобно организовать с помощью класса `JComboBox`. Выбранный элемент виден в окне компонента, остальные элементы списка раскрываются при щелчке кнопкой мыши по стрелке, находящейся справа в поле компонента. Раскрывшееся окно — это экземпляр класса `JPopupMenu`.

Создать экземпляр раскрывающегося списка можно конструктором по умолчанию `JComboBox()`, а затем заносить в него элементы методами `addItem(Object)` и `insertItemAt(Object, int)`.

Однако чаще бывает удобнее занести элементы в список сразу же при его создании конструктором `JComboBox(Object[])` или `JComboBox(Vector)`, предварительно создав массив или вектор, содержащий элементы. Например:

```
String[] data = {"Иванов", "Петров", "Сидоров"};
JComboBox cb = new JComboBox(data);
```

Если в списке будет использована модель, отличная от модели, принятой по умолчанию, то элементы сначала заносятся в нее, а затем конструктором `JComboBox(ComboBoxModel)` создается объект, связанный с этой моделью.

В список можно занести и изображения, например:

```
Object[] data = {new ImageIcon("apple.gif"),
                 new ImageIcon("grape.gif"),
                 new ImageIcon("pear.gif")};
JComboBox fruits = new JComboBox(data);
```

Есть возможность заносить в список объекты и других типов. Мы поговорим о реализации этой возможности в *разд. "Визуализация элементов списков" данной главы*.

После создания списка в его окне виден первый элемент. Чтобы поместить в окно какой-то другой элемент, следует обратиться к методам `setSelectedItem(Object)` или `setSelectedIndex(int)`. Если в этих методах указать в качестве параметра `null` или `-1`

соответственно, то окно будет пустым. Это удобно в случае редактируемого списка, в окно которого можно вводить новый элемент или редактировать выбранный. Список становится редактируемым после выполнения метода `setEditable(true)`. Для редактирования привлекается текстовый редактор — экземпляр класса, реализующего интерфейс `ComboBoxEditor`. По умолчанию используется одна из реализаций этого интерфейса — класс `BasicComboBoxEditor`, открывающий для редактирования поле ввода класса `JTextField`.

Редактирование выбранного элемента списка в окне не приводит к изменению этого элемента в списке, а влияет только на объект, возвращаемый методом `getSelectedItem()`.

Текст HTML интерпретируется в элементах списка, но в окне редактируемого списка при использовании в качестве редактора объекта класса `BasicComboBoxEditor` появляется в "плоском" ASCII-виде и в таком же виде возвращается методом `getSelectedItem()`.

По умолчанию список раскрывается полностью. Чтобы ограничить раскрывающееся окно несколькими строками, нужно обратиться к методу

```
setMaximumRowCount(int);
```

Если в списке больше элементов, чем выделено строк этим методом, то в раскрывающемся окне появится полоса прокрутки.

При выборе элемента или окончании редактирования (нажатии клавиши <Enter>) в раскрывающемся списке происходит событие класса `ActionEvent` и одно или два события класса `ItemEvent`, а при раскрытии и свертывании списка — событие класса `PopupMenuEvent`. Обработчик события может получить выбранный элемент методом `getSelectedItem()`, а его индекс — методом `getSelectedIndex()`.

Моделью данных для класса `JComboBox` служит класс `DefaultComboBoxModel`, реализующий сразу три интерфейса: `ListModel`, `ComboBoxModel` и `MutableComboBoxModel`, и расширяющий класс `AbstractListModel`. Нет никакой необходимости в непосредственном обращении к методам этой модели, поскольку они дублируются методами класса `JComboBox`, за одним исключением. В модели данных класса `DefaultComboBoxModel` при изменении списка происходит событие класса `ListDataEvent`, не отслеживаемое классом `JComboBox`. Но эта модель не реагирует на события `ActionEvent` и `ItemEvent`.

Больше того, сам класс `JComboBox` зачем-то реализует интерфейсы `ActionListener`, `EventListener`, `ListDataListener`, но использовать его как слушателя событий нельзя.

Есть еще несколько нестыковок. В классе `JComboBox` элементы называются `Item`, например такое имя использовано в названии метода `getItemAt(int)`. В классе `DefaultComboBoxModel` аналогичный метод называется `getElementAt(int)`. Это приходится учитывать при создании собственной модели данных.

Наконец, попытка задать в одном списке `JComboBox` элементы с текстом и изображением приведет к их смешению. Причину этого рассмотрим в *разд. "Визуализация элементов списков" данной главы*.

Список выбора *JList*

Вместо группы кнопок выбора можно создать список класса `JList`. В таком списке допустимо выбирать не только один элемент, но и группу подряд идущих элементов, и несколько таких групп. Кроме конструктора по умолчанию `JList()`, создающего пустой список, можно задать список с заданным массивом объектов конструктором `JList(Object[])`, с за-

данным вектором при помощи конструктора `JList(Vector)` или с определенной заранее моделью `JList(ListModel)`. Это делается так же, как и при создании экземпляра класса `JComboBox`.

Список типа `JList` выглядит на экране просто как столбец из всех своих элементов. Чтобы ограничить число видимых на экране строк и снабдить список полосой прокрутки для показа остальных строк, следует поместить список на панель типа `JScrollPane`. После этого можно задать число видимых строк методом `setVisibleRowCount(int)`. С учетом всего этого определение списка выбора может выглядеть так:

```
   JFrame f = new JFrame();
   String[] data = {"Иванов", "<html><font color=red>Петров", "Сидоров"};
JList list = new JList(data);
list.setVisibleRowCount(2);
   list.addListSelectionListener(this);
   JScrollPane sp = new JScrollPane(list);
f.getContentPane().add(sp);
```

Так же, как и в раскрывающийся список `JComboBox`, в список `JList` можно занести не только текст, но и изображения.

По умолчанию в списке можно выбрать любое число любых элементов, держа клавишу <Ctrl> нажатой. После применения метода

```
setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

в списке можно будет выбрать только один элемент. Третья возможность — отметить один диапазон подряд идущих элементов — достигается использованием в качестве параметра этого метода константы `SINGLE_INTERVAL_SELECTION`. Эти три возможности выбора элементов списка — результат реализации интерфейса `ListSelectionModel` классом `DefaultListSelectionModel`. Данная реализация модели выбора применяется в классе `JList` по умолчанию. Если такая реализация почему-либо не устраивает разработчика, то он может реализовать интерфейс `ListSelectionModel` своим классом и установить созданную модель выбора методом `setSelectionModel(ListSelectionModel)`.

Один (первый) выбранный элемент можно получить методом `getSelectedValue()`, массив типа `Object[]` всех выбранных элементов — методом `getSelectedValues()`. Индекс первого выбранного элемента выдает метод `getSelectedIndex()`, массив индексов всех выбранных элементов — метод `getSelectedIndices()`.

Кроме модели выбора — реализации интерфейса `ListSelectionModel` — класс `JList` связан еще с моделью данных. Она описана интерфейсом `ListModel` и частично реализована абстрактным классом `AbstractListModel`. Класс `JList` использует расширение этого класса — класс `DefaultListModel`. Класс `DefaultListModel` хранит данные в закрытом поле `delegate` типа `Vector` на основе идеи делегирования и дублирует фактически все методы класса `Vector`, обращаясь к классу-представителю, например:

```
public Object getElementAt(int index){
    return delegate.elementAt(index);
}
```

Класс JList отслеживает событие ListSelectionEvent, происходящее при смене выделенного элемента списка. Его модель данных отслеживает, кроме того, событие ListDataEvent, возникающее при всяком изменении списка.

Визуализация элементов списков

Компоненты классов JList и JComboBox могут содержать десятки и сотни элементов, имеющих тип String или Icon. Создание графического объекта для каждого элемента списка приведет к колоссальному расходу оперативной памяти и к большим затратам времени на создание объектов. Чтобы избежать этого расхода ресурсов, для изображения элементов списков назначается объект-рисовальщик. Он последовательно выводит элементы на экран или на принтер, переходя от одного элемента к другому. Короче говоря, реализуется design pattern, известный под именем Flyweight.

Кроме экономии ресурсов такой подход дает возможность вывода каждого элемента списка по-своему, меняя вид элемента, например шрифт или цвет. Класс-рисовальщик описан в интерфейсе ListCellRenderer, имеющем только один метод

```
public Component getListCellRendererComponent(
  JList list,           // Список, элементы которого выводятся на экран
  Object value,         // Элемент списка, который будет выведен
  int index,            // Порядковый индекс этого элемента
  boolean isSelected,   // Выбран ли этот элемент?
  boolean cellHasFocus  // Имеет ли фокус этот элемент?
);
```

Этот метод должен сформировать компонент и поместить в него текущий элемент списка value, имеющий порядковый номер index. Вид компонента может зависеть не только от его класса или порядкового номера, но и от того, выбран ли он isSelected (обычно выбранный элемент выделяется синим цветом фона) и имеет ли фокус cellHasFocus (обычно обводится тонкой рамкой). Полученный компонент затем выводится на экран своим методом paint().

В библиотеке Swing интерфейс ListCellRenderer реализован классами BasicComboBoxRenderer и DefaultListCellRenderer, расширяющими класс JLabel. Именно потому, что выводом элементов фактически занимается класс JLabel, можно использовать в элементах списка текст или изображение. Интересный эффект получится, если смешать в одном списке типа JComboBox и текст, и изображения. Класс BasicComboBoxRenderer попытается вывести их вместе. Список типа JList, в котором текст перемежается изображениями, будет выведен правильно. Все дело в разной реализации интерфейса ListCellRenderer. Вот фрагмент реализации:

```
public class DefaultListCellRenderer extends JLabel
       implements ListCellRenderer, Serializable{

  public Component getListCellRendererComponent(
      JList list, Object value, int index,
         boolean isSelected, boolean cellHasFocus){
```

```
        setComponentOrientation(list.getComponentOrientation());

    if (isSelected){
      setBackground(list.getSelectionBackground());
      setForeground(list.getSelectionForeground());
    }else{
      setBackground(list.getBackground());
      setForeground(list.getForeground());
       }

    if (value instanceof Icon){
      setIcon((Icon)value);
      setText("");
    }else{
      setIcon(null);
      setText((value == null) ? "" : value.toString());
       }

    setEnabled(list.isEnabled());
    setFont(list.getFont());
    setBorder((cellHasFocus) ?
      UIManager.getBorder("List.focusCellHighlightBorder") :
         noFocusBorder);

    return this;
  }
. . . . .
}
```

Как видно из этого фрагмента, в каждый формирующийся компонент — один элемент списка `JList` — может быть помещено либо изображение типа `Icon`, либо текст типа `String`. Всякий другой объект будет преобразован в строку его методом `toString()`, в том числе и объект класса `JLabel`. Легко изменить эту реализацию, убрав условие `if(value instanceof Icon)` из приведенного фрагмента и применив унаследованные от класса `JLabel` методы

```
setText(((JLabel)value).getText());
setIcon(((JLabel)value).getIcon());
```

После этого элементами списка могут служить объекты класса `JLabel`. Но это еще не все. Метод `getSelectedValue()` по-прежнему будет возвращать строку, выдаваемую методом `JLabel.toString()`, а не ссылку на объект. Значит, надо еще расширить класс `JLabel`, переписав метод `toString()`.

Итак, если разработчику нужно создать список, содержащий объекты других типов, отличных от `String` и `Icon`, то он должен написать класс, экземпляры которого будут служить элементами списка. В данном классе следует переопределить, кроме методов `get`*Xxx*`()`/`set`*Xxx*`()`, метод `toString()`, а при необходимости и метод `paint()`. Экземпляры этого класса записываются в конструктор `JList(Object[])` и передаются методу `getListCellRendererComponent()` как параметр `value`.

Потом следует написать свою реализацию интерфейса `ListCellRenderer`. Обычно она расширяет класс `JLabel` или `JPanel`.

Упражнение

2. Перепишите листинг 10.2 с использованием компонентов Swing.

Счетчик *JSpinner*

Перебирать элементы списка часто бывает удобнее с помощью счетчика — небольшого редактируемого поля с текущим значением списка и двумя стрелками вверх и вниз, с помощью которых можно заменять текущее значение предыдущим или следующим. Самый простой такой счетчик создается конструктором по умолчанию `JSpinner()`. Его текущее значение — 0, следующее — 1, предыдущее — –1, и так можно перебирать все целые числа.

Для создания более сложного счетчика конструктором `JSpinner(SpinnerModel)` придется сначала определять модель данных. Она описана интерфейсом `SpinnerModel`, частично реализована абстрактным классом `AbstractSpinnerModel` и полностью реализована в трех классах. Класс `SpinnerDateModel` реализует модель, содержащую даты, класс `SpinnerListModel` — модель, содержащую коллекцию типа `List`, в частности, массив произвольных объектов типа `Object[]`. Класс `SpinnerNumberModel` содержит целые или вещественные числа или объекты класса `Number`. Например, следующая строка:

```
JSpinner sp = new JSpinner(new SpinnerNumberModel(50, 0, 100, 5));
```

создает счетчик с текущим значением 50, диапазоном значений от 0 до 100 и шагом изменения значений 5. В поле можно ввести любое значение из указанного диапазона, например 47, тогда предыдущее значение будет равно 42, а следующее — 52.

С помощью класса `SpinnerNumberModel` можно создать еще модель с вещественными числами конструктором

```
SpinnerNumberModel(double current, double min, double max, double step);
```

и числовую модель общего вида конструктором

```
SpinnerNumberModel(Number current, Comparable min,
                   Comparable max, Number step);
```

Значения `min` и `max` могут быть `null`, в таком случае нижняя или верхняя границы не существуют.

Текущее, предыдущее и следующее значение можно получить от счетчика `JSpinner` методами `getValue()`, `getPreviousValue()` и `getNextValue()` соответственно. Эти методы возвращают объект класса `Object`. Если значения выходят за заданные в конструкторе границы, то указанные методы возвращают `null`.

При всяком изменении текущего значения, а также окончании ввода в поле нового значения путем нажатия клавиши <Enter>, в счетчике происходит событие класса `ChangeEvent`. Поэтому получать значения счетчика надо примерно так:

```
sp.addChangeListener(this);
// . . . .
public void stateChanged(ChangeEvent e){
```

```
  comp.setValue((int)sp.getValue());
}
```

Вторая модель данных класса, `SpinnerDateModel`, позволяет выбирать даты из заданного списка. Конструктор по умолчанию `SpinnerDateModel()` создает счетчик с текущей датой и временем, предыдущее его значение — это то же время вчерашнего дня, следующее значение — то же время завтрашнего дня, ограничений на выбор нет.

Конструктор

```
SpinnerDateModel(Date value, Comparable first,
                 Comparable last, Date step);
```

задает произвольную текущую дату `value`, диапазон значений дат от `first` до `last` и шаг `step`. Если одно или оба значения диапазона равны `null`, то соответствующая граница отсутствует. Шаг `step` определяет также и форму представления даты и может принимать значения одной из констант: `ERA`, `YEAR`, `MONTH`, `WEEK_OF_YEAR`, `WEEK_OF_MONTH`, `DAY_OF_MONTH`, `DAY_OF_YEAR`, `DAY_OF_WEEK`, `DAY_OF_WEEK_IN_MONTH`, `AM_PM`, `HOUR`, `HOUR_OF_DAY`, `MINUTE`, `SECOND`, `MILLISECOND` класса `Calendar`.

Более широкие возможности предоставляет третья модель данных класса — `SpinnerListModel`. В конструкторе `SpinnerListModel(Object[])` этого класса можно задать массив произвольных объектов, например:

```
String[] data = {"Дворник", "Уборщица", "Программист", "Сторож"};
SpinnerListModel model = new SpinnerListModel(data);
JSpinner emp = new JSpinner(model);
```

В другом конструкторе, `SpinnerListModel(List)`, задается экземпляр коллекции, реализующей интерфейс `List`, например экземпляр класса `Vector`.

Хотя в счетчик можно заложить любые объекты, в поле будет показана только строка, полученная методом `toString()` текущего объекта. Это происходит потому, что редактор по умолчанию, заложенный в класс `JSpinner`, — это экземпляр класса `JFormattedTextField`. Определяет его вложенный в `JSpinner` класс `DefaultEditor` и его подклассы `DateEditor`, `ListEditor` и `NumberEditor`.

Полосы прокрутки *JScrollBar*

Полосы прокрутки используются многими компонентами и контейнерами Swing. В массе случаев достаточно поместить компонент на панель типа `JScrollPane`, как это сделано в листинге 11.3, чтобы обеспечить прокрутку содержимого компонента.

Полосы прокрутки определяются четырьмя числами-свойствами, хранящимися в модели данных. Это наименьшее значение полосы `minimum`, наибольшее значение `maximum`, текущее значение `value` и шаг изменения `extent`. От последнего числа зависит размер видимой области компонента, связанного с полосой прокрутки, и длина ползунка на полосе прокрутки. Действия с этими числами описаны интерфейсом `BoundedRangeModel`. При всяком изменении данных чисел происходит событие `ChangeEvent`. Класс `JScrollBar` использует в качестве модели данных реализацию `DefaultBoundedRangeModel` этого интерфейса.

Полосы прокрутки создаются конструктором

```
JScrollBar(int orientation, int value, int extent, int min, int max);
```

или конструктором `JScrollBar(int orientation)`, устанавливающим значения `min = 0`, `max = 100`, `value = 0`, `extent = 10`. Вертикальная или горизонтальная ориентация задается константами `VERTICAL` или `HORIZONTAL`.

Все значения можно потом изменить методами `setMinimum(int)`, `setMaximum(int)`, `setValue(int)` и `setExtent(int)`.

Текущее значение возвращается методом `getValue()`. Метод `getUnitIncrement(int)` возвращает величину перемещения на одну единицу вверх, если параметр этого метода равен –1, или вниз, если параметр равен 1. Эта величина устанавливается равной 1 при создании полосы и может быть изменена методом `setUnitIncrement(int)`. Аналогично метод `getBlockIncrement(int)` возвращает величину перемещения вверх или вниз на один блок. Размер блока вначале равен величине `extent`, затем ее можно изменить методом `setBlockIncrement(int)`.

Полоса прокрутки реагирует на событие `AdjustmentEvent`, происходящее при каждом изменении ее модели данных.

Ползунок *JSlider*

Ползунок представляет собой линейку с указателем, которым можно установить какое-то значение `value` из диапазона `min–max`. Внутренне ползунок устроен так же, как и полоса прокрутки.

Компонент `JSlider` тоже использует модель данных класса `DefaultBoundedRangeModel` с наименьшим, наибольшим и текущим значениями, а также шагом изменения. Впрочем, можно применить другую модель, задав ее в конструкторе `JSlider(BoundedRangeModel)` или установив методом `setModel(BoundedRangeModel)`.

Основной конструктор:

```
JSlider(int orientation, int min, int max, int value);
```

В других конструкторах отсутствует тот или иной параметр, при этом устанавливается горизонтальный ползунок со значениями `min = 0`, `max = 100`, `value = (min + max)/2`.

Рядом с линейкой ползунка можно разметить шкалу со штрихами, отстоящими друг от друга на расстояние, устанавливаемое методом `setMajorTickSpacing(int)`. Вначале это расстояние равно нулю. После определения расстояния шкала задается методом `setPaintTicks(true)`. К штрихам можно добавить числовые значения методом `setPaintLabels(true)`. Между штрихами допускается размещение более мелких штрихов методом `setMinorTickSpacing(int)`. Если применить метод `setSnapToTicks(true)`, то движок ползунка будет останавливаться только против штрихов.

Основную линейку ползунка можно убрать методом `setPaintTrack(false)`, оставив только шкалу.

Числовые значения в шкале ставятся против каждого штриха. Методом `createStandardLabels(int incr, int start)` можно изменить это правило, задав другой шаг расстановки чисел `incr` на шкале и другой начальный отсчет `start`. Затем этот шаг надо установить на шкале методом `setLabelTable(Dictionary)`. Все это удобно делать вместе, например после определений:

```
   JSlider sl = new JSlider();
sl.setMajorTickSpacing(10);
```

```
sl.setMinorTickSpacing(5);
sl.setPaintTicks(true);
sl.setPaintLabels(true);
sl.setLabelTable(sl.createStandardLabels(20, 28));
```

получим отмеченные значения 28, 48, 68, 88, как показано на рис. 11.3.

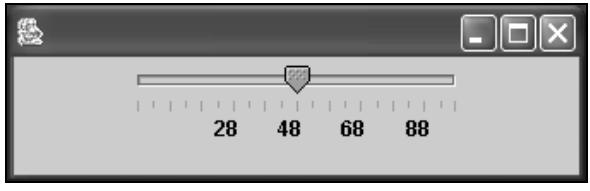


Рис. 11.3. Ползунок `JSlider`

Метод `setLabelTable(Dictionary)` позволяет сделать и более сложные изменения, установив в качестве меток не только числа, но и какие-то другие значения словаря типа `Dictionary`.

Внешний вид ползунка определяется абстрактным классом `SliderUI`. У него два расширения — классы `BasicSliderUI` и `MultiSliderUI`. На рис. 11.3 ползунок нарисован расширением класса `BasicSliderUI` — классом `MetalSliderUI`. При желании можно создать свой класс-рисовальщик, расширив один из этих классов и установив новый класс методом `setUI(SliderUI)`.

При перемещении движка в ползунке происходит событие `ChangeEvent`. В процессе обработки этого события можно получить значение ползунка методом `getValue()`.

Упражнение

3. Перепишите листинг 10.4, заменив полосы прокрутки ползунками Swing.

Индикатор *JProgressBar*

Индикатор, часто называемый "градусником", показывает степень выполнения какого-то процесса, чаще всего в процентах. Процесс должен вырабатывать какое-нибудь целое число. В конструкторе индикатора

```
JProgressBar(int orientation, int min, int max);
```

задаются наименьшее `min` и наибольшее `max` значения этого числа. В других конструкторах опущены некоторые из указанных параметров. При этом ориентация считается горизонтальной, `min = 0`, `max = 100`.

По мере выполнения процесса он должен передавать степень своего выполнения в индикатор методом `setValue(int)`. Это значение немедленно отражается в индикаторе. После обращения к методу `setStringPainted(true)` в окне индикатора появится еще число — процент выполнения процесса.

Если время выполнения процесса, связанного с индикатором, не определено, то можно перевести индикатор в *неопределенный режим* (indeterminate mode). Это делается методом `setIndeterminate(true)`. В этом режиме индикатор мигает, показывая, что процесс выполняется. Когда окончание процесса определится, надо занести наибольшее значение процесса в индикатор методом `setMaximum(int)`, текущее значение методом `setValue(int)` и перевести индикатор в обычный режим методом `setIndeterminate(false)`.

Внешний вид индикатора описывается абстрактным классом `ProgressBarUI`. У него два расширения — классы `BasicProgressBarUI` и `MultiProgressBarUI`. Стандартный вид Java L&F обеспечивается классом `MetalProgressBarUI`. При необходимости изменения внешнего вида индикатора следует расширить один из этих классов и установить новый вид методом `setUI(ProgressBarUI)`.

Индикатор может работать в отдельном окне, эту возможность предоставляет класс `ProgressMonitor`.

Дерево объектов *JTree*

Дерево `JTree` располагает объекты в иерархическую структуру. Она создается только на экране, но не в оперативной памяти. На уровне 0 находится один *корневой* (root) объект, на уровнях 1, 2 и т. д. размещаются его *потомки* (child) — *узловые* (node) объекты, имеющие своих потомков и одного *предка* (parent). На самом нижнем уровне расположены *листья* (leaf). Это узлы, не имеющие потомков.

Для экономии ресурсов дерево не определяется рекурсивно, его узлы не являются ссылками типа `JTree`. Вместо этого узел описан интерфейсом `TreeNode` и его расширением — интерфейсом `MutableTreeNode`. Это расширение добавляет методы замены объекта, находящегося в узле, а также методы добавления и удаления потомков из узла. Оно реализовано классом `DefaultMutableTreeNode` из пакета `javax.swing.tree`.

Узел дерева `JTree` создается конструктором `DefaultMutableTreeNode(Object)`, в котором задается ссылка на содержащийся в узле объект.

Каждый узел класса `DefaultMutableTreeNode` содержит ссылку на своего предка, которую можно получить методом `getParent()`. Если этот метод возвращает `null`, значит, узел корневой, но для такой проверки есть специальный логический метод `isRoot()`.

Узел хранит ссылки на потомков в структуре типа `Vector`, получить их можно многочисленными методами `getXxx()`. Метод `getLevel()` показывает уровень узла относительно корня, а метод `getDepth()` — количество уровней поддерева, начинающегося с данного узла. Метод `insert(MutableTreeNode, int)` добавляет к узлу нового потомка в позицию, указанную вторым параметром. Метод `setUserObject(Object)` меняет ссылку на объект, расположенную в узле.

Узлы дерева можно сделать редактируемыми методом `setEditable(true)`. Редактор должен реализовать методы интерфейса `TreeCellEditor`. По умолчанию используется реализация `DefaultTreeCellEditor`, открывающая для редактирования окно класса `JTreeField`.

Дерево класса `JTree` создается одним из семи конструкторов. Проще всего воспользоваться конструктором `JTree(TreeNode)`, создающим корень дерева, а затем создавать и добавлять к дереву новые узлы.

Например:

```
   DefaultMutableTreeNode root = new DefaultMutableTreeNode("Корень");
   JTree tr = new JTree(root);
DefaultMutableTreeNode subtree1 = new DefaultMutableTreeNode("Узел 1");
   root.add(subtree1);
   subtree1.add(new DefaultMutableTreeNode("Лист 2a"));
DefaultMutableTreeNode subtree2 = new DefaultMutableTreeNode("Узел 2");
   subtree1.add(subtree2);
subtree2.add(new DefaultMutableTreeNode("Лист 3a"));
subtree2.add(new DefaultMutableTreeNode("Лист 3b"));
   subtree2.add(new DefaultMutableTreeNode("Лист 3c"));
   subtree1.add(new DefaultMutableTreeNode("Лист 2b"));
root.add(new DefaultMutableTreeNode("Лист 1"));
// и т. д....
```

Для простоты каждый узел в этом дереве содержит строку класса `String`. Полученное дерево показано на рис. 11.4.

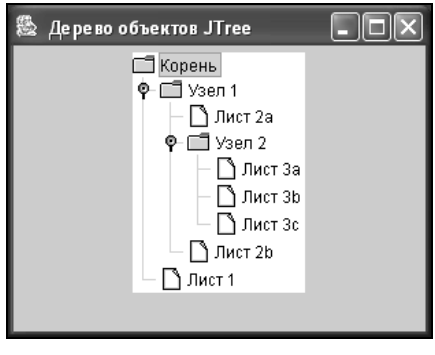


Рис. 11.4. Дерево класса `JTree`

Против каждого узла выводится содержимое его объекта, преобразованное методом `toString()`.

Второй способ — сформировать вектор узлов дерева и воспользоваться конструктором `JTree(Vector)`. Поддеревья создаются вложенными векторами. Вот как будет создано предыдущее дерево:

```
   Vector root = new Vector();
Vector subtree1 = new Vector();
   root.add(subtree1);
```

```
  subtree1.add("Лист 2a");
     Vector subtree2 = new Vector();

  subtree1.add(subtree2);
    subtree2.add("Лист 3a");
    subtree2.add("Лист 3b");
    subtree2.add("Лист 3c");
     subtree1.add("Лист 2b");

   root.add("Лист 1");

JTree tr = new JTree(root);
```

Недостаток этого метода в том, что у дерева, построенного с помощью вектора, нет корневого узла. Кроме того, в обоих вариантах узлы оказываются неподписанными, на экран выводится содержимое узлов в виде строки.

Второй недостаток можно устранить использованием конструктора `JTree(Hashtable)`, аргумент которого — хеш-таблица. Вот как будет создано предыдущее дерево этим способом:

```
   Hashtable root = new Hashtable();

   Hashtable subtree1 = new Hashtable();

   root.put("Узел 1", subtree1);

  Hashtable subtree2 = new Hashtable();
  subtree1.put("Лист 2a", new Integer(21));
  subtree1.put("Узел 2", subtree2);
    subtree2.put("Лист 3a", new Integer(31));
    subtree2.put("Лист 3b", new Integer(32));
    subtree2.put("Лист 3c", new Integer(33));
     subtree1.put("Лист 2b", new Integer(22));

   root.put("Лист 1", new Integer(1));

JTree tr = new JTree(root);
```

Из каждой пары "ключ — значение", хранящейся в хеш-таблице, ключ выводится на экран в виде строки. У этого дерева тоже нет корня. Кроме того, его узлы выводятся не в том порядке, в котором помещаются в хеш-таблицу, т. к. она "перемешивает" свои данные.

Дерево с большим числом узлов удобно поместить на панель типа `JScrollPane` и указать число строк, помещающихся в окне, методом `setVisibleRowCount(int)`, например:

```
JTree tr = new JTree(root);
JScrollPane sp = new JScrollPane(tr);
tr.setVisibleRowCount(8);
```

В дереве можно выделить один или несколько узлов. Все узлы дерева пронумерованы сверху вниз, начиная от корневого узла, имеющего номер 0. Метод `getSelectionRows()` возвращает массив типа `int[]` номеров выделенных узлов. Говоря точнее, нумерация узлов

дерева описана интерфейсом `RowMapper`. В пакете `javax.swing.tree` есть абстрактный класс `AbstractLayoutCache`, реализующий этот интерфейс, и два его подкласса: `FixedHeightLayoutCache` и `VariableHeightLayoutCache`.

Иногда удобнее получить путь к узлу в виде последовательности объектов, хранящихся в узлах, ведущих от корня к данному узлу. Такая последовательность хранится в классе `TreePath`. Метод `getSelectionPath()` возвращает экземпляр этого класса, метод `getSelectionPaths()` возвращает массив таких экземпляров для всех выделенных узлов. Есть и другие аналогичные методы получения экземпляров класса `TreePath` для различных узлов дерева.

Получив экземпляр класса `TreePath`, можно выбрать из него массив типа `Object[]` объектов, содержащихся в узлах, методом `getPath()`. Первый элемент этого массива всегда содержит объект, хранящийся в корне. Другие методы класса `TreePath` позволяют получить отдельные элементы этого массива.

Вся информация о выделенных элементах дерева находится в модели выбора, описанной интерфейсом `TreeSelectionModel` и реализованной классом `DefaultTreeSelectionModel`. Данный класс хранит массив типа `TreePath[]` путей к выделенным узлам и методы работы с этим массивом. Он предлагает три готовые модели выбора: выбор только одного узла дерева `SINGLE_TREE_SELECTION`, выбор диапазона узлов `CONTIGUOUS_TREE_SELECTION` и выбор нескольких диапазонов `DISCONTIGUOUS_TREE_SELECTION`. Последний выбор принимается по умолчанию. Изменить модель выбора в дереве `tr` можно так:

```
tr.getSelectionModel().setSelectionMode(
   TreeSelectionModel.SINGLE_TREE_SELECTION);
```

Модель выбора реагирует на событие класса `TreeSelectionEvent`, происходящее при всяком новом выборе узлов. В отличие от большинства классов событий, класс `TreeSelectionEvent` содержит несколько полезных при обработке события методов, например метод `getPaths()` выдает массив типа `TreePath[]`.

Вся информация об узлах дерева хранится в модели данных, описанной в интерфейсе `TreeModel`. Этот интерфейс не описывает класс узлов дерева. Узел может быть экземпляром любого класса. Но реализация `DefaultTreeModel`, принятая в классе `JTree` по умолчанию, уже требует, чтобы узел имел интерфейс `TreeNode`.

Модель данных реагирует на событие класса `TreeModelEvent`, происходящее при изменении узлов дерева. Этот класс события позволяет получить полезные данные о дереве, в частности, метод `getTreePath()` возвращает в виде объекта класса `TreePath` путь к предку тех узлов, которые были изменены, добавлены или удалены.

Подобно классу `JList` дерево типа `JTree` делегирует визуализацию классу-представителю, реализующему интерфейс `TreeCellRenderer`. Интерфейс описывает один метод:

```
Component getTreeCellRendererComponent(JTree tree, Object value,
   boolean selected, boolean expanded, boolean leaf,
   int row, boolean hasFocus);
```

Как видно из этого описания, изображение узла `value` дерева `tree` может меняться в зависимости от его порядкового номера в дереве `row` и от того, выбран ли этот узел `selected`,

раскрыта ли соответствующая ветвь дерева `expanded`, является ли узел листом `leaf` и имеет ли узел фокус `hasFocus`.

Данный интерфейс реализован классом `DefaultTreeCellRenderer`, добавляющим массу свойств и методов и расширяющим класс `JLabel`. Последнее свойство позволяет вывести на экран различные значки для узлов и листьев, открытых и закрытых ветвей дерева.

В состав J2SE JDK входит пример построения дерева шрифтов, расположенный в каталоге $JAVA_HOME/demo/jfc/SampleTree/. В нем показано, как можно изменить стандартную визуализацию дерева.

Построение меню средствами Swing

Все компоненты, составляющие меню в библиотеке Swing: строка меню, пункты меню, всплывающее меню, имеют один и тот же тип, описанный интерфейсом `MenuElement`. Это позволяет объединять элементы какого-либо подменю или все меню, находящиеся в линейке меню, в массив с помощью метода `getSubElements()` и рассматривать их как одно целое.

Для создания системы меню сначала следует создать строку меню и установить ее в контейнер типа `JFrame`, `JApplet`, `JDialog` методом `setJMenuBar(JMenuBar)`. Этот метод располагает строку меню горизонтально ниже строки заголовка окна.

Строка меню *JMenuBar*

Строка меню создается единственным конструктором класса `JMenuBar()`. Полученная строка не содержит ни одного меню, их надо добавлять методом `add(JMenu)` по мере создания. Добавляемые меню будут располагаться слева направо в порядке обращения к методам `add(JMenu)`. В некоторых графических системах меню **Справка** (Help) располагается справа. Чтобы учесть эту особенность, в класс `JMenuBar` включен специальный метод `setHelpMenu(JMenu)`. Впрочем, этот метод реализован далеко не во всех выпусках JDK.

Начнем создавать примерное меню:

```
   JFrame f = new JFrame("Пример системы меню");

JMenuBar mb = new JMenuBar());
   f.setJMenuBar(mb);

JMenu file = new JMenu("<html><u>Ф</u>айл"));
JMenu edit = new JMenu("<html><u>П</u>равка"));
JMenu view = new JMenu("<html><u>В</u>ид"));
   JMenu help = new JMenu("<html><u>С</u>правка"));

mb.add(file); mb.add(edit);
mb.add(view); mb.add(help);
```

Меню *JMenu*

Каждое меню по существу представляет собой два компонента: "кнопку" с текстом и всплывающее меню типа `JPopupMenu`, появляющееся при щелчке кнопкой мыши по этой "кноп-

ке". Как видно из рис. 11.1, меню `JMenu` относится к типу кнопок, расширяющих класс `AbstractButton`. Кроме того, класс `JMenu` непосредственно расширяет класс пункта меню `JMenuItem`. Следовательно, объект класса `JMenu` может служить пунктом какого-то другого меню, образуя таким образом подменю.

Меню создается конструктором `JMenu(String)`. Второй конструктор `JMenu(String, boolean)` создает *плавающее* (tear-off) меню, если второй параметр равен `true`. Это возможно не во всех графических системах.

Вновь созданное меню не содержит ни одного пункта. Пункты меню добавляются один за другим методом `add(JMenuItem)` или методом `add(String)`. Интересно, что эти методы возвращают ссылку на объект класса `JMenuItem`, а второй метод сам создает такой объект. Еще один метод, `add(Component)`, добавляет к меню произвольный компонент. Это означает, что пунктом меню может служить любой компонент, но для встраивания в систему меню он должен реализовать интерфейс `MenuElement`. Например, иногда пунктом меню служит раскрывающийся список `JComboBox`. Но чаще среди пунктов меню встречаются экземпляры подклассов класса `JMenuItem`: подменю — объекты класса `JMenu`, кнопки выбора класса `JCheckBoxMenuItem` и радиокнопки класса `JRadioButtonMenuItem`.

В меню можно отделить одну группу пунктов от другой горизонтальной чертой с помощью метода `addSeparator()`.

Пункты меню, включая разделительную черту, нумеруются сверху вниз, начиная от нуля. Методы `insert(JMenuItem, int)`, `insert(String, int)` и `add(Component, int)` позволяют вставить новый пункт в указанную вторым параметром позицию, а метод `insertSeparator(int)` вставляет горизонтальную разделительную черту в указанную позицию.

Методы `remove(Component)`, `remove(int)`, `remove(JMenuItem)` и `removeAll()` удаляют пункты из меню. В сочетании с методами `add()` и `insert()` они позволяют динамически перестроить меню при изменении содержимого окна.

Меню, как и всякой кнопке, можно назначить командную клавишу методом `setMnemonic(int)`. Добавим командные клавиши-акселераторы к меню нашего примера:

```
file.setMnemonic(KeyEvent.VK_A);
edit.setMnemonic(KeyEvent.VK_G);
view.setMnemonic(KeyEvent.VK_D);
help.setMnemonic(KeyEvent.VK_C);
```

Меню реагирует на событие класса `MenuEvent`, происходящее при раскрытии, выборе пунктов и закрытии меню.

Пункт меню *JMenuItem*

Класс `JMenuItem` расширяет класс `AbstractButton` и поэтому во многом наследует поведение кнопки. При создании пункта меню в нем можно задать текст `JMenuItem(String)`, изображение `JMenuItem(Icon)` или сразу и то и другое конструктором `JMenuItem(String, Icon)`. Взаимное положение текста и изображения можно отрегулировать так же, как это делалось для кнопки.

Добавим пункты и разделительную черту к меню **Файл** нашего примера. Забегая вперед, добавим и методы обработки событий. Объяснение этих методов будет дано в *главе 15*.

```
   JMenu nw = new JMenu("Создать");

file.add(nw);           // Добавляем как подменю
  nw.add("Файл");       // Пункты подменю
  nw.add("Сообщение");
  nw.add("Образ");
   JMenuItem open = file.add("Открыть...");

   JMenuItem close = file.add("Закрыть");

   file.addSeparator();

// Другой способ:
JMenuItem exit = new JMenuItem("Выход");
   file.add(exit);

// Обрабатываем события:
open.addActionListener(new ActionListener(){
  public void actionPerformed(ActionEvent e){
    JFileChooser fch = new JFileChooser();
    fch.showOpenDialog(null);
  }
   });

exit.addActionListener(new ActionListener(){
  public void actionPerformed(ActionEvent e){
    System.exit(0);
  }
});
```

При создании пункта меню, содержащего текст, можно сразу же задать командную клавишу-акселератор, используя конструктор `JMenuItem(String, int)`. Потом это сделать не удастся, поскольку метод `setMnemonic(int)` не реализован в классе `JMenuItem`, точнее говоря, он переопределен так, что только выбрасывает исключение. Назначать командную клавишу следует специальным методом `setAccelerator(KeyStroke)`, при этом в пункт меню добавляется описание командной клавиши, например **Ctrl+O**. Добавим эту командную клавишу к пункту **Открыть** нашего меню **Файл**:

```
open.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_O,
   Event.CTRL_MASK));
```

Класс `JMenuItem` отслеживает, кроме унаследованных событий `ChangeEvent`, `ActionEvent`, `ItemEvent`, еще и события классов `MenuKeyEvent`, происходящие при нажатии и отпускании клавиш, и `MenuDragMouseEvent`, происходящие при прохождении курсора мыши над пунктом меню.

Пункт меню *JCheckBoxMenuItem*

Вставить в меню кнопки выбора удобнее всего с помощью специально включенного в библиотеку Swing класса `JCheckBoxMenuItem`. Этот класс наследует все свойства своего суперкласса `JMenuItem` и добавляет логический метод `getState()`, позволяющий отследить

состояние кнопки. Впрочем, можно пользоваться и унаследованным методом `isSelected()`.

Добавим кнопки выбора к меню **Вид** нашего примера:

```
JCheckBoxMenuItem cbm1 = new JCheckBoxMenuItem("Текст");
JCheckBoxMenuItem cbm2 = new JCheckBoxMenuItem("Значки");
   JCheckBoxMenuItem cbm3 = new JCheckBoxMenuItem("Рисунки");

   view.add(cbm1); view.add(cbm2); view.add(cbm3);

   view.addSeparator();

cbm1.addItemListener(new ItemListener(){
  public void itemStateChanged(ItemEvent e){
    if (e.getStateChange() == ItemEvent.SELECTED)
       ch.setText(txt);
    else ch.setText("");
  }
});
```

Пункт меню *JRadioButtonMenuItem*

Для размещения в меню группы радиокнопок удобно воспользоваться классом `JRadioButtonMenuItem`. Он не добавляет функциональности своему суперклассу `JMenuItem` и используется точно так же, как в обычной группе радиокнопок.

Добавим группу радиокнопок к меню **Вид** нашего примера:

```
JRadioButtonMenuItem rbm1 =
     new JRadioButtonMenuItem("Большой");

JRadioButtonMenuItem rbm2 =
     new JRadioButtonMenuItem("Средний");

JRadioButtonMenuItem rbm3 =
     new JRadioButtonMenuItem("Малый");

   view.add(rbm1);view.add(rbm2); view.add(rbm3);

   ButtonGroup bg = new ButtonGroup();

bg.add(rbm1); bg.add(rbm2); bg.add(rbm3);
```

Всплывающее меню *JPopupMenu*

Всплывающее меню (pop-up menu) используется обычно как контекстное меню. Оно появляется в MS Windows и Java L&F при отпускании правой кнопки мыши. В некоторых графических системах для появления контекстного меню надо нажать среднюю кнопку мыши. Контекстное меню обычно содержит перечень действий, доступных в компоненте, над которым находится курсор мыши, при данном положении курсора мыши. Поэтому контекст-

ное меню не привязывается к строке меню или другому меню, а соотнесено с одним или несколькими компонентами. Оно добавляется к компонентам при обработке событий мыши класса `MouseEvent`.

Для того чтобы отследить событие, при наступлении которого в данной графической системе следует вызвать всплывающее меню, в классе `MouseEvent` есть логический метод `isPopupTrigger()`. К этому методу следует обращаться при всяком событии мыши. Когда метод `isPopupTrigger()`вернет `true`, показав тем самым, что надо вызывать всплывающее меню, следует обратиться к методу `show(Component, int, int)`. В нем первый параметр — это компонент, над которым появится окно всплывающего меню, второй и третий параметры — координаты курсора мыши в системе координат этого компонента. Вот стандартная конструкция, в которой `popup` — это экземпляр класса `JPopupMenu`:

```
public void processMouseEvent(MouseEvent e){
    if (e.isPopupTrigger())
         popup.show(e.getComponent(), e.getX(), e.getY());
    else super.processMouseEvent(e);
}
```

При этом следует убедиться, что компонент, для которого вызывается всплывающее меню и в котором записан приведенный выше метод `processMouseEvent()`, отслеживает события мыши, т. е. к нему присоединен методом `addMouseListener()` хотя бы один слушатель или в нем есть обращение к методу `enableEvents(AWTEvent.MOUSE_EVENT_MASK)`.

Действия со всплывающим меню похожи на действия с обычным меню. Сначала создается пустое меню конструктором `JPopupMenu()` или `JPopupMenu(String)`. Строка, записанная во втором конструкторе, должна быть заголовком меню, но она отображается на экран не всеми графическими системами. Эту строку можно добавить потом методом `setLabel(String)`. Затем в созданное всплывающее меню методами `add(Action)`, `add(JMenuItem)` или `add(String)` добавляются пункты меню. Методы `insert()` и `remove()` позволяют динамически перестроить меню.

Всплывающее меню перед своим появлением на экране, исчезновением с экрана и перед уничтожением вызывает событие класса `PopupMenuEvent`.

Панель выбора цвета *JColorChooser*

Многие приложения, связанные с рисованием, обработкой текстов и изображений, требуют задания определенных цветов. Библиотека Swing предлагает простой класс `JColorChooser`, предоставляющий панель с палитрами цветов в моделях RGB и HSB. Есть три способа использования этого класса.

- Самый простой способ — создать диалоговое окно, содержащее панель выбора цветов статическим методом `showDialog(Component, String, Color)`. Первый параметр этого метода задает окно верхнего уровня, в которое помещается панель. Значение `null` указывает, что окном будет служить экземпляр класса `Frame`, располагающийся в центре экрана. Второй параметр дает заголовок этому окну, а третий параметр задает начальный цвет, причем `null` устанавливает белый цвет. Например:

```
Color c = JColorChooser.showDialog(null, "Цвет", null);
```

На экране появляется модальное диалоговое окно с цветовой палитрой, ползунками, задающими интенсивность цвета, и кнопками **OK**, **Cancel** и **Reset**. Метод `showDialog()` возвра-

щает выбранный цвет после щелчка по кнопке **OK** и `null`, если щелчок был сделан по кнопке **Cancel**. После этого диалоговое окно удаляется с экрана, но не уничтожается. Оно запоминает выбранный цвет и при следующем выборе этот цвет можно восстановить щелчком по кнопке **Reset**. При первом появлении окна на экране щелчок по кнопке **Reset** возвращает начальный цвет, заданный третьим параметром метода.

- Более гибкий и сложный способ — создать диалоговое окно с панелью цветов статическим методом `createDialog(Component, String, boolean, JColorChooser, ActionListener, ActionListener)`. Этот метод, кроме окна верхнего уровня и его заголовка, позволяет задать третьим параметром модальность диалогового окна. Четвертый параметр обеспечивает выбор цвета не только экземпляра класса `JColorChooser`, но и любого расширения этого класса. Пятый и шестой параметры позволяют задать нестандартную обработку щелчков по кнопкам **OK** и **Cancel** соответственно. При этом выбранный цвет можно получить методом `getColor()`. Например:

```
JDialog d = JColorChooser.createDialog(
    new JFrame(), "Выбор цвета", false,
    cc = new JColorChooser(),
    new OkColor(), new CancelColor());

   d.setVisible(true);

class OkColor implements ActionListener{
     public void actionPerformed(ActionEvent e){
        comp.setColor(cc.getColor());
     }
   }

class CancelColor implements ActionListener{
     public void actionPerformed(ActionEvent e){
        comp.setColor(defColor);
     }
}
```

- Если нет необходимости создавать отдельное диалоговое окно, а нужна только панель выбора цветов, то можно воспользоваться конструктором `JColorChooser(Color)`, в котором задается начальный цвет, или конструктором `JColorChooser()`, устанавливающим белый цвет в качестве исходного.

В этом случае определять момент окончания выбора цвета придется самостоятельно. Класс `JColorChooser` отслеживает только событие `PropertyChangeEvent` изменения свойств Java Bean, которое можно обработать, присоединив обработчик унаследованным от класса `JComponent` методом

```
addPropertyChangeListener(PropertyChangeListener);
```

Второй способ отследить выбор цвета — обратиться к модели данных, все интерфейсы и классы которой собраны в пакет `javax.swing.colorchooser`.

Текущий выбор цвета хранится в модели данных, описанной интерфейсом `ColorSelectionModel`. Класс `JColorChooser` использует реализацию этого интерфейса классом `DefaultColorSelectionModel`. Экземпляр данного класса можно получить методом `getSelectionModel()`. Эта модель отслеживает событие `ChangeEvent`.

При необходимости изменения модели данных можно сделать другую реализацию интерфейса `ColorSelectionModel` и установить ее при создании объекта конструктором `JColorChooser(ColorSelectionModel)`.

Упражнение

4. Перепишите "рисовалку" листинга 10.9 или листинга 10.10 с использованием компонентов Swing.

Окно выбора файла *JFileChooser*

Подавляющему большинству приложений приходится работать с файлами. Библиотека Swing имеет в своем составе законченный компонент `JFileChooser`, соответствующий стандартному окну выбора файла большинства операционных систем.

Для создания экземпляра окна выбора файла есть несколько конструкторов. Конструктор `JFileChooser(File)` или `JFileChooser(String)` создает окно, в котором показан каталог с указанным файлом. Конструктор по умолчанию `JFileChooser()` открывает окно с начальным каталогом пользователя. Он соответствует `JFileChooser(null)`. Еще в трех конструкторах задается объект класса `FileSystemView`, позволяющий получить различные атрибуты файла.

По умолчанию окно показывает только файлы (режим `FILES_ONLY`). Перед выводом окна на экран можно установить режим показа только каталогов `DIRECTORIES_ONLY` или файлов и каталогов `FILES_AND_DIRECTORIES`. Эти режимы устанавливаются методом `setFileSelectionMode(int)`.

По умолчанию окно не отображает скрытые (hidden) файлы. Чтобы задать их показ, надо обратиться к методу `setFileHidingEnabled(false)`.

По умолчанию в окне можно отметить один файл. Возможность выбора нескольких файлов задается методом `setMultiSelectionEnabled(true)`.

Фильтр файлов *FileFilter*

По умолчанию окно показывает все файлы в выбранном каталоге. Установив фильтр, можно ограничить отображение отдельными файлами. Для этого надо расширить абстрактный класс `FileFilter` из пакета `javax.swing.filechooser` (не перепутайте с интерфейсом `FileFilter` из пакета `java.io`) и установить полученный фильтр в окне выбора файла методом `addChoosableFileFilter(FileFilter)`. Этот метод можно применить несколько раз с разными параметрами, определив несколько фильтров в одном окне.

Класс `FileFilter` содержит всего два абстрактных метода. Логический метод `accept(File)` возвращает `true`, если его параметр следует показать в окне. Метод `getDescription()` возвращает строку описания данного фильтра, которая будет отображена в поле **Тип файлов:** (Files of type:) окна выбора.

Вот пример фильтра, отбирающего только файлы с расширением java. Вид окна открытия файла с соответствующим фильтром показан на рис. 11.5.

```
class JavaFileFilter extends javax.swing.filechooser.FileFilter{
  public boolean accept(File f){
```

```
    if (f != null){
         String name = f.getName();

         int i = name.lastIndexOf('.');

      if (i>0 && i < name.length() - 1)
        return name.substring(i + 1).equalsIgnoreCase("java");
    }
    return false;
     }

  public String getDescription(){
    return "Файлы Java";
  }
}
```

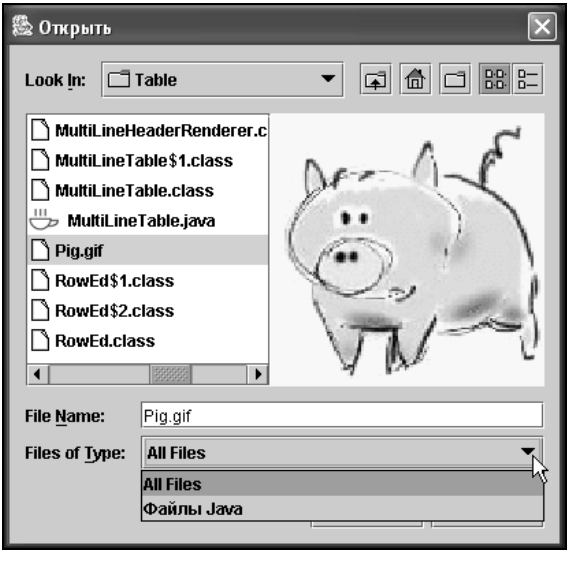


Рис. 11.5. Окно открытия файла

После создания экземпляра окна выбора и установки режимов и фильтров окно можно показать на экране как модальное окно открытия файла методом `showOpenDialog(Component)`, как модальное окно сохранения файла `showSaveDialog(Component)` или как модальное окно произвольного выбора `showDialog(Component, String)`. В последнем методе второй параметр задает произвольную надпись вместо надписи **Открыть** (Open) или **Сохранить как** (Save). Первый параметр этих методов задает компонент, от которого зависит и над контейнером которого будет расположено окно выбора файла. Чаще всего это `Frame`, который можно задать просто параметром `null`.

Итак, создать и показать окно выбора файла очень просто:

```
   JFileChooser fch = new JFileChooser();

fch.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
```

```
fch.setFileHidingEnabled(false);
fch.setMultiSelectionEnabled(true);
fch.addChoosableFileFilter(new JavaFileFilter());
   fch.addChoosableFileFilter(new AnotherFileFilter());

switch (fch.showDialog(null, "Открыть")){
  case JFileChooser.APPROVE_OPTION:
    File selectedFile = fch.getSelectedFile();
    File directory = fch.getCurrentDirectory();
    break;
  case JFileChooser.CANCEL_OPTION:
    break;
  case JFileChooser.ERROR_OPTION:
    System.err.println("Error"); break;
}
```

Как получить выбранный файл

Как видно из этого примера, методы вида `showXxxDialog()` возвращают целое число — одну из трех констант, соответствующих выбору файла и щелчку по кнопке **Открыть** (Open) или **Сохранить** (Save) — константа `APPROVE_OPTION`, щелчку по кнопке **Отмена** (Cancel) — константа `CANCEL_OPTION` или появлению ошибки — константа `ERROR_OPTION`.

После того как пользователь отметил файл в окне выбора, этот файл можно получить методом `getSelectedFile()` в виде экземпляра класса `File`, как показано выше. Если установлен режим выбора и файлов, и каталогов — `FILES_AND_DIRECTORIES`, — то при выборе каталога этот метод возвращает `null`. Если же установлен режим выбора только каталогов `DIRECTORIES_ONLY`, то возвращается каталог. Если задан выбор нескольких файлов, то их массив типа `File[]` можно получить методом `getSelectedFiles()`. Каталог, в котором находится выбранный файл, можно получить в виде экземпляра класса `File` методом `getCurrentDirectory()`, как показано выше.

Итак, стандартная работа с окном выбора файлов выглядит следующим образом:

```
   JFileChooser fch = new JFileChooser();

   int state = fch.showOpenDialog(null);

   File f = fch.getSelectedFile();

if (f != null && state == JFileChooser.APPROVE_OPTION)
    JOptionPane.showMessageDialog(null, f.getPath());
else if (state == JFileChooser.CANCEL_OPTION)
    JOptionPane.showMessageDialog(null, "Canceled");
```

Разумеется, окно выбора файла не открывает и не сохраняет файл на самом деле, оно только предоставляет этот файл вызвавшей окно выбора программе.

Дополнительный компонент

Возможности окна выбора файлов легко расширить, добавив в него произвольный компонент методом `setAccessory(JComponent)`. Чаще всего добавляется небольшое окно предварительного просмотра отмеченного файла, хотя можно добавить что угодно, даже еще одно окно выбора файла. Вот, например, класс, формирующий окно предварительного просмотра файлов с изображениями:

```
 class ImagePreviewer extends Jlabel implements PropertyChangeListener{

  public ImagePreviewer(JFileChooser fch){
    if (fch == null)
      throw new IllegalArgumentException("fileChooser must be non-null");
    fch.addPropertyChangeListener(this);
     }

  public void loadImageFromFile(File f){
    Icon icon = null;
    if (f != null){
     ImageIcon im = new ImageIcon(f.getPath());
        Dimension size = getSize();

     if (im.getIconWidth() != size.width)
       icon = new ImageIcon(im.getImage().getScaledInstance(
          size.width, size.height, Image.SCALE_DEFAULT));
     else icon = im;
    }
    setIcon(icon);
     }

  public void propertyChange(PropertyChangeEvent e){
    String prop = e.getPropertyName();
    if (prop.equals(JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)){
      File f = (File)e.getNewValue();
      if (isShowing()){
        loadImageFromFile(f);
        repaint();
      }
    }
  }
}
```

После определения объекта класса `JFileChooser`, но до его вывода на экран, присоединяем этот компонент:

```
JFileChooser fch = new JFileChooser();
ImagePreviewer ip = new ImagePreviewer(fch);
ip.setPreferredSize(new Dimension(200, 200));
fch.setAccessory(ip);
```

Получившееся окно открытия файла показано на рис. 11.5.

Замена изображений

В окне выбора файла перед каталогами и файлами выводятся стандартные значки. Их можно заменить какими-то другими изображениями. Для этого надо расширить абстрактный класс `FileView`, переопределив пять его методов, и установить новые изображения методом `setFileView(FileView)`. Например, выведем перед исходными файлами Java изображение дымящейся чашечки кофе:

```
   class JavaFileView extends FileView{

  public String getName(File f){
     return null;        // Оставляем системную реализацию
     }

  public String getDescription(File f){
    String ext = getExtension(f);
    if (ext != null && ext.equals("java"))
       return "Исходный файл Java";
    else return null;
     }

  public String getTypeDescription(File f){
     return getDescription(f);
     }

     Icon icon = new ImageIcon("javacup.gif");

  public Icon getIcon(File f){
    String ext = getExtension(f);
    if (ext != null && ext.equals("java"))
       return icon;
    return null;
     }

   public Boolean isTraversable(File f){
     return null;       // Оставляем системную реализацию
      }

   protected String getExtension(File f){
      if (f != null){
         String name = f.getName();
         int i = name.lastIndexOf('.');
         if (i > 0 && i < name.length() - 1)
             return name.substring(i + 1).toLowerCase();
       }
       return null;
   }
}
```

Теперь достаточно к описанию объекта `fch` класса `JFileChooser` добавить строку

```
fch.setFileView(new JavaFileView());
```

и стандартный значок перед исходными файлами Java с расширением java будет заменен изображением из файла javacup.gif, как показано на рис. 11.5.

Более содержательный пример создания окна выбора файлов имеется в составе J2SE JDK в каталоге $JAVA_HOME/demo/jfc/FileChooserDemo/.

Окно выбора файлов чаще всего создается в отдельном диалоговом окне, но это обычный компонент Swing, и его можно поместить в контейнер. При этом можно убрать кнопки выбора и отмены выбора методом `setControlButtonsAreShown(boolean)`.

Русификация Swing

Как видно из примеров этой главы, компоненты библиотеки Swing правильно отображают кириллицу по обычным правилам, изложенным в *главе 5*. Остается лишь русифицировать стандартные надписи, что видно на рис. 11.6. Для этого надо написать несколько файлов ресурсов (properties), которые, если они есть, просматриваются классами-рисовальщиками компонентов перед выводом на экран. Эти файлы уже написаны Сергеем Астаховым и лежат в файле swing_ru.jar, ссылку на который можно найти, например, по адресу **http://lib.juga.ru/article/articleprint/125/-1/53/**. Достаточно поместить архив swing_ru.jar, не распаковывая, в каталог $JAVA_HOME/lib/ext/, и все надписи будут сделаны по-русски, как показано на рис. 11.6.

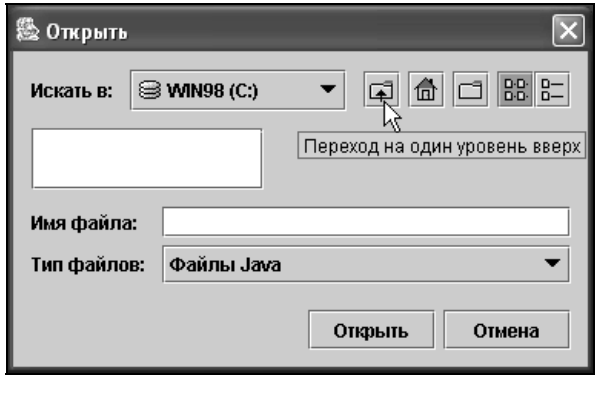


Рис. 11.6. Русифицированное окно открытия файла

На том же сайте: **http://lib.juga.ru/article/archive/19/**, да и на других страницах Интернета находится статья Сергея Астахова "Русские буквы и не только...", в которой собрано все касающееся русификации Java. По всем вопросам, возникающим при работе с кириллицей, безусловно, нужно обращаться к этому ресурсу.

Вопросы для самопроверки

1. Чем отличаются компоненты Swing от компонентов AWT?
2. Какая конструктивная схема использована в компонентах Swing?
3. В каких случаях приходится изменять Модель компонента?
4. В каких случаях приходится изменять Вид компонента?
5. Какими компонентами Swing можно создать кнопку с двумя состояниями?
6. Какие компоненты Swing создают радиокнопки?
7. Как в Swing показать каталоги и имена файлов в виде дерева?
8. Можно ли средствами Swing создать всплывающее меню?

Глава 12



Текстовые компоненты

При создании приложения с графическим интерфейсом очень часто приходится использовать поля для ввода текста. Такое поле может состоять из одной или нескольких строк, быть редактируемым или не редактируемым. В редактируемом окне часто приходится задавать возможность смены шрифта, цвета, вставку дополнительных символов. Библиотека Swing предоставляет для этого большие возможности, предлагая пакеты интерфейсов и классов `javax.swing.text`, `javax.swing.text.html`, `javax.swing.text.html.parser` и `javax.swing.text.rtf`.

На вершине иерархии текстовых компонентов стоит класс `JTextComponent` — непосредственное расширение класса `JComponent`. Это абстрактный класс, вобравший в себя общие свойства всех текстовых компонентов. У него три расширения: однострочное текстовое поле `JTextField`, многострочная текстовая область `JTextArea` и небольшой, но мощный текстовый редактор `JEditorPane`, имеющий расширение — класс `JTextPane`, методами которого можно оформить текст в каком-то определенном стиле.

У класса `JTextField` есть два расширения — поле для ввода пароля `JPasswordField`, в котором вместо вводимых символов показывается один заранее определенный символ, по умолчанию звездочка, и поле для редактирования форматированных объектов `JFormattedTextField`, например даты — объекта класса `Date`, или чисел, заданных в определенном формате.

Рассмотрим текстовые компоненты подробнее и начнем с вершины их иерархии.

Компонент *JTextComponent*

Абстрактный класс `JTextComponent` стоит на вершине иерархии текстовых компонентов и содержит их общие свойства.

Текстовые компоненты построены по схеме "Model-View-Controller", которая отражена во внутреннем устройстве класса `JTextComponent`. Рассмотрим это устройство подробнее. Начнем с модели данных текстовых компонентов.

Модель данных — документ

Модель данных схемы MVC текстовых компонентов описана интерфейсом `Document` и называется *документом*. Документ может быть простым, "плоским", или сложным, структурированным. Интерфейс `Document` описывает *простой текст* (content), содержащийся в компо-

ненте, как последовательность символов Unicode. Количество символов в тексте можно узнать методом `getLength()`. Последовательность символов нумеруется, начиная от нуля. Каждый символ имеет свой порядковый номер, называемый *позицией* (position, location или offset) символа в тексте. Точнее говоря, модель данных задает позицию не символа, а позицию *между* символами, перед текущим символом текста, что удобно для вставки текста.

Если текст документа сложный, структурированный, то, чтобы скрыть сложность определения текущей позиции при частых вставках и удалениях текста, удобно воспользоваться объектом, описанным интерфейсом `Position`. Экземпляр класса, реализующего интерфейс `Position`, создается методом `createPosition(int)`. Методы `getStartPosition()` и `getEndPosition()` возвращают начальную и конечную позиции текста в виде объекта, реализующего интерфейс `Position`. В интерфейсе `Position` всего один метод — `getOffset()`, возвращающий позицию символа в виде целого числа типа `int`. Кроме того, в интерфейсе `Position` есть вложенный класс `Bias`, в котором определены два экземпляра класса `Bias`: поле `Forward` и поле `Backward`. Они уточняют символ, позиция которого определена: текущий или предыдущий. Позиция символа используется, например, при получении текста методами

```
String getText(int offset, int length);
void getText(int offset, int length, Segment text);
```

извлекающими из документа фрагмент текста длиной `length` символов, начиная от позиции `offset`. Второй метод заносит извлеченный фрагмент в экземпляр `text` класса `Segment`. Опишем этот небольшой, но удобный класс.

Строка символов *Segment*

Класс `Segment` представляет строку символов в виде, удобном для быстрого просмотра. У строки класса `Segment` всегда есть текущий символ, который можно получить методом `current()`. Позицию текущего символа можно узнать методом `getIndex()`, а установить — методом `setIndex(int)`. Класс `Segment` содержит методы `previous()` и `next()`, возвращающие предыдущий и следующий символ строки, а также методы `first()` и `last()`, возвращающие первый и последний символ строки. Будьте внимательны: эти четыре метода меняют текущую позицию строки!

Интересно, что для быстроты доступа строка класса `Segment` хранится в *открытом* (public) поле — массиве с именем `array` типа `char[]`, начиная с индекса `offset`, и занимает в этом массиве `count` элементов. Это открытые поля, так что к массиву символов можно обращаться напрямую и менять его элементы, хотя при этом теряется главное назначение класса `Segment` — быстро просматривать текст. Метод `getBeginIndex()` возвращает индекс начала строки в массиве `array`, т. е. число `offset`, а `getEndIndex()` — индекс элемента массива, следующего за последним символом строки.

Заканчивая обзор класса `Segment`, скажем, что для большей надежности выполнения метода `getText(int, int, Segment)` в класс `Segment` введен метод `setPartialReturn(boolean)`. Если в этом методе задать параметр `true`, то передача текста методом `getText()` в экземпляр класса `Segment` будет происходить, по возможности, без дополнительного копирования. Значение параметра по умолчанию — `false`. С учетом этого, работа с классом `Segment` начинается примерно так:

```
Segment seg = new Segment();
seg.setPartialReturn(true);
```

```
doc.getText(0, doc.getLength(), seg);
// Работаем с объектом seg...
```

Запись текста в документ

Прежде чем использовать текст документа, его надо записать в документ. Запись выполняется методом

```
void insertString(int offset, String text, AttributeSet attr);
```

Новый текст `text` вставляется перед символом с позицией `offset`, позиция этого и следующих символов увеличивается на длину вставленного текста.

Атрибуты текста

Как видно из сигнатуры метода `insertString()`, у вносимого текста могут быть атрибуты, например: имя шрифта, размер шрифта, цвет. Если у текста нет атрибутов, то третьему параметру метода надо дать значение `null`.

Атрибуты записываются в виде пар "имя — значение" в объект, реализующий интерфейс `AttributeSet`. Этот интерфейс описывает неизменяемое множество атрибутов, в нем не описаны методы добавления и удаления атрибутов. Такие методы внесены в его расширение — интерфейс `MutableAttributeSet`. В библиотеке Swing есть реализация данного интерфейса — класс `SimpleAttributeSet`. С помощью этого класса можно определить любые пары "имя — значение", но общепринятые атрибуты удобнее задавать с использованием констант и статических методов класса `StyleConstants` и четырех его подклассов, которые в то же время вложены в него: `CharacterConstants`, `ColorConstants`, `FontConstants` и `ParagraphConstants`.

Объект, реализующий интерфейс `AttributeSet`, может содержать ссылку на другой, "родительский" объект того же типа. Ссылка хранится как значение атрибута, имеющего имя `ResolveAttribute`. Так можно получить цепочку объектов, содержащих атрибуты текста. Если какая-то пара "имя — значение" не найдена в первом объекте методом `getAttribute(Object)`, то она отыскивается в родительском объекте, который определяется методом `getResolveParent()`, затем поиск идет далее по цепочке.

У интерфейса `MutableAttributeSet` есть свое расширение — интерфейс `Style`. Это расширение дает возможность получить имя множества атрибутов методом `getName()`, создав так называемый *стиль* (style), и присоединить к множеству слушателя события `ChangeEvent` методом

```
void addChangeListener(ChangeListener chl);
```

Заданное имя затем можно использовать как значение другого атрибута, а с помощью слушателя отслеживать добавление и удаление атрибутов.

Для создания стилей и работы с ними очень полезен класс `StyleContext`. Его экземпляр с общепринятым набором атрибутов можно получить следующим образом:

```
StyleContext stc = StyleContext.getDefaultStyleContext();
```

Затем в полученный объект `stc` можно добавить новые атрибуты методами `addAttribute()`, удалить атрибуты методами `removeAttribute()`, создать цепочку стилей.

Удаление текста из документа

Обратная операция — удаление части или всего текста из документа — выполняется методом

```
remove(int offset, int length);
```

Он удаляет `length` символов, начиная от символа, находящегося в позиции `offset`.

Фильтрация документа

Операции занесения текста в документ методом `insertString()` и удаления методом `remove()` можно изменить так, чтобы они производили различные проверки и модификации, фильтруя таким способом вставляемый или удаляемый текст. Это можно сделать прямым расширением класса, реализующего `Document`, и переопределением его методов `insertString()` и `remove()`. Но есть еще один способ, не изменяющий документ.

Сначала надо расширить класс `DocumentFilter`, переопределив его методы. В классе `DocumentFilter` всего три метода:

```
void insertString(DocumentFilter.FilterBypass fb, int offset,
                  String text, AttributeSet attr);
void remove(DocumentFilter.FilterBypass fb, int offset, int length);
void replace(DocumentFilter.FilterBypass fb, int offset, int length,
             String text, AttributeSet attr);
```

Их стандартная реализация просто вызывает соответствующие методы вложенного абстрактного класса `FilterBypass`, которые оставлены абстрактными.

После расширения класса `DocumentFilter`, вызванного переопределением методов, полученный фильтр надо установить в документ методом

```
void setDocumentFilter(DocumentFilter filter);
```

класса `AbstractDocument`. Далее всякое обращение к методам `insertString()` и `remove()` документа будет пропускаться через методы созданного фильтра.

Пример использования фильтра, пропускающего только цифры, приведен в листинге 12.1.

Внесение структуры в документ

В модель данных можно внести структуру дерева, например разбить документ на главы, параграфы, разделы. Каждый элемент разбиения описывается интерфейсом `Element`. Элемент занимает какую-то область текста с начальной позицией, возвращаемой методом `getStartOffset()`, и конечной позицией `getEndOffset()`. У элемента может быть родительский элемент, который легко получить методом `getParentElement()`. Область текста, занимаемая родительским элементом, полностью включает в себя область исходного элемента. У элемента могут быть дочерние элементы, чья область текста полностью лежит внутри области самого элемента. Их можно получить методом `getElement(int)`. Число дочерних элементов возвращает метод `getElementCount()`, а индекс дочернего элемента — метод `getElementIndex(int)`. Элементу можно дать имя, а затем получить его методом `getName()`.

Интерфейс `Element` частично реализован абстрактным классом `AbstractElement`, вложенным в класс `AbstractDocument`, и полностью реализован еще двумя вложенными

в AbstractDocument классами — BranchElement и LeafElement, расширяющими класс AbstractElement. Основная разница между ними в том, что у класса BranchElement могут быть дочерние элементы, а у класса LeafElement — нет.

Класс BranchElement, в свою очередь, расширяется классом SectionElement, вложенным в класс DefaultStyledElement, и классом BlockElement, вложенным в класс HTMLDocument.

Класс LeafElement расширяется классом RunElement, вложенным в класс HTMLDocument.

Элементы создаются методами

```
Element createLeafElement(Element parent, AttributeSet attr,
                          int pos1, int pos2);
Element createBranchElement(Element parent, AttributeSet attr);
```

Для каждого структурного элемента можно задать свое множество атрибутов методом addAttribute(Object name, Object value) или методом addAttributes(AttributeSet).

В документе допускается задание несколько независимых структур. Их корневые элементы можно получить методом getRootElements(), возвращающим массив типа Element[]. Один из корневых элементов можно сделать корневым элементом по умолчанию и получать его методом getDefaultRootElement(), возвращающим элемент в виде объекта, реализующего интерфейс Element. Получив корневые элементы структурного дерева, легко обойти его, используя метод getElement(int) интерфейса Element или метод children(), имеющийся в его реализациях.

События в документе

При всяком изменении документа или его структуры происходит событие, описанное интерфейсом DocumentEvent. Он предлагает метод getDocument(), позволяющий узнать, в каком документе произошло событие, методы getOffset() и getLength(), сообщающие о начальной позиции и длине измененного текста. Вложенный интерфейс ElementChange содержит метод getElement(), позволяющий узнать элемент, в котором произошло событие, и методы, помогающие отследить добавление и удаление элемента из документа.

Реализация интерфейса DocumentEvent — класс DefaultDocumentEvent, вложенный в класс AbstractDocument, — добавляет к методам интерфейса метод undo(), отменяющий изменения, метод redo(), восстанавливающий изменения, и еще несколько информационных методов.

Если модель данных позволяет отменять и восстанавливать изменения (undo/redo), то при каждом таком действии в ней происходит событие класса UndoableEditEvent.

Реализации документа

Интерфейс Document частично реализован абстрактным классом AbstractDocument. Этот класс вносит понятие блокировки документа. Документ могут просматривать несколько подпроцессов-"читателей" и один подпроцесс-"писатель". Доступ их к документу блокируется методами readLock() и writeLock(). Блокировки снимаются методами readUnlock() и writeUnlock().

Класс AbstractDocument обычно не расширяется непосредственно, а используются или расширяются его подклассы PlainDocument и DefaultStyledDocument.

Класс PlainDocument задает модель простого документа с "плоским" текстом, которая используется полями ввода JTextField, JPasswordField, JTextArea. Текст в этой модели имеет структуру: структурные элементы текста — это строки. Корневой элемент структуры можно получить методом getDefaultRootElement(). Метод getParagraphElement(int offset) возвращает элемент структуры — строку в виде объекта типа Element, к которому принадлежит позиция offset. Каждой строке, как любому элементу структуры, можно придать атрибуты. Отдельные символы атрибутов не имеют.

Более сложную структуру вносит в документ модель класса DefaultStyledDocument, используемая в текстовом редакторе JTextPane. В этой модели не только строке, но и каждому символу текста можно задать свой стиль.

Наконец, класс DefaultStyledDocument расширяется классом HTMLDocument. Это модель разметки языка HTML. В ней можно определить *таблицы стилей* (style sheets).

Установка модели данных

После того как новая модель данных определена, ее надо установить в компонент методом setDocument(Document) класса JTextComponent.

Вид

Вторая часть схемы MVC — *Вид* — построена для текстовых компонентов так, что для каждого элемента документа создается свой вид, значит, каждый элемент может получить свое графическое оформление.

Эта идея частично реализована абстрактным классом View. Он задает для каждого документа целую структуру *Видов*, отвечающую структуре элементов документа. Каждый вид из этой структуры при своем создании получает ссылку на определенный элемент документа, которую можно отследить методом getElement(). Документ, к которому относится *Вид*, можно получить методом getDocument(). У каждого *Вида* есть родительский вид, который определяется методами getParent() и setParent(), и множество дочерних видов. Их число можно определить методом getViewCount(), получить дочерние виды можно методом getView(int).

Создание дочерних *Видов* выполняется методом create(Element), описанным в интерфейсе ViewFactory. В документе у каждого элемента есть объект, реализующий этот интерфейс, который можно получить методом getViewFactory(). Еще несколько методов занимаются добавлением видов в иерархию и удалением их оттуда.

Каждый *Вид* должен преобразовать линейный порядок следования символов в элементе документа, определяемый их позициями, в двумерное отображение символов на экран, состоящее из нескольких строк экрана. Это выполняется методом

```
Shape modelToView(int startPos, Position.Bias b0,
                  int endPos, Position.Bias b1, Shape fig);
```

который возвращает двумерную фигуру класса Shape. Параметры метода задают начальную startPos и конечную endPos позиции текста, направления b0 и b1 от этих позиций к пре-

образуемому тексту, которые могут принимать одно из двух значений: `Position.Bias.Backward` или `Position.Bias.Forward`, и окружающую фигуру `fig`.

Обратное преобразование выполняется методом

```
int viewToModel(float x, float y, Shape fig, Position.Bias[] b);
```

возвращающим позицию символа, имеющего координаты `(x, y)` на экране в фигуре `fig`. Кроме того, метод вычисляет массив направлений `b`, уточняющий положение символа в модели данных.

Для вывода на экран в каждом виде создается графический контекст — экземпляр класса `Graphics`. Непосредственный вывод элемента на экран выполняется методом `paint(Graphics)` подобно выводу компонента. Графическим контекстом можно воспользоваться не только в методе `paint(Graphics)`, но и непосредственно, получив его методом `getGraphics()`.

Каждый *Вид* устанавливает свой размер методом `setSize(float width, float height)`, у него есть минимальный, предпочтительный и максимальный размер.

В этом класс `View` напоминает класс `Component`. Сходство усиливается тем, что подобно тому, как класс `Component` порождает множество компонентов, создающих на экране графический интерфейс, класс `View` порождает множество подклассов-видов, отображающих на экране различные типы документов. Их иерархия показана на рис. 12.1.

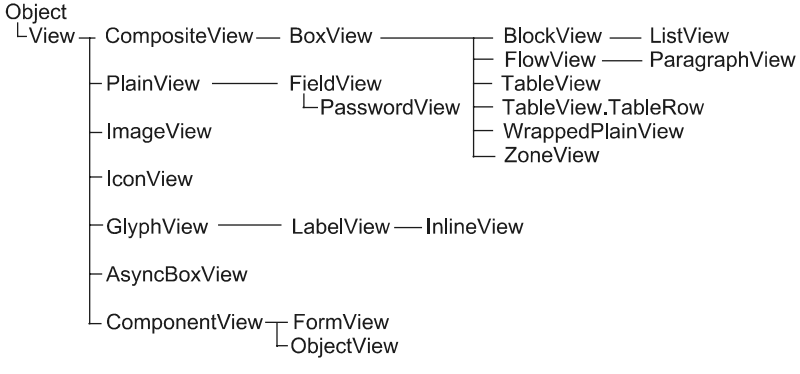


Рис. 12.1. Иерархия классов-видов

Как видно из рис. 12.1, иерархия *Видов* обширна и разветвлена. Большинство классов связано с интерпретацией языка HTML, например класс `ImageView` обрабатывает тег `<IMG>`, класс `FormView` — тег `<FORM>` и относящиеся к нему теги `<INPUT>`, `<SELECT>`, `<TEXTAREA>`, класс `ObjectView` — тег `<OBJECT>`.

Класс `PlainView` создает несколько строк текста с одним шрифтом и используется в области ввода `JTextArea`, класс `FieldView` образует одну строку и применяется в поле ввода `JTextField`, класс `PasswordView` — в поле ввода пароля `JPasswordView`.

Класс `BoxView` и его подклассы создают прямоугольный вид текста разного вида.

Такое разнообразие видов обычно удовлетворяет запросы разработчика, и созданием собственных видов приходится заниматься редко.

Контроллер — редактор текста

Третья часть модели MVC — *Контроллер* — реализована в текстовых компонентах набором редакторов текста. Их иерархия начинается с абстрактного класса `EditorKit`, в котором описаны абстрактные методы чтения из байтового и символьного входного потока, например, связанного с клавиатурой, в модель данных

```
void read(InputStream in, Document doc, int pos);
void read(Reader in, Document doc, int pos);
```

и методы записи из модели данных в байтовый и символьный потоки

```
void write(OutputStream out, Document doc, int pos, int length);
void write(Writer out, Document doc, int pos, int length);
```

Класс позволяет создать свой документ методом

```
Document createDefaultDocument();
```

и получить связанную с ним "фабрику" видов методом

```
ViewFactory getViewFactory();
```

Класс определяет курсор как текущую позицию вида, предназначенную для вставки и удаления текста методом

```
Caret createCaret();
```

Опишем курсор подробнее.

Курсор

Курсор описан интерфейсом `Caret` и реализован классом `DefaultCaret` как тонкая вертикальная черта, отмечающая позицию между символами. Текущая позиция называется *точкой* (dot). Для работы с выделенным текстом отмечается еще одна позиция — другой конец выделенного текста — называемая *меткой* (mark). Пока выделения нет, метка совпадает с точкой. При перемещении точки методом `moveDot(int)` метка остается на месте и создается выделенный участок текста. При установке точки методом `setDot(int)` метка переносится в точку и выделение отменяется.

Выделенный текст можно отметить на экране или никак не отмечать. Это регулируется методом `setSelectionVisible(boolean)`, а отслеживается логическим методом `isSelectionVisible()`.

Отследить положение точки и метки можно методами `getDot()` и `getMark()`, возвращающими их линейную позицию в части *Вид* схемы MVC. Определить двумерную текущую позицию курсора можно методом

```
Point getMagicCaretPosition();
```

а установить — методом

```
void setMagicCaretPosition(Point);
```

Во время перемещения курсора эта позиция имеет значение `null`.

Визуализация курсора осуществляется методом `paint(Graphics)`. Класс `DefaultCaret` расширяет класс `Rectangle`, определяя прямоугольник, внутри которого вычерчивается курсор.

Мерцание курсора можно задать методом `setBlinkRate(int)`, в котором указывается задержка в миллисекундах.

Курсор можно показать на экране или убрать его с экрана методом `setVisible(true)`, а проследить за этим — логическим методом `isVisible()`.

Изменение положения курсора вызывает событие `ChangeEvent`, которое обрабатывается обычным образом. Кроме того, курсор реагирует на события мыши и изменение фокуса ввода, при этом метод `paint(Graphics)` может изменить форму и цвет курсора.

Таким образом, библиотека Swing предоставляет все возможности для создания курсора любого вида.

Ограничение перемещения курсора

Подобно тому, как класс `DocumentFilter` отбирает символы перед занесением их в документ, класс `NavigationFilter` позволяет отследить перемещения курсора и выделения текста и ограничить их, например для того чтобы не допустить выделения какого-то участка текста.

Для того чтобы задать ограничения курсору, надо расширить класс `NavigationFilter`, переопределив следующие его методы:

- `void setDot(NavigationFilter.FilterBypass fb, int dot, Position.Bias bias)` — определяет положение следующей позиции точки `dot` и метки курсора;
- `void moveDot(NavigationFilter.FilterBypass fb, int dot, Position.Bias bias)` — определяет положение следующей позиции точки `dot`, оставляя метку на месте;
- `int getNextVisualPositionFrom(JTextComponent tc, int pos, Position.Bias bias, int direction, Position.Bias[] biasRet)` — вычисляет следующую позицию относительно текущей позиции `pos` с учетом направления движения курсора `direction`. Стандартная реализация вызывает метод класса `View` с тем же названием.

После расширения класса `NavigationFilter`, вызванного переопределением методов, надо установить полученный ограничитель курсора в текстовый компонент методом `setNavigationFilter(NavigationFilter)` класса `JTextComponent`.

Вот пример фильтра, ограничивающего перемещения курсора и область выделения текста первыми тридцатью символами.

```
class CursorFilter extends NavigationFilter{
  public void setDot(NavigationFilter.FilterBypass fb,
                     int dot, Position.Bias bias){
    super.setDot(fb, (dot < 30)? dot:30, bias);
     }

  public void moveDot(NavigationFilter.FilterBypass fb,
                      int dot, Position.Bias bias){
```

```
    super.moveDot(fb, (dot < 30)? dot:30, bias);
  }
}
```

После этого определяем текстовый компонент и устанавливаем в него фильтр, например:

```
JTextArea ta = new JTextArea(5, 60);
ta.setNavigationFilter(new CursorFilter());
```

Реализации редактора

Прямая реализация абстрактного класса `EditorKit` — класс `DefaultEditorKit` — кроме определения абстрактных методов вводит еще множество статических полей, задающих реакцию на нажатие специальных клавиш и перемещение курсора. Такой контроллер используется по умолчанию в компоненте `JTextComponent`.

Класс `DefaultEditorKit` расширяется классом `StyledEditorKit`. Этот класс позволяет внести в текст атрибуты, изменяющие стиль текста: шрифт, цвет, подчеркивание, курсив и другие. Такой редактор применяется по умолчанию в компоненте `JTextPane`.

От класса `StyledEditorKit` порождены два класса: `HTMLEditorKit` и `RTFEditorKit`, редактирующие тексты в форматах HTML и RTF. Возможности последнего класса пока сильно ограничены, а класс `HTMLEditorKit` собрал вокруг себя два пакета: `javax.swing.text.html` и `javax.swing.text.html.parser`. Классы этих пакетов позволяют задать нестандартную интерпретацию тегов HTML, создать и применить описание DTD документа и таблицы стилей CSS, разработать свой интерпретатор (parser) документа HTML.

Как видите, схема MVC получила в компоненте `JTextComponent` развитое детальное воплощение. Разработчик при желании может совершенно изменить стандартную реализацию текстовых компонентов. Правда, нужда в этом возникает редко. Чаще всего приходится переопределять метод `insertString()`, чтобы наложить какие-то ограничения на вводимые символы. Примеры этому будут даны далее.

Как принято в библиотеке Swing, класс `JTextComponent` дублирует многие методы своей схемы MVC. Большинство действий с текстовым компонентом можно выполнить без непосредственного обращения к *Модели*, *Виду* и *Контроллеру* компонента.

Раскладка клавиатуры

Еще одна возможность, заложенная в класс `JtextComponent`, — задать раскладку клавиатуры для ввода текста.

Для обеспечения такой возможности при нажатии или отпускании клавиши создается объект класса `KeyStroke`. Он содержит код клавиши и состояние клавиш-модификаторов <Shift>, <Alt>, <Ctrl> и <Meta>.

Класс `KeyStroke` предоставляет "фабричные" методы для создания своих объектов одним из следующих статических методов:

- `KeyStroke getKeyStroke(char keyChar)` — задается символ `keyChar`, отвечающий нажатию клавиши;

- `KeyStroke getKeyStroke(int keyCode, int modifiers)` — задается код клавиши `keyCode` константой вида `VK_*` класса `KeyEvent` и отмечается нажатие клавиш <Shift>, <Alt>, <Ctrl> и <Meta> в виде побитовой дизъюнкции констант `SHIFT_MASK`, `ALT_MASK`, `CTRL_MASK` и `META_MASK` класса `InputEvent`. Отсутствие модификаторов отмечается нулем;
- `KeyStroke getKeyStroke(Character keyChar, int modifiers)` — символ задается объектом класса `Character`;
- `KeyStroke getKeyStroke(int keyCode, int modifiers, boolean onRelease)` — последний параметр отмечает, что объект создается при нажатии (`false`) или отпускании клавиши (`true`);
- `KeyStroke getKeyStroke(String keyString)` — все параметры записаны в одну строку `keyString`. Правила записи приведены в документации.

Полученный в результате нажатия или отпускания клавиши объект класса `KeyStroke` используется затем объектом, реализующим интерфейс `Keymap`. Этот объект задает реакцию на нажатие клавиши в виде объекта, реализующего интерфейс `Action`, и хранит набор пар "клавиша — действие" типа "`KeyStroke` — `Action`". Интерфейс `Action` описан в *главе 14*.

Такой механизм задания реакции на действия с клавиатурой заменяет обычную обработку событий клавиатуры через добавление слушателя к текстовому компоненту.

Новая пара "клавиша — действие" добавляется в набор методом

```
void addActionForKeyStroke(KeyStroke key, Action a);
```

Весь набор действий в виде массива типа `Action[]` можно получить методом `getBoundActions()`, а набор объектов `KeyStroke[]` — методом `getBoundKeyStrokes()`. Отдельное действие `Action` возвращается методом `getAction(KeyStroke)`.

Объекты типа `Keymap` можно связать в цепочку, задав родительский объект методом `setResolveParent(Keymap)`. Если пара "клавиша — действие" не окажется найденной в данном объекте, то она будет отыскиваться в родительском объекте и далее по цепочке.

Библиотека Swing не реализует интерфейс `Keymap` открытым классом. Вместо этого реализация осуществляется закрытым полем класса `JTextComponent`. При создании текстового компонента это поле заполняется раскладкой по умолчанию `DEFAULT_KEYMAP`. Получить текущую раскладку можно методом `getKeymap()`, а установить новую — методом `setKeymap(Keymap)`. Добавить новую раскладку в цепочку можно статическим методом

```
addKeymap(String name, Keymap parent);
```

Печать текста документа

Несколько логических методов `print()` класса `JTextComponent` вызывают появление на экране стандартного диалогового окна печати, помогающего выбрать принтер и распечатать на нем содержимое компонента. Методы возвращают `false`, если пользователь отменил печать, щелкнув мышью по кнопке **Cancel** в диалоговом окне, и `true` в противном случае.

- `print()` — печать текста компонента без колонтитулов.
- `print(MessageFormat header, MessageFormat footer)` — добавляется печать верхнего `header` и нижнего `footer` колонтитулов, оформленных как объекты класса `MessageFormat` из пакета `java.text`.

Остальные, более сложные методы печати используют сервер печати, но это выходит за рамки нашей книги.

Поле ввода *JTextField*

Хотя внутреннее строение текстовых компонентов сложно, их обычное использование не представляет никаких трудностей.

Однострочное поле ввода создается одним из конструкторов:

- `JTextField(int columns)` — пустое поле ввода с окном, размер которого достаточен для размещения `columns` символов. В поле можно вводить сколько угодно символов, окно будет прокручиваться;
- `JTextField(String text)` — поле ввода с начальным текстом `text`;
- `JTextField(String text, int columns)` — поле ввода с начальным текстом `text` и шириной `columns` символов;
- `JTextField(Document doc, String text, int columns)` — задается модель данных `doc`.

Модель данных можно заменить методом `setDocument(Document)`. Допустимо заменять не всю модель, а только шрифт — методом `setFont(Font)`.

Методы, унаследованные от `JTextComponent`, позволяют занести текст в поле ввода методом `setText(String)`, получить весь текст методом `getText()`, часть текста методом `getText(int offset, int length)` или только выделенную часть текста методом `getSelectedText()`.

Выделенный в поле текст можно заменить другим текстом `content` методом `replaceSelection(String content)`.

По умолчанию текст в поле прижимается влево. Изменить это правило можно методом `setHorizontalAlignment(int)`, задав в нем одну из констант: `LEFT`, `CENTER`, `RIGHT`, `LEADING`, `TRAILING` класса `JTextField`.

По умолчанию текст в поле можно редактировать, но разрешается создавать поле только для чтения унаследованным методом `setEditable(false)`.

В поле можно установить новый курсор методом `setCaret(Caret)`. Допускается просто изменять цвет курсора методом `setCaretColor(Color)`. Позицию курсора можно отследить методом `getCaretPosition()`, а задать программно — методом `setCaretPosition(int)`. Переместить курсор программно, выделив таким способом участок текста, можно методом `moveCaretPosition(int)`.

Границы выделенного участка возвращают методы `getSelectionStart()` и `getSelectionEnd()`, а устанавливают — методы `setSelectionStart(int)` и `setSelectionEnd(int)`.

Цвет выделенного текста можно задать методом `setSelectedTextColor(Color)`, а цвет фона выделенного текста — методом `setSelectionColor(Color)`.

Работу с системным *буфером обмена* (clipboard) обеспечивают методы `cut()`, `copy()`, отправляющие в буфер выделенный участок текста, и метод `paste()`, вставляющий в поле содержимое буфера.

Итак, основные действия с полем ввода легко выполняются без обращения к модели данных — документу, виду или редактору. Для более сложных действий надо получить ссылку на документ методом `getDocument()`.

В листинге 12.1 приведен пример текстового поля для ввода одних только цифр.

Листинг 12.1. Поле ввода цифр

```
import java.awt.*;
import javax.swing.*;
   import javax.swing.text.*;

public class NumberText extends JFrame{
  JTextField tf = new JTextField(5);
     JLabel l = new JLabel("Вводите цифры:");

  NumberText(){
    super("text");
    Container c = getContentPane(); // Необязательно, начиная с версии 5.0.
       c.setLayout(new FlowLayout());

    // Вставляем фильтр вводимых символов
    ((PlainDocument)tf.getDocument()).
           setDocumentFilter(new NumberFilter());

    // Текст будет выделяться только красным цветом
    tf.setSelectedTextColor(Color.red);
    // При выделении текста фон останется белым
    tf.setSelectionColor(Color.white);
    // Курсор будет красным
    tf.setCaretColor(Color.red);
       l.setLabelFor(tf);

    c.add(l);
       c.add(tf);

    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
     }

  public static void main(String[] args){
    new NumberText();
  }
         // Фильтр вводимых данных
  class NumberFilter extends DocumentFilter{
         // Переопределяем только один метод
    public void insertString(FilterBypass fb, int pos,
                             String text, AttributeSet attr)
```

```
            throws BadLocationException{
      try{
          Integer.parseInt(text); // Введена цифра?
      }catch(Exception e){
        // Если не цифра, то символ не вводим
        super.insertString(fb, 0, "", attr);
        return;
      }
      // Если введена цифра, то заносим ее в поле
      super.insertString(fb, pos, text, attr);
      }
    }
}
```

Поле ввода пароля *JPasswordField*

Класс `JPasswordField` непосредственно расширяет класс `JTextField`, значит, к нему относится все сказанное ранее. Одно отличие заключается в том, что в этом поле вместо вводимых символов повторяется один символ, по умолчанию — звездочка. Звездочку можно заменить другим символом с помощью метода `setEchoChar(char)`.

Второе отличие заключается в том, что вместо метода `getText()` для получения текста из поля пароля используется метод `getPassword()`, возвращающий массив символов типа `char[]`, а не строку.

Редактор объектов *JFormattedTextField*

Еще одно расширение класса `JTextField` — класс `JFormattedTextField` — предназначено для работы с объектами, содержащими символы, такими как `Date`, `Number`. Впрочем, конструктор класса `JFormattedTextField(Object)` и метод `setValue(Object)` позволяют включить в редактор любой объект, при этом в окно редактирования будет выведен результат преобразования этого объекта в текстовую строку.

Типичное применение редактора:

```
JFormattedTextField ftf = new JFormattedTextField(new Date());
ftf.addActionListener(this);
// . . . . . .
// Редактируем дату... Потом нажимаем клавишу <Enter>
// . . . . . .
public void actionPerformed(ActionEvent e){
   newDate = (Date)ftf.getValue();
}
```

Метод `getValue()` возвращает объект типа `Object`, полученный в результате обратного преобразования отредактированной в окне строки в первоначальный объект.

Преобразованием объекта в строку и обратно занимается вложенный в JFormattedTextField абстрактный класс AbstractFormatter. Для этого в нем есть методы valueToString(Object) и stringToValue(String). Эти методы оставлены абстрактными. После их определения следует установить полученный "преобразователь" в редактор методом install(JFormattedTextField) класса AbstractFormatter или воспользоваться конструктором

```
JFormattedTextField(JFormattedTextField.AbstractFormatter);
```

или методом setFormatter(AbstractFormatter) класса JFormattedTextField.

Кроме преобразования объекта, AbstractFormatter определяет еще экземпляр класса DocumentFilter и экземпляр класса NavigationFilter, которые можно получить методами getDocumentFilter() и getNavigationFilter(). Это придает редактору объектов вторую роль — проверять и фильтровать вводимый в поле текст, допуская только определенные символы.

В библиотеке Swing есть реализация класса AbstractFormatter — его расширение DefaultFormatter. Для преобразования объекта в строку метод valueToString(Object) в классе DefaultFormatter определен очень просто: он использует метод toString() этого объекта. Метод stringToValue(String) обратного преобразования строки в объект использует конструктор класса с единственным параметром типа String. Если такого конструктора нет, то возвращается строка.

Класс DefaultFormatter применяется редко, он употребляется как базовый класс для полезных расширений. Несколько готовых расширений есть в библиотеке Swing.

От класса DefaultFormatter порождены два класса. Класс MaskFormatter определяет маску ввода, накладывающую ограничения на вводимые значения подобно классу Format и его подклассам из пакета java.text. Например, класс

```
MaskFormatter mf = new MaskFormatter("###.##");
```

накладывает маску действительных чисел с двумя знаками после десятичной точки.

Второй класс InternationalFormatter прямо использует класс Format для своих преобразований, который задается в конструкторе

```
InternationalFormatter(Format);
```

Он расширен двумя классами: класс NumberFormatter использует класс DecimalFormat для фильтрации чисел, а класс DateFormatter — класс DateFormat для отбора даты и времени.

Для того чтобы облегчить создание классов-преобразователей, в классе JFormattedTextField есть вложенный абстрактный класс AbstractFormatterFactory, расширенный классом DefaultFormatterFactory. Этот класс содержит несколько методов get*Xxx*Formatter(), возвращающих тот или иной объект-преобразователь.

Область ввода *JTextArea*

Класс JTextArea представляет многострочную область ввода с "плоским" текстом, в котором не меняются атрибуты шрифта. Для редактирования сложного текста удобнее использовать JTextEditor или JTextPane.

В область ввода `JTextArea` не заложена возможность прокрутки большого текста. Если в этом есть необходимость, то область надо поместить в контейнер `JScrollPane`:

```
JTextArea ta = new JTextArea(5, 50);
JScrollPane sp = new JScrollPane(ta);
container.add(sp);
```

При этом следует задать размеры области — число строк и столбцов — как это сделано выше, или предпочтительный размер `JScrollPane`.

По умолчанию слово, не поместившееся в видимой части строки, не переносится целиком на следующую строку. Эту возможность надо включить методом `setWrapStyleWord(true)`.

Аналогично, по умолчанию весь текст в области показывается в виде одной строки, выходящей за пределы окна. Если область ввода помещена в контейнер `JScrollPane`, то появляется горизонтальная линейка прокрутки. Чтобы строки не выходили за пределы окна, надо включить перенос строк методом `setLineWrap(true)`.

Текст в область ввода можно не только занести методом `setText(String)`, но и добавить в конец уже имеющегося текста методом `append(String)` и вставить программно в определенную позицию методом `insert(String, int)`.

Область ввода допускает изменение размера табуляции методом `setTabSize(int)`.

В остальном область ввода ведет себя как поле ввода класса `JTextField`.

Текстовый редактор *JEditorPane*

Текстовый редактор класса `JEditorPane` по умолчанию распознает три MIME-типа текста: text/plain, text/html и text/rtf, вызывая для изменения редакторы `DefaultEditorKit`, `HTMLEditorKit` или `RTFEditorKit` соответственно. Для того чтобы учесть MIME-тип текста, применяется конструктор

```
JEditorPane(String type, String text);
```

Он вызывает метод `setContentType(type)`, задающий MIME-тип текста, а затем — метод `setText(text)`. Например:

```
JEditorPane ep= new JEditorPane("text/html;Content-Type=windows-1251",
                                "<html>Документ HTML");
JScrollPane sp = new JScrollPane(ep);
container.add(sp);
```

Метод `setText(String)` не меняет выбранный редактор. Поэтому MIME-тип его параметра должен соответствовать имеющемуся редактору. Впрочем, можно установить новый редактор методом `setEditor(EditorKit)`. Следует учитывать, что при этом сменится и документ!

Для определенного MIME-типа методом

```
setEditorKitForContentType(String type, EditorKit editor);
```

можно задать редактор, который будет вызываться для обработки текста этого типа.

Еще два конструктора позволяют занести начальный текст в редактор прямо с адреса, заданного в форме URL. Адрес задается объектом класса `URL` или одной из строк:

```
JEditorPane(URL url);
JEditorPane(String url);
```

Информацию с адреса URL можно занести в редактор в любое время методом `setPage(URL)` или `setPage(String url)`. Учтите, что при изменении MIME-типа текста поменяется тип документа и редактора. Если занесена страница HTML, то будут установлены модель данных класса `HTMLDocument` и редактор класса `HTMLEditorKit`.

Самый общий способ загрузки текста — занести текст из входного потока методом `read(InputStream in, Object obj)`. Если установлен редактор `HTMLEditorKit` и параметр `obj` имеет тип `HTMLDocument`, то текст HTML будет проинтерпретирован. В других случаях будет обрабатываться "плоский" текст.

Текстовый редактор показывает и изображения, определенные HTML-тегом `<IMG>`. Но для работы с изображениями и компонентами более удобен редактор класса `JTextPane`.

Редактор *JTextPane*

Класс `JTextPane` непосредственно расширяет класс `JEditorPane` и наследует все его свойства. Кроме этого он позволяет работать со структурированным текстом с различными стилями, поскольку по умолчанию наделен моделью данных типа `DefaultStyledDocument`.

В редактор легко добавить новые стили методом

```
addStyle(String name, Style parent);
```

Можно задать множество атрибутов для отдельных символов, которые выделены в тексте или которые будут вводиться в текст, методом

```
setCharacterAttributes(AttributeSet attr, boolean replace);
```

Можно задать атрибуты сразу целому элементу, который выделен или в котором находится текущая позиция, методом

```
setParagraphAttributes(AttributeSet attr, boolean replace);
```

Если второй аргумент этих методов равен `true`, то существующие атрибуты будут заменены новыми.

Можно задать и новую модель данных методом

```
setStyledDocument(StyledDocument);
```

Редактор позволяет вставить в текущую позицию текста изображение методом `insertIcon(Icon)`. Если часть текста была выделена, то изображение будет вставлено вместо выделенного текста.

Более того, в текущую позицию текста или вместо выделенного текста можно вставить любой компонент методом `insertComponent(Component)`. Изображение и компонент хранятся в модели данных как атрибут одного символа.

Вопросы для самопроверки

1. Как используется модель MVC в текстовых компонентах?
2. В чем отличие текстовых компонентов Swing от аналогичных компонентов AWT?
3. Можно ли в текстовых компонентах Swing менять шрифт?
4. Можно ли в текстовых компонентах Swing использовать разные шрифты в одной строке?
5. В каких случаях удобно использовать готовые текстовые редакторы Swing?
6. Можно ли средствами Swing написать свой текстовый редактор Swing?
7. Можно ли в текстовых редакторах Swing использовать разные шрифты в одном документе?



Глава 13

Таблицы

В графическом интерфейсе пользователя очень часто встречаются таблицы разного вида. Некоторые из них только показывают список табличных данных, другие позволяют заполнять их информацией и даже выполнять простейшие расчеты.

В подавляющем большинстве информационных систем данные хранятся в таблицах реляционной базы данных. Результат запроса к базе данных тоже представляется в виде таблицы. Очень часто этот результат необходимо отразить на экране или в отчете также в виде таблицы, снабженной заголовком, комментариями, оформленной цветом и различными шрифтами.

В графической библиотеке Swing для решения таких задач имеются классы `JTable`, `JTableHeader`, `TableColumn` и сопутствующие им классы, составляющие пакет `javax.swing.table`. Эти классы позволяют создать не только статичные таблицы, отражающие результаты запроса, но и редактируемые таблицы, изменяющие данные, и даже полномасштабные электронные таблицы. Рассмотрим последовательно эти классы.

Класс *JTable*

Класс `JTable` очень велик. Он определяет множество методов построения таблиц и работы с ними. С ним связано несколько классов-делегатов. В своей работе класс `JTable` использует класс `DefaultTableModel` для хранения данных, класс `JTableHeader` для построения заголовков столбцов, класс `TableCellEditor` для редактирования ячеек, класс `TableCellRenderer` для отображения ячеек на экране, класс `TableColumn` для сбора информации о свойствах каждого столбца. Для изменения свойств таблицы эти классы нужно заменить их расширениями или реализовать соответствующие интерфейсы.

Таблицы сконструированы по схеме "Model-View-Controller". Для хранения содержимого таблицы и ее характеристик имеются три модели данных: модель ячеек таблицы, описанная интерфейсом `TableModel`, модель столбцов таблицы, описанная интерфейсом `TableColumnModel`, и модель выделения ячеек таблицы, в качестве которой взята модель выделения списков `ListSelectionModel` (о ней мы уже говорили в *главе 11*).

Для создания таблицы предлагается несколько конструкторов.

- Конструктор по умолчанию `JTable()` создает пустую таблицу без строк и столбцов с пустыми моделями данных.
- Конструктор `JTable(int rows, int columns)` формирует пустую редактируемую таблицу с заданным числом `rows` строк и `columns` столбцов с моделью данных по умол-

чанию, которая предполагает хранить в таблице объекты типа `Object` в виде вектора типа `Vector`, состоящего из векторов. В нее можно вводить данные прямо с клавиатуры.

- Конструктор `JTable(Object[][] data, Object[] colNames)` создает таблицу, заполненную объектами `data`. Параметр `colNames` содержит имена столбцов таблицы. Все строки массива `data` должны содержать столько же элементов, сколько существует в массиве `colNames`. Пример создания таблицы этим конструктором приведен в листинге 13.1.
- Конструктор `JTable(Vector data, Vector colNames)` делает то же самое, но параметры заданы векторами. Пример его использования представлен в листинге 13.3.

  Надо заметить, что заголовки столбцов автоматически появляются на экране, только если таблица заключена в панель `JScrollPane`. Если при этом у столбцов не заданы имена, то они помечаются буквами A, B, C и т. д., как принято в электронных таблицах.

Остальные конструкторы определяют таблицу с заранее заданными моделями данных.

- Конструктор `JTable(TableModel)` использует заданную параметром модель ячеек таблицы. Модель столбцов и модель выделения данных определяются по умолчанию.
- Конструктор `JTable(TableModel, TableColumnModel)` оставляет модель выделения данных по умолчанию.
- Конструктор `JTable(TableModel, TableColumnModel, ListSelectionModel)` задает все три модели данных.

Листинг 13.1 показывает простой пример определения таблицы с помощью массива объектов, в данном случае — строк.

Листинг 13.1. Простое определение таблицы

```
import java.awt.*;
   import javax.swing.*;

   public class SimpTable extends JFrame{

  SimpTable(){
    super("  My Table");
    Container c = getContentPane(); // Излишне для JDK 5.0 и выше.
       c.setLayout(new FlowLayout());

    String[][] data = {{"-27", "32"}, {"-45", "55"}};
       String[] colNames = {"Вчера", "Сегодня"};

       JTable t1 = new JTable(data, colNames);

       c.add(new JScrollPane(t1));

    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
     }
```

```
  public static void main(String[] args){
    new SimpTable();
  }
}
```

Таблицу можно заполнить методом

```
setValueAt(Object data, int row, int column);
```

Он заменяет старое содержимое ячейки `(row, column)` *Вида* объектом `value`. Следует учесть, что в этом методе параметр `column` указывает номер столбца в *Виде*, но не в *Модели* данных. При выводе содержимого таблицы из *Модели* данных в *Вид* порядок столбцов можно изменить, да и необязательно выводить все столбцы таблицы.

В таблице можно задать или заменить *Модель* ячеек таблицы методом `setModel(TableModel)` и модель столбцов таблицы с помощью метода `setColumnModel(TableColumnModel)`. Рассмотрим их подробнее.

Модель данных таблицы

Таблица класса `JTable` пользуется тремя моделями данных для хранения своих элементов. Две модели, описанные интерфейсами `TableModel` и `TableColumnModel`, специфичны для таблиц, третья модель — `ListSelectionModel` — заимствована у списков `JList`. Она уже рассматривалась нами в *главе 11*.

Модель ячеек таблицы

Модель хранения содержимого ячеек таблицы описана интерфейсом `TableModel`, который частично реализован абстрактным классом `AbstractTableModel` и полностью реализован его подклассом `DefaultTableModel`.

Эта модель предполагает, что в ячейках таблицы могут храниться объекты любого типа и в разных ячейках даже одного столбца могут храниться объекты разных типов.

Строки и столбцы пронумерованы, начиная от нуля. Общий суперкласс всех ячеек столбца с индексом `ind` можно получить методом `getColumnClass(int ind)`. Текущее число строк в таблице можно узнать методом `getRowCount()`, число столбцов — методом `getColumnCount()`. У столбца может быть имя, получить которое можно методом `getColumnName(int)`, возвращающим строку класса `String`.

Содержимое ячейки таблицы можно получить из модели данных в виде объекта класса `Object` методом `getValueAt(int rowInd, int colInd)`, а установить в модель, если ячейка редактируема, — методом `setValueAt(Object data, int rowInd, int colInd)`.

Проверить, редактируема ячейка или нет, можно логическим методом `isCellEditable(int rowInd, int colInd)`.

Легко создать свою модель ячеек таблицы, расширив класс `AbstractTableModel`. При этом необходимо определить три метода:

```
int getRowCount();
int getColumnCount();
Object getValueAt(int, int);
```

Остальные методы переопределяются по усмотрению разработчика. Но если этого не сделать, то получится нередактируемая таблица только для чтения, поскольку метод `isEditable()` в классе `AbstractTableModel` возвращает `false`, а метод `setValueAt()` пуст. Для того чтобы таблица стала редактируемой, надо переопределить и эти два метода.

В листинге 13.2 приведен пример расширения класса `AbstractTableModel` — модель ячеек таблицы, запрещающая редактировать первый столбец таблицы.

Листинг 13.2. Модель запрещает редактировать первый столбец

```
import java.awt.*;
import javax.swing.*;
   import javax.swing.table.*;

   public class SimpTable extends JFrame{

  SimpTable(){
    super("  Таблица с неизменяемым первым столбцом");
    Container c = getContentPane(); // Излишне для JDK 5.0 и выше.
       c.setLayout(new FlowLayout());

       JTable t1 = new JTable(new FirstColumnTableModel());

       c.add(new JScrollPane(t1));

    setSize(400, 400);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
     }

  public static void main(String[] args){
    new SimpTable();
  }
   }

class FirstColumnTableModel extends AbstractTableModel{
  protected Object[][] data = {
    {"Текст", Color.black, new Boolean(true)},
    {"Фон", new Color(130, 56, 187), new Boolean(true)},
    {"Рамка", new Color(200, 45, 125), new Boolean(false)},
  };
     protected String[] colNames = {"Элемент", "Цвет", "Выбрать"};

     public FirstColumnTableModel(){ super(); }

  // Определяем обязательные методы
  public int getRowCount(){ return data.length; }
  public int getColumnCount(){ return data[0].length; }
     public Object getValueAt(int row, int col){ return data[row][col]; }
```

```
  // Запрещаем заполнять ячейки первого столбца
  public void setValueAt(Object value, int row, int col){
    if (col != 0) data[row][col] = value;
     }

  // Сообщаем, что первый столбец нельзя редактировать
  public boolean isCellEditable(int row, int col){
    return col != 0;
     }

  // Для визуализации содержимого ячеек графическим
  // компонентом определяем класс ячеек столбца
  public Class getColumnClass(int col){
    return data[0][col].getClass();
     }

  // Для показа имен в строке заголовков
  public String getColumnName(int col){
    return colNames[col];
  }
}
```

Как видно из рис. 13.1, объект класса `Boolean` показан в таблице компонентом `JCheckBox`. Это результат действия метода `getColumnClass()`. На рис. 13.2 тот же объект изображен в стандартном виде. Но объект класса `Color` по-прежнему показан строкой — результатом действия метода `toString()`.

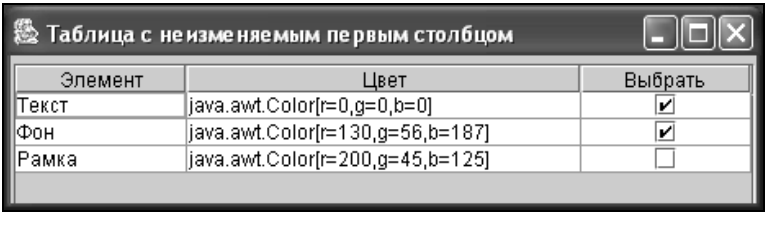


Рис. 13.1. Таблица со сложными объектами

Класс `DefaultTableModel` кроме реализации методов интерфейса `TableModel` добавляет несколько конструкторов и полезных методов работы с моделью данных. Он создает редактируемую модель ячеек таблицы. Для хранения данных он создает вектор строк класса `Vector`, хранящий строки опять-таки в виде вектора. Получившийся вектор векторов хранится в *защищенном* (protected) поле `dataVector`.

Конструктор по умолчанию `DefaultTableModel()` создает объект с нулевым количеством строк и столбцов.

Конструктор `DefaultTableModel(int rowCount, int colCount)` создает модель данных с заданным числом строк и столбцов. В ячейках этой модели — пустые ссылки `null`.

Конструкторы

```
DefaultTableModel(Vector data, Vector colNames);
DefaultTableModel(Object[][] data, Object[] colNames);
```

сразу же заполняют модель данными.

Еще два конструктора:

```
DefaultTableModel(Vector colNames, int rowCount);
DefaultTableModel(Object[] colNames, int rowNames);
```

задают модель с поименованными столбцами и пустыми строками.

Методы класса `DefaultTableModel`, как и положено модели данных, добавляют в конец модели строки методами `addRow()` и столбцы методами `addColumn()`, вставляют строку в указанное место методами `insertRow()`. Аргументы этих методов могут быть типа `Object` или `Vector`.

Модель позволяет заполнить ячейки объектами методом

```
setValueAt(Object value, int row, int column);
```

Многочисленные методы `get`*`Xxx`*`()` предоставляют сведения о модели.

В модели можно переставить строки методом

```
moveRow(int start, int end, int to);
```

Первые два аргумента — `start` и `end` — задают диапазон переставляемых строк, последний аргумент `to` — новый индекс строки, имевшей до перестановки индекс `start`.

Наконец, методы `setDataVector()` позволяют вообще заменить все содержимое модели, т. е. содержимое поля `dataVector`.

При создании модели ячеек таблицы часто бывает удобнее расширить класс `DefaultTableModel`, а не абстрактный класс `AbstractTableModel`. В листинге 13.3 показан пример такого расширения — модель заполняется сведениями о файлах, полученными из каталога, имя которого указывается в командной строке.

Листинг 13.3. Заполнение модели ячеек таблицы данными из каталога

```
import javax.swing.*;
import javax.swing.table.*;
import java.io.*;
   import java.util.*;

   public class FileTable extends JFrame{

  public FileTable(File dir){
       super("  Таблица файлов");

       JTable table = new JTable(new FileTableModel(dir));

    getContentPane().add(new JScrollPane(table));
    setSize(600, 400);
    setVisible(true);
```

```
    setDefaultCloseOperation(EXIT_ON_CLOSE);
     }

  public static void main(String[] args){
    File dir = args.length > 0 ? new File(args[0]) :
             new File(System.getProperty("user.home"));

    new FileTable(dir);
  }
   }

   class FileTableModel extends DefaultTableModel{

  protected File dir;
     protected String[] fName;

  protected String[] colName = new String[]{
    "Имя", "Размер", "Дата и время",
    "Каталог", "Для чтения", "Для записи"
     };

  protected Class[] colClass = new Class[]{
    String.class, Long.class, Date.class,
    Boolean.class, Boolean.class, Boolean.class
     };

  public FileTableModel(File dir){
    super(dir.list().length, 6);
    this.dir = dir;
    fName = dir.list();
     }

  public String getColumnName(int col){ return colName[col]; }
     public Class getColumnClass(int col){ return colClass[col]; }

  public Object getValueAt(int row, int col){
    File f = new File(dir, fName[row]);
    switch(col){
      case 0: return fName[row];
      case 1: return new Long(f.length());
      case 2: return new Date(f.lastModified());
      case 3: return f.isDirectory() ?
              Boolean.TRUE : Boolean.FALSE;
      case 4: return f.canRead() ?
              Boolean.TRUE : Boolean.FALSE;
      case 5: return f.canWrite() ?
              Boolean.TRUE : Boolean.FALSE;
      default: return null;
    }
  }
}
```

При всяком изменении модели или ее содержимого в ней происходит событие класса `TableModelEvent`. Этот класс различает три типа событий: `INSERT`, `DELETE` и `UPDATE` и возвращает тип события методом `getType()`. Кроме того, класс отслеживает диапазон строк, в которых произошло событие, и столбец. Эти сведения можно получить методами `getFirstRow()`, `getLastRow()` и `getColumn()`. Если последний из этих методов вернул константу `ALL_COLUMNS`, то это означает, что событие затронуло все столбцы. Слушатель данного события может быть присоединен к модели методом `addTableModelListener(TableModelEvent)`.

Есть еще две комбинации типов события `TableModelEvent`. Сочетание `UPDATE`, `ALL_COLUMNS` и константы `MAX_VALUE` в качестве значения метода `getLastRow()` говорит о том, что была изменена вся *Модель* и *Вид* должен переписать на экране всю таблицу. Сочетание `UPDATE`, `ALL_COLUMNS` и `HEADER_ROW` — результат применения метода `getFirstRow()`, — сообщает о том, что структура таблицы изменена — добавлен или удален столбец, *Виду* надо изменить структуру столбцов и перечертить таблицу на экране. Впрочем, это изменение отслеживается событием `TableColumnModelEvent`.

После определения модели ячеек ее надо установить в таблицу методом `setModel(TableModel)` или воспользоваться конструктором таблицы, как показано в листинге 13.2.

Свойства столбца таблицы *TableColumn*

Класс `TableColumn` хранит информацию о столбце таблицы: минимальную, максимальную и текущую ширину столбца, возможность изменения ширины, заголовок, индекс столбца в модели данных, ссылки на объекты классов `TableCellEditor` и `TableCellRenderer`. Для доступа к этой информации в классе есть методы `get`*`Xxx`*`()`/`set`*`Xxx`*`()`.

Особенно часто приходится использовать методы `setMinWidth(int)`, `setMaxWidth(int)` и `setPreferredWidth(int)`, поскольку ширина столбцов и всей таблицы, задаваемая по умолчанию, редко удовлетворяет разработчика.

Модель столбцов таблицы

Модель хранения столбцов таблицы описана интерфейсом `TableColumnModel`, который реализован классом `DefaultTableColumnModel`. Эта модель собирает сведения обо всех столбцах таблицы в виде вектора экземпляров класса `TableColumn`. Кроме того, запоминается ширина промежутков между колонками и общая ширина таблицы. Модель хранит объект класса `ListSelectionModel`, определяющий правила выделения столбцов и содержащий сведения о выделенных столбцах таблицы.

Порядок следования столбцов в модели может не совпадать с их порядком в модели ячеек таблицы. Модель может переставлять столбцы методом `moveColumn(int oldInd, int newInd)`, эти перестановки не меняют модель ячеек таблицы.

При каждом изменении модели: добавлении или удалении столбцов, их перестановках, выделении столбцов, происходит событие класса `TableColumnModelEvent`, которое модель может отследить методом

```
addColumnModelListener(ColumnModelListener);
```

Заголовки столбцов таблицы *JTableHeader*

В начале таблицы можно вывести строку с именами столбцов. Она позволяет изменять ширину столбцов с помощью мыши в заданных для столбцов пределах.

Строка заголовков появляется автоматически, если таблица помещена в панель класса `JScrollPane`, даже если имена столбцов не были заданы. В таком случае в строку заголовков выводятся заглавные латинские буквы A, B, C и т. д., как определено в модели ячеек таблицы класса `DefaultTableModel`.

Строка заголовков — это объект класса `JTableHeader` из пакета `javax.swing.table`. Данный объект хранит заголовок в модели столбцов типа `TableColumnModel`.

Экземпляр класса с моделью столбцов таблицы по умолчанию создается конструктором `JTableHeader()`. Второй конструктор, `JTableHeader(TableColumnModel)`, создает объект с заданной моделью столбцов.

После создания строки заголовков ее надо связать с таблицей методом `setTable(JTable)`.

Второй способ создания строки заголовков — получить ее экземпляр методом `getTableHeader()` класса `JTable`.

Полученная строка заголовков — самостоятельный объект, который надо отдельно выводить на экран. Например, в листинге 13.4 заголовок таблицы выводится на "север", а сама таблица — в "центр" размещения `BorderLayout`. Таким способом строку заголовков можно вывести последней строкой таблицы, поместив ее на "юг" и превратив в итоговую строку таблицы. О классе `BorderLayout` и вообще о менеджерах размещения компонентов речь пойдет в следующей главе.

Листинг 13.4. Отдельный вывод заголовка таблицы

```
import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;
public class HeadTable extends JFrame{
  HeadTable(){
    super("  Сотрудники");
    Container c = getContentPane(); // Излишне для JDK 5.0 и выше.
    Vector data = new Vector();
     Vector row = new Vector();
     row.addElement("Иванов");
     row.addElement(new Integer(1970));
     row.addElement(new Boolean(false));
    data.addElement(r);
     row = new Vector();
     row.addElement("Петров");
     row.addElement(new Integer(1980));
     row.addElement(new Boolean(true));
       data.addElement(r);
```

```
    Vector col = new Vector();
     col.addElement("Фамилия");
     col.addElement("Год рождения");
     col.addElement("Семейное положение");
    JTable t2 = new JTable(data, col);
    JTableHeader th = t2.getTableHeader();
    c.add(th, BorderLayout.SOUTH);
    c.add(t2, BorderLayout.CENTER);
    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
  }
  public static void main(String[] args){
    new HeadTable();
  }
}
```

На рис. 13.2 показан вывод программы, записанной в листинге 13.4. Как видно из рисунка, объект класса `Boolean` отображен строкой — результатом действия метода `toString()`.

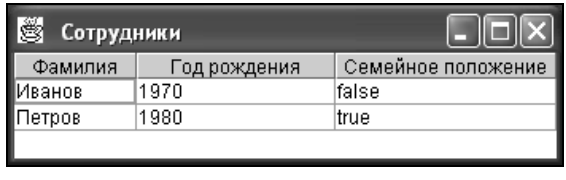


Рис. 13.2. Отдельный вывод заголовка таблицы

Границы между заголовками столбцов служат для изменения ширины столбца с помощью мыши. Изменение может быть сделано в пределах от минимальной до максимальной ширины. Поведение остальных столбцов при изменении ширины одного из них регулируется методом `setAutoResizeMode(int)` класса `JTable`. Аргумент этого метода — одна из следующих констант:

- `AUTO_RESIZE_OFF` — не изменять ширину остальных столбцов;
- `AUTO_RESIZE_NEXT_COLUMN` — изменить ширину следующего столбца;
- `AUTO_RESIZE_SUBSEQUENT_COLUMNS` — изменить пропорционально ширину следующих столбцов (по умолчанию);
- `AUTO_RESIZE_LAST_COLUMN` — изменить ширину последнего столбца;
- `AUTO_RESIZE_ALL_COLUMNS` — изменить пропорционально ширину всех столбцов.

Модель выделения ячеек

Правила выделения ячеек таблицы регулируются интерфейсом `ListSelectionModel`. В классе `JTable` по умолчанию используется реализация `DefaultListSelectionModel` этого интерфейса. Она была описана в *главе 11* при рассмотрении класса `JList`. Эта модель

задает три режима выделения: выделение отдельного элемента SINGLE_SELECTION, выделение смежных элементов SINGLE_INTERVAL_SELECTION, выделение нескольких участков смежных элементов MULTIPLE_INTERVAL_SELECTION.

Режим выделения можно установить непосредственно методами модели выделения или методом setSelectionMode(int) класса JTable, в котором надо задать одну из трех указанных выше констант. Первый способ позволяет установить разные режимы выделения для строк и столбцов таблицы. Для строк режим выделения можно задать просто методом setSelectionModel(ListSelectionModel) класса JTable. Чтобы задать режим выделения столбцов, надо предварительно получить ссылку на объект типа TableColumnModel, а потом установить новую модель. Все это можно сделать так:

```
table.getColumnModel().setSelectionModel(new SomeSelectionModel());
```

Цвет текста и фона выделенных ячеек можно задать методами setSelectionForeground(Color) и setSelectionBackground(Color).

Остальные свойства выделения можно задать и узнать методами модели выделения, получив ссылку на нее методом getSelectionModel() класса JTable.

Визуализация ячеек таблицы

Непосредственным выводом содержимого ячеек на экран — его визуализацией — занимается еще один делегат класса JTable, описанный интерфейсом TableCellRenderer.

Интерфейс TableCellRenderer описывает всего один метод:

```
Component getTableCellRendererComponent(
   JTable table, Object value,
   boolean isSelected, boolean hasFocus,
   int row, int col);
```

Как видно из описания, для каждой ячейки с индексами (row, col) можно задать свой способ визуализации. Этот способ может также меняться в зависимости от содержимого value ячейки, от того, выделена ли ячейка isSelected, имеет ли она фокус ввода hasFocus, и даже от того, какая таблица table использует этот метод. По этим данным метод должен сформировать компонент, содержащий значение value, и вернуть его. Обычно возвращается объект класса, определяющего этот метод, т. е. this. Затем полученный объект рисует себя на экране своим методом paint(). Поэтому удобно реализовать интерфейс каким-нибудь графическим компонентом, имеющим метод paint().

Все это похоже на визуализацию элементов списка JList, описанную в *главе 11*.

В библиотеке Swing интерфейс TableCellRenderer реализован классом DefaultTableCellRenderer, расширяющим класс JLabel. Но класс DefaultTableCellRenderer не реализует даже все возможности класса JLabel, например не отображаются изображения типа Icon. Используется только метод setText(String) класса JLabel в таком виде:

```
protected void setValue(Object value){
  setText((value == null) ? "" : value.toString());
}
```

Это означает, что, хотя ячейка таблицы может содержать любой объект, на экране появляется только строка, полученная методом toString() этого объекта.

Для того чтобы показать графические объекты, хранящиеся в ячейках таблицы, в графическом виде, чаще всего достаточно переопределить метод `setValue(Object)` класса `DefaultTableCellRenderer`. Например, в программе листинга 13.2 объекты класса `Color` выводятся текстовой строкой, как видно на рис. 13.1. Чтобы в ячейке показать цвет, надо расширить класс `DefaultTableCellRenderer`, переопределив его метод `setValue()`:

```
class ColorRenderer extends DefaultTableCellRenderer{
  public void setValue(Object value){
    setBackground((Color)value);
  }
}
```

Для того чтобы этот способ визуализации применялся только к объектам класса `Color`, следует обратиться к методу `setDefaultRenderer(Class, TableCellRenderer)` класса `JTable`. В примере листинга 13.2 в конструктор класса `SimpTable` надо вставить строку

```
t1.setDefaultRenderer(Color.class, new ColorRenderer());
```

Этот метод можно применить несколько раз для разных классов, задав таким путем свой класс-рисовальщик для объекта каждого класса. Например, можно выводить на экран изображения, хранящиеся в ячейках таблицы, определив класс:

```
class IconRenderer extends DefaultTableCellRenderer{
    public void setValue(Object value){ setIcon((Icon)value); }
}
```

и добавив строку

```
t1.setDefaultRenderer(Icon.class, new IconRenderer());
```

Возможности класса `JLabel`, который фактически рисует на экране содержимое ячеек таблицы в классе `DefaultTableCellRenderer`, ограничены. Для более сложного вывода на экран следует непосредственно реализовать интерфейс `TableCellRenderer`. Листинг 13.5 показывает, как можно реализовать его для вывода многострочных ячеек, воспользовавшись текстовой областью класса `JTextArea`, описанного в *главе 12*.

Листинг 13.5. Несколько строк в одной ячейке таблицы

```
import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.border.*;
import java.util.*;
public class MultiLineTable extends JFrame{
  MultiLineTable(int lineCount){
       super("  Таблица с многострочными ячейками");

    // Расширяем модель ячеек, переопределяя метод,
    // возвращающий класс содержимого столбца
    DefaultTableModel tm = new DefaultTableModel(){
      public Class getColumnClass(int col){
```

```
        return getValueAt(0, col).getClass();
      }
       };

    // Заносим в модель ячеек данные
   tm.setDataVector(
    new Object[][]{
      {"Имя\nФамилия","Иван\nПетров","Петр\nИванов"},
      {"Отдел\nДолжность","Сбыт\nВодитель","Сбыт\nЭкспедитор"}
    },
    new Object[]{"Данные","1","2"}
     );

   // Создаем таблицу с новой моделью ячеек
      JTable t = new JTable(tm);

   // Изменяем высоту ячеек на экране, чтобы поместились
   // все строки содержимого ячейки
      t.setRowHeight( t.getRowHeight() * lineCount);

   // Устанавливаем новый класс-рисовальщик
      t.setDefaultRenderer(String.class, new MultiLineCellRenderer());

      getContentPane().add(new JScrollPane(t));

   setSize(400, 400);
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   setVisible(true);
     }

  public static void main(String[] args){
    new MultiLineTable(2);
  }
}
         // Класс-рисовальщик
class MultiLineCellRenderer extends JTextArea
          implements TableCellRenderer{

  public MultiLineCellRenderer(){
    setLineWrap(true);
    setWrapStyleWord(true);
    setOpaque(true);
     }

  public Component getTableCellRendererComponent(
       JTable table, Object value,
       boolean isSelected, boolean hasFocus,
       int row, int col){
    if (isSelected){
      setForeground(table.getSelectionForeground());
```

```
      setBackground(table.getSelectionBackground());
    }else{
      setForeground(table.getForeground());
      setBackground(table.getBackground());
       }

    if (hasFocus){
      setBorder(UIManager.getBorder("Table.focusCellHighlightBorder"));
      if (table.isCellEditable(row, col)){
         setForeground(UIManager.getColor("Table.focusCellForeground"));
         setBackground( UIManager.getColor("Table.focusCellBackground"));
      }
       }else setBorder(new EmptyBorder(1, 2, 1, 2));

   setFont(table.getFont());
   setText((value == null) ? "" : value.toString());
   return this;
  }
}
```

Часто требуется поместить несколько строк в заголовки столбцов таблицы. Для этого можно воспользоваться программой листинга 13.5, написав новый метод `getTableCellRendererComponent()`:

```
public Component getTableCellRendererComponent(
       JTable table, Object value,
       boolean isSelected, boolean hasFocus,
          int row, int col){

   if (table != null){
      JTableHeader header = table.getTableHeader();
      if (header != null){
         setForeground(header.getForeground());
         setBackground(header.getBackground());
         setFont(header.getFont());
      }
   }
   setText((value == null) ? "" : value.toString());
   setBorder(UIManager.getBorder("TableHeader.cellBorder"));
   return this;
}
```

Затем надо установить экземпляр измененного класса (назовем его `MultiLineHeader`) в заголовок каждого столбца, взяв заголовки из модели заголовков таблицы:

```
MultiLineHeader mlh = new MultiLineHeader();
Enumeration e = t.getColumnModel().getColumns();
while (e.hasMoreElements())
   ((TableColumn)e.nextElement()).setHeaderRenderer(mlh);
```

Редактор ячеек таблицы

По умолчанию таблица создается редактируемой. Это означает, что содержимое ее ячеек можно изменять вводом новых значений с клавиатуры. Для редактирования ячеек используется еще один класс-делегат. Им может стать любой класс, реализующий интерфейс `TableCellEditor`.

Интерфейс `TableCellEditor` расширяет интерфейс `CellEditor` и добавляет к его методам только один метод

```
public Component getTableCellEditorComponent(JTable table,
     Object value, boolean isSelected, int row, int column);
```

который должен выполнять любой редактор ячеек таблицы. Этот метод формирует компонент, пригодный для редактирования, например объект класса `JTextField`, для заданной ячейки с индексами `(row, column)` и содержимым `value`. Дополнительно указывается таблица `table`, в которой происходит редактирование. Метод может учесть параметр `isSelected`, показывающий, выделена ячейка или нет.

Сам же интерфейс `CellEditor` описывает несколько методов, из которых наиболее важны два. Один из них — логический метод `stopCellEditing()` — возвращает `true`, если редактирование ячейки завершено и следует сохранить сделанные изменения. Он возвращает `false`, если редактирование еще не завершено. В это время удобно сделать проверку измененной ячейки. Второй метод `cancelCellEditing()` отменяет редактирование.

При изменении содержимого ячейки происходит событие класса `ChangeEvent`, которое можно отследить, присоединив к редактору слушателя события методом `addCellEditorListener(CellEditorListener)`.

Интерфейс `TableCellEditor` реализован классом `DefaultCellEditor`. Конструкторы этого класса применяют в качестве конкретных редакторов компоненты `JCheckBox`, `JComboBox` и `JTextField`.

По умолчанию таблицы используют редактор с полем ввода `JTextField`. Хотя в ячейках таблицы могут располагаться любые объекты, этот редактор обрабатывает только текст, получающийся применением метода `toString()` такого объекта.

Компонент `JCheckBox` используется для изображения в виде флажка логического содержимого ячейки типа `boolean`, как показано на рис. 13.1. Этот редактор позволяет изменять истинность содержимого ячейки.

Компонент `JComboBox` применяется для ввода в ячейку одного из нескольких значений, содержащихся в раскрывающемся списке.

В листинге 13.6 приведен пример программы, позволяющей устанавливать разные редакторы для изменения содержимого различных ячеек таблицы. В примере ячейка в третьей строке и во втором столбце редактируется компонентом `JComboBox`. Результат показан на рис. 13.3.

Листинг 13.6. Разные редакторы ячеек таблицы

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.*;
import javax.swing.table.*;
   import javax.swing.event.*;

public class RowEd extends JFrame{
  public RowEd(){
       super("  Редактор строки");

    DefaultTableModel dm = new DefaultTableModel();
    dm.setDataVector(
      new Object[][]{{"Имя", "Иван"},
                     {"Фамилия", "Петров"},
                     {"Пол", "Мужской"}},
        new Object[]{"Сотрудник", "Сведения"});

       JTable table = new JTable(dm);

    JComboBox cb = new JComboBox();
    cb.addItem("Мужской");
       cb.addItem("Женский");

    RowEditor rowEd = new RowEditor(table);
    rowEd.setEditorAt(2, new DefaultCellEditor(cb));
       table.getColumn("Сведения").setCellEditor(rowEd);

    getContentPane().add(new JScrollPane(table));
    setSize(400, 100);
    setVisible(true);
     }

  public static void main(String[] args){
    new RowEd();
  }
   }

   class RowEditor implements TableCellEditor{

  protected Hashtable editors;
  protected TableCellEditor editor, defEditor;
     JTable table;

  public RowEditor(JTable table){
    this.table = table;
    editors = new Hashtable();
    defEditor = new DefaultCellEditor(new JTextField());
     }

  public void setEditorAt(int row, TableCellEditor editor){
    editors.put(new Integer(row), editor);
     }
```

```
  public Component getTableCellEditorComponent(JTable table,
         Object value, boolean isSelected, int row, int column){

        return editor.getTableCellEditorComponent(table,
          value, isSelected, row, column);
     }

  public Object getCellEditorValue(){
    return editor.getCellEditorValue();
  }
  public boolean stopCellEditing(){
    return editor.stopCellEditing();
  }
  public void cancelCellEditing(){
    editor.cancelCellEditing();
  }
  public boolean isCellEditable(EventObject anEvent){
    selectEditor((MouseEvent)anEvent);
    return editor.isCellEditable(anEvent);
  }
  public void addCellEditorListener(CellEditorListener l){
    editor.addCellEditorListener(l);
  }
  public void removeCellEditorListener(CellEditorListener l){
    editor.removeCellEditorListener(l);
  }
  public boolean shouldSelectCell(EventObject anEvent){
    selectEditor((MouseEvent)anEvent);
    return editor.shouldSelectCell(anEvent);
     }

  protected void selectEditor(MouseEvent e){
    int row = (e == null) ?
      table.getSelectionModel().getAnchorSelectionIndex() :
         table.rowAtPoint(e.getPoint());

    editor = (TableCellEditor)editors.get(new Integer(row));
    if (editor == null) editor = defEditor;
  }
}
```



Рис. 13.3. Компонент `JComboBox` в ячейке таблицы

Сортировка строк таблицы

Вы знаете, что, работая с электронными таблицами, такими как MS Excel, можно отсортировать строки таблицы по какому-нибудь столбцу, просто щелкнув мышью по заголовку столбца. Повторный щелчок вызовет сортировку в обратном порядке.

Такая возможность есть и в классе `JTable`. Самый простой способ реализовать ее — это включить сортировку методом `setAutoCreateRowSorter(true)`:

```
JTable table = new JTable();
table.setAutoCreateRowSorter(true);
```

Все! После этого щелчок мышью по заголовку столбца таблицы вызовет сортировку строк таблицы по этому столбцу, а повторный щелчок — сортировку в обратном порядке.

Сортировку обеспечивает абстрактный настраиваемый класс `RowSorter<M>`. Он рассчитан не только на таблицы, а вообще на компоненты, сконструированные по схеме MVC *(см. главу 3)*, и осуществляет связь между *Видом* и *Моделью* в этой схеме. Порядок строк меняется только во View, строки в Model остаются без изменения.

Класс `RowSorter<M>` расширен абстрактным классом `DefaultRowSorter<M, I>`, рассчитанным на модель, в которой данные хранятся в виде таблицы.

Для сортировки строк таблицы сделано его расширение — класс `TableRowSorter<M extends TableModel>`. Объект этого класса осуществляет связь View и Model таблицы, которую можно записать следующим образом:

```
FileTableModel model = new FileTableModel();
JTable table = new JTable(model);
table.setRowSorter(new TableRowSorter(model));
```

Разумеется, тип элементов столбца, по которому ведется сортировка, должен допускать сравнение элементов по величине, реализовав интерфейс `Comparator` *(см. главу 6)*. Это можно проверить логическим методом `isSortable(int column)` класса `DefaultRowSorter`. Если это не сделано, то можно поправить дело методом

```
setComparator(int column, Comparator comparator);
```

того же класса. Если же и это не сделано, то сортируются строки, полученные из элементов столбца их методами `toString()`.

В тех случаях, когда надо отсортировать строки сразу по нескольким столбцам, следует задать очередность сортировки столбцов. Для этого создается список типа `List`, содержащий объекты вложенного класса `RowSorter.SortKey`. При создании этих объектов указывается номер столбца и порядок его сортировки. Порядок сортировки, прямой или обратный, определяется константой `ASCENDING` или `DESCENDING` перечисления `SortOrder`. Очередность столбцов при сортировке соответствует порядку объектов в списке. Все это вместе выглядит так:

```
List<RowSorter.SortKey> keys = new ArrayList<RowSorter.SortKey>();
keys.add(new RowSorter.SortKey(1, SortOrder.ASCENDING));
keys.add(new RowSorter.SortKey(0, SortOrder.ASCENDING));
TableRowSorter sorter = new TableRowSorter(model));
sorter.setSortKeys(keys);
```

Как уже говорилось ранее, перестановка строк при сортировке происходит только при выводе их из Model на экран, точнее говоря, во View. Методами

```
int modelRowIndex = table.convertRowIndexToModel(viewRowIndex);
int viewRowIndex  = table.convertRowIndexToView(modelRowIndex);
```

можно отследить соответствие порядкового номера строки `viewRowIndex` во View и ее порядкового номера `modelRowIndex` в Model.

Фильтрация строк таблицы

Кроме сортировки, класс `RowSorter` позволяет выбрать строки таблицы из Model по какому-нибудь критерию для вывода их во View. Для этого надо создать фильтр, отбирающий строки — объект абстрактного настраиваемого класса `RowFilter<M, I >`, и передать ссылку на него методом `setRowFilter()` класса `DefaultRowSorter`. Это можно сделать по такой схеме:

```
TableRowSorter sorter = new TableRowSorter(model));
RowFilter filter = RowFilter.regexFilter("java", 0);
sorter.setRowFilter(filter);
```

В этом примере отбираются строки таблицы, содержащие в нулевом столбце строку символов, в которой встречается подстрока “java”.

Для создания фильтра в классе `RowFilter` есть несколько статических методов. Кроме использованного ранее метода

```
public static <M, I > RowFilter<M, I > regexFilter(String regex, int... indi-
ces);
```

фильтрующего строки регулярным выражением `regex`, полезны еще два метода:

```
public static <M, I > RowFilter<M, I >
    numberFilter(RowFilter.ComparisionType type, Number number, int... indi-
ces);

public static <M, I > RowFilter<M, I >
       dateFilter(RowFilter.ComparisionType type, Date date, int... indices);
```

Параметр `type` этих методов — одна из констант `AFTER`, `BEFORE`, `EQUAL` или `NOT_EQUAL` вложенного перечисления `RowFilter.ComparisionType`. Эти константы показывают, что надо передать во View только строки со значением большим, меньшим, равным или не равным значению `number` или `date`, служащему вторым параметром методов. Последние параметры `indices` всех трех методов перечисляют индексы столбцов, значения которых сравниваются со вторым аргументом. Отсутствие параметров `indices` означает просмотр всех столбцов таблицы.

В более сложных случаях, например, когда задан один или несколько диапазонов значений, можно связать несколько фильтров методами

```
public static <M, I > RowFilter<M, I > notFilter(RowFilter<M, I > filter);

public static <M, I > RowFilter<M, I >
    andFilter(Iterable<? extends RowFilter<? super M, ? super I >> filters);
```

```
public static <M, I > RowFilter<M, I >
        orFilter(Iterable<? extends RowFilter<? super M, ? super I >>
   filters);
```

Первый из этих трех методов, `notFilter()`, создает фильтр, пропускающий те и только те строки таблицы, которые были бы отвергнуты его аргументом. Второй метод, `andFilter()`, дает фильтр, пропускающий те и только те строки, которые проходят через все фильтры коллекции `filters`. Третий метод, `orFilter()`, пропускает строки, удовлетворяющие хотя бы одному фильтру из коллекции `filters`.

Для более сложной фильтрации строк таблицы нужно расширить класс `RowFilter`. При этом достаточно переопределить только один метод

```
public boolean include(RowFilter.Entry<? extends M, ? extends I > row);
```

Объект класса `DefaultRowSorter` обращается к этому методу при просмотре Model, передавая ему каждую строку таблицы. Метод возвращает `true`, если строка отвечает фильтру и ее надо передать во View.

Параметр этого метода — ссылка на строку таблицы, представленную в виде объекта вложенного настраиваемого класса `RowFilter.Entry`. Класс `RowFilter.Entry` предоставляет информацию о строке следующими методами:

- `I getIdentifier()` — возвращает идентификатор строки в модели;
- `M getModel()` — возвращает ссылку на модель;
- `String getStringValue(int index)` — возвращает значение столбца с индексом `index` в виде строки;
- `Object getValue(int index)` — возвращает значение столбца с индексом `index` в виде ссылки;
- `int getValueCount()` — возвращает количество столбцов в данной строке.

Эту информацию можно использовать в методе `include()`, чтобы принять решение об отборе строки. В следующем примере создан фильтр для отбора тех строк таблицы, у которых в первом столбце записан заданный текст, а сумма числовых значений остальных столбцов больше заданного целого числа

```
class SumRowFilter extends RowFilter{
    private String text;
    private int limit;
    public SumRowFilter(String text, int limit){
        this.text = text;
        this.limit = limit;
    }
    public boolean include(Entry entry){
        int sum = 0;
        if (entry.getStringValue(0).equals(text)){
            for (int i = 1; i < entry.getValueCount() - 1; i++){
                sum += ((Number)entry.getValue(i)).intValue();
            }
            if ( sum > limit) return true;
         }
```

```
            return false;
        }
    }
```

Печать таблицы

Несколько логических методов `print()` вызывают на экран стандартное диалоговое окно печати, позволяющее выбрать принтер и распечатать содержимое таблицы. Методы возвращают `false`, если пользователь отменил печать, щелкнув мышью по кнопке **Cancel** в диалоговом окне, и `true` в противном случае.

- `print(JTable.PrintMode mode)` — печать без колонтитулов с выбором сжатия столбцов. Столбцы таблицы при печати будут сжиматься под ширину листа бумаги, если задана константа `FIT_WIDTH` вложенного в класс `JTable` перечисления `JTable.PrintMode`, или не будут — константа `NORMAL`.
- `print()` — печать содержимого таблицы без колонтитулов со сжатием столбцов в режиме `JTable.PrintMode.FIT_WIDTH`.
- `print(JTable.PrintMode mode, MessageFormat header, MessageFormat footer)` — добавляется печать верхнего `header` и нижнего `footer` колонтитулов, оформленных как объекты класса `MessageFormat` из пакета `java.text`.

Остальные, более сложные методы печати используют сервер печати, что выходит за рамки нашей книги.

Примеры создания таблиц и связи их с базами данных имеются в документации Java SE в каталоге $JAVA_HOME/demo/jfc/TableExample/.

Вопросы для самопроверки

1. Как конструктивная схема MVC использована для создания классов таблиц?
2. Можно ли хранить в таблицах класса `JTable` образцы цвета?
3. Можно ли хранить в ячейках таблицы раскрывающиеся списки?
4. Можно ли сделать таблицу с ячейками-кнопками?
5. Можно ли сделать отдельные ячейки таблицы не редактируемыми?
6. Можно ли делать вычисления в таблице класса `JTable`, как это делается в электронных таблицах?
7. Можно ли сортировать строки таблиц класса `JTable`?
8. Можно ли сделать электронную таблицу средствами класса `JTable`?

Глава 14



Размещение компонентов и контейнеры Swing

В предыдущих главах мы размещали компоненты главным образом "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. Мы уже использовали некоторые из них в предыдущих главах без всякого объяснения. Библиотека Swing добавляет к указанным классам свои менеджеры размещения, используемые контейнерами Swing.

Каждый программист может создать собственные 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. 14.1. 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 14.1 создаются кнопка `JButton`, метка `JLabel`, кнопка выбора `JCheckBox`, раскрывающийся список `JComboBox`, поле ввода `JtextField` и все это размещается в контейнере `JFrame`. Рисунок 14.1 содержит вид перечисленных компонентов при разных размерах контейнера.

Листинг 14.1. Менеджер размещения `FlowLayout`

```
import java.awt.*;
   import javax.swing.*;
   class FlowTest extends JFrame{
  FlowTest(String s){
    super(s);
    Container c = getContentPane(); // Излишне, начиная с версии 5.0.
    c.setLayout(new FlowLayout(FlowLayout.LEFT, 10, 10));
    c.add(new JButton("Кнопка"));
    c.add(new JLabel("Метка"));
    c.add(new JCheckBox("Выбор"));
    c.add(new JComboBox());
    c.add(new JTextField("Ввод", 10));
    setSize(300, 100);
    setVisible(true);
  }
  public static void main(String[] args){
    JFrame f= new FlowTest("  Менеджер FlowLayout");
    f.setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
}
```

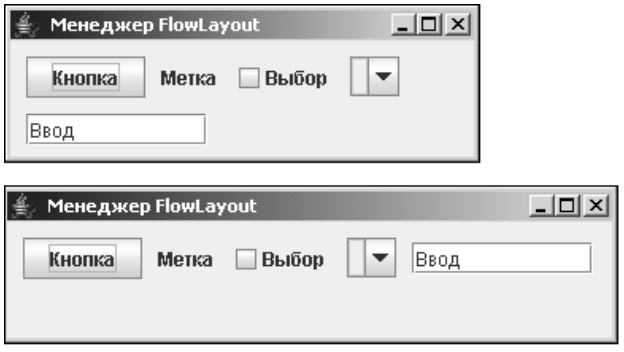


Рис. 14.1. Размещение компонентов с помощью `FlowLayout`

Менеджер *BorderLayout*

Менеджер размещения `BorderLayout` делит контейнер на пять неравных областей, полностью заполняя каждую область одним компонентом, как показано на рис. 14.2. 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 ранее констант: `add(Component comp, String region)`.

Обычный метод `add(Component comp)` с одним аргументом помещает компонент в область `CENTER`.

Ссылку на компонент, помещенный в определенную область, можно получить методом

```
getLayoutComponent(Object region);
```

параметром которого служит одна из перечисленных ранее констант.

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 заметить на рис. 14.3. Если не занята область `CENTER`, то компоненты прижимаются к границам контейнера.

В листинге 14.2 создаются пять кнопок, размещаемых в контейнере. Обратите внимание на отсутствие установки менеджера в контейнере `setLayout()` — менеджер `BorderLayout` установлен в контейнере `JFrame` по умолчанию. Результат размещения показан на рис. 14.2.

Листинг 14.2. Менеджер размещения `BorderLayout`

```
import java.awt.*;
   import javax.swing.*;

   class BorderTest extends JFrame{
```

```
  BorderTest(String s){
    super(s);
    Container c = getContentPane(); // Излишне, начиная с версии 5.0.
    c.add(new JButton("North"), BorderLayout.NORTH);
    c.add(new JButton("South"), BorderLayout.SOUTH);
    c.add(new JButton("West"), BorderLayout.WEST);
    c.add(new JButton("East"), BorderLayout.EAST);
    c.add(new JButton("Center"));
    setSize(300, 200);
    setVisible(true);
  }
  public static void main(String[] args){
    JFrame f= new BorderTest("  Менеджер BorderLayout");
    f.setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
}
```

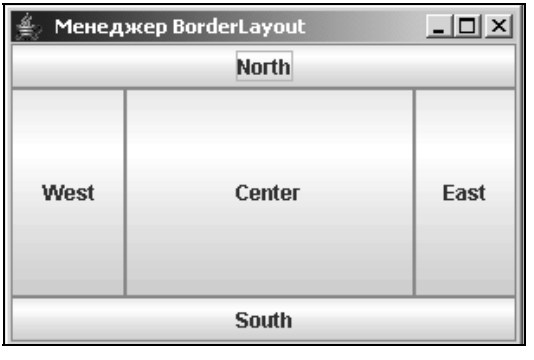


Рис. 14.2. Области размещения `BorderLayout`

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 14.3 и показано на рис. 14.3. Напомним, что на панелях `Panel` и `JPanel` менеджер размещения по умолчанию — `FlowLayout`.

Листинг 14.3. Сложная компоновка

```
import java.awt.*;
   import javax.swing.*;

   class BorderPanelTest extends JFrame{

  BorderPanelTest(String s){
    super(s);
    Container c = getContentPane(); // Излишне для JDK выше 1.4.
        // Создаем панель p2 с тремя кнопками
```

```
    JPanel p2 = new JPanel();
    p2.add(new JButton("Выполнить"));
    p2.add(new JButton("Отменить"));
       p2.add(new JButton("Выйти"));

    JPanel p1 = new JPanel();
    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 JTextField("Поле ввода", 20), BorderLayout.NORTH);
        // Область ввода помещается на панель с прокруткой
    JScrollPane sp = new JscrollPane(
         new JTextArea("Область ввода", 5, 20));
        // Панель прокрутки помещается в центр панели p1
       p1.add(sp), BorderLayout.CENTER);

        // Панель p1 помещаем в "центре" контейнера
    c.add(p1, BorderLayout.CENTER);
    setSize(300, 200);
    setVisible(true);
  }
  public static void main(String[] args){
    JFrame f= new BorderPanelTest("  Сложная компоновка");
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}
```

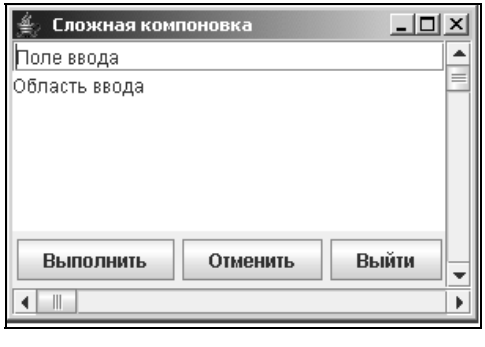


Рис. 14.3. Компоновка с помощью `FlowLayout` и `BorderLayout`

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 14.4 выстраиваются кнопки для калькулятора, а рис. 14.4 показывает, как выглядит это размещение.

Листинг 14.4. Менеджер `GridLayout`

```
import java.awt.*;
import javax.swing.*;
import java.util.*;
class GridTest extends JFrame{
  GridTest(String s){
    super(s);
    Container c = getContentPane(); // Излишне для JDK выше 1.4.
    c.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 JButton(st.nextToken()));
    setSize(200, 200);
    setVisible(true);
  }
  public static void main(String[] args){
    JFrame f= new GridTest("  Менеджер GridLayout");
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}
```



Рис. 14.4. 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 14.5 менеджер размещения `cl` работает с панелью `p`, помещенной в "центр" контейнера `JFrame`. Панель `p` указывается как аргумент `parent` в методах `next()` и `show()`. На "север" контейнера `JFrame` отправлена панель `p2` с меткой и раскрывающимся списком `ch`. Рисунок 14.5 демонстрирует результат работы программы.

Листинг 14.5. Менеджер `CardLayout`

```
import java.awt.*;
   import javax.swing.*;

class CardTest extends JFrame{
  CardTest(String s){
       super(s);

    Container c = getContentPane(); // Излишне для JDK 5.0 и выше.
    JPanel p = new JPanel();
    CardLayout cl = new CardLayout();
    p.setLayout(cl);
    p.add(new JButton("Русская страница"),"page1");
    p.add(new JButton("English page"),    "page2");
    p.add(new JButton("Deutsche Seite"),  "page3");
    c.add(p);
    cl.next(p);
       cl.show(p, "page1");
```

```
    JPanel p2 = new JPanel();
       p2.add(new JLabel("Выберите язык:"));

    JComboBox ch = new JComboBox();
    ch.addItem("Русский");
    ch.addItem("Английский");
       ch.addItem("Немецкий");

    p2.add(ch);
       c.add(p2, BorderLayout.NORTH);

    setSize(400, 300);
    setVisible(true);
  }
  public static void main(String[] args){
    JFrame f= new CardTest("  Менеджер CardLayout");
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}
```

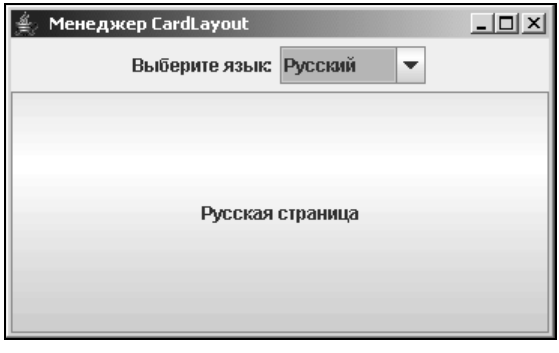


Рис. 14.5. Менеджер размещения `CardLayout`

Менеджер *GridBagLayout*

Менеджер размещения `GridBagLayout` расставляет компоненты наиболее гибко, позволяя задавать размеры и положение каждого компонента.

В классе `GridBagLayout` есть только один конструктор, 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. 14.1. Эти данные определяются конструктором, имеющим одиннадцать параметров по числу полей. Второй конструктор — конструктор по умолчанию — присваивает параметрам значения, заданные по умолчанию.

Таблица 14.1. *Поля класса* `GridBagConstraints`

Поле	Значение
anchor	Направление размещения компонента в контейнере. Константы: абсолютные — CENTER, NORTH, EAST, NORTHEAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, NORTHWEST; относительные — PAGE_START, PAGE_END, LINE_START, LINE_END, FIRST_LINE_START, FIRST_LINE_END, LAST_LINE_START, LAST_LINE_END, относительно базовой линии — BASELINE, BASELINE_LEADING, BASELINE_TRAILING, ABOVE_BASELINE, ABOVE_BASELINE_LEADING, ABOVE_BASELINE_TRAILING, BELOW_BASELINE, BELOW_BASELINE_LEADING, BELOW_BASELINE_TRAILING, 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, 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();
JButton b1 = new JButton();        // Создаем компонент.
c.gridwidth = 2;                   // Меняем правила размещения.
add(b1, c);                        // Помещаем компонент b1 в контейнер
                                   // по указанным правилам размещения c.
JButton b2 = new JButton();        // 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.

Контейнеры Swing

Поскольку класс `JComponent` библиотеки Swing — прямой наследник класса AWT `Container`, а компоненты Swing расширяют класс `JComponent`, все они являются контейнерами. Мы уже видели в предыдущих главах, что одни компоненты могут содержать в себе другие компоненты. Но в библиотеке Swing есть и компоненты, специально предназначенные для размещения других компонентов. Некоторые используют специально разработанные для них менеджеры размещения. Сейчас мы дадим обзор таких контейнеров. Кроме того, в библиотеке Swing есть много готовых диалоговых окон, которые будут рассмотрены в конце этой главы.

Панель *JPanel*

Класс `JPanel` реализует простую невидимую панель для размещения компонентов. Он очень похож на панель класса `Panel` библиотеки AWT.

Конструктор по умолчанию `JPanel()` применяет к панели менеджер размещения компонентов `FlowLayout`. Напомним, что этот менеджер располагает компоненты рядами, укладывая их слева направо и сверху вниз как кирпичи в порядке обращения к методам `add()`, устанавливая при этом такой размер каждого компонента, который возвращает метод `getPreferredSize()` этого компонента. При изменении размера панели компоненты перестраиваются, увеличивая или уменьшая количество рядов. Если метод `getPreferredSize()` не определен, то компонент не будет виден на панели. В таком случае надо обратиться к методу `setPreferredSize(Dimension)` компонента и установить его подходящий размер.

Конструктор `JPanel(LayoutManager)` применяет к панели заданный менеджер размещения. Это можно сделать и позднее методом `setLayout(LayoutManager)`, унаследованным от класса `Container`.

Панель класса `JPanel` применяет *двойную буферизацию* (double buffering) для перерисовки своего содержимого. Описание метода двойной буферизации мы дадим в *главе 20*. Если двойная буферизация не нужна, то следует создать панель конструктором `JPanel(false)`.

Наконец, четвертый конструктор, `JPanel(LayoutManager, boolean)`, задает менеджер размещения и применение двойной буферизации.

Как правило, применение панели ограничивается созданием ее экземпляра и размещением на ней компонентов унаследованными от класса `Container` методами:

- `add(Component comp)` — добавляет компонент `comp` в конец списка компонентов, лежащих на панели;
- `add(Component comp, Object constraints)` — добавляет компонент `comp` в конец списка компонентов, лежащих на панели, и передает менеджеру размещения параметры `constraints`, суть которых зависит от типа менеджера;
- `add(Component comp, int ind)` — вставляет компонент `comp` в указанную позицию `ind` списка компонентов панели;
- `add(Component comp, Object constraints, int ind)` — содержит все эти параметры.

Панель прокрутки *JScrollPane*

Класс `JScrollPane` содержит один компонент, обеспечивая прокрутку его содержимого и снабжая при необходимости линейками прокрутки. Это удобно для текстовой области `JTextArea`, для таблиц `JTable`, списков, изображений и других компонентов, чье содержимое не умещается в окне компонента. Возможность прокрутки не встроена в эти компоненты, чтобы можно было легко отказаться от нее в тех случаях, когда содержимое компонента не должно прокручиваться, но они реализуют интерфейс `Scrollable`, описывающий методы предоставления информации линейкам прокрутки.

Компонент помещается в панель прокрутки сразу же при ее создании конструктором `JScrollPane(Component)` или позднее методом `setViewportView(Component)`. Полосы прокрутки могут всегда находиться на экране, появляться при необходимости или не появляться вообще. Это определяется методами:

```
void setVerticalScrollBarPolicy(int);
void setHorizontalScrollBarPolicy(int);
```

Аргументом первого метода служит одна из констант класса `JScrollPane`:

- `VERTICAL_SCROLLBAR_ALWAYS`;
- `VERTICAL_SCROLLBAR_AS_NEEDED`;
- `VERTICAL_SCROLLBAR_NEVER`,

а второго — одна из констант этого же класса:

- `HORIZONTAL_SCROLLBAR_ALWAYS`;
- `HORIZONTAL_SCROLLBAR_AS_NEEDED`;
- `HORIZONTAL_SCROLLBAR_NEVER`.

Точнее говоря, эти и другие константы собраны в интерфейсе `ScrollPaneConstants`, реализованном классом `JScrollPane`.

Панель прокрутки имеет сложное строение. На самом деле кроме своего содержимого и двух полос прокрутки она может содержать еще шесть компонентов: заголовок, устанавливаемый методом `setColumnHeaderView(Component)`, столбец слева, задаваемый методом `setRowHeaderView(Component)`, и четыре компонента по углам, размещаемые методом `setCorner(String, Component)`, применение которого можно посмотреть в листинге 14.6. Все это показано на рис. 14.6, который нарисован программой листинга 14.6. Размещением всех девяти компонентов занимается специально разработанный для этого менеджер размещения `ScrollPaneLayout`. Он жестко определяет место и размер каждого дополнительного компонента. К панели прокрутки, как ко всякому контейнеру, разрешается применить другой менеджер размещения методом `setLayout(LayoutManager)`, но новый менеджер может быть только расширением менеджера размещения `ScrollPaneLayout`.

Листинг 14.6. Компоненты панели прокрутки

```
import java.awt.*;
import java.awt.event.*;
   import javax.swing.*;

   public class ScrollComps extends JFrame{

  ScrollComps(){
    super("  Компоненты панели прокрутки");
    Container c = getContentPane(); // излишне для JDK 5.0 и выше.
       c.setLayout(new FlowLayout());

       JScrollPane sp = new JScrollPane(new JTextArea(5,30));

       sp.setPreferredSize(new Dimension(200, 200));

    sp.setCorner(JScrollPane.LOWER_LEFT_CORNER, new JLabel(" LL"));
    sp.setCorner(JScrollPane.LOWER_RIGHT_CORNER, new JLabel("LR"));
    sp.setCorner(JScrollPane.UPPER_LEFT_CORNER, new JLabel(" UL"));
       sp.setCorner(JScrollPane.UPPER_RIGHT_CORNER, new JLabel("UR"));

    JLabel lh = new JLabel("  Header");
    lh.setBorder(BorderFactory.createEtchedBorder());
       sp.setColumnHeaderView(lh);

    JLabel lr = new JLabel("Row");
    lr.setBorder(BorderFactory.createEtchedBorder());
       sp.setRowHeaderView(lr);

    sp.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
       sp.setViewportBorder(BorderFactory.createEtchedBorder());
```

```
        c.add(sp);

    setSize(400, 400);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
     }

  public static void main(String[] args){
    new ScrollComps();
  }
}
```

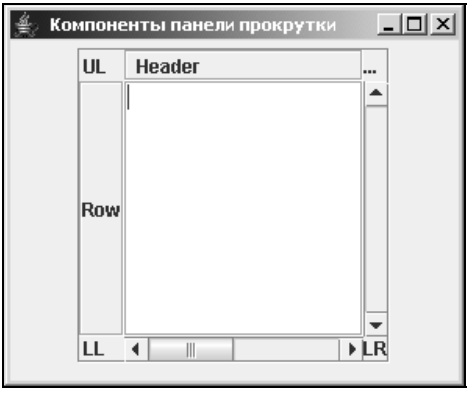


Рис. 14.6. Компоненты панели прокрутки

Видимая часть компонента, находящегося на панели прокрутки, а также заголовок и столбец слева представлены экземплярами класса `JViewport`. Задача этого класса — выбрать участок компонента и быстро показать его в "смотровом окне" (viewport), а также обеспечить быструю и правильную прокрутку компонента. Поэтому компонент `JViewport` может содержать в себе только один компонент, расположением которого занимается специально разработанный менеджер размещения `ViewportLayout`. Этот менеджер "растягивает" размещаемую область компонента до размеров смотрового окна.

В последнее время большое распространение получила мышь с колесиком, с помощью которого удобно выполнять прокрутку. По умолчанию панель прокрутки предоставляет возможность прокрутки колесиком, но ее можно отключить методом

```
setWheelScrollingEnabled(false);
```

Проверить, допускает ли панель прокрутку колесиком мыши, можно логическим методом

```
public boolean isWheelScrollingEnabled();
```

Двойная панель *JSplitPane*

Панель класса `JSplitPane` содержит два компонента, разделенных тонкой полосой, которую можно перемещать с помощью мыши, меняя таким образом взаимные размеры компонентов. Компоненты могут располагаться по горизонтали, что определяется константой

HORIZONTAL_SPLIT класса JSplitPane, или по вертикали — VERTICAL_SPLIT. Эти константы указываются в конструкторе JSplitPane(int orientation). Конструктор по умолчанию JSplitPane() задает горизонтальное расположение компонентов.

Панель JSplitPane при перемещении разделительной черты может перерисовывать компоненты сразу же по мере передвижения или после окончательной установки разделительной черты. Это определяется вторым параметром конструктора JSplitPane(int orientation, boolean continuous).

В перечисленных конструкторах не задаются размещаемые компоненты. Вместо них конструктор создает и размещает две кнопки JButton с надписями, возвращаемыми статическими методами UIManager.getString("SplitPane.leftButtonText") и UIManager.getString("SplitPane.rightButtonText"). Это просто надписи "left button" и "right button", как показано на рис. 14.7.



Рис. 14.7. Двойная панель

Четвертый конструктор, JSplitPane(int orientation, Component left, Component right), кроме расположения orientation сразу задает компонент, который будет располагаться слева (сверху) left, и тот, что будет располагаться справа (снизу) right.

Очень часто компоненты, размещаемые в панели, — это панели прокрутки класса JScrollPane, содержащие в себе текст, изображение, таблицу или другие компоненты.

Наконец, пятый конструктор задает все свойства:
JSplitPane(int orientation, boolean continuous, Component left, Component right).

Положение разделительной черты отмечается числом пикселов от левого (верхнего) края панели. Его можно получить методом getDividerLocation(), а установить из программы — методом setDividerLocation(int). В некоторых графических системах единица измерения может быть другой. В таком случае удобнее использовать метод setDividerLocation(double), задающий положение разделительной черты в процентах ширины (высоты) панели числом от 0.0 до 1.0.

Панель хранит и предыдущее положение разделительной черты. Его можно получить методом getLastDividerLocation(), а установить из программы — методом setLastDividerLocation(int).

Толщина разделительной черты назначается методом setDividerSize(int), параметр которого задается в пикселах. По умолчанию в Java L&F толщина равна 8 пикселов.

Разделительную черту нельзя переместить так, чтобы компонент стал меньше своего минимального размера, определенного методом getMinimumSize(). Границы ее перемещения

определяются методами `getMinimumDividerLocation()` и `getMaximumDividerLocation()`. Однако можно поместить на разделительную черту две небольшие кнопки с треугольными стрелками методом `setOneTouchExpandable(true)`. Они видны на рис. 14.7. При щелчке кнопкой мыши на одной из этих кнопок один компонент распахивается на всю панель, а другой исчезает полностью.

Компоненты можно установить в панель или заменить другими компонентами с помошью методов `setLeftComponent(Component)`, `setRightComponent(Component)`, `setTopComponent(Component)`, `setBottomComponent(Component)`, причем можно всегда пользоваться только первой или только последней парой методов независимо от фактического горизонтального или вертикального расположения компонентов.

Панель с вкладками *JTabbedPane*

Класс `JTabbedPane` создает сразу несколько панелей. На экране видна только одна из них, для остальных панелей показаны *вкладки* (tabs). Щелчок кнопкой мыши по вкладке вызывает на экран связанную с ней панель.

Конструктор по умолчанию `JTabbedPane()` создает одну пустую панель без вкладок. Конструктор `JTabbedPane(int pos)` задает расположение вкладок. Параметр этого конструктора `pos` — одна из констант класса `JtabbedPane`: `TOP`, `BOTTOM`, `LEFT`, `RIGHT`. Как правило, вкладки помещаются сверху (`TOP`), но, как видите, их можно поместить снизу, слева и справа.

Если все вкладки не помещаются в окно панели в один ряд, то они могут располагаться несколькими рядами или прокручиваться, для чего в строке вкладок появляются кнопки прокрутки, как показано на рис. 14.8. Первый метод расположения вкладок обозначается константой `WRAP_TAB_LAYOUT`, второй — константой `SCROLL_TAB_LAYOUT` класса `JTabbedPane`. Третий конструктор, `JTabbedPane(int pos, int tab)`, задает своим вторым параметром один из этих двух методов.

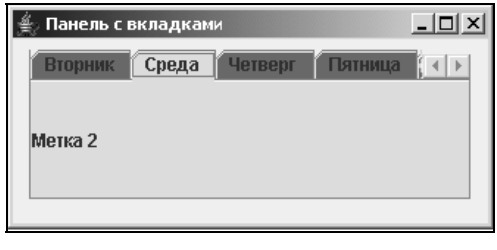


Рис. 14.8. Панель с вкладками

Как видите, конструкторы класса не создают содержащиеся в нем панели и не помещают на них компоненты. Это выполняется после создания объекта класса `JTabbedPane` следующими методами:

- `Component add(Component)` — добавляет компонент в последнюю панель и пишет на вкладке имя компонента;
- `Component add(String, Component)`

  `void addTab(String, Component)` — пишут на вкладке строку, записанную в первом параметре;

- `void add(Component, Object)` — помещает на вкладку объект, определенный вторым параметром. Обычно это изображение типа `Icon`;
- `Component add(Component, int)` — вставляет компонент в указанную позицию;
- `void add(Component, Object, int)` — объединяет возможности остальных методов;
- `void addTab(String, Icon, Component)` — помещает на вкладку строку и/или изображение;
- `void addTab(String title, Icon image, Component comp, String tip)` — последний параметр `tip` задает всплывающую подсказку.

Все эти методы, так или иначе, обращаются к основному методу

```
void insertTab(String title, Icon image, Component comp, String tip. int ind);
```

которым можно пользоваться во всех случаях.

Многочисленные методы `set`*`Xxx`*`()` позволяют установить отдельные элементы панелей и вкладок. Кроме того, можно задать цвет фона методом `setBackgroundAt(Color)`, как показано на рис. 14.8 и в листинге 14.7. Это удобно для того, чтобы разметить вкладки разными цветами.

Листинг 14.7. Панель с разноцветными вкладками

```
import java.awt.*;
import javax.swing.*;
public class Tabbed extends JFrame{
   Tabbed(){
     super("  Панель с вкладками");
     Container c = getContentPane(); // Излишне для JDK 5.0 и выше.
     c.setLayout(new FlowLayout());
     String[] day = {"Понедельник", "Вторник", "Среда",
        "Четверг", "Пятница", "Суббота", "Воскресенье"};
     JTabbedPane sp = new JTabbedPane(JTabbedPane.TOP,
           JTabbedPane.SCROLL_TAB_LAYOUT);
     sp.setPreferredSize(new Dimension(300, 100));
     for (int i = 0; i < 7; i++){
         sp.add(day[i], new JLabel("Метка " + i));
         sp.setBackgroundAt(i, new Color(16*i, 0, 16*(7-i)));
     }
     c.add(sp);
     setSize(400, 400);
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     setVisible(true);
   }
   public static void main(String[] args){
     new Tabbed();
   }
}
```

Линейная панель *Box*

Класс `Box` расставляет компоненты в одну строку или в один столбец, выравнивая их ширину или высоту по размеру наибольшего компонента. Этот класс был разработан для создания панелей инструментальных кнопок `JToolBar`, но его можно применять и для других целей. В классе есть только один конструктор — `Box(int)`, в котором задается одна из двух констант класса `BoxLayout`: константа `X_AXIS` — размещение компонентов в одну строку, или `Y_AXIS` — размещение компонентов в один столбец.

Еще один способ создания линейной панели — воспользоваться статическим методом `createHorizontalBox()` или статическим методом `createVerticalBox()`. Эти методы всего лишь обращаются к конструктору с соответствующей константой.

Сами компоненты добавляются к панели `Box` унаследованными от класса `Container` методами `add(Component)`, `add(Component, int)`.

Расположением компонентов в классе `Box` занимается специально разработанный менеджер размещения `BoxLayout`. Применить другой менеджер к этому классу нельзя, метод `setLayout(LayoutManager)` выбрасывает исключение, но менеджер `BoxLayout` может с успехом применяться в контейнерах иных типов. Рассмотрим его подробнее.

Менеджер размещения *BoxLayout*

Экземпляры класса `BoxLayout` создаются конструктором `BoxLayout(Container, int)`. Первым параметром указывается контейнер, размещением компонентов в котором будет управлять создаваемый менеджер размещения. Второй параметр задает способ расположения компонентов одной из констант `X_AXIS` — расположение слева направо, `Y_AXIS` — расположение сверху вниз, `LINE_AXIS` и `PAGE_AXIS` — расположение определяется контейнером. Создание и применение менеджера выглядят примерно так:

```
JPanel p = new JPanel();
p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
p.add(new JLabel("Введите имя: ", JLabel.RIGHT));
p.add(new JTextField(30));
```

Если задано горизонтальное расположение компонентов, то менеджер пытается сделать высоту всех компонентов одинаковой, равной высоте самого высокого компонента. При вертикальном расположении менеджер старается выровнять ширину компонентов по самому широкому компоненту. Если это сделать не удается, например потому, что задан максимальный размер компонентов, то по умолчанию компоненты размещаются в центре панели. Точнее говоря, это зависит от того, какое значение возвращают методы `getAlignmentX()` и `getAlignmentY()` самого компонента. Возвращаемое этими методами значение меняется от 0.0f — компонент прижимается влево (вверх), до 1.0f — компонент прижимается вправо (вниз) относительно других компонентов.

Компоненты-заполнители

Казалось бы, в панели `Box` нет ничего хитрого, но ее возможности расширяются тем, что в число размещаемых на панели компонентов можно включить невидимые компоненты-заполнители трех видов.

Первый вид заполнителя — невидимая *разделительная область* (rigid area), имеющая фиксированные размеры. Она создается статическим методом

```
static Component Box.createRigidArea(Dimension);
```

и вставляется между компонентами, создавая промежуток фиксированного размера между ними.

Заполнитель второго вида — невидимая *"распорка"* (strut) — имеет только один фиксированный размер. У горизонтальной распорки, создаваемой статическим методом

```
static Component Box.createHorizontalStrut(int width);
```

фиксирована только ширина. При горизонтальном расположении компонентов распорку можно использовать для создания промежутков между компонентами, а при вертикальном расположении — для задания ширины всей панели.

У вертикальной распорки, создаваемой статическим методом

```
static Component Box.createVerticalStrut(int height);
```

фиксирована высота. Она используется аналогично горизонтальной распорке.

Третий вид заполнителя — невидимая *"надувная подушка"* (glue), "раздуваясь", заполняет все выделенное ей пространство, раздвигая остальные компоненты и прижимая их к краям панели, если они имеют фиксированный максимальный размер. Этот заполнитель создается одним из статических методов:

- `static Component Box.createGlue()` — "подушка" раздувается во все стороны;
- `static Component Box.createHorizontalGlue()` — "подушка" раздается в ширину;
- `static Component Box.createVerticalGlue()` — "подушка" раздается в высоту.

Кроме этих трех компонентов-разделителей можно использовать невидимый компонент с фиксированным минимальным, максимальным и предпочтительным размерами. Он является объектом класса `Filler`, вложенного в класс `Box`, и создается конструктором

```
Box.Filler(Dimension min, Dimension pref, Dimension max);
```

Преимущество этого объекта в том, что он может поменять размеры методом

```
void changeShape(Dimension min, Dimension pref, Dimension max);
```

Листинг 14.8 показывает пример размещения текстовой области и двух кнопок на панели класса `Box`. Для того чтобы сдвинуть кнопки вправо, применена "подушка".

Листинг 14.8. Размещение компонентов на панели `Box`

```
import java.awt.*;
   import javax.swing.*;

public class MyBox extends JFrame{
  JButton  b1 = new JButton("Первая");
  JButton  b2 = new JButton("Вторая");
     JTextArea ta = new JTextArea(5, 30);

  MyBox(){
    super("  Линейная панель");
```

```
    Container c = getContentPane(); // Излишне для JDK 5.0 и выше.
       c.setLayout(new FlowLayout());

    Box out = Box.createVerticalBox();
    Box in1 = Box.createHorizontalBox();
       Box in2 = Box.createHorizontalBox();

    out.add(in1);
       out.add(in2);

       in1.add(ta);

    in2.add(Box.createHorizontalGlue());
    in2.add(b1);
       in2.add(b2);

       c.add(outer);

    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
     }

  public static void main(String[] args){
    new MyBox();
  }
}
```

Итак, класс `Box`, использующий разделители, становится удобным и гибким контейнером. Нужно еще учесть, что компоненты можно обвести рамкой необходимой ширины и то, что контейнеры класса `Box` могут вкладываться друг в друга в разных сочетаниях. Все это делает класс `Box` вполне приемлемым контейнером для размещения самых разных компонентов, а не только инструментальных кнопок.

Менеджер размещения *SpringLayout*

Контейнер `Box`, управляемый менеджером размещения `BoxLayout`, оказался удобным средством компоновки компонентов, но для сложного размещения большого числа компонентов приходится вкладывать экземпляры класса `Box` друг в друга. К тому же возникает необходимость часто применять компоненты-разделители. Это приводит к неоправданной сложности компоновки.

Вообще говоря, возможности размещения компонентов в контейнерах графических библиотек AWT и Swing очень велики и изменяются в широком диапазоне.

На одном конце этого диапазона — абсолютное размещение, при котором прямо указываются координаты `(x, y)`, ширина `width` и высота `height` компонента в координатной системе контейнера. Это выполняет метод

```
setBounds(int x, int y, int width, int height);
```

или пара методов

```
setLocation(int x, int y);
setSize(int width, int height);
```

При этом компоненты совершенно точно помещаются в контейнер, но их положение и размеры не меняются при изменении размеров контейнера. Можно связать координаты и размеры компонента с размерами контейнера, например:

```
JPanel p = new JPanel();
p.setLayout(null);
int w = p.getSize().width, h = p.getSize().height;
JButton b = new JButton("Выход");
b.setBounds(w/10, h/10, w/5, h/8);
p.add(b);
```

Это сложно и требует долгой кропотливой подгонки.

На другом конце диапазона — менеджер размещения `FlowLayout`, располагающий компоненты просто по их предпочтительному размеру, и менеджер `GridLayout`, подгоняющий размеры всех компонентов под размер контейнера.

В состав Java SE, начиная с версии JDK 1.4, введен новый менеджер размещения `SpringLayout`, пытающийся совместить точность и гибкость размещения компонентов. Для определения положения и размеров компонента этот менеджер пользуется координатами `(x, y)` и размерами `width`, `height`, но это не числа, а объекты специально разработанного небольшого класса `Spring`. Опишем этот класс.

Размеры *Spring*

Абстрактный класс `Spring` описывает объекты, хранящие размеры. Каждый объект хранит минимальный, предпочтительный и максимальный размеры. Эти размеры могут использоваться как размеры промежутков между компонентами. Поскольку класс `Spring` абстрактный, объект с размерами задается не конструктором, а статическим методом

```
public static Spring constant(int min, int pref, int max);
```

Второй статический метод, `constant(int size)`, возвращает объект класса `Spring` с совпадающими между собой размерами, равными `size`.

Два статических метода берут координату `height` или координату `width` у минимального, предпочтительного и максимального размеров заданного компонента `comp`:

```
public static Spring height(Component comp);
public static Spring width(Component comp);
```

Итак, на объект класса `Spring` можно смотреть как на трехмерный вектор `(min, pref, max)`.

Кроме этих трех размеров, объект хранит еще и текущее значение `value`, устанавливаемое методом `setValue(int value)`. Значение `value` должно лежать между минимальным и максимальным значениями. Начальное значение `value` совпадает с предпочтительным размером.

Менеджер размещения получает эти размеры, обращаясь к методам `getMinimumValue()`, `getMaximumValue()`, `getPreferredValue()`, `getValue()`.

Очень часто менеджер размещения `SpringLayout` использует несколько объектов `Spring`. При этом их размеры складываются, вычитаются, берется наибольший или наименьший размер. Все операции выполняются, как операции с векторами, покоординатно. Для удобства их выполнения в класс `Spring` введены статические методы таких вычислений:

- `public static Spring max(Spring s1, Spring s2)` — возвращает новый объект, размеры которого составлены из наибольших значений объектов `s1` и `s2`;
- `public static Spring minus(Spring s)` — возвращает новый объект, размеры которого равны размерам объекта `s` с обратным знаком;
- `public static Spring sum(Spring s1, Spring s2)` — возвращает новый объект, размеры которого равны сумме соответствующих размеров объектов `s1` и `s2`.

Для вычисления минимального из двух значений `s1` и `s2` нет специального метода, оно вычисляется так:

```
Spring sp = Spring.minus(Spring.max(Spring.minus(s1), Spring.minus(s2)));
```

Промежутки *Constraints*

Объект класса `Spring` — всего лишь четверка чисел. Он не может определить пространство в контейнере, а используется только для построения объекта вложенного класса `SpringLayout.Constraints`.

Объект класса `Constraints`, подобно прямоугольнику, содержит координаты `(x, y)`, ширину `width` и высоту `height`, но эти величины — не числа, а объекты класса `Spring`. Объекты `x` и `y` — не жестко фиксированные координаты левого верхнего угла, как в прямоугольнике. Они имеют наименьшее, наибольшее, предпочтительное и текущее значения, которыми пользуется менеджер размещения `SpringLayout`. Он варьирует положение левого верхнего угла компонента в заданных объектами `x` и `y` пределах. Поэтому они обозначаются статическими строковыми константами `WEST` и `NORTH` класса `SpringLayout`. Ширина `width` и высота `height` — это наименьшая, наибольшая, предпочтительная и текущая ширина и высота компонента в контейнере, управляемом менеджером размещения `SpringLayout`. Чаще всего они совпадают с соответствующими значениями самого компонента. Величины `x + width` и `y + height` обозначаются статическими строковыми константами `EAST` и `SOUTH` и определяют положение правого нижнего угла компонента.

Для получения и установки всех этих значений в классе `Constraints` есть методы-"сеттеры" и "геттеры": `setX(Spring)`, `getX()`, `setY(Spring)`, `getY()`, `setWidth(Spring)`, `getWidth()`, `setHeight(Spring)`, `getHeight()`. Методы `setConstraint(String, Spring)` и `getConstraint(String)` устанавливают и выдают объект класса `Spring` по заданному имени `NORTH`, `WEST`, `SOUTH` или `EAST`.

Размещение компонентов

После обсуждения этих вспомогательных классов можно объяснить принцип работы менеджера размещения `SpringLayout` на примере.

Допустим, мы хотим расположить несколько компонентов `comp[0]`, `comp[1]`, `comp[2]` и т. д. в одну строку с фиксированными промежутками между ними величиной в 6 пикселов. Кроме того, мы решили оставить промежутки в 10 пикселов от границ контейнера. Листинг 14.9 содержит программу, выполняющую такое размещение, а рис. 14.9 показывает результат размещения.

Листинг 14.9. Размещение компонентов `SpringLayout`

```
import java.awt.*;
   import javax.swing.*;

public class SpringWin extends JFrame{
   JComponent[] comp = {
     new JButton("Длинная надпись"),
     new JButton("<html>Надпись с<p> двумя строками"),
     new JButton("OK")
    };
   public SpringWin(){
     super(" Размещение SpringLayout");
     Container c = getContentPane(); // Излишне для JDK 5.0 и выше.
     SpringLayout sl = new SpringLayout();
        c.setLayout(sl);

      // Задаем величину промежутка между компонентами
        Spring xPad = Spring.constant(6);

      // Задаем величину отступа от границ контейнера
        Spring yPad = Spring.constant(10);

      // Текущее положение левого верхнего угла
        Spring currX = yPad;

      // Наибольшая высота компонента, пока 0
        Spring maxHeight = Spring.constant(0);

     for (int i = 0; i < comp.length; i++){
       c.add(comp[i]);
         // Получаем размер i-го компонента
       SpringLayout.Constraints cons = sl.getConstraints(comp[i]);
         // Устанавливаем положение i-го компонента
       cons.setX(currX);
       cons.setY(yPad);
         // Перемещаем текущее положение угла
       currX = Spring.sum(xPad, cons.getConstraint("East"));
         // Изменяем наибольшую высоту
       maxHeight = Spring.max(maxHeight, cons.getConstraint("South"));
     }
```

```
        // Получаем размеры контейнера
         SpringLayout.Constraints pCons = sl.getConstraints(c);

        // Устанавливаем размеры всего содержимого контейнера
      pCons.setConstraint(SpringLayout.EAST, Spring.sum(currX, yPad));
         pCons.setConstraint(SpringLayout.SOUTH, Spring.sum(maxHeight, yPad));

      pack();
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setVisible(true);
       }

   public static void main(String args[]){
     new SpringWin();
   }
}
```

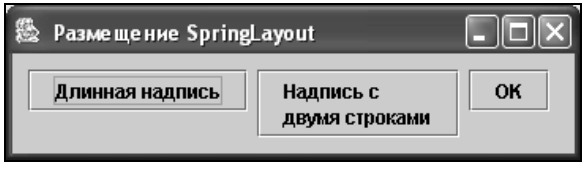


Рис. 14.9. Размещение `SpringLayout`

Панель инструментальных кнопок *JToolBar*

Класс `JToolBar` создает панели инструментальных кнопок. Обычно такая панель занимает строку ниже строки меню или столбец слева. Очень часто панель делают плавающей — ее можно перемещать по экрану — или всплывающей.

Пустая горизонтальная панель создается конструктором `JToolBar()`.

Конструктор `JToolBar(int)` задает расположение панели: горизонтальное — константа `HORIZONTAL`, вертикальное — константа `VERTICAL`.

Конструктор `JToolBar(String)` определяет заголовок горизонтальной панели.

Наконец, конструктор `JToolBar(String, int)` определяет заголовок и расположение панели.

Документация Java SE рекомендует управлять контейнером, в который помещают панель инструментов, с помощью менеджера размещения `BorderLayout` и ничего не помещать в граничные области этого контейнера. В этом случае панель чаще всего помещают на "север". Слева (сверху) панели имеется полоса с "насечкой". Она видна на рис. 14.10. Наведя курсор мыши на эту полосу, панель можно перемещать по контейнеру. При перенесении панели на "запад", "восток" или на "юг", она занимает эту область, располагая свои компоненты по вертикали или по горизонтали. При перенесении панели в "центр" или вынесении ее за окно контейнера, панель автоматически оформляется в отдельное окно класса `JFrame`. В строке заголовка отдельного окна появляется строка, заданная в конструкторе. Полоса с

"насечкой" сохраняется в этом окне, с ее помощью можно вернуть панель на прежнее место. Панель возвращается на свое первоначальное место и при закрытии ее окна.

Панель инструментов можно сделать неперемещаемой методом `setFloatable(false)`. Полоска с "насечкой" исчезает, панель нельзя передвигать по экрану.

Обычно инструментальные кнопки на панели обведены тонкой рамкой, как на **Панели 2** рис. 14.10, но после применения метода `setRollover(true)` рамка будет появляться только при наведении курсора мыши на кнопку.

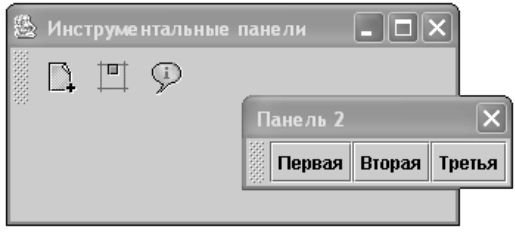


Рис. 14.10. Инструментальные панели

Для расположения компонентов класс `JToolBar` применяет свой внутренний менеджер размещения `DefaultToolBarLayout`, основанный на менеджере `BoxLayout`, следовательно, панель инструментов может использовать свойства этого менеджера размещения. Листинг 14.10 дает пример создания панели инструментов.

Листинг 14.10. Панели инструментальных кнопок

```
import java.awt.*;
   import javax.swing.*;

   public class MyTool extends JFrame{

  MyTool(){
    super(" Инструментальные панели");
       Container c = getContentPane(); // Излишне для JDK 5.0 и выше.

    JToolBar tb1 = new JToolBar(" Панель 1"),
                tb2 = new JToolBar(" Панель 2");

       tb1.setRollover(true);

    tb1.add(new JButton(new ImageIcon("Add24.gif")));
    tb1.add(new JButton(new ImageIcon("AlignTop24.gif")));
       tb1.add(new JButton(new ImageIcon("About24.gif")));

    tb2.add(new JButton("Первая"));
    tb2.add(new JButton("Вторая"));
       tb2.add(new JButton("Третья"));
```

```
    c.add(tb1, BorderLayout.NORTH);
       c.add(tb2, BorderLayout.WEST);

    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
     }

  public static void main(String[] args){
    new MyTool();
  }
}
```

В составе Java SE в каталоге $JAVA_HOME/demo/jfc/Notepad/ есть пример текстового редактора с панелью инструментальных кнопок класса `JToolBar`. Его можно запустить, перейдя в этот каталог и набрав в командной строке

```
java -jar Notepad.jar
```

На панель инструментов можно поместить любой компонент методом `add(Component)`, но, как правило, на ней располагаются кнопки с ярлычками. Эти кнопки дублируют некоторые, чаще всего используемые, пункты меню. Для того чтобы облегчить связь кнопок и пунктов меню с их действиями, в библиотеке Swing разработан интерфейс `Action`.

Интерфейс *Action*

Интерфейс `Action` разработан для того, чтобы собрать в одном месте все, относящееся к какому-то действию: командную клавишу, клавишу-ускоритель, изображение-ярлык, строку описания в пункте меню, всплывающую подсказку.

Он расширяет интерфейс `ActionListener`, добавляя к его единственному методу `actionPerformed(ActionEvent)` несколько статических полей класса `String` — имен определяемых интерфейсом объектов, — и методы определения этих полей `putValue(String key, Object value)`, `getValue(String key)`.

Интерфейс описывает объекты с такими именами:

- `ACCELERATOR_KEY` — имя клавиши-ускорителя, объекта класса `KeyStroke`;
- `ACTION_COMMAND_KEY` — имя командной клавиши класса `KeyMap`;
- `DEFAULT` — строка значений по умолчанию;
- `DISPLAYED_MNEMONIC_INDEX_KEY` — целое число, индекс действия;
- `LARGE_ICON_KEY` — имя изображения;
- `LONG_DESCRIPTION` — описание действия для всплывающей справки;
- `MNEMONIC_KEY` — код командной клавиши типа `int`;
- `NAME` — имя действия, записываемое в пункт меню;
- `SELECTED_KEY` — выбранное значение;
- `SHORT_DESCRIPTION` — краткое описание действия для всплывающей подсказки;
- `SMALL_ICON` — имя изображения на инструментальной кнопке.

Поля определяются, например, так:

```
act.putValue(Action.SMALL_ICON, new ImageIcon("save.gif"));
```

и используются потом так:

```
ImageIcon ic = (ImageIcon)act.getValue(Action.SMALL_ICON);
```

Объект, реализующий интерфейс `Action`, может быть доступен (enabled) или недоступен (disabled). Это регулируется методом `setEnabled(boolean)`. Например, при создании текстового редактора команды **Вырезать**, **Копировать** следует сделать недоступными, пока не выделен текст в окне редактирования.

Интерфейс `Action` частично реализован абстрактным классом `AbstractAction`, в котором не определен только метод `actionPerformed(ActionEvent)`. Достаточно расширить класс `AbstractAction`, определив этот метод, и можно использовать его возможности.

Некоторые контейнеры, в число которых входят `JMenu`, `JPopupMenu` и `JToolBar`, умеют использовать объекты, реализующие интерфейс `Action`. Достаточно обратиться к методу `add(Action act)` такого контейнера, и он создаст соответствующий компонент со свойствами, определенными в объекте `act`.

Например, метод `tb.add(act)` панели инструментов `tb` класса `JToolBar` создаст инструментальную кнопку класса `JButton` с изображением, командной клавишей, всплывающей подсказкой и прочими свойствами инструментальной кнопки, содержащимися в объекте `act`. Метод `m.add(act)` меню `m` класса `JMenu` с тем же объектом `act` создаст пункт меню класса `JMenuItem` с надписью, ярлычком, всплывающей подсказкой, взятыми у того же объекта `act`.

Контейнер автоматически присоединит к созданному компоненту обработчик события `ActionEvent`, описанный в методе `actionPerformed(ActionEvent)`. Кроме того, контейнер будет автоматически отслеживать изменения в самом объекте типа `Action` и приводить созданный компонент в соответствие с этими изменениями.

Классы, реализующие интерфейс `Action`, очень удобно использовать в тех случаях, когда одно и то же действие выполняется несколькими элементами графического интерфейса. Например, сохранение файла может быть вызвано пунктом меню **Сохранить**, инструментальной кнопкой с изображением дискеты или из всплывающего контекстного меню. В любом случае сохранение файла выполняет один объект типа `Action`, присоединенный к этим графическим элементам их методом `add(Action)`.

Слоеная панель *JLayeredPane*

Слоеная панель (layered pane) — это экземпляр класса `JLayeredPane`. Она состоит из множества лежащих друг на друге слоев, в которых располагаются компоненты. Компоненты, лежащие в верхних слоях, визуально перекрывают компоненты, находящиеся в нижних слоях. Слои в панели нумеруются целыми числами, представленными объектами класса `Integer`. Слой, номер которого больше, располагается выше слоя с меньшим номером. Можно рассматривать слои как третью координату `z` на экране — глубину.

Метод `add(Component comp, Object constraints)` класса `Container` переопределен в классе `JLayeredPane` так, что второй параметр задает слой, в который помещается компонент. Например, компонент `comp` можно поместить в слой с номером 50 методом `add(comp, new Integer(50))`.

Шесть слоев обозначены статическими константами типа `Integer`. Они интенсивно используются методами самой библиотеки Swing.

- `FRAME_CONTENT_LAYER` — слой с номером `Integer(-30000)`. Такой маленький номер гарантирует, что этот слой окажется ниже всех слоев. Данный слой используется классом `JRootPane` для размещения компонентов и строки меню.
- `DEFAULT_LAYER` — слой с номером `Integer(0)`. Стандартная панель для размещения компонентов.
- `PALETTE_LAYER` — слой с номером `Integer(100)`. В нем обычно располагаются плавающие панели инструментальных кнопок и палитры.
- `MODAL_LAYER` — слой с номером `Integer(200)`. Здесь располагаются модальные диалоговые окна.
- `POPUP_LAYER` — слой с номером `Integer(300)`. Сюда помещают окна, всплывающие над модальными диалоговыми окнами.
- `DRAG_LAYER` — слой с номером `Integer(400)`. Сюда переводится компонент на время его перетаскивания с помощью мыши. После перетаскивания и отпускания кнопки мыши компонент погружается в свой слой.

Слоеная панель не управляется никаким менеджером размещения, размеры и положение компонентов на ней следует задавать унаследованным методом

```
setBounds(int x, int y, int width, int height);
```

или парой методов

```
setLocation(int x, int y);
setSize(int width, int height);
```

Класс `JLayeredPane` расширяет класс `JComponent` и является компонентом Swing. Каждый слой ведет себя как обычный контейнер класса `Container`, но компоненты в нем могут перекрываться, причем компонент с меньшим номером перекрывает компонент с большим номером, но компонент с номером –1 лежит под всеми другими компонентами этого слоя.

Компоненты можно перемещать в своем слое методами:

- `moveToFront(Component comp)` — переместить компонент `comp` в позицию с номером 0;
- `moveToBack(Component comp)` — переместить компонент `comp` в позицию с номером –1;
- `setPosition(Component comp, int ind)` — переместить компонент `comp` в позицию `ind`.

Чтобы поместить компонент в другой слой, надо сначала указать ему новый слой методами:

- `setLayer(Component comp, int layer)` — указать компоненту `comp` слой с номером `layer` и позицию –1 в новом слое;
- `setLayer(Component comp, int layer, int pos)` — указать компоненту `comp` слой `layer` и позицию `pos` в новом слое.

После этого надо поместить компонент в новый слой методом `add()`.

Слоеная панель обычно не используется напрямую. Она содержится в *корневой панели* класса `JRootPane` наряду с другими панелями.

Корневая панель *JRootPane*

Класс `JRootPane` определяет *корневую панель* (root pane), которая располагается в окнах `JWindow`, `JDialog`, `JFrame`, `JInternalFrame`, `JApplet`, но может использоваться и в других контейнерах, реализующих интерфейс `RootPaneContainer`.

Корневая панель сама содержит несколько панелей размером во все окно, наложенных друг на друга. Нельзя просто положить компонент на корневую панель. Его надо положить на одну из панелей, содержащихся в корневой панели.

Ниже всех панелей лежит *панель содержимого* (content pane), которую можно получить методом `getContentPane()`. Это контейнер класса `Container`, в котором обычно размещается большинство компонентов и строка меню. Управляет их расположением по умолчанию менеджер размещения `BorderLayout`. Чтобы заменить менеджер размещения панели содержимого, надо вызвать ссылку на нее следующим образом:

```
getContentPane().setLayout(new FlowLayout());
```

Таким же образом надо помещать компоненты на панель содержимого, например:

```
Container c = getContentPane();
c.add(new JLabel("Окно регистрации", JLabel.CENTER), BorderLayout.NORTH);
c.add(new JTextArea(5, 50));
```

Начиная с пятой версии Java SE, панель содержимого сделана панелью по умолчанию. Приведенные выше строки теперь не требуют указания ссылки на контейнер `c`, можно написать просто

```
add(new JLabel("Окно регистрации", JLabel.CENTER), BorderLayout.NORTH);
add(new JTextArea(5, 50));
```

Панель содержимого располагается в нижнем слое `FRAME_CONTENT_LAYER` слоеной панели класса `JLayeredPane`. Кроме панели содержимого в этом же слое, в его верхней части, может находиться необязательная строка меню класса `JMenuBar`.

Итак, корневая панель `JRootPane` не содержит панель содержимого непосредственно. Она хранит экземпляр класса `JLayeredPane`, который и помещает панель содержимого в свой нижний слой.

Компоненты можно помещать в различные слои слоеной панели, получив ссылку на нее:

```
getLayeredPane().add(toolBar, JLayeredPane.PALETTE_LAYER);
```

На самом верху корневой панели, выше слоеной панели, лежит невидимая *"прозрачная панель"* (glass pane). На самом деле это не панель, а экземпляр класса `Component`, следовательно, на нее нельзя поместить компоненты. Она служит, главным образом, для обработки событий мыши.

Дело в том, что обычно действия мыши обрабатываются компонентом, над которым расположен ее курсор, точнее, обработчиком событий мыши `MouseEvent`, присоединенным к этому компоненту. Такой обработчик можно присоединить к прозрачной панели и обрабатывать события мыши одинаково на всей корневой панели, независимо от того, над каким компонентом расположен курсор мыши. Это можно сделать, например, так:

```
getGlassPane().addMouseInputListener(this);
```

Кроме того, прозрачная панель удобна для рисования. Линии и фигуры, нарисованные на ней, будут видны поверх всех компонентов, свободно пересекая их границы. По умолчанию прозрачная панель невидима. Для того чтобы рисунки, сделанные на ней, были видны, надо обратиться к методу `setVisible(true)`.

Всеми панелями корневой панели распоряжается специально разработанный менеджер размещения. Его замена может привести к нарушению взаимодействия компонентов, находящихся на корневой панели. Но всегда можно создать новый экземпляр панели содержимого, слоеной панели или прозрачной панели и поместить его в корневую панель методами `setContentPane(Container)`, `setLayeredPane(JLayeredPane)` и `setGlassPane(Component)`.

Очень часто разработчика не устраивает то, что панель содержимого — это простой контейнер библиотеки AWT, экземпляр класса `Container`. Ее можно легко заменить панелью другого типа, например:

```
JFrame fr = new JFrame("Окно верхнего уровня");
JPanel c = new JPanel();
c.add(new JTextField(50));
c.add(new JButton("OK"));
fr.setContentPane(c);
```

Обычно окно, в котором расположена корневая панель, оформляется по правилам текущего оконного менеджера графической оболочки операционной системы. Но метод `setWindowDecorationStyle(int)` позволяет задать другой стиль оформления. Аргумент этого метода может принимать одно из значений `NONE` (по умолчанию), `FRAME`, `PLAIN_DIALOG`, `INFORMATION_DIALOG`, `ERROR_DIALOG`, `COLOR_CHOOSER_DIALOG`, `FILE_CHOOSER_DIALOG`, `QUESTION_DIALOG`, `WARNING_DIALOG`. Подробнее о стилях оформления Look and Feel написано в *главе 17*.

Окно *JWindow*

Класс `JWindow` представляет простейшее окно верхнего уровня, которое может располагаться в любом месте экрана. Класс `JWindow` расширяет класс `Window`, применяемый в библиотеке AWT. Это один из четырех "тяжелых" компонентов библиотеки Swing. Возможности окна класса `JWindow` невелики. У него нет рамки, строки заголовка с кнопками. Оно не регистрируется у оконного менеджера операционной системы, следовательно, не перемещается по экрану и не изменяет свои размеры. Чаще всего окно класса `JWindow` используют как *предварительное окно* (splash window), появляющееся на экране на время загрузки основного окна приложения.

Окно `JWindow` может быть связано с уже существующим, *"родительским"* (parent) окном. Для этого его надо создать конструктором `JWindow(Frame)`, `JWindow(Window)` или `JWindow(Window, GraphicsConfiguration)`. На такое окно можно передать фокус ввода и, например, вводить текст в поле ввода, находящееся в окне, передав затем введенный текст родительскому окну. Окно располагается сверху родительского окна.

Если окно создано конструктором `JWindow()` или `JWindow(GraphicsConfiguration)`, то оно не связано с родительским окном. На него нельзя передать фокус ввода, следовательно, в нем нельзя ничего сделать. Оно может скрываться под другими окнами.

Окно JWindow содержит непосредственно всего один компонент — корневую панель класса JRootPane. На эту корневую панель и кладутся все компоненты, как описано в предыдущем пункте. Можно получить ссылку на корневую панель методом getRootPane(), но в классе JWindow есть методы прямого доступа к панели содержимого getContentPane(), слоеной панели getLayeredPane() и к прозрачной панели getGlassPane(). Разумеется, эти методы обращаются к соответствующим методам корневой панели. Вот, например, исходный код одного из этих методов:

```
public Container getContentPane(){
   return getRootPane().getContentPane();
}
```

Окно, как и всякий контейнер, наследует метод setLayout(LayoutManager), но замена менеджера размещения может привести к разрушению структуры окна. Поэтому метод setLayout() переопределен так, что он проверяет значение флага — защищенного логического поля rootPaneCheckingEnabled — и только если значение этого поля false, меняет менеджер размещения. По умолчанию значение этого поля true, изменить его можно методом setRootPaneCheckingEnabled(boolean). Данный метод *защищен* (protected), он предназначен для использования при расширении класса JWindow.

Такое же правило действует при попытке добавить компонент унаследованными методами add(), addImpl() непосредственно в окно JWindow, минуя находящийся в нем объект класса JRootPane.

Тем не менее всегда можно заменить всю корневую панель методом setRootPane(JRootPane) или ее панели методами setContentPane(Container), setLayeredPane(JLayeredPane) и setGlassPane(Component).

Унаследованный от класса Window метод dispose() уничтожает окно, освобождая все занятые им ресурсы.

Диалоговое окно *JDialog*

Класс JDialog расширяет класс Dialog библиотеки AWT *(см. главу 10)* и является "тяжелым" компонентом. Он создает модальные (modal) или немодальные диалоговые окна. Из модального окна нельзя удалить фокус ввода, не проделав все находящиеся в нем действия.

Каждое диалоговое окно обязательно связано с родительским окном класса Window, Dialog или Frame. Даже конструктор по умолчанию JDialog() создает скрытое родительское окно.

Конструкторы JDialog(Frame), JDialog(Frame, String), JDialog(Dialog), JDialog(Dialog, String) JDialog(Window), JDialog(Window, String) создают немодальные диалоговые окна с заголовком или без заголовка.

Конструкторы JDialog(Frame, boolean), JDialog(Frame, String, boolean), JDialog(Dialog, boolean), JDialog(Dialog, String, boolean) создают модальные диалоговые окна типа модальности DEFAULT_MODALITY_TYPE, если последний параметр равен true, с заголовком или без заголовка.

Конструкторы JDialog(Window, Dialog.ModalityType), JDialog(Window, String, Dialog.ModalityType) создают диалоговые окна с заданной модальностью.

Модальность окна и его заголовок можно изменить унаследованными методами setModalityType(Dialog.ModalityType) и setTitle(String).

Диалоговое окно, как и окно `JWindow`, непосредственно содержит только один компонент — корневую панель `JRootPane` и точно так же дает непосредственный доступ к панелям корневой панели методами `getContentPane()`, `getLayeredPane()`, `getGlassPane()` и к самой корневой панели методом `getRootPane()`.

Все компоненты следует помещать на панели, расположенные в корневой панели. Относительно помещения компонентов непосредственно в диалоговое окно применяется та же политика, что и для окна `JWindow`.

Так же как и для окна `JWindow`, для диалогового окна `JDialog` можно подготовить другие экземпляры панелей и установить их методами `setRootPane(JRootPane)`, `setContentPane(Container)`, `setLayeredPane(JLayeredPane)` и `setGlassPane(Component)`.

Диалоговое окно снабжено рамкой и строкой заголовка, в которую помещается строка, записанная в конструкторе. В строке заголовка есть кнопка **Закрыть**, реакцию на которую, а заодно и реакцию на нажатие комбинации клавиш <Alt>+<F4>, можно установить методом

```
setDefaultCloseOperation(int);
```

Реакция задается одной из трех констант:

- `DO_NOTHING_ON_CLOSE` — отсутствие всякой реакции;
- `HIDE_ON_CLOSE` — окно становится невидимым (по умолчанию);
- `DISPOSE_ON_CLOSE` — окно ликвидируется, освобождая оперативную память.

Если разработчик хочет задать какую-нибудь другую реакцию на попытку закрыть окно, то ему сначала надо отключить стандартную реакцию, например:

```
setDefaultCloseOperation(JDialog.DO_NOTHUNG_ON_CLOSE);
addWindowListener(new WindowAdapter(){
   public void windowClosing(WindowEvent e){
      ta.setText(tf.getText()); dispose();
   }
});
```

По умолчанию диалоговое окно может изменять свои размеры, но это правило можно поменять унаследованным методом `setResizable(boolean)`.

По умолчанию диалоговое окно появляется в рамке, со строкой заголовка, оформленными по правилам графической оболочки операционной системы. Данное оформление можно убрать методом `setUndecorated(true)`, но вместе с этим будет потеряна и стандартная реакция на действия мыши с элементами оформления. После применения этого метода можно установить оформление текущего Look and Feel методом `setWindowDecorationStyle(int)` класса `JRootPane`. Подробно о Look and Feel написано в *главе 17*.

Очень часто диалоговые окна создаются для вывода сообщений, предупреждений, для подтверждения или отмены каких-то действий. В библиотеке Swing в классе `JOptionPane` собрана богатая коллекция готовых диалоговых окон. Речь о них пойдет немного позднее.

Окно верхнего уровня *JFrame*

Класс `JFrame` расширяет класс `Frame` графической библиотеки AWT. Это полноценное самостоятельное окно верхнего уровня, снабженное рамкой и строкой заголовка с кнопками

системного меню **Свернуть**, **Развернуть** и **Закрыть**, как принято в графической оболочке операционной системы.

Конструкторы класса `JFrame()`, `JFrame(String)` создают невидимое окно с заголовком или без заголовка. Чтобы вывести его на экран, надо воспользоваться методом `setVisible(true)`.

Окно `JFrame` непосредственно содержит только один компонент — корневую панель класса `JRootPane`. К нему относится все, что сказано в предыдущих разделах, за исключением модальности и родительского окна.

Реакция на щелчок кнопкой мыши по кнопке закрытия окна тоже определяется методом `setDefaultCloseOperation(int)`, но параметр может принять еще одно, четвертое значение: `EXIT_ON_CLOSE` — завершить работу приложения методом `System.exit(0)`. Это значение не следует применять в апплетах.

Напомним еще, что окно `JFrame` наследует от класса `Frame` возможность заменить ярлычок кнопки системного меню — дымящуюся чашечку кофе — другим ярлычком методом `setIconImage(Image)`.

Как и в диалоговых окнах, можно убрать оформление окна, выполненное по правилам оконного менеджера графической оболочки операционной системы, методом `setUndecorated(true)`, установив затем оформление текущего Look and Feel методом

```
getRootPane().setWindowDecorationStyle(JRootPane.FRAME);
```

Подробнее об этом написано в *главе 17*.

Внутреннее окно *JInternalFrame*

Класс `JInternalFrame` создает окно, очень похожее на окно класса `JFrame`, но существующее только внутри другого окна, обычно окна класса `JDesktopPane`. Его можно перемещать, менять размеры, сворачивать и разворачивать, но все это можно делать, не выходя за пределы объемлющего окна.

Поскольку внутреннее окно не зависит от операционной системы, а создается полностью средствами Java, оно получает по умолчанию Java L&F. В отличие от окна `JFrame` программа сама может управлять окном класса `JInternalFrame`: внутреннему окну можно разрешить или запретить менять размеры, сворачиваться в ярлык и разворачиваться на все объемлющее окно, закрываться. По умолчанию все эти четыре свойства отсутствуют. У внутреннего окна, созданного конструктором по умолчанию `JInternalFrame()`, отсутствуют кнопки **Свернуть**, **Развернуть**, **Закрыть**, при установке курсора мыши на границу окна курсор не меняет свой вид и не позволяет менять размеры окна, заголовок отсутствует. Кроме того, окно по умолчанию невидимо.

Возможности изменения окна устанавливаются конструкторами класса или методами `setClosable(boolean)`, `setTitle(String)`, `setIconifiable(boolean)`, `setMaximizable(boolean)`, `setResizable(boolean)`, `setVisible(boolean)`.

Основной конструктор класса регулирует все эти возможности:

```
JInternalFrame(String title, boolean resizable, boolean closable,
               boolean maximizable, boolean iconifiable)
```

У остальных конструкторов отсутствует один или несколько параметров, отсутствующие параметры получают значение `false`. На рис. 14.11 показаны два внутренних окна, созданных разными конструкторами. Первое окно сдвинуто влево вниз, оно частично обрезано границами внешнего окна. В листинге 14.11 приведена программа, создающая эти окна.

Листинг 14.11. Внутренние окна

```
import java.awt.*;
   import javax.swing.*;

   public class IntFrame extends JFrame{

  IntFrame(){
    super(" Окно с внутренними окнами");
    c = getContentPane();  // Излишне для JDK 5.0 и выше.
       c.setLayout(new FlowLayout());

    JInternalFrame ifr1 =
      new JInternalFrame(" Первое окно", true, true, true, true);
    ifr1.getContentPane().add(new JLabel(" Это первое внутреннее окно"));
    ifr1.setPreferredSize(new Dimension(200, 200));
       ifr1.setVisible(true);

       c.add(ifr1);

    JInternalFrame ifr2 = new JInternalFrame(" Второе окно");
    ifr2.getContentPane().add(new JButton("Это второе внутреннее окно"));
    ifr2.setPreferredSize(new Dimension(200, 200));
       ifr2.setVisible(true);

       c.add(ifr2);

    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
     }

  public static void main(String[] args){
    new IntFrame();
  }
   }
```

Как и все окна Swing, внутреннее окно `JInternalFrame` содержит всего один компонент — корневую панель `JRootPane` — и обладает всеми методами работы с ней, перечисленными в предыдущих разделах.

К внутреннему окну можно применить все методы окна `JFrame`, за исключением остановки JVM — метод `setDefaultCloseOperation(int)` не принимает параметр `EXIT_ON_CLOSE`.

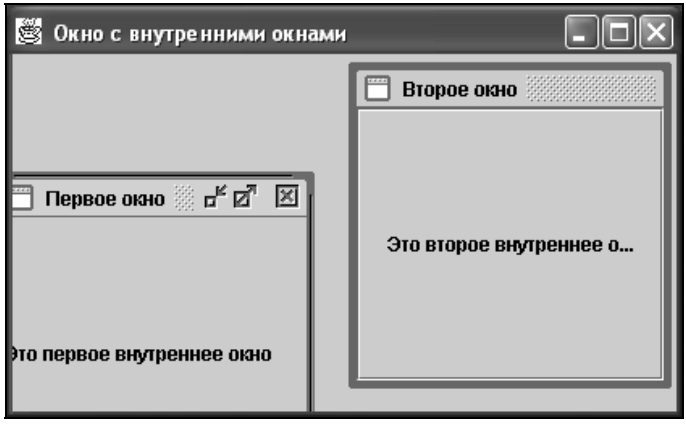


Рис. 14.11. Внутренние окна

Рабочий стол *JDesktopPane*

Хотя внутренние окна можно поместить в любой контейнер, удобнее всего расположить их в специально разработанном внутреннем "рабочем столе" `JDesktopPane`. На нем размещаются окна верхнего уровня подобно тому, как на экране располагаются окна приложений.

Класс `JDesktopPane` расширяет класс `JLayeredPane`, следовательно, имеет множество слоев, благодаря чему компоненты могут перекрываться, но не управляется никаким менеджером размещения. Поэтому позицию и размеры компонента следует задавать методом `setBounds()`, как показано в листинге 14.12. На рис. 14.12 представлен вывод программы листинга 14.12.

Листинг 14.12. Внутренние окна на внутреннем рабочем столе

```
import java.awt.*;
   import javax.swing.*;

   public class Desk extends JFrame{

  Desk(){
       super(" Внутренний рабочий стол");

       JDesktopPane dp = new JDesktopPane();

       setContentPane(dp);

     JInternalFrame ifr1 =
       new JInternalFrame(" Первое окно", true, true, true, true);
     ifr1.getContentPane().add(new JLabel(" Это первое внутреннее окно"));
     ifr1.setBounds(10,10, 200,200);
        ifr1.setVisible(true);
```

```
        dp.add(ifr1);

    JInternalFrame ifr2 = new JInternalFrame(" Второе окно");
    ifr2.getContentPane().add(new JButton("Это второе внутреннее окно"));
    ifr2.setBounds(150, 200, 200, 100);
        ifr2.setVisible(true);

        dp.add(ifr2);

    setSize(400, 400);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
     }

  public static void main(String[] args){
    new Desk();
  }
}
```

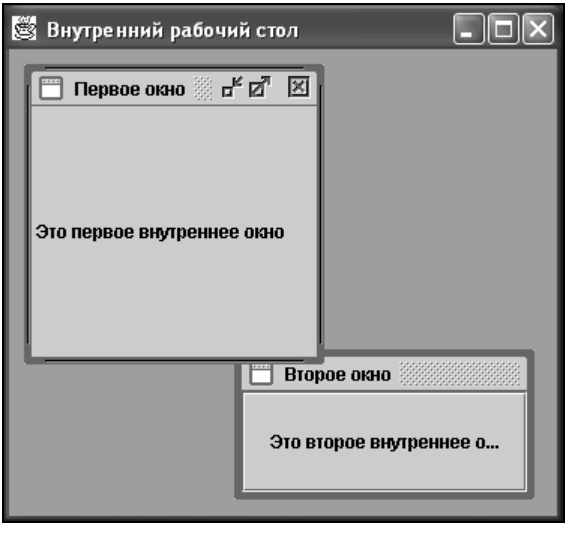


Рис. 14.12. Внутренний рабочий стол

Перемещение внутреннего окна внутри содержащего его контейнера можно ускорить, если заменить перерисовку всего окна во время перемещения перерисовкой только рамки методом

```
setDragMode(JDesktopPane.OUTLINE_DRAG_MODE);
```

Стандартные диалоги *JOptionPane*

Хотя класс `JDialog` позволяет создавать любые диалоговые окна, в подавляющем большинстве случаев их назначение сводится к выводу *сообщений* (message), подтверждению или отмене каких-то *действий* (confirm) и вводу коротких *сведений* (input).

Класс `JOptionPane` предоставляет более двадцати статических методов для создания модальных диалоговых окон. Это три метода `showMessageDialog()` с разными аргументами, создающие окна сообщений, четыре метода `showConfirmDialog()`, формирующие окна подтверждения, шесть методов `showInputDialog()`, создающие окна ввода, и один метод, создающий диалоговое окно общего вида — `showOptionDialog()`. Перечисленные методы создают диалоговые окна класса `JDialog`. Еще более десяти методов вида `showInternalXxxDialog()` предназначено для формирования внутренних диалоговых окон класса `JInternalFrame`.

Окна сообщений, в свою очередь, бывают нескольких типов, снабжаемых определенными ярлыками и обозначаемых следующими константами:

- `INFORMATION_MESSAGE` — просто сообщение, в Java L&F это синий круг с буквой "i" внутри, как на рис. 14.20 (по умолчанию);
- `WARNING_MESSAGE` — предупреждение, желтый треугольник с восклицательным знаком внутри, как на рис. 14.15;
- `ERROR_MESSAGE` — сообщение об ошибке, красный шестиугольник с "кирпичом" внутри, как на рис. 14.13;
- `QUESTION_MESSAGE` — вопрос, зеленый квадрат с вопросительным знаком внутри, как на рис. 14.14;
- `PLAIN_MESSAGE` — произвольное сообщение без определенного ярлыка.

Сообщение любого типа можно снабдить произвольным ярлыком типа `Icon`, как на рис. 14.16. Это делается самым развитым из данной группы методов статическим методом

```
static void showMessageDialog(Component parent, Object message,
                              String title, int type, Icon icon);
```

Для работы указанного метода задается родительское окно `parent`, строка заголовка `title`, сообщение `message`, его тип `type` — одна из констант, перечисленных ранее, и ярлык `icon`.

Как видите, сообщение `message` задается объектом самого общего типа `Object`. В качестве сообщения может появиться компонент класса `Component`, изображение типа `Icon`. Все остальные объекты преобразуются в строку методом `toString()`, строка помещается в созданный объект класса `JLabel`, который и выводится в окно сообщения. Параметром `message` может быть и массив типа `Object[]`. В этом случае в окно будет выведено несколько сообщений.

На рис. 14.13 показано окно сообщения, созданное методом

```
JOptionPane.showMessageDialog(this,
   " В поле \"Код\" могут быть только цифры.",
   " Ошибка", JOptionPane.ERROR_MESSAGE);
```

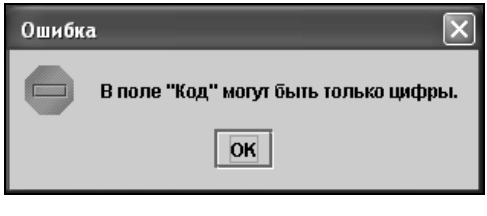


Рис. 14.13. Окно сообщения типа "Ошибка"

Окно подтверждения или отмены действий содержит две или три кнопки: **Yes** (Да), **No** (Нет), **Cancel** (Отмена). Это обозначается константами `YES_NO_OPTION` или `YES_NO_CANCEL_OPTION` и регулируется параметром `optType` методов `showConfirmDialog()`. Сигнатура метода этого типа с самым большим числом параметров выглядит так:

```
static int showConfirmDialog(Component parent, Object message,
   String title, int optType, int messType, Icon icon);
```

Метод возвращает одну из констант `YES_OPTION`, `NO_OPTION`, `CANCEL_OPTION` в зависимости от того, какая кнопка была нажата. При закрытии окна без всякого выбора возвращается значение `CLOSED_OPTION`. На рис. 14.14 показано окно подтверждения, используемое, например, так:

```
int n = JOptionPane.showConfirmDialog(this,
    "Сохранить этот/предыдущий вариант (Yes/No)?",
    "  Сохранение документа", JOptionPane.YES_NO_CANCEL_OPTION,
     JOptionPane.QUESTION_MESSAGE);
switch(n){
   case JOptionPane.YES_OPTION: saveDoc(); break;
   case JOptionPane.NO_OPTION: restore(); saveDoc(); break;
   case JOptionPane.CANCEL_OPTION:
   case JOptionPane.CLOSED_OPTION: break;
   default:
}
```

Диалоговое окно ввода содержит поле для ввода краткого ответа, возвращаемого методом, или список выбора, а также кнопки **OK** и **Cancel** (Отмена). На рис. 14.15 показано окно, созданное методом

```
String s = (String)JOptionPane.showInputDialog(this,
   " Запишите ответ: ", " Ответ", JOptionPane.WARNING_MESSAGE);
```

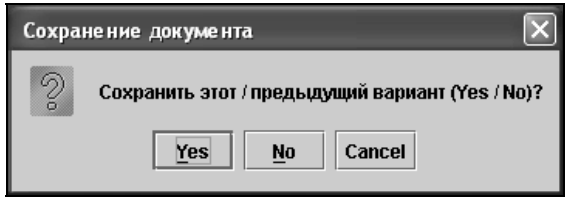


Рис. 14.14. Окно подтверждения типа "Вопрос"

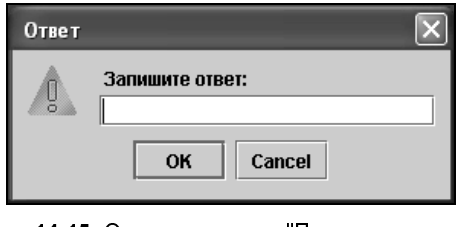


Рис. 14.15. Окно ввода типа "Предупреждение"

На рис. 14.16 показано окно, созданное методами

```
   String[] vars = {"Первый", "Второй", "Третий"};

String s = (String)JOptionPane.showInputDialog(this,
   "Выберите вариант ответа:", " Варианты ответа",
    JOptionPane.QUESTION_MESSAGE, new ImageIcon("bird.gif"),
    vars, "Второй");
```

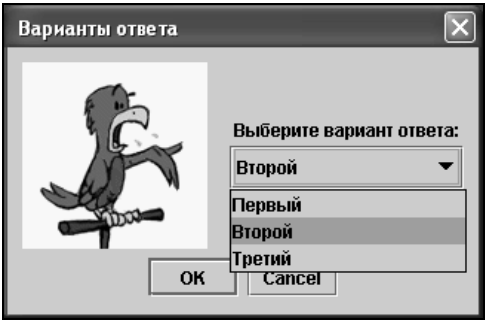


Рис. 14.16. Окно ввода с вариантами ответа

Для российских программистов, вечно озабоченных русификацией своих приложений, удобнее четвертый тип стандартных диалоговых окон, создаваемый методом

```
static int showOptionDialog(Component parent, Object message,
   String title, int optType, int messType, Icon icon,
   Object[] options, Object init);
```

Предпоследний параметр `options` задает надписи на кнопках диалогового окна или графические компоненты, выводимые в окно вместо кнопок. Последний параметр `init` выделяет одну из кнопок или графических компонентов.

Например, окно, показанное на рис. 14.14, будет лучше выглядеть, если его создать методами

```
   String[] vars = {"Этот", "Предыдущий", "Не сохранять"};

int n = JOptionPane.showOptionDialog(this,
   " Сохранить этот или предыдущий вариант?",
   " Сохранение документа", JOptionPane.YES_NO_CANCEL_OPTION,
    JOptionPane.QUESTION_MESSAGE, null, vars, "Этот");
```

как показано на рис. 14.17.

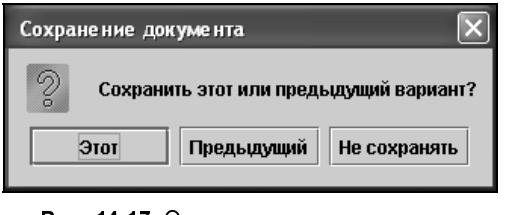


Рис. 14.17. Окно с русскими надписями

Как видно из рисунков, каждое стандартное диалоговое окно содержит элементы: предопределенный выбранным L&F или собственный ярлык, сообщение, кнопки и, может быть, поле ввода. Если такое строение диалогового окна не устраивает разработчика, то он может создать собственное диалоговое окно класса `JDialog`, в которое поместить диалоговую панель `JOptionPane`. Для этого имеется семь конструкторов. Конструктор по умолчанию `JOptionPane()` создает диалоговую панель с кнопкой **OK** и стандартной строкой сообщения. Она показана на рис. 14.18. Этот рисунок создан методами

```
JOptionPane op = new JOptionPane();
JDialog d = op.createDialog(this, " Простейшее диалоговое окно");
d.setVisible(true);
```

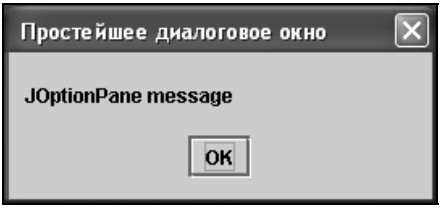


Рис. 14.18. Простейшее диалоговое окно

Конструктор с наибольшим числом параметров выглядит так:

```
JOptionPane(Object message, int messType, int optType,
   Icon icon, Object[] options, Object init);
```

у остальных конструкторов приняты значения по умолчанию отсутствующих параметров.

На рис. 14.19 представлено окно, созданное методами

```
   String[] opts = {"Применить", "Отменить", "Перейти", "Завершить"};
JOptionPane op = new JOptionPane(
   "<html><font size=+2 >Дальнейшие действия?",
   JOptionPane.QUESTION_MESSAGE,
   JOptionPane.YES_NO_CANCEL_OPTION,
      null, opts, opts[2]);
   JDialog d = op.createDialog(this, " Собственное диалоговое окно");
d.setVisible(true);
```

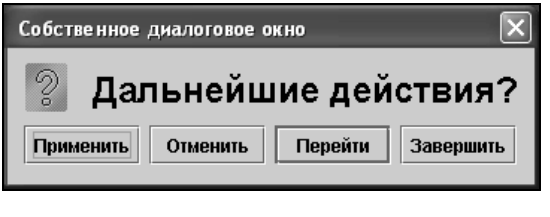


Рис. 14.19. Собственное диалоговое окно

Окно с индикатором *ProgressMonitor*

Еще один вид диалоговых окон, предназначенный для слежения за протеканием какого-нибудь процесса, предоставляет класс `ProgressMonitor`. Окно этого класса показывает сообщение, индикатор-"градусник" — объект класса `JProgressBar` — и кнопки **OK** и **Cancel**.

Единственный конструктор класса

```
ProgressMonitor(Component parent, Object message, String note, int min,
int max);
```

кроме ссылки `parent` на родительское окно, сообщения `message`, наименьшего `min` и наибольшего `max` значений "градусника" содержит параметр `note`. Это строка, значение которой можно менять во время ожидания методом `setNote(String)`.

Для смены значения индикатора выполняемый процесс должен обращаться к методу `setProgress(int pos)`, задавая в нем текущее значение `pos`, лежащее между значением `min` и значением `max`. Это похоже на работу с классом `JProgressBar`, описанным в *главе 11*.

В листинге 14.13 приведен пример использования окна индикатора. Для изменения значения индикатора запущен простейший подпроцесс. Рисунок 14.20 показывает вывод программы листинга 14.13.

Листинг 14.13. Создание окна индикатора

```
import java.awt.*;
   import javax.swing.*;

   public class Progress extends JFrame{

  Progress(){
      super("  Progress...");

   final ProgressMonitor mon = new ProgressMonitor(this,
         "Идет процесс.", "Осталось ", 0, 100);

      Runnable runnable = new Runnable(){

       public void run(){

      for (int i = 1; i < 100; i++){
        try{
          mon.setNote( "Осталось " + (100 - i) + " %");
             mon.setProgress(i);

          if (mon.isCanceled()){
            mon.setProgress(100);
            break;
          }
             Thread.sleep(100);
```

```
        }catch(InterruptedException e){}
         }

      mon.close();
    }
     };

    Thread thread = new Thread(runnable);
       thread.start();

    setSize(400, 400);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
     }

  public static void main(String[] args){
    new Progress();
  }
}
```

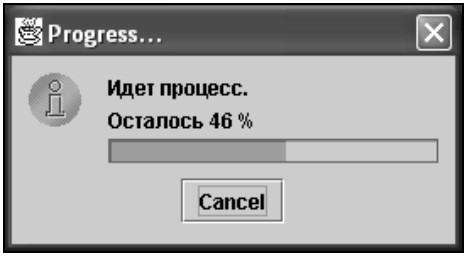


Рис. 14.20. Окно индикатора

В заключение следует сказать, что следить за загрузкой большого входного потока класса `InputStream` удобно с помощью специально разработанного фильтра входного потока — класса `ProgressMonitorInputStream`, расширяющего класс `FilterInputStream`.

Заключение

Все менеджеры размещения написаны полностью на языке Java, в состав Sun Java SE входят их исходные тексты. Если вы решили написать свой менеджер размещения, реализовав интерфейс `LayoutManager` или `LayoutManager2`, то посмотрите эти исходные тексты.

Вопросы для самопроверки

1. Что такое менеджер размещения?
2. Почему менеджеры размещения удобнее абсолютной расстановки компонентов?
3. В каких контейнерах можно установить менеджер размещения?

4. В каких компонентах можно установить менеджер размещения?
5. Почему менеджер размещения `BorderLayout` столь популярен?
6. Какой менеджер размещения установлен по умолчанию в окне класса `JFrame`?
7. Какой менеджер размещения установлен по умолчанию в классе `JPanel`?
8. Какой менеджер размещения установлен по умолчанию в классе `JScrollPane`?
9. Какой менеджер размещения установлен по умолчанию в классе `JWindow`?
10. Какой менеджер размещения установлен по умолчанию в классе `JDialog`?
11. Можно ли написать свой собственный менеджер размещения?



Глава 15

Обработка событий

В предыдущих главах мы написали много программ, создающих компоненты интерфейса с пользователем, но, собственно говоря, 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, т. е. 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 AWT показана на рис. 15.1. Все классы, отображенные на рисунке, кроме класса `AWTEvent`, собраны в пакет `java.awt.event`.

- События типа `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent` возникают во всех компонентах.
- События типа `ContainerEvent` — в контейнерах класса `Container`, а значит, и во всех компонентах графической библиотеки Swing.
- События типа `WindowEvent` возникают в окнах класса `Window` и в его наследниках.
- События типа `TextEvent` генерируются только в компонентах `TextComponent`, `TextArea`, `TextField`.
- События типа `ActionEvent` проявляются в компонентах `Button`, `List`, `TextField`, `JComboBox`, `JTextField`, кнопках класса `AbstractButton` и его наследниках.
- События типа `ItemEvent` возникают в компонентах `Checkbox`, `JCheckbox`, `Choice`, `JComboBox`, `List` и в кнопках класса `AbstractButton` и его наследниках.
- Наконец, события типа `AdjustmentEvent` возникают только в полосах прокрутки `Scrollbar` и `JScrollBar`.

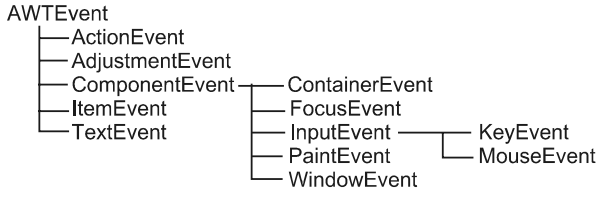


Рис. 15.1. Иерархия классов, описывающих события AWT

Узнать, в каком объекте произошло событие, можно методом `getSource()` класса `EventObject`. Этот метод возвращает ссылку на объект типа `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.

Графическая библиотека Swing добавляет еще несколько классов-событий, собранных в пакет `javax.swing.event`. Большинство этих классов наследуют напрямую от класса `EventObject`.

Событие нельзя обработать произвольно написанным методом. У каждого события есть свои методы, к которым обращается исполняемая система Java при его возникновении. Они описаны в интерфейсах-*слушателях* (listener). Для каждого показанного на рис. 15.1 типа событий, кроме `InputEvent` (оно 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 *обратным вызовом* (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 ею 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 существуют 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 вида `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 прорыв водопроводной трубы, его не интересуют. При наступлении "своего" события он действует по контракту, записанному в методе обработки события.

> ***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 `JFrame` помещено поле ввода `tf` типа `JTextField`, нередактируемая область ввода `ta` типа `JTextArea` и кнопка `b` типа `JButton`. 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 есть всего один метод `actionPerformed()`, который надо определить.

Итак, пишем:

```
class TextMove implements ActionListener{
  private JTextField tf;
  private JTextArea ta;
  TextMove(JTextField tf, JTextArea 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 15.1.

Листинг 15.1. Обработка события `ActionEvent`

```
import java.awt.*;
import java.awt.event.*;
   import javax.swing.*;

   class MyNotebook extends JFrame{

  MyNotebook(String title){
       super(title);

       Container c = getContentPane();  // Излишне для JDK больше 1.4.

    JTextField tf = new JTextField("Вводите текст", 50);
       c.add(tf, BorderLayout.NORTH);

    JTextArea ta = new JTextArea();
    ta.setEditable(false);
       c.add(ta);

    JPanel p = new JPanel();
       c.add(p, BorderLayout.SOUTH);

    JButton b = new JButton("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){
    JFrame f = new MyNotebook("  Обработка ActionEvent");
    f.setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
}
// Текст класса TextMove
// ...
```

На рис. 15.2 показан результат работы с этой программой.

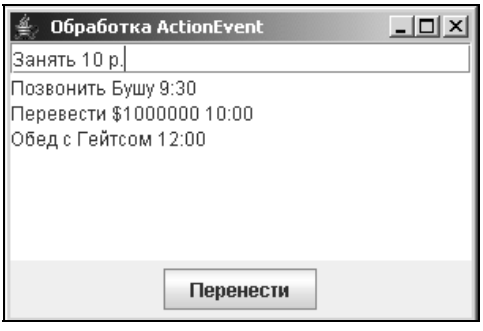


Рис. 15.2. Обработка события `ActionEvent`

В листинге 15.1 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.

Самообработка событий

Класс, содержащий источники события, может сам обрабатывать его. Возвращаясь к примеру с установкой охранной сигнализации, можно сказать, что вы способны 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 15.2.

Листинг 15.2. Самообработка события `ActionEvent`

```
import java.awt.*;
import java.awt.event.*;
   import javax.swing.*;

class MyNotebook extends JFrame implements ActionListener{
  private JTextField tf;
     private JTextArea ta;

  MyNotebook(String title){
       super(title);

       Container c = getContentPane();
```

```
    tf = new JTextField("Вводите текст", 50);
       c.add(tf, BorderLayout.NORTH);

    ta = new JTextArea();
    ta.setEditable(false);
       c.add(ta);

    JPanel p = new JPanel();
       c.add(p, BorderLayout.SOUTH);

    JButton b = new JButton("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){
    JFrame f = new MyNotebook("  Обработка ActionEvent");
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}
```

Здесь поля `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`.

Обработка вложенным классом

Еще один вариант — сделать обработчик вложенным классом. Это позволяет обойтись без переменных экземпляра и конструктора в классе-обработчике `TextMove`, как показано в листинге 15.3.

Листинг 15.3. Обработка вложенным классом

```
import java.awt.*;
import java.awt.event.*;
   import javax.swing.*;
```

```
class MyNotebook extends JFrame{
  private JTextField tf;
     private JTextArea ta;

  MyNotebook(String title){
    super(title);
    Container c = getContentPane(); // Излишне для JDK выше 1.4.
    tf = new JTextField("Вводите текст", 50);
    add(tf, BorderLayout.NORTH);
    ta = new JTextArea();
    ta.setEditable(false);
    c.add(ta);
    JPanel p = new JPanel();
    add(p, BorderLayout.SOUTH);
    JButton b = new JButton("Перенести");
       p.add(b);

    tf.addActionListener(new TextMove());
       b.addActionListener(new TextMove());

    setSize(300, 200);
    setVisible(true);
  }
  public static void main(String[] args){
    JFrame f = new MyNotebook("  Обработка ActionEvent");
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
  // 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 AWT. 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 позднее.

Перейдем к детальному рассмотрению разных типов событий.

Упражнение

1. Реализуйте обработку события безымянным вложенным классом.

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:

- `getActionCommand()` — возвращает в виде строки `String` надпись на кнопке `JButton`, точнее, то, что установлено методом `setActionCommand(String s)` класса `JButton`, выбранный пункт меню или списка `JList`, или что-то другое, зависящее от компонента;
- `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>.

Вот как, например, можно отследить нажатие клавиши <Alt> при щелчке по кнопке **Open**:

```
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 параметра `e` координаты курсора мыши в системе координат компонента методами `e.getX()`, `e.getY()` или одним методом `e.getPoint()`, возвращающим экземпляр класса `Point`. Координаты курсора мыши относительно всего экрана, определяемые объектом класса `GraphicsConfiguration`, можно получить методами `e.getXOnScreen()`, `e.getYOnScreen()` или `e.getLocationOnScreen()`.

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 15.4 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. 15.3 показан пример работы с этой программой.

Листинг 15.4. Простейшая программа рисования

```
import java.awt.*;
import java.awt.event.*;
   import javax.swing.*;

   public class ScribbleTest extends JFrame{

  public ScribbleTest(String s){
       super(s);

       Container c = getContentPane(); // Излишне для JDK выше 1.4.

    Scribble scr = new Scribble(this, 500, 500);
    JScrollPane pane = new JScrollPane(scr);
    pane.setSize(300, 300);
       c.add(pane, BorderLayout.CENTER);

    JPanel p = new JPanel();
       c.add(p, BorderLayout.SOUTH);
```

```
    JButton b1 = new JButton("Красный");  p.add(b1);
    b1.addActionListener(scr);
    JButton b2 = new JButton("Зеленый");  p.add(b2);
    b2.addActionListener(scr);
    JButton b3 = new JButton("Синий");    p.add(b3);
    b3.addActionListener(scr);
    JButton b4 = new JButton("Черный");   p.add(b4);
    b4.addActionListener(scr);
    JButton b5 = new JButton("Очистить"); p.add(b5);
       b5.addActionListener(scr);

    setDefaultCloseOperation(EXIT_ON_CLOSE);
    pack();
    setVisible(true);
  }
  public static void main(String[] args){
     new ScribbleTest("  \"Рисовалка\"");
  }
   }

class Scribble extends JPanel implements
        ActionListener, MouseListener, MouseMotionListener{
  protected int lastX, lastY, w, h;
  protected Color currColor = Color.black;
     protected JFrame f;

  public Scribble(JFrame 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. 15.3. 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 кнопки и перемещение мыши, и нам нужны были лишь методы `mousePressed()` и `mouseDragged()`. Для остальных методов мы задали пустые реализации.

Чтобы облегчить задачу реализации интерфейсов, имеющих более одного метода, созданы классы-адаптеры.

Упражнение

2. Реализуйте оставшиеся методы интерфейсов-слушателей событий мыши.

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 void mouseMoved(MouseEvent e){}
    public void mouseDragged(MouseEvent e){}
    public void mouseWheelMoved(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`.

Управление колесиком мыши

Недавно появившееся колесико, дополнившее среднюю кнопку мыши, оказалось очень удобным и быстро завоевало популярность. Его использование встроено в Java SE, начиная с версии JDK 1.4, и применяется, например, в контейнере `JScrollPane`. Если же разработчику нужно создать свой компонент, в котором прокрутка осуществляется колесиком мыши, то он может задать обработку события `MouseWheelEvent`.

Класс `MouseWheelEvent` расширяет класс `MouseEvent`, следовательно, содержит все его поля и методы. Дополнительно он различает два типа прокрутки: прокрутку блока, отмечаемую статическим полем `WHEEL_BLOCK_SCROLL`, и прокрутку одной единицы — статическое поле `WHEEL_UNIT_SCROLL`. Тип прокрутки, а также смысл понятий "блок" и "единица" определяется графической оболочкой операционной системы и возвращается методом `getScrollType()`. Например, блок — это страница текста, а единица — одна строка.

Если тип прокрутки — `WHEEL_UNIT_SCROLL`, то метод `getUnitsToScroll()` возвращает количество прокрученных единиц с учетом направления вращения колесика: на себя — положительное число, от себя — отрицательное. Метод `getScrollAmount()` возвращает то же число без знака. Конкретное значение зависит от платформы.

Если же разработчик не хочет использовать понятие "единица", то он может методом `getWheelRotation()` отследить количество "щелчков", сделанных колесиком во время прокрутки. При этом положительное число возвращается при повороте колесика на себя и означает прокрутку вниз, а отрицательное — при повороте колесика от себя, прокрутка вверх.

Интерфейс `MouseWheelListener` описывает один метод

```
public void mouseWheelMoved(MouseWheelEvent e);
```

которому передается объект-событие, происшедшее в результате вращения колесика мыши. В листинге 15.5 приведен тренировочный пример работы с колесиком мыши.

Листинг 15.5. Событие колесика мыши

```
import java.awt.*;
import java.awt.event.*;
   import javax.swing.*;

public class WheelEv extends JFrame
               implements MouseWheelListener{

      JTextArea ta = new JTextArea(5, 30);

   WheelEv(){
     super(" Колесико мыши");
        getContentPane().add(ta);

        ta.addMouseWheelListener(this);

     setSize(400, 400);
     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     setVisible(true);
      }

  public void mouseWheelMoved(MouseWheelEvent e){
     if (e.getScrollType() == MouseWheelEvent.WHEEL_UNIT_SCROLL)
        ta.append("  Единицы = " + e.getUnitsToScroll() +
          "  Количество = "+ e.getScrollAmount() +
          "  Вращение = " +  e.getWheelRotation() + "\n");
      }

   public static void main(String[] args){
     new WheelEv();
   }
}
```

Рисунок 15.4 показывает конкретные значения, полученные этой программой в операционной системе Windows XP.



Рис. 15.4. Значения, полученные от колесика мыши

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 15.3 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){}
}
```

Упражнение

3. Реализуйте обработку нажатия различных клавиш.

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:

```
JTextComponent tc = (JTextComponent)e.getSource();
String s = tc.getText();
// Дальнейшая обработка
```

Событие изменения *ChangeEvent*

Еще одно событие общего характера ChangeEvent происходит при изменении состояния компонента Swing: щелчке кнопкой мыши на кнопке Swing любого типа, выборе пункта меню, движении ползунка и "градусника", смене страниц на панели со вкладками. Большинство этих компонентов реагируют на собственные события, более удобные для обработки, но для окна выбора цвета JColorChooser и ползунка JSlider это основное событие.

Поскольку событие класса `ChangeEvent` носит такой общий характер, в этом классе нет собственных методов, только унаследованный метод `getSource()`.

Слушатель этого события `ChangeListener` обладает всего одним методом:

```
public void stateChanged(ChangeEvent);
```

При обработке события следует получить ссылку на источник и извлечь из него всю необходимую информацию, например:

```
 import javax.swing.event.*;

 class SliderHandler implements ChangeListener{

private int threshold;
   private JLabel l;

public SliderHandler(int threshold, JLabel l){
  this.threshold = threshold; this.l = l;
   }

public void stateChanged(ChangeEvent e){
  JSlider sl = (JSlider)e.getSource();
  if (!sl.getValueIsAdjusting())
    if (threshold <= (int)sl.getValue())
      l.setText("Порог достигнут");
}
}
```

Обработка действий с окном

Событие `WindowEvent` может произойти по двенадцати причинам:

- окно открылось — идентификатор `WINDOW_OPENED`;
- окно закрылось — идентификатор `WINDOW_CLOSED`;
- попытка закрытия окна — идентификатор `WINDOW_CLOSING`;
- окно получило фокус — идентификатор `WINDOW_ACTIVATED`;
- процесс получения фокуса — идентификатор `WINDOW_GAINED_FOCUS`;
- окно потеряло фокус — идентификатор `WINDOW_DEACTIVATED`;
- процесс потери фокуса — идентификатор `WINDOW_LOST_FOCUS`;
- окно свернулось в ярлык — идентификатор `WINDOW_ICONIFIED`;
- окно развернулось — идентификатор `WINDOW_DEICONIFIED`;
- всякое изменение состояния окна — `WINDOW_STATE_CHANGED`;
- окно стало первым — `WINDOW_FIRST`;
- окно стало последним — `WINDOW_LAST`.

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 componentMoved(ComponentEvent e);
    public void componentResized(ComponentEvent e);
    public void componentHidden(ComponentEvent e);
    public void componentShown(ComponentEvent e);
}
```

Аргумент `e` методов этого интерфейса предоставляет ссылку на компонент-источник события методом `e.getComponent()`.

Событие *ContainerEvent*

Данное 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 перечеркивается 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*

Данное событие возникает при выборе или отказе от выбора элемента в списке `List`, `Choice`, `JComboBox`, или кнопки выбора `Checkbox`, а также в кнопках Swing, наследующих от класса `AbstractButton`, 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 *главе 10* мы уже видели примеры использования этого события.

Событие *AdjustmentEvent*

Это событие возникает для полосы прокрутки `Scrollbar` или `JScrollBar` 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 представлен в листинге 15.6.

Листинг 15.6. 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 15.1—15.3, мы привели пример класса `TextMove`, слушающего сразу два компонента: поле ввода `tf` типа `JTextField` и кнопку `b` типа `JButton`.

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 осуществляет peer-объект.

В классе `FlowerButton` листинга 10.7 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. Таким 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.

Задачу 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 *главе 10* мы уже применили такое переопределение в листинге 10.10 для вызова всплывающего меню.

Создание собственного события

Вы можете создать собственное событие и определить источник и условия его возникновения.

В листинге 15.7 приведен пример создания события `MyEvent`, любезно предоставленный Вячеславом Педаком.

Событие `MyEvent` говорит о начале работы программы (`START`) и окончании ее работы (`STOP`).

Листинг 15.7. 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.

Вопросы для самопроверки

1. Какая модель обработки событий выбрана в AWT?
2. Что нового добавила в обработку событий библиотека Swing?
3. Какие компоненты отслеживают события мыши?
4. Какие компоненты отслеживают события клавиатуры?
5. Может ли одно и то же событие возникнуть сразу в нескольких компонентах?
6. Может ли одно действие вызвать сразу несколько событий?
7. Можно ли сделать обработку нескольких событий одним методом?
8. Можно ли обработать одно событие сразу несколькими методами?
9. Как в AWT осуществляется диспетчеризация событий?



Глава 16

Оформление рамок

Каждый стандартный графический компонент — это прямоугольная область на экране со сторонами, параллельными сторонам экрана. Стороны прямоугольника выделяются каким-то образом на экране. Например, стороны кнопки `JButton` нарисованы так, что создают впечатление ее выпуклости. При нажатии кнопки мыши оформление сторон графической кнопки меняется, создавая впечатление ее "вдавленности".

Некоторые компоненты, например `JLabel`, вообще не оформляют свои границы.

В любом случае библиотека Swing позволяет изменить оформление границ всякого компонента, в том числе и контейнера, обведя его рамкой, причем рамка может быть самого разного вида.

Самые общие свойства всех рамок описаны интерфейсом `Border` из пакета `javax.swing.border`. Основное свойство — вычерчивание рамки методом

```
public void paintBorder(Component c, Graphics g,
                        int x, int y, int width, int height);
```

Здесь задается компонент `c`, который обводится рамкой, экземпляр `g` класса `Graphics`, обладающего методами рисования, и размеры рамки, которые обычно совпадают с размерами компонента, чуть больше или чуть меньше их.

Рамка может быть прозрачной или не прозрачной. Это отмечается логическим методом `isBorderOpaque()`.

Последний метод интерфейса, `getBorderInsets(Component c)`, возвращает пространство, занятое рамкой данного компонента `c`, в виде экземпляра класса `Insets`. Напомним, что в классе `Insets` это пространство определяется толщиной рамки сверху `top`, слева `left`, справа `right` и снизу `bottom`. Все четыре поля класса `Insets` — просто целочисленные переменные, и получить толщину рамки сверху можно так:

```
int d = b.getBorderInsets(this).top;
```

Интерфейс `Border` частично реализован абстрактным классом `AbstractBorder`, в котором сделана пустая реализация метода `paintBorder()`, метод `isBorderOpaque()` возвращает `false`, а метод `getBorderInsets()` — объект с нулевыми значениями. Кроме этих реализаций, в классе есть метод

```
public static Rectangle getInteriorRectangle(Component c, Border b,
         int x, int y, int width, int height);
```

который удобно использовать для определения размеров самого компонента без рамки.

Класс `AbstractBorder` расширяют около двадцати классов, вычерчивающих самые разнообразные рамки. Для удобства работы с ними в пакете `javax.swing` имеется класс `BorderFactory`, в котором собраны статические методы вида `createXxxBorder()` для разных типов рамок с разными параметрами. Чаще всего для создания рамки достаточно воспользоваться одним из этих методов, а затем установить полученную рамку в компонент методом `setBorder(Border)` класса `JComponent`. Например, на рис. 16.1 один из компонентов создан методами:

```
JLabel l2 = new JLabel("  LineBorder(Color.blue, 3)  ");
l2.setBorder(BorderFactory.createLineBorder(Color.blue, 3));
```

Рассмотрим подробнее некоторые типы рамок. Простые типы рамок показаны на рис. 16.1, созданном программой листинга 16.1.

Листинг 16.1. Простые рамки

```
import java.awt.*;
import javax.swing.*;
   import javax.swing.border.*;

   public class SimpBorders extends JFrame{

  SimpBorders(){
    super("  Простые рамки");
    Container c = getContentPane(); // Излишне для JDK выше 1.4.
       c.setLayout(new FlowLayout());

    JButton l1 = new JButton("  EmptyBorder()  ");
    l1.setBackground(Color.white);
       l1.setBorder(BorderFactory.createEmptyBorder());

    JLabel l2 = new JLabel("  LineBorder(Color.blue, 3)  ");
       l2.setBorder(BorderFactory.createLineBorder(Color.blue, 3));

    JLabel l3 = new JLabel("  BevelBorder(BevelBorder.RAISED)  ");
       l3.setBorder(BorderFactory.createBevelBorder(BevelBorder.RAISED));

    JLabel l4 = new JLabel("  BevelBorder(BevelBorder.LOWERED)  ");
       l4.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));

    JLabel l5 = new JLabel("  Объемная двухцветная рамка  ");
    l5.setBorder(BorderFactory.createBevelBorder(BevelBorder.RAISED,
           Color.black, Color.white, Color.black, Color.white));

    JLabel l6 = new JLabel("  EtchedBorder()  ");
       l6.setBorder(BorderFactory.createEtchedBorder());

    c.add(l1); c.add(l2); c.add(l3);
       c.add(l4); c.add(l5); c.add(l6);
```

```
    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
     }

  public static void main(String[] args){
    new SimpBorders();
  }
}
```

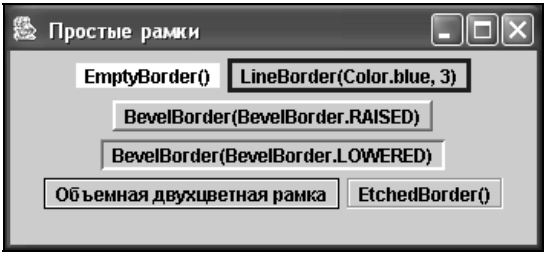


Рис. 16.1. Простые рамки

Пустая рамка *EmptyBorder*

Класс `EmptyBorder` представляет самую простую рамку. Это пустое пространство, окружающее компонент, как показано на рис. 16.1 сверху слева. Конструкторы класса

```
EmptyBorder(Insets);
EmptyBorder(int top, int left, int bottom, int right);
```

задают толщину рамки.

Статический метод `createEmptyBorder()` класса `BorderFactory` создает пустую рамку с нулевыми размерами, а статический метод

```
createEmptyBorder(int top, int left, int bottom, int right);
```

рамку с заданными размерами.

Рамка невидима — метод `isBorderOpaque()` возвращает `false`, метод `paintBorder()` не вычерчивает ничего. Метод `getBorderInsets()` без аргументов возвращает размеры рамки в виде экземпляра класса `Insets`.

Употребление пустой невидимой рамки сводится к определению экземпляра класса `EmptyBorder` и установке его в компонент методом `setBorder(Border)` класса `JComponent`.

Прямолинейная рамка *LineBorder*

Класс `LineBorder` определяет одноцветную рамку заданной толщины, одинаковой на всех сторонах рамки. Она показана на рис. 16.1 сверху справа. Рамка заданного цвета толщиной

в один пиксел создается конструктором `LineBorder(Color)`. Конструктор `LineBorder(Color, int)` определяет вторым аргументом толщину линий. Конструктор `LineBorder(Color, int, boolean)`, если третий аргумент равен `true`, создает рамку с закругленными краями.

Два статических метода, `createBlackLineBorder()` и `createGrayLineBorder()`, создают рамку с черными и серыми краями толщиной в один пиксел.

В классе `BorderFactory` есть два статических метода, аналогичные конструкторам класса `LineBorder`: метод `createLineBorder(Color)` и метод `createLineBorder(Color, int)`.

Объемная рамка *BevelBorder*

Рамка класса `BevelBorder` состоит из двух линий: светлой и темной. Если светлая линия расположена сверху и слева, а темная справа и снизу, то создается впечатление падения света сверху слева и компонент выглядит выпуклым. Это тип `RAISED`, он показан на рис. 16.1 во второй строке. Если же поменять местами темные и светлые линии, то компонент выглядит вдавленным в поверхность контейнера. Это тип `LOWERED`, на рис. 16.1 он показан в третьей строке. Именно так создается кнопка `JButton`.

Конструктор `BevelBorder(int type)` рисует рамку заданного типа `type` со светлыми линиями светлее фона контейнера и темными линиями темнее фона контейнера. Точно такие же рамки создаются статическими методами

```
createBevelBorder(int type);
createRaisedBevelBorder();
createLoweredBevelBorder();
```

класса `BorderFactory`.

Конструктор

```
BevelBorder(int type, Color highlight, Color shadow);
```

или статический метод

```
createBevelBorder(int type, Color highlight, Color shadow);
```

класса `BorderFactory` создают рамку с заданным светлым `highlight` и темным `shadow` цветом.

Объемная рамка может состоять из двойных линий разных цветов. Конструктор

```
BevelBorder(int type, Color highlightOuter, Color highlightInner,
                      Color shadowOuter, Color shadowInner);
```

или статический метод

```
createBevelBorder(int type, Color highlightOuter, Color highlightInner,
                            Color shadowOuter, Color shadowInner);
```

создают объемную двухцветную рамку. Внутренние линии имеют цвета `highlightInner` и `shadowInner`, а внешние — цвета `highlightOuter` и `shadowOuter`.

Закругленная объемная рамка *SoftBevelBorder*

Класс `SoftBevelBorder` расширяет класс `BevelBorder`, создавая рамки со слегка закругленными, смягченными краями. Такие рамки создаются тремя конструкторами, аналогичными конструкторам класса `BevelBorder`:

```
SoftBevelBorder(int type);
SoftBevelBorder(int type, Color highlight, Color shadow);
SoftBevelBorder(int type, Color highlightOuter, Color highlightInner,
                          Color shadowOuter, Color shadowInner);
```

Врезанная рамка *EtchedBorder*

Рамка класса `EtchedBorder` похожа на объемную рамку, но имеет такие тонкие границы, что компонент с этой рамкой выглядит врезанным в контейнер, чуть-чуть выступая, если задана константа `RAISED`, или чуть-чуть вдавливаясь, если задана константа `LOWERED`. Такая рамка показана на рис. 16.1 в нижней строке справа. Она характерна для "приборного" стиля Java L&F, ранее называвшегося “Metal”.

Стандартная врезанная рамка с цветами чуть светлее и чуть темнее цвета фона контейнера создается конструктором по умолчанию `EtchedBorder()` или статическим методом `createEtchedBorder()` класса `BorderFactory`.

Тип рамки `RAISED` или `LOWERED` задается конструктором `EtchedBorder(int)` или статическим методом `createEtchedBorder(int)`.

Цвета чуть выпуклой кнопки определяются конструктором

```
EtchedBorder(Color highlight, Color shadow);
```

или статическим методом

```
createEtchedBorder(Color highlight, Color shadow);
```

Наконец, можно задать и тип, и цвета конструктором

```
EtchedBorder(int type, Color highlight, Color shadow);
```

или статическим методом

```
createEtchedBorder(int type, Color highlight, Color shadow);
```

Рамка с изображением *MatteBorder*

Рамка класса `MatteBorder` может состоять из повторяющегося изображения, как показано на рис. 16.2 сверху, или из линий разной толщины, но одного и того же цвета, как показано на том же рисунке внизу.

Рамка с изображением создается конструктором `MatteBorder(Icon)`. При этом ширина рамки определяется величиной изображения.

Ширину рамки с изображением или цветом можно определить конструкторами

```
MatteBorder(Insets, Icon);
MatteBorder(Insets, Color);
```

```
MatteBorder(int top, int left, int bottom, int right, Icon);
MatteBorder(int top, int left, int bottom, int right, Color);
```

или статическими методами

```
createMatteBorder(int top, int left, int bottom, int right, Icon);
createMatteBorder(int top, int left, int bottom, int right, Color);
```

класса `BorderFactory`.

Рамки этого типа очень просты в использовании, но если они содержат изображения, то следует тщательно подбирать размеры рамки. Листинг 16.2 содержит программу, создавшую рис. 16.2.

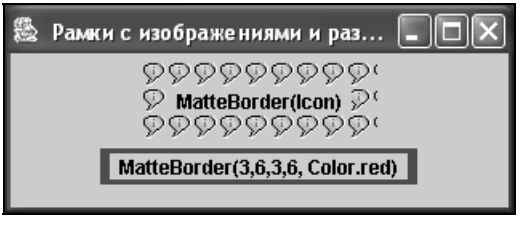


Рис. 16.2. Рамка с изображениями и рамка с линиями разной толщины

Листинг 16.2. Рамки с изображениями и разной толщины

```
import java.awt.*;
import javax.swing.*;
   import javax.swing.border.*;

   public class MatBorders extends JFrame{

  MatBorders(){
    super(" Рамки с изображениями и разной толщины");
    Container c = getContentPane(); // Излишне для JDK выше 1.4.
       c.setLayout(new FlowLayout());

    JLabel l1 = new JLabel("  MatteBorder(Icon)  ");
       l1.setBorder(new MatteBorder(new ImageIcon("about16.gif")));

    JLabel l2 = new JLabel("  MatteBorder(3,6,3,6, Color.red)  ");
       l2.setBorder(BorderFactory.createMatteBorder(3,6,3,6, Color.red));

    c.add(l1);
       c.add(l2);

    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
     }
```

```
  public static void main(String[] args){
    new MatBorders();
  }
}
```

Рамки с надписями *TitledBorder*

Класс `TitledBorder` позволяет создать рамку с надписью. В простейшем случае конструктор `TitledBorder(String)` или статический метод `createTitledBorder(String)` класса `BorderFactory` создает простую рамку толщиной в один пиксел, в которую слева сверху вставлена строка. Это показано на рис. 16.3 сверху.

Надпись можно вставить в рамку любого типа. Для этого используется конструктор

```
TitledBorder(Border, String);
```

или статический метод

```
createTitledBorder(Border, String);
```

На рис. 16.3 во второй строке надпись вставлена в рамку класса `EtchedBorder`.

Надпись можно вставить в верхнюю границу рамки, `TOP`, написать выше верхней границы, `ABOVE_TOP`, или ниже верхней границы, `BELOW_TOP`. То же самое можно сделать снизу: `BOTTOM`, `ABOVE_BOTTOM`, `BELOW_BOTTOM`. Эти константы — параметр `pos` в конструкторах и методах, описанных далее.

По умолчанию надпись располагается слева, `LEFT`, но ее можно расположить по центру, `CENTER`, или справа, `RIGHT`. Эти константы — параметр `just` в конструкторах и методах, описанных далее.

Все константы, определяющие место надписи, перечислены в листинге 16.5.

Все восемнадцать возможностей реализуются конструктором

```
TitledBorder(Border, String, int just, int pos);
```

или статическим методом

```
createTitledBorder(Border, String, int just, int pos);
```

Некоторые из этих возможностей показаны на рис. 16.3.

Кроме различного расположения надписи, для нее можно задать шрифт конструктором

```
TitledBorder(Border, String, int just, int pos, Font);
```

или статическим методом

```
createTitledBorder(Border, String, int just, int pos, Font);
```

Наконец, кроме расположения и шрифта можно определить еще и цвет надписи конструктором

```
TitledBorder(Border, String, int just, int pos, Font, Color);
```

или статическим методом класса `BorderFactory`

```
createTitledBorder(Border, String, int just, int pos, Font, Color);
```

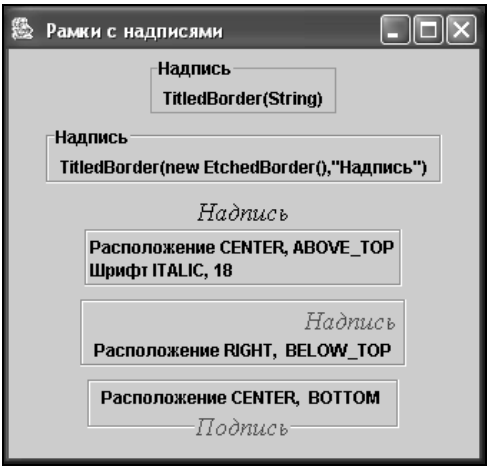


Рис. 16.3. Рамки с надписями

Листинг 16.3 содержит программу, создавшую рис. 16.3. В ней приведены все описанные возможности создания рамок с надписями.

Листинг 16.3. Рамки с надписями

```
import java.awt.*;
import javax.swing.*;
   import javax.swing.border.*;

   public class TitBorders extends JFrame{

  TitBorders(){
    super(" Рамки с надписями");
    Container c = getContentPane(); // Излишне для JDK выше 1.4.
       c.setLayout(new FlowLayout());

    JLabel l1 = new JLabel("  TitledBorder(String)  ");
       l1.setBorder(new TitledBorder("Надпись"));

    JLabel l2 = new JLabel(
         "  TitledBorder(new EtchedBorder(),\"Надпись\")  ");
    l2.setBorder(BorderFactory.createTitledBorder(
            BorderFactory.createEtchedBorder(), "Надпись"));

    JLabel l3 = new JLabel(
         "<html>  Расположение CENTER," +
         "  ABOVE_TOP<p>Шрифт ITALIC, 18  ");
    l3.setBorder(BorderFactory.createTitledBorder(
         BorderFactory.createEtchedBorder(), "Надпись",
         TitledBorder.CENTER, TitledBorder.ABOVE_TOP,
            new Font("Times New Roman", Font.ITALIC, 18)));
```

```
    JLabel l4 =
         new JLabel("  Расположение RIGHT,  BELOW_TOP  ");
    l4.setBorder(BorderFactory.createTitledBorder(
         BorderFactory.createEtchedBorder(), "Надпись",
         TitledBorder.RIGHT, TitledBorder.BELOW_TOP,
         new Font("Times New Roman", Font.ITALIC, 18),
            Color.red ));

    JLabel l5 =
         new JLabel("  Расположение CENTER,  BOTTOM   ");
    l5.setBorder(BorderFactory.createTitledBorder(
         BorderFactory.createEtchedBorder(), "Подпись",
         TitledBorder.CENTER, TitledBorder.BOTTOM,
         new Font("Times New Roman", Font.ITALIC, 18),
            Color.red ));

       c.add(l1); c.add(l2); c.add(l3); c.add(l4); c.add(l5);

    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
     }

  public static void main(String[] args){
    new TitBorders();
  }
}
```

Сдвоенные рамки *CompoundBorder*

Класс `CompoundBorder` создает рамку, состоящую из двух вложенных рамок любых типов. Это делается конструктором `CompoundBorder(Border out, Border in)` или статическим методом `createCompoundBorder(Border out, Border in)` класса `BorderFactory`.

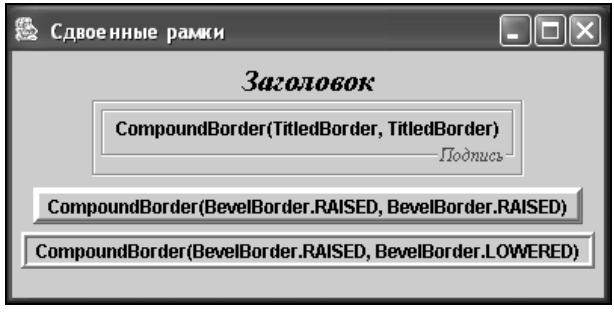


Рис. 16.4. Сдвоенные рамки

Число возможных комбинаций двух рамок очень велико. На рис. 16.4 показаны некоторые из них, а в листинге 16.4 приведена программа, создающая это окно.

Листинг 16.4. Сдвоенные рамки

```
import java.awt.*;
import javax.swing.*;
   import javax.swing.border.*;

   public class CompBorders extends JFrame{

  CompBorders(){
    super(" Сдвоенные рамки");
    Container c = getContentPane(); // Излишне для JDK выше 1.4.
       c.setLayout(new FlowLayout());

    JLabel l1 = new JLabel(
        "  CompoundBorder(TitledBorder, TitledBorder)  ");
       l1.setBorder(new CompoundBorder(

      BorderFactory.createTitledBorder(
        BorderFactory.createEtchedBorder(), "Заголовок",
        TitledBorder.CENTER, TitledBorder.ABOVE_TOP,
           new Font("Times New Roman", Font.ITALIC|Font.BOLD, 20)),

      BorderFactory.createTitledBorder(
        BorderFactory.createEtchedBorder(), "Подпись",
        TitledBorder.RIGHT, TitledBorder.BOTTOM,
        new Font("Times New Roman", Font.ITALIC, 12),
        Color.red)
       ));

    JLabel l2 = new JLabel(
      "  CompoundBorder(BevelBorder.RAISED, BevelBorder.RAISED)  ");
    l2.setBorder(new CompoundBorder(
      BorderFactory.createBevelBorder(BevelBorder.RAISED),
      BorderFactory.createBevelBorder(BevelBorder.RAISED)
       ));

    JLabel l3 = new JLabel(
      "  CompoundBorder(BevelBorder.RAISED, BevelBorder.LOWERED)  ");
    l3.setBorder(new CompoundBorder(
      BorderFactory.createBevelBorder(BevelBorder.RAISED),
      BorderFactory.createBevelBorder(BevelBorder.LOWERED)
       ));

       c.add(l1); c.add(l2); c.add(l3);
```

```
    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
     }

  public static void main(String[] args){
    new CompBorders();
  }
}
```

Создание собственных рамок

Хотя библиотека Swing предоставляет множество готовых рамок и фабричный класс `BorderFactory` для их быстрого создания, часто возникает необходимость сконструировать оригинальную рамку.

Для ее создания можно расширить абстрактный класс `AbstractBorder`, определив хотя бы один конструктор и переопределив методы `paintBorder()` и `getBorderInsets()`. Если рамка не прозрачна, то надо переопределить метод `isBorderOpaque()` так, чтобы он возвращал `true`.

Свою рамку можно создать, расширив какой-либо класс рамок. В листинге 16.5 приведен пример рамки, расширяющей класс `TitledBorder`, в заголовок которой можно вставить не надпись, а компонент класса `JComponent`.

Поскольку класс `TitledBorder` не является контейнером, его расширение `PlaceBorder` служит только для рисования границ рамки и для определения места заголовка. Для вставки компонента в заголовок класс `PlaceBorder` погружается в контейнер `PlaceBorderPane`, расширяющий `JPanel`. Компонент помещается в этот контейнер на место, определенное классом `PlaceBorder`. Все это описано в листинге 16.5 и показано на рис. 16.5.

Листинг 16.5. Рамка с заголовком, содержащим компонент

```
import java.awt.*;
import javax.swing.*;
   import javax.swing.border.*;

interface BorderConstants{
  static public final int DEFAULT_POSITION = 0;
  static public final int ABOVE_TOP        = 1;
  static public final int TOP              = 2;
  static public final int BELOW_TOP        = 3;
  static public final int ABOVE_BOTTOM     = 4;
  static public final int BOTTOM           = 5;
     static public final int BELOW_BOTTOM     = 6;

  static public final int DEFAULT_JUSTIFICATION = 0;
  static public final int LEFT     = 1;
  static public final int CENTER   = 2;
```

```
  static public final int RIGHT    = 3;
  static public final int LEADING  = 4;
     static public final int TRAILING = 5;

  static public final int EDGE_SPACING = 2;
  static public final int TEXT_SPACING = 2;
  static public final int TEXT_INSET_H = 5;
   }

public class CompTitledTest extends JFrame
      implements BorderConstants{
  public CompTitledTest(){
    super(" Рамка с компонентом");
       Container c = getContentPane(); // Излишне для JDK выше 1.4.

    JLabel lab = new JLabel(" PlaceBorder(JLabel)  ",
         new ImageIcon("middle.gif"), JLabel.LEFT);

    PlaceBorderPane pbp =
         new PlaceBorderPane(new EtchedBorder(), lab, CENTER, TOP);

       c.add(pbp);

    setSize(300, 300);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
     }

  public static void main(String[] args){
    new CompTitledTest();
  }
   }

class PlaceBorderPane extends JPanel implements BorderConstants{
  protected JComponent comp;
  protected JPanel p;
     protected PlaceBorder border;

  public PlaceBorderPane(){
    this(new JLabel("Заголовок"));
     }

  public PlaceBorderPane(JComponent c){
    this(null, c, LEFT, TOP);
     }

  public PlaceBorderPane(Border b, JComponent c, int just, int pos){
    super(); comp = c;
    border = new PlaceBorder(b, c, just, pos);
    setBorder(border);
```

```
    setLayout(null);
    add(comp);
    p = new JPanel();
    add(p);
     }

  public JPanel getContentPane(){
    return p;
     }

  public void doLayout(){
       Insets insets = getInsets();

    Rectangle r = getBounds();
       r.x = 0; r.y = 0;

    comp.setBounds(border.getComponentRect(r,insets));
    r.x += insets.left;
    r.y += insets.top;
    r.width  -= insets.left + insets.right;
    r.height -= insets.top  + insets.bottom;
    p.setBounds(r);
     }

class PlaceBorder extends TitledBorder{
  public PlaceBorder(JComponent c){
    this(null, c, LEFT, TOP);
     }

  public PlaceBorder(Border b){
    this(b, null, LEFT, TOP);
     }

  public PlaceBorder(Border b, JComponent c){
    this(b, c, LEFT, TOP);
     }

  public PlaceBorder(Border b, JComponent c, int just, int pos){
    super(b, null, just, pos, null, null);
    if (b == null)
      border = super.getBorder();
     }

  public void paintBorder(Component c, Graphics g,
                          int x, int y, int width, int height){
    Rectangle r = new Rectangle(
         x + EDGE_SPACING, y + EDGE_SPACING,
            width  - (EDGE_SPACING * 2), height - (EDGE_SPACING * 2));
```

```
    Insets bIns = (border != null) ?
         border.getBorderInsets(c) : new Insets(0, 0, 0, 0);

    Rectangle rect = new Rectangle(x, y, width, height);
    Insets insets = getBorderInsets(c);
    Rectangle compR = getComponentRect(rect, insets);
    int diff;
    switch (titlePosition){
      case ABOVE_TOP:
        diff = compR.height + TEXT_SPACING;
        r.y += diff;
        r.height -= diff;
        break;
      case TOP:
      case DEFAULT_POSITION:
        diff = insets.top/2 - bIns.top - EDGE_SPACING;
        r.y += diff;
        r.height -= diff;
        break;
      case BELOW_TOP:
      case ABOVE_BOTTOM:
        break;
      case BOTTOM:
        diff = insets.bottom/2 - bIns.bottom - EDGE_SPACING;
        r.height -= diff;
        break;
      case BELOW_BOTTOM:
        diff = compR.height + TEXT_SPACING;
        r.height -= diff;
    }
    border.paintBorder(c, g, r.x, r.y, r.width, r.height);
    Color col = g.getColor();
    g.setColor(c.getBackground());
    g.fillRect(compR.x, compR.y, compR.width, compR.height);
    g.setColor(col);
    comp.repaint();
     }

  public Insets getBorderInsets(Component c, Insets insets){
    Insets bIns = (border != null) ?
          border.getBorderInsets(c) : new Insets(0, 0, 0, 0);

    insets.top    = EDGE_SPACING + TEXT_SPACING + bIns.top;
    insets.right  = EDGE_SPACING + TEXT_SPACING + bIns.right;
    insets.bottom = EDGE_SPACING + TEXT_SPACING + bIns.bottom;
       insets.left   = EDGE_SPACING + TEXT_SPACING + bIns.left;
```

```
    if (c == null || comp == null) return insets;

       int h = (comp != null) ? comp.getPreferredSize().height : 0;

    switch (titlePosition){
      case ABOVE_TOP:
        insets.top += h + TEXT_SPACING;
        break;
      case TOP:
      case DEFAULT_POSITION:
        insets.top += Math.max(h, bIns.top) - bIns.top;
        break;
      case BELOW_TOP:
        insets.top += h + TEXT_SPACING;
        break;
      case ABOVE_BOTTOM:
        insets.bottom += h + TEXT_SPACING;
        break;
      case BOTTOM:
        insets.bottom += Math.max(h, bIns.bottom) - bIns.bottom;
        break;
      case BELOW_BOTTOM:
        insets.bottom += h + TEXT_SPACING;
    }
    return insets;
     }

  public Rectangle getComponentRect(Rectangle rect, Insets bIns){
    Dimension d = comp.getPreferredSize();
    Rectangle r = new Rectangle(0, 0, d.width, d.height);
    switch (titlePosition){
      case ABOVE_TOP:
        r.y = EDGE_SPACING;
        break;
      case TOP:
      case DEFAULT_POSITION:
        r.y = EDGE_SPACING +
              (bIns.top - EDGE_SPACING - TEXT_SPACING - d.height)/2;
        break;
      case BELOW_TOP:
        r.y = bIns.top - d.height - TEXT_SPACING;
        break;
      case ABOVE_BOTTOM:
        r.y = rect.height - bIns.bottom + TEXT_SPACING;
        break;
```

```
      case BOTTOM:
        r.y = rect.height - bIns.bottom + TEXT_SPACING +
              (bIns.bottom - EDGE_SPACING - TEXT_SPACING - d.height)/2;
        break;
      case BELOW_BOTTOM:
        r.y = rect.height - d.height - EDGE_SPACING;
    }
    switch (titleJustification) {
      case LEFT:
      case DEFAULT_JUSTIFICATION:
        r.x = TEXT_INSET_H + bIns.left;
        break;
      case RIGHT:
        r.x = rect.width - bIns.right -TEXT_INSET_H - r.width;
        break;
      case CENTER:
        r.x = (rect.width - r.width) / 2;
    }
    return r;
  }
}
}
```

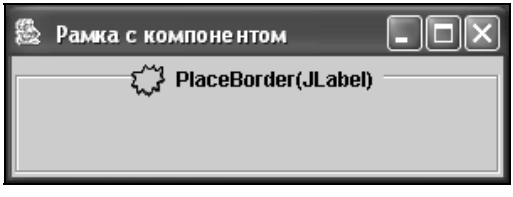


Рис. 16.5. Рамка с компонентом `JLabel`

Вопросы для самопроверки

1. Какие компоненты Swing можно окружить рамкой?
2. Можно ли окружить рамкой контейнер?
3. Можно ли сделать несколько рамок для одного и того же компонента?
4. Можно ли поменять рамку при наступлении какого-нибудь события?
5. Можно ли поменять цвет рамки при наступлении какого-нибудь события?
6. Можно ли сделать рамку не для всех, а только для одной или нескольких сторон компонента?
7. Можно ли сделать разные рамки для разных сторон компонента?
8. Как меняется рамка при изменении размеров окна?



Глава 17

Изменение внешнего вида компонента

Одна из самых замечательных особенностей библиотеки Swing — возможность изменять внешний вид и поведение графических элементов приложения.

Вид (look) каждого графического компонента задают его форма, тип и цвет рамки, цвет фона, цвет, тип и размер шрифта, форма курсора мыши. *Поведение* компонента (feel) определяют та или иная реакция на действия мыши, набор командных клавиш, способ перемещения окон и т. д. Набор таких свойств всех компонентов приложения определяет его вид и поведение, Look and Feel, сокращенно — L&F. Всего набирается несколько сотен свойств, определяющих L&F приложения.

Самые общие методы получения и задания сведений о виде и поведении приложения Swing собраны в абстрактном классе `LookAndFeel` пакета `javax.swing`. При создании какого-то конкретного стиля L&F надо расширить этот класс, заполнив его характеристиками конкретного L&F.

В библиотеке Swing собраны все необходимые сведения о стандартном виде и поведении графического приложения на нескольких наиболее распространенных графических платформах. Часть этих сведений, общая для всех платформ, образует набор *системных сведений* (system defaults). Системные сведения собраны в абстрактном классе `BasicLookAndFeel`, расширяющем класс `LookAndFeel`. Класс `BasicLookAndFeel` и его вспомогательные классы составляют пакет `javax.swing.plaf.basic`.

Конкретные сведения, специфичные для трех наиболее распространенных платформ, собраны в трех классах, расширяющих класс `BasicLookAndFeel`:

- `MotifLookAndFeel` — вид и поведение, характерные для графической оболочки CDE (Common Desktop Environment), основанной на библиотеке графических функций Motif. Эта графическая оболочка применяется как "родная" в операционной системе Solaris. В документации этот вид и поведение называются CDE/Motif;
- `WindowsLookAndFeel` — вид и поведение Win32, характерные для платформы MS Windows;
- `MacLookAndFeel` — вид и поведение, принятое на платформе Apple Macintosh.

Вид и поведение CDE/Motif можно реализовать на любой платформе, вид и поведение Windows и Macintosh, из-за лицензионных ограничений, доступны лишь на соответствующей платформе.

Технология Java определяет и свой собственный, независимый от платформы, Java Look and Feel, сокращенно — Java L&F, еще короче — JLF. Он неформально называется “Metal” за схожесть этого L&F с гравировкой по металлу. Такой стиль в русской технической литературе называется "приборным" стилем за сходство с оформлением алюминиевых панелей научных приборов. JLF позволяет создать различные темы оформления. По умолчанию начиная с пятой версии Java SE установлена тема “Ocean”, в прежних версиях была тема “Steel”. Тему “Steel” можно сделать темой по умолчанию, установив системное свойство `swing.metalTheme=steel`.

Приложение, работающее в стиле JLF, будет выглядеть одинаково на всех платформах. Именно этот стиль использован во всех примерах, приведенных в предыдущих главах.

Сведения о виде и поведении Java L&F собраны в четвертом наследнике класса `BasicLookAndFeel` — классе `MetalLookAndFeel`.

Сведения, собранные в каждом из этих классов, образуют *платформенные сведения* (look and feel defaults).

Кроме системных и платформенных сведений есть еще *пользовательские сведения* (user defaults), задаваемые приложением при его запуске или во время работы.

Наивысший приоритет имеют пользовательские сведения. Если они не заданы, то исполняющая система Swing отыскивает платформенные сведения. Не найдя их, она берет нужные значения из системных сведений.

Итак, на каждой платформе можно установить три стандартных L&F: "родной" для данной платформы, CDE/Motif и Java L&F. Они выглядят как на рис. 17.1.

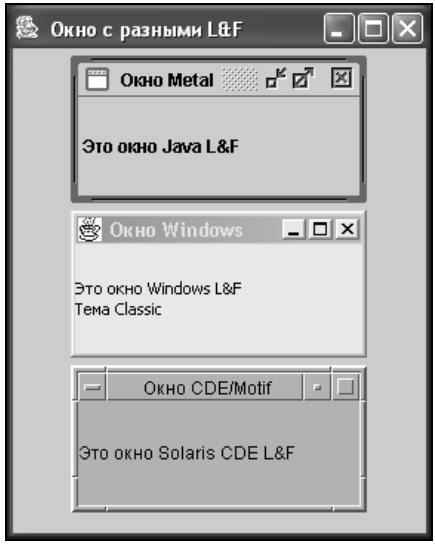


Рис. 17.1. Стандартные L&F

Окно верхнего уровня класса `JFrame` или `JDialog`, зарегистрированное в оконном менеджере графической подсистемы операционной системы, оформляется по правилам этого оконного менеджера.

Его внешний вид можно изменить только специальными ухищрениями, а именно статическими методами

```
JFrame.setDefaultLookAndFeelDecorated(true);
JDialog.setDefaultLookAndFeelDecorated(true);
JFrame newLAFWin = new JFrame();
```

После выполнения этих методов, если оконный менеджер может создавать окна без оформления и если текущий L&F способен оформлять окна, то все создаваемые окна класса `JFrame` и `JDialog` будут оформлены текущим L&F, как показано на рис. 17.4.

Чтобы оформить отдельное окно верхнего уровня текущим L&F, надо сначала отключить его оформление оконным менеджером, а затем установить новый стиль оформления методом `setWindowDecorationStyle(int)` класса `JRootPane`:

```
JFrame fr = new JFrame();
fr.setUndecorated(true);
fr.getRootPane().setWindowDecorationStyle(JRootPane.FRAME);
```

Все остальные элементы графического интерфейса пользователя могут тоже оформляться, как принято текущим оконным менеджером, следовать правилам другого оконного менеджера или быть оформлены в своей манере. Вид и поведение графического приложения определяются перед его загрузкой, но можно изменить их уже во время работы приложения.

Это свойство графических элементов библиотеки Swing получило название Pluggable Look and Feel, сокращенно — PL&F, PLAF или plaf.

Получение свойств L&F

Для получения свойств текущего L&F и установки нового L&F нужно заменить классы *Xxx*`LookAndFeel` на класс `UIManager` из пакета `javax.swing`. Он содержит массу статических методов, позволяющих получить сведения об элементах L&F, изменить некоторые элементы или вообще сменить L&F.

Объект класса `UIManager` ищет название L&F сначала как значение системного свойства swing.defaultlaf, созданного, например, при запуске приложения из командной строки с ключом `-D`:

```
java -Dswing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
   SomeSwingApplication
```

Если такое системное свойство не определено, объект ищет файл swing.properties, обычно лежащий в каталоге $JAVA_HOME/lib. Если он существует, то в нем отыскивается значение ключа swing.defaultlaf, например:

```
swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
```

Если такого ключа нет или вообще отсутствует файл swing.properties, то устанавливается Java L&F.

Для хранения свойств текущего L&F класс `UIManager` использует модель данных — объект класса `UIDefaults`. Класс `UIDefaults` расширяет класс `Hashtable`, следовательно, является хеш-таблицей, состоящей из пар "ключ — значение" (key — value). Кроме обычного

для хеш-таблицы метода `get(Object key)`, возвращающего значение `value` ключа `key`, и метода `put(Object key, Object value)`, устанавливающего значение `value` с ключом `key`, класс `UIDefaults` содержит специализированные методы для определенных типов данных, хранящихся в таблице. Они позволяют избавиться от приведения типов, так надоедающего при вызове метода `get(Object)`.

Например, метод `getBoolean(Object key)` возвращает значение ключа `key`, если оно имеет тип `boolean`. Аналогично действуют методы `getBorder(Object)`, `getColor(Object)`, `getDimension(Object)`, `getFont(Object)`, `getIcon(Object)`, `getInsets(Object)`, `getInt(Object)`, `getString(Object)`. У каждого из них есть парный метод `get`*`Xxx`*`(Object, Locale)`, возвращающий значение ключа для данной локали.

Экземпляр класса `UIDefaults`, используемый в классе `UIManager`, — это *закрытое* (private) поле. Поэтому методы класса `UIDefaults` дублируются статическими методами класса `UIManager` с теми же именами.

Например, текущий шрифт, которым делаются надписи класса `Jlabel`, можно получить так:

```
Font labelFont = UIManager.getFont("Label.font");
```

Кроме того, статическим методом `getDefaults()` можно получить ссылку на экземпляр класса `UIDefaults`, используемый классом `UIManager`.

Просмотреть все несколько сотен свойств, хранящихся в модели данных `UIDefaults`, можно так:

```
   UIDefaults defs = UIManager.getDefaults();

Enumeration keys = defs.keys();
   Enumeration elem = defs.elements();

while (keys.hasMoreElements() && elem.hasMoreElements())
   System.out.println(
      keys.nextElement() + ": " + elem.nextElement());
```

Статический метод

```
put(Object key, Object value);
```

меняет только пользовательские установки различных свойств, оставляя установки L&F без изменения. Посмотреть установки L&F можно статическим методом `geLookAndFeelDefaults()`, возвращающим экземпляр класса `UIDefaults`.

Для того чтобы легче переключать вид и поведение приложения, класс `UIManager` хранит несколько L&F под произвольно данными именами в виде массива объектов вложенного класса `UIManager.LookAndFeelInfo`. Новый элемент заносится в этот массив статическим методом

```
installLookAndFeel(String name, String className);
```

вызывающим конструктор класса `UIManager.LookAndFeelInfo`, или статическим методом

```
installLookAndFeel(UIManager.LookAndFeelInfo);
```

По умолчанию хранятся три L&F: платформенный L&F, CDE/Motif и Java L&F.

Посмотреть имена всех имеющихся в массиве L&F можно так:

```
UIManager.LookAndFeelInfo[] info = UIManager.getInstalledLookAndFeels();
for (int i = 0; i < info.length; i++)
   System.out.println(info[i].getName());
```

Задание стандартного L&F

Очень легко задать один из стандартных PL&F, воспользовавшись одним из статических методов `setLookAndFeel(LookAndFeel)` или `setLookAndFeel(String)` класса `UIManager`. Аргумент второго из этих методов — строка, содержащая полное имя нужного класса *Xxx*`LookAndFeel` со всеми подпакетами. Например, строка `"javax.swing.plaf.metal.MetalLookAndFeel"` задает имя класса, определяющего Java L&F. Поскольку это имя может измениться в следующих версиях J2SE, то для получения имени класса Java L&F лучше пользоваться статическим методом `getCrossPlatformLookAndFeelClassName()` класса `UIManager`. Впрочем, Java L&F устанавливается по умолчанию, как можно видеть из многочисленных примеров, приведенных в предыдущих главах. Еще один статический метод `getSystemLookAndFeelClassName()` класса `UIManager` возвращает полное имя класса, определяющего стандартный L&F для данной платформы. Обычный способ задания платформенного L&F выглядит так:

```
public static void main(String[] args){
  try{
    UIManager.setLookAndFeel(
      UIManager.getSystemLookAndFeelClassName());
     }catch(Exception e){}

  new SomeSwingApplication();
}
```

Обрабатывать исключение здесь не нужно, потому что в ответ на его выбрасывание устанавливается Java L&F.

Для получения строки с полным именем класса CDE/Motif в классе `UIManager` никакого метода нет, ее надо задавать прямо:

```
UIManager.setLookAndFeel(
   "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
```

В листинге 17.1 приведена программа, создавшая рис. 17.1. В ней каждое внутреннее окно строится по правилам одного из стандартных L&F.

Листинг 17.1. Внутренние окна со стандартными L&F

```
import java.awt.*;
   import javax.swing.*;

   public class DiffLAF extends JFrame{
```

```
  DiffLAF(){
    super(" Окно с разными L&F");
    Container c = getContentPane(); // Излишне для JDK выше 1.4.
       c.setLayout(new FlowLayout());

    JInternalFrame ifr1 =
      new JInternalFrame(" Окно Metal", true, true, true, true);
    ifr1.getContentPane().add(new JLabel(" Это окно Java L&F"));
    ifr1.setPreferredSize(new Dimension(200, 100));
    ifr1.setVisible(true)
       c.add(ifr1);

  try{
    UIManager.setLookAndFeel(
         UIManager.getSystemLookAndFeelClassName());

    JInternalFrame ifr2 =
      new JInternalFrame(" Окно Windows", true, true, true, true);
    ifr2.getContentPane().add(
      new JLabel("<html>Это окно Windows L&F<p>Тема Classic"));
    ifr2.setPreferredSize(new Dimension(200, 100));
    ifr2.setVisible(true);
       c.add(ifr2);

    UIManager.setLookAndFeel(
      "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
     }catch(Exception e){}

    JInternalFrame ifr3 =
      new JInternalFrame(" Окно CDE/Motif", true, true, true, true);
    ifr3.getContentPane().add(
      new JLabel("Это окно Solaris CDE L&F"));
    ifr3.setPreferredSize(new Dimension(200, 100));
    ifr3.setVisible(true);
       c.add(ifr3);

    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
     }

  public static void main(String[] args){
    new DiffLAF();
  }
}
```

Дополнительные L&F

Кроме текущего L&F библиотека Swing может одновременно использовать *дополнительные* (auxiliary) L&F. Они являются расширениями класса `MultiLookAndFeel` из пакета `javax.swing.plaf.multi`. Полные имена их классов вида *Xxx*`LookAndFeel` задаются в файле $JAVA_HOME/lib/swing.properties, в строке с ключом swing.auxiliarylaf, через запятую. При загрузке объектов L&F исполняющая система Java вначале создает и загружает объекты текущего L&F, а затем объекты дополнительных L&F в порядке их записи в строке swing.auxiliarylaf в файле swing.properties.

Дополнительные L&F обычно связаны с дополнительными устройствами ввода/вывода: клавиатурой Бройля, речевым вводом/выводом и тому подобными устройствами. Ничто не мешает дополнительному L&F выводить свои объекты на экран, но при этом могут возникнуть конфликты с текущим L&F. Поэтому рекомендуется не использовать в качестве дополнительных графических L&F классы, унаследованные от `BasicLookAndFeel`, а наследовать их прямо от `MultiLookAndFeel`.

Смена всего L&F

Разработчик GUI может сменить весь L&F своего приложения или его отдельные свойства. Для смены всего L&F сначала устанавливается новый L&F:

```
UIManager.setLookAndFeel(new MyCoolLookAndFeel());
```

Затем надо привести вид и поведение приложения в соответствие с новым L&F.

При каждом изменении какого-либо свойства, входящего в L&F, или всего L&F целиком, происходит событие класса `PropertyChangeEvent`. Класс `UIManager` присоединяет обработчик этого события обычным методом `addPropertyChangeListener(PropertyChangeListener)`. В обработчике события следует сообщить всем компонентам приложения о смене L&F. Это удобно сделать статическим методом `updateComponentTreeUI(Component)` класса `SwingUtilities`. В аргументе данного метода достаточно указать контейнер верхнего уровня, метод передаст сообщение всем вложенным контейнерам и компонентам.

Итак, обработка изменения L&F выглядит следующим образом:

```
JFrame frame = JFrame("Главное окно");
UIManager.addPropertyChangeListener(
  new PropertyChangeListener(){
    public void propertyChange(PropertyChangeEvent e){
        SwingUtilities.updateComponentTreeUI(frame);
    }
});
```

Если при этом надо установить первоначальные размеры окна, то в обработчик события следует вставить обращение к методу `frame.pack()`.

Метод `updateComponentTreeUI(frame)` рекурсивно просматривает все вложенные контейнеры и компоненты. Для каждого компонента вызывается его метод `updateUI()` класса `JComponent`, который, в свою очередь, вызывает метод `setUI(ComponentUI)` с аргументом соответствующего типа, устанавливающий новый L&F для этого компонента.

В листинге 17.2 приведен пример программы, дающей возможность пользователю менять L&F во время работы приложения, выбрав нужный пункт меню. На рис. 17.2 показан вывод этой программы.

Листинг 17.2. Смена L&F во время работы приложения

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.util.*;
   import javax.swing.*;

   public class ChLAF extends JFrame{

  ChLAF(){
    super(" Смена L&F");
    Container c = getContentPane(); // Излишне для JDK выше 1.4.
       c.setLayout(new FlowLayout());

    JMenuBar mb = new JMenuBar();
       setJMenuBar(mb);

    JMenu serv = new JMenu("Сервис");
       mb.add(serv);

    JMenu laf = new JMenu("Вид");
       serv.add(laf);

       ButtonGroup bg = new ButtonGroup();

    UIManager.LookAndFeelInfo[] info =
         UIManager.getInstalledLookAndFeels();

    for (int i = 0; i < info.length; i++){
      JRadioButtonMenuItem item =
           new JRadioButtonMenuItem(info[i].getName());

         item.addItemListener(new LAFChange(info[i].getClassName()));

      bg.add(item);
      laf.add(item);
       }

       JButton b = new JButton("Кнопка");

       c.add(b);

    UIManager.addPropertyChangeListener(
      new PropertyChangeListener(){
```

```
        public void propertyChange(PropertyChangeEvent e){
            SwingUtilities.updateComponentTreeUI(c);
        }
       });

    setSize(400, 400);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
     }

  public static void main(String[] args){
    new ChLAF();
     }

    class LAFChange implements ItemListener{

      private String className;

    public LAFChange(String className){
      this.className = className;
       }

       public void itemStateChanged(ItemEvent e){

      if(e.getStateChange() == ItemEvent.SELECTED)
        try{
          UIManager.setLookAndFeel(className);
        }catch(Exception ex){}
    }
  }
}
```

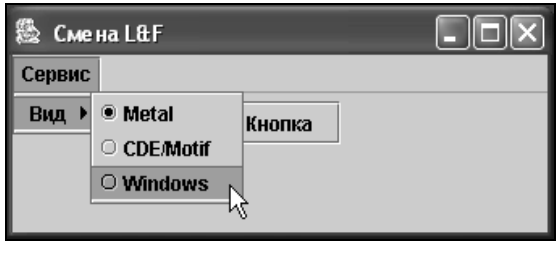


Рис. 17.2. Смена L&F

В составе Java SE, в каталоге $JAVA_HOME/demo/jfc/SwingSet2/ приведен пример, позволяющий менять L&F и темы во время работы приложения. Для его запуска достаточно перейти в указанный каталог и набрать в командной строке

```
java -jar SwingSet2.jar
```

Замена отдельных свойств L&F

Во время работы приложения можно заменить не весь текущий L&F, а только некоторые его свойства. Для этого надо воспользоваться статическим методом `put(Object key, Object value)` класса `UIManager`. Поскольку при этом меняется только модель данных — класс `UIDefaults` — к нему следует присоединить обработчик событий, получив ссылку на него методом `getDefaults()`. Потом нужно оповестить все заинтересованные компоненты о сделанных изменениях методом `updateComponentTreeUI(Component)`.

В листинге 17.3 приведена программа, в которой выполнены эти действия. Она позволяет заменить размер шрифта в текстовых полях другим размером, выбранным из меню.

Листинг 17.3. Смена размера шрифта в полях ввода

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.util.*;
import javax.swing.*;
   import javax.swing.plaf.*;

public class PropCh extends JFrame{
  PropCh(){
    super(" Смена размера шрифта");
    Container c = getContentPane(); // Излишне для JDK выше 1.4.
       c.setLayout(new FlowLayout());

    JMenuBar mb = new JMenuBar();
       setJMenuBar(mb);

    JMenu serv = new JMenu("Сервис");
       mb.add(serv);

    JMenu laf = new JMenu("Размер шрифта");
       serv.add(laf);

       ButtonGroup bg = new ButtonGroup();

       FontChange fch = new FontChange();

    for (int i = 10; i < 22; i += 2){
      JRadioButtonMenuItem item = new JRadioButtonMenuItem(""+ i);
      item.addItemListener(fch);
      bg.add(item);
      laf.add(item);
       }

    JTextArea      ta = new JTextArea(5, 20);
```

```
    JTextField     tf = new JTextField(20);
       JPasswordField pf = new JPasswordField(20);

       c.add(ta); c.add(tf); c.add(pf);

    PropertyChangeListener pcl =
      new PropertyChangeListener(){
        public void propertyChange(PropertyChangeEvent e){
            SwingUtilities.updateComponentTreeUI(c);
        }
         };

    UIManager.addPropertyChangeListener(pcl);
       UIManager.getDefaults().addPropertyChangeListener(pcl);

    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
     }

  public static void main(String[] args){
    new PropCh();
     }

  class FontChange implements ItemListener{
       public FontChange(){}

       public void itemStateChanged(ItemEvent e){

      if (e.getStateChange() == ItemEvent.SELECTED){
        JMenuItem mi = (JMenuItem)e.getSource();
        int  n = Integer.parseInt(mi.getText());
        Font f = UIManager.getFont("TextArea.font");
        String name = f.getName();
           int  style = f.getStyle();

           FontUIResource fr = new FontUIResource(name, style, n);

        UIManager.put("TextArea.font", fr);
//      UIManager.put("TextField.font", fr);
//      UIManager.put("PasswordField.font", fr);
        UIManager.put("EditorPane.font", fr);
        UIManager.put("TextPane.font", fr);
        UIManager.put("FormattedTextField.font", fr);
      }
    }
  }
}
```

На рис. 17.3 показан результат работы программы из листинга 17.3. В области ввода установлен шрифт величиной 20 пунктов. Поскольку две строчки листинга закомментированы, в поле ввода и в поле пароля остался прежний системный размер шрифта.

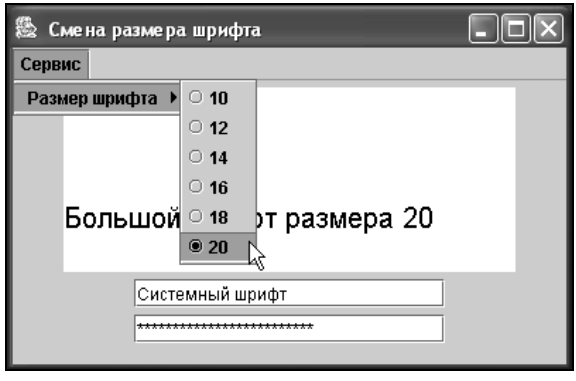


Рис. 17.3. Поля ввода с измененным размером шрифта

Темы Java L&F

Вид и поведение Java L&F базируются на трех *основных* (primary) цветах и трех *дополнительных* (secondary) фоновых цветах, из которых образуется колорит внешнего вида приложения. Кроме того, выбран цвет для ввода текста и цвет фона текстовых полей. Выбор каких-либо шести цветов в качестве основных и дополнительных, а также шрифтов для заголовка, надписей, ввода текста образуют определенную *тему* (theme) внешнего вида.

Методы, возвращающие цвета и шрифты выбранной темы, частично определены абстрактным классом `MetalTheme` из пакета `javax.swing.plaf.metal`. Полное определение темы, выбираемой по умолчанию, дано классом `DefaultMetalTheme`. В этой теме основные цвета — темно-синий, синий и голубой, точнее, цвет primary1 в модели RGB равен (102, 102, 153), цвет primary2 — (153, 153, 204), цвет primary3 — (204, 204, 255). Дополнительные цвета — это темно-серый, серый и светло-серый, точнее, secondary1 равен (102, 102, 102), secondary2 — (153, 153, 153), secondary3 — (204, 204, 204). Цвет primary1 используется рамками активного компонента, надписями на компонентах. Цвет primary2 выделяет пункты меню, цвет primary3 выделяет текст в полях ввода. Цвет secondary1 оттеняет "выпуклые" компоненты, цвет secondary2 используется рамками неактивных компонентов, цвет secondary3 — цвет фона неактивных компонентов.

Кроме перечисленных цветов тема `DefaultMetalModel` определяет черный цвет для текста, вводимого в текстовые поля, и белый цвет для фона текстовых полей.

В теме класса `OceanTheme`, расширяющего класс `DefaultMetalModel`, основные цвета — это светло-синий, голубой и светло-голубой, точнее, в модели RGB это цвета (99, 130, 191), (163, 184, 204) и (184, 207, 229). Дополнительные цвета — серо-голубой, светло-голубой и светло-серый, точнее, (122, 138, 153), (184, 207, 229) и (238, 238, 238).

Для создания собственной темы достаточно расширить класс `MetalTheme`, определив методы, устанавливающие и возвращающие цвета: `getPrimary1()`, `getPrimary2()`,

getPrimary3(), getSecondary1(), getSecondary2(), getSecondary3(), и методы, задающие и возвращающие шрифты: getControlTextFont(), getMenuTextFont(), getSubTextFont(), getSystemTextFont(), getUserTextFont(), getWindowTitleFont(). Если нужно изменить только отдельный цвет или шрифт, то достаточно расширить класс DefaultMetalTheme или класс OceanTheme.

Методы, создающие цвета, должны возвращать объект класса ColorUIResource. Для создания такого объекта есть четыре конструктора:

```
ColorUIResource(Color);
ColorUIResource(int red, int green, int blue);
ColorUIResource(float red, float green, float blue);
ColorUIResource(int rgb);
```

Поэтому метод, возвращающий первый основной цвет, может выглядеть так:

```
public ColorUIResource getPrimary1(){
  return new ColorUIResource(36, 124, 225);
}
```

Методы, формирующие вид шрифтов, должны возвращать объект класса FontUIResource, для создания которого есть два конструктора:

```
FontUIResource(Font);
FontUIResource(String name, int style, int size);
```

Метод, создающий системный шрифт, может выглядеть так:

```
public FontUIResource getSystemTextFont(){
  return FontUIResource("Times New Roman", Font.PLAIN, 10);
}
```

Хотя тема определяет, главным образом, шрифты, их цвета и цвета фона, но она может изменить любые свойства Java L&F. Для этого в классе MetalTheme есть метод addCustomEntriesToTable(UIDefaults), позволяющий занести в модель данных UIDefaults не только новые цвета и шрифты, но и какие-нибудь другие свойства. Достаточно переопределить этот метод в расширении класса MetalTheme, чтобы задать изменение свойств. В листинге 17.4 приведен пример метода, устанавливающего новые рамки текстовых полей.

После того как написан класс темы, расширяющий класс MetalTheme, класс DefaultMetalTheme или класс OceanTheme, надо установить новую тему статическим методом setCurrentTheme(MetalTheme) класса MetalLookAndFeel, как показано в листинге 17.4. В нем заданы только серые цвета, что удобно для печати иллюстраций в книге. Результат показан на рис. 17.4.

Листинг 17.4. Окна, оформленные в "серой" теме

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
```

```
import javax.swing.plaf.*;
import javax.swing.plaf.metal.*;
public class ContTheme extends JFrame{
  ContTheme(){
    super(" Окно с серой темой");
    JDesktopPane dp = new JDesktopPane();
    setLayout(new FlowLayout());
    setContentPane(dp);
    JInternalFrame ifr1 = new JInternalFrame(
      " Окно GrayMetalTheme", true, true, true, true);
    ifr1.getContentPane().setLayout(new FlowLayout());
    ifr1.getContentPane().add(
      new JLabel("<html>Это окно Java L&F<p>Серая тема" ));
    ifr1.setBounds(0,0, 200,200);
    ifr1.setVisible(true);
    dp.add(ifr1);
    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
     }

  public static void main(String[] args){
    JFrame.setDefaultLookAndFeelDecorated(true);
    JDialog.setDefaultLookAndFeelDecorated(true);
    MetalLookAndFeel.setCurrentTheme(new GrayMetalTheme());
    new ContTheme();
  }
}
class GrayMetalTheme extends DefaultMetalTheme{
  public ColorUIResource getPrimary1(){
    return getSecondary1();
  }
  public ColorUIResource getPrimary2(){
    return getSecondary2();
  }
  public ColorUIResource getPrimary3(){
    return getSecondary3();
  }
  public void addCustomEntriesToTable(UIDefaults table){
    super.addCustomEntriesToTable(table);
    BorderUIResource b = new BorderUIResource(
      new CompoundBorder(
          new LineBorder(Color.gray),
          new LineBorder(Color.white)));
    table.put("TextField.border", b);
    table.put("PasswordField.border", b);
```

```
    table.put("TextArea.border", b);
    table.put("TextPane.font", b);
  }
}
```

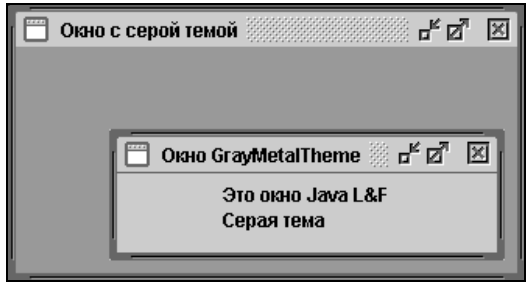


Рис. 17.4. Окна, оформленные в "серой" теме

В стандартной поставке Java SE, в каталоге $JAVA_HOME/demo/jfc/MetalWorks/ лежат файлы приложения `MetalWorks` с примерами тем Java L&F. Запустить это приложение можно из командной строки, перейдя в этот каталог и набрав

```
java -jar MetalWorks.jar
```

Там же есть исходные тексты приложения, из которых видно, как создаются и используются темы Java L&F.

Вопросы для самопроверки

1. Что такое вид и поведение (Look and Feel) приложения?
2. Что такое тема (theme) оформления и чем она отличается от вида и поведения приложения?
3. Какие L&F входят в стандартную поставку графической библиотеки Swing?
4. Можно ли стандартными средствами Swing создать собственный L&F?
5. Какие темы предлагает Java L&F?
6. Можно ли создать свои темы в Java L&F?



Глава 18

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 — таким окном служит окно браузера. Апплеты не запускаются с помощью JVM — их загружает браузер, который сам запускает JVM для выполнения апплета. Эти особенности отражаются на написании программы апплета.

С точки зрения графической библиотеки AWT, 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.

В графической библиотеке Swing всякий апплет расширяет класс `JApplet`, расширяющий класс `Applet`. Главные дополнения Swing к свойствам апплета AWT заключаются в возможности добавления системы меню методом `setJMenuBar(JMenuBar)` и в наличии множества панелей, как в классе `JFrame`. На панели, получаемой методом `getContentPane()`, по умолчанию установлен менеджер размещения `BorderLayout`.

Еще одна особенность апплета, вытекающая из того, что он не запускается машиной JVM, заключается в том, что отпадает необходимость в методе `main()`, его нет в апплетах.

В апплетах редко встречается конструктор. Дело в том, что при запуске апплета создается его контекст. Во время выполнения конструктора контекст еще не сформирован полностью, 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 автоматически 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 летом на участок, вы 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 18.1 записан простейший апплет библиотеки AWT, выполняющий вечную программу `HelloWorld`.

Листинг 18.1. 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 18.2.

Листинг 18.2. 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 Интернета. Браузер, просматривая HTML-файл, выполнит тег `<applet>` и загрузит апплет. После загрузки апплет появится в окне браузера, как показано на рис. 18.1.

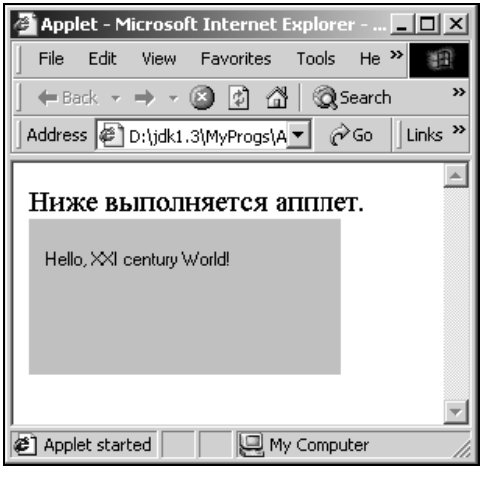


Рис. 18.1. Апплет `HelloWorld` в окне Internet Explorer

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 даже сделать апплет невидимым, задав его размер, равный одному пикселу. Кроме того, размер апплета разрешается указывать в процентах по отношению к размеру окна браузера, например:

```
<applet code = "HelloWorld.class" width = "100%" height = "100%">
```

Во-вторых, как видно на рис. 18.1, 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, который 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 полноценного обозревателя, можно воспользоваться им. Программа `appletviewer` запускается из командной строки:

```
appletviewer HelloWorld.html
```

На рис. 18.2 `appletviewer` показывает апплет `HelloWorld`.



Рис. 18.2. 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 18.3 приведен апплет, записывающий в строку состояния браузера "бегущую строку", а в листинге 18.4 — соответствующий HTML-файл.

Листинг 18.3. 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 18.4. 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 18.5 и 18.6, результат — на рис. 18.3.

Листинг 18.5. 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 18.6. 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>
```

Замечание по отладке

Браузеры помещают загруженные апплеты в свой кэш, поэтому после нажатия кнопки **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**.

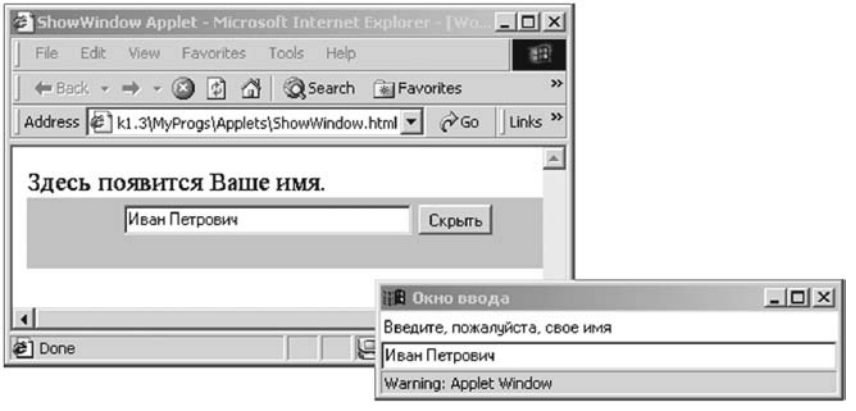


Рис. 18.3. Апплет, создающий окно

Упражнения

1. Создайте апплет — "записную книжку", как в *главе 15*.
2. Создайте апплет — "рисовалку" по схеме *главы 15*.
3. Запрограммируйте игру в "крестики-нолики" в виде апплета.

Передача параметров в апплет

При запуске приложения интерпретатором `java` из командной строки в него можно передать параметры в виде аргумента метода `main(String[] args)`. В апплеты тоже можно передавать параметры, но другим путем.

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 18.7 показан измененный файл HelloWorld.html.

Листинг 18.7. 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 18.8 показан переработанный апплет `HelloWorld`. В нем назначен белый фон, а шрифт устанавливается с параметрами, извлеченными из HTML-файла.

Листинг 18.8. 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 задайте начальные значения переменным в апплете, как это сделано в листинге 18.8.

На рис. 18.4 показан работающий апплет.

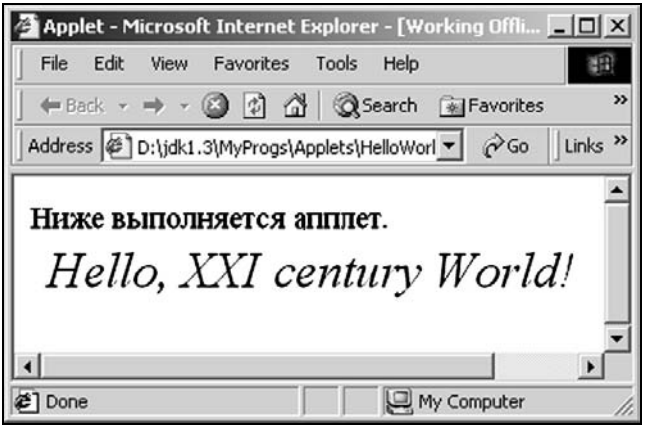


Рис. 18.4. Апплет с измененным шрифтом

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, в котором лежит соответствующий HTML-файл;
- `archive` — файлы всех классов, составляющих апплет, могут быть упакованы архиватором ZIP или специальным архиватором JAR, который мы опишем в *главе 25*, 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` — 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, каскадная таблица стилей);
- `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 записи тега `<applet>`:

```
<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 jar-архивы и указывайте их в параметре `archive` в HTML-файле. Это значительно ускорит загрузку апплета.

Следует еще сказать, что начиная с версии 4.0 в языке HTML 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.

Упражнение

4. Напишите апплет, собирающий все сведения о своем контексте.

Изображение и звук в апплетах

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 *главе 20*, 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 и 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 Интернета.

Слежение за процессом загрузки

Если вы хотя бы раз видели, как изображение загружается из Интернета, 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` хранит 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 выяснения 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 *главе 20* в листинге 20.5 мы применим эти методы для ожидания загрузки изображения.

Изображение, находящееся в объекте класса Image, можно создать непосредственно по пикселам или 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 каталога 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 18.9 приведен простой пример загрузки изображения и звука из файлов, находящихся в том же каталоге, что и HTML-файл. На рис. 18.5 показано, как выглядит изображение, увеличенное в два раза.

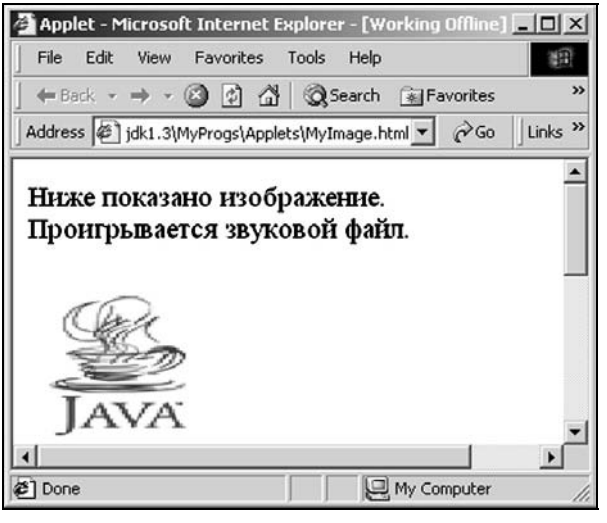


Рис. 18.5. Вывод изображения

Листинг 18.9. 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(); }
}
```

Перед выводом на экран изображение можно преобразовать, но об этом мы поговорим в *главе 20*.

Упражнения

5. Напишите апплет, показывающий несколько изображений, меняющихся при нажатии кнопки.
6. Напишите аналогичный апплет, в котором изображения меняются каждые 3 секунды.

Защита от апплета

Как видно из предыдущего текста, апплету в браузере позволено очень немного. Это не случайно. Апплет, появившийся в браузере откуда-то из Интернета, способен 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.

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 способны усилить или ослабить эти ограничения, например разрешить локальным апплетам, загруженным с той же машины, где они выполняются, доступ к файловой системе. Наименьшие ограничения имеют *доверенные* (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.

Апплеты в библиотеке Swing

Как уже говорилось ранее, апплеты, созданные средствами графической библиотеки Swing, расширяют класс `JApplet`, непосредственно расширяющий класс `Applet`. Основные расширения заключаются в возможности добавить систему меню и в наличии множества панелей подобно классу `JFrame`. Впрочем, в апплетах Swing чаще всего окно создают в отдельном классе. В листинге 18.10 показан шаблон создания апплета.

Листинг 18.10. Шаблон апплета, использующего Swing

```
import java.awt.*;
import java.awt.event.*;
   import javax.swing.*;

   public class SwingAppletTemplate extends JApplet{

   public void init(){
         Container c = getContentPane();

         c.setLayout(new XxxLayout()); // По умолчанию BorderLayout

      c.add(new MainFrame());       // Главное окно апплета
   }
   public void start(){
   }
```

```
    public void stop(){
    }
    public void destroy(){
    }
}
class MainFrame extends JFrame{
    // Главное окно, показываемое в апплете
}
```

Апплет *JApplet*

Как сказано ранее, апплеты, созданные средствами библиотеки Swing, расширяют класс `JApplet`, являющийся непосредственным расширением класса `Applet`. Поэтому, подобно любому апплету, они наследуют четверку методов: `init()`, `start()`, `stop()`, `destroy()`. Как всякое окно верхнего уровня библиотеки Swing, апплеты Swing содержат всего один компонент — корневую панель класса `JRootPane`. Поэтому к апплетам относится все сказанное ранее о размещении компонентов в окне верхнего уровня `JFrame`. В частности, панелью содержимого по умолчанию управляет менеджер размещения `BorderLayout`.

Если в браузере, выполняющем апплет, имеется библиотека Swing, как, например, в Netscape Communicator 6.*x*, то апплет класса `JApplet` можно вызвать тегом `<APPLET>` обычным образом. Если библиотеки Swing нет, то в браузер надо загрузить дополнительный модуль Java Plug-in. Он входит в состав JRE и автоматически инсталлируется при инсталляции JRE. Если же модуля Java Plug-in на компьютере нет, то браузер предлагает загрузить его из Интернета при первой попытке выполнения апплета. Загрузить Java Plug-in можно с сайта **http://java.sun.com/** или **http://www.microsoft.com/java/** или указать свой собственный сайт.

Для подключения модуля Java Plug-in старых версий вместо тега `<applet>` использовался тег `<object>` для браузера Internet Explorer и тег `<embed>` для Netscape Communicator. Для того чтобы апплет мог работать со старой версией JRE в любом браузере, все теги объединялись, например, следующим образом:

```
<html>
<body>
<!--"CONVERTED_APPLET"-->
<!-- HTML CONVERTER -->
<object classid = "clsid:CAFEEFAC-0014-0000-0000-ABCDEFFEDCBA"
        width = 100 height = 100
        codebase="http://java.sun.com/products/plugin/1.4/
        jinstall-140-win32.cab#Version=1,4,0,0">
<param name = CODE value = some.class >
<param name = ARCHIVE value = some.jar >
<param name="type" value="application/x-java-applet;jpi-version=1.4">
<param name="scriptable" value="false">
<comment>
```

```
<embed type="application/x-java-applet;jpi-version=1.4"
       code = some.class archive = some.jar
       width = 100 height = 100
       scriptable = false
       pluginspage="http://java.sun.com/products/plugin/1.4/
                    plugin-install.html">
<noembed></noembed></embed>
</comment>
</object>
<!--
<applet code = some.class archive = some.jar width = 100 height = 100>
</applet>
-->
<!--"END_CONVERTED_APPLET"-->
</body>
</html>
```

Параметр `codebase` в теге `<object>` и параметр `pluginspage` в теге `<embed>` показывают сайт **java.sun.com/products/**, с которого загружается Java Plug-in. Если вам требуется загрузить этот модуль с другого Web-сервера, замените адрес сайта своим адресом.

В состав J2SE JDK входит утилита $JAVA_HOME/lib/htmlconverter.jar. Она переписывает HTML-файл, содержащий тег `<APPLET>`, добавляя теги `<OBJECT>` и `<EMBED>`. Написанный ранее HTML-текст получен этой утилитой из файла dummy.html, содержавшего первоначально текст:

```
<html>
  <body>
    <APPLET CODE = some.class ARCHIVE = some.jar WIDTH = 100 HEIGHT = 100>
    </APPLET>
  </body>
</html>
```

Для того чтобы получить преобразованный файл, достаточно набрать командную строку

```
java -jar htmlconverter.jar dummy.html
```

Новым версиям Java JRE, начиная с JDK 1.4, для запуска апплета не нужен тег `<OBJECT>`. Им достаточно тега `<APPLET>`, следовательно, нет нужды в утилите htmlconverter.

В документации Java SE, в каталоге $JAVA_HOME/docs/technotes/guides/plugin/, есть руководство разработчика, подробно объясняющее правила работы с Java Plug-in.

Упражнение

7. Перепишите апплеты предыдущих упражнений для библиотеки Swing.

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 Интернету, хорошие примеры апплетов собраны в JDK в каталоге demo/applets.

В JDK вошел целый пакет `java.applet`, в который корпорация 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 Интернета. Такая возможность есть сейчас у всех музыкальных и видеопроигрывателей.

Более того, апплеты широко применяются в сотовых телефонах и других мобильных устройствах, поэтому рано говорить о том, что они устарели и выходят из употребления.

Тем не менее компания Sun Microsystems больше не развивает пакет `java.applet`. В нем так и остался один класс и три интерфейса.

Вопросы для самопроверки

1. Что такое апплет?
2. Чем апплет отличается от приложения?
3. Можно ли написать программу, которую смогут выполнять и интерпретатор, и браузер, т. е. одновременно апплет и приложение?
4. Можно ли написать конструктор в апплете?
5. Будет ли выполняться конструктор в апплете?
6. Как передать параметры в апплет?
7. Может ли апплет читать файлы на машине браузера?
8. Может ли апплет читать файлы на машине, с которой он загружен?
9. Может ли апплет передавать данные по сети?



Глава 19

Прочие свойства Swing

В этой главе собраны разнообразные сведения о библиотеке Swing, не вошедшие в предыдущие главы.

Свойства экземпляра компонента

Каждый экземпляр компонента Swing содержит небольшую хеш-таблицу, унаследованную от класса `JComponent`. Чаще всего она пуста, хотя при создании экземпляра компонента в нее могло быть занесено несколько значений конструктором класса.

Перед работой с таблицей ее надо заполнить методом `putClientProperty(Object key, Object value)`. После того как таблица заполнена, любой объект может получить значение `value` ключа `key` методом `getClientProperty(Object key)`, возвращающим значение `value` в виде объекта класса `Object`. Разумеется, метод `getClientProperty(Object)` может вернуть значение только уже определенного ранее ключа, иначе он вернет `null`.

Именно через свойство экземпляра панель класса `JToolBar` определяет, надо ли показывать рамку у расположенных на ней инструментальных кнопок *(см. рис. 14.10)*. Метод `setRollover(boolean)` этого класса определяет свойство `JToolBar.isRollover`. Вот исходный текст этого метода:

```
public void setRollover(boolean rollover){
  putClientProperty("JToolBar.isRollover",
      rollover ? Boolean.TRUE : Boolean.FALSE);
}
```

Метод `isRollover()` в своей работе использует это свойство. Вот его исходный код:

```
public boolean isRollover(){
  Boolean rollover =
    (Boolean)getClientProperty("JToolBar.isRollover");
  if (rollover != null)
    return rollover.booleanValue();
  return false;
}
```

Аналогично можно поступать с любым наследником `comp` класса `JComponent`: сначала определить какое-то свойство, например:

```
comp.putClientProperty("Number", new Integer(k++));
```

а затем воспользоваться им, скажем, чтобы отличить один экземпляр от другого:

```
Integer numb = (Integer)comp.getClientProperty("Number");
```

Подобная схема применена в листинге 19.1.

Прокрутка содержимого компонента

Содержимое компонентов Swing, находящихся на панели `JScrollPane`, легко прокручивать с помощью полос прокрутки или колесика мыши. Но возможность прокрутки заложена не только в компоненты, реализующие интерфейс `Scrollable`. Она есть у каждого компонента Swing.

Возможность прокрутки содержимого компонента активизируется методом `setAutoscrolls(true)`. Если после его выполнения вывести курсор мыши за пределы компонента при нажатой кнопке мыши (drag), то начинает происходить событие мыши. Чтобы обработать это событие, надо присоединить к компоненту обработчик события мыши, обратившись в нем к методу `scrollRectToVisible(Rectangle rect)`. Например:

```
JPanel p = new JPanel();
   p.setAutoscrolls(true);

p.addMouseMotionListener(new MouseMotionAdapter(){
  public void mouseDragged(MouseEvent e){
    Rectangle r = new Rectangle(e.getX(), e.getY(), 1, 1);
    ((JPanel)e.getSource()).scrollRectToVisible(r);
  }
});
```

Метод `scrollRectToVisible(r)` обращается к подобному методу объемлющего контейнера. Некоторые контейнеры, например `JViewport`, переопределяют этот метод таким образом, чтобы в окне появился прямоугольник `rect`.

Передача фокуса ввода

Каждый компонент Swing является контейнером и может содержать внутри себя другие компоненты. Только один из компонентов в контейнере обладает фокусом ввода — он реагирует на действия мыши и клавиатуры. На большинстве платформ по умолчанию передача фокуса ввода другому компоненту происходит при щелчке на нем кнопкой мыши или нажатии клавиши <Tab>, для текстовых полей — <Ctrl>+<Tab>. После нажатия клавиш фокус передается следующему компоненту, причем следующим, как правило, считается компонент, помещенный в контейнер после данного компонента. Обычно это расположение компонентов слева направо и сверху вниз. Обратная передача фокуса ввода происходит при нажатии комбинации клавиш <Shift>+<Tab>, для текстовых полей — <Ctrl>+<Shift>+<Tab>.

Передача фокуса может быть временной, что означает возможность быстро вернуть фокус. Временная передача фокуса обычно производится при просмотре пунктов меню, передвижении бегунка по полосе прокрутки, перемещении окна.

Компонент класса `Component` и его наследники могут запросить фокус защищенными логическими методами `requestFocus(boolean)` и `requestFocusInWindow(boolean)`. Значение параметра `true` означает временную передачу фокуса, `false` — постоянную передачу. Второй из этих методов может применяться компонентом, который расположен в контейнере, уже имеющем фокус. Он введен для того, чтобы запретить передачу фокуса из одного окна в другое окно, независимое от первого. Методы возвращают `false`, если запрос отвергнут, и `true`, если запрос обрабатывается.

Иногда возникает необходимость изменить порядок следования компонентов в контейнере. Так же как и обработку событий, систему передачи фокуса Swing наследует от AWT. В пакете `java.awt` есть интерфейс `KeyEventDispatcher`, описывающий всего один логический метод `dispatchKeyEvent(KeyEvent)`. Этот метод должен возвращать `false`, если событие `KeyEvent` передается следующему компоненту, реализующему данный интерфейс, и `true`, если передача события на этом заканчивается. Событие поступает этому методу *до* объекта-слушателя типа `KeyListener`, следовательно, метод `dispatchKeyEvent(KeyEvent)` может решить, передавать ли событие обработчику.

Второй интерфейс, `KeyEventPostProcessor`, тоже определяет один метод `postProcessKeyEvent(KeyEvent)`, к которому компонент обращается уже после передачи фокуса.

Оба интерфейса частично реализует абстрактный класс `KeyboardFocusManager`, расширенный классом `DefaultKeyboardFocusManager`.

Кроме своей основной работы — распределять события класса `KeyEvent` между компонентами — этот класс может назначить навигационные клавиши. В нем определены статические константы:

- `FORWARD_TRAVERSAL_KEY` — обычно <Tab> или <Ctrl>+<Tab>;
- `BACKWARD_TRAVERSAL_KEY` — обычно <Shift>+<Tab>;
- `UP_CYCLE_TRAVERSAL_KEY` — переход к первому компоненту, значения по умолчанию нет;
- `DOWN_CYCLE_TRAVERSAL_KEY` — переход к последнему компоненту, значения по умолчанию нет.

Перечисленные константы используются в качестве первого параметра метода `setDefaultFocusTraversalKeys(int, Set)`, применяемого для определения новых навигационных клавиш вместо клавиш <Tab> и <Shift>. Второй параметр содержит описание новых навигационных клавиш в виде множества `Set` объектов класса `AWTKeyStroke`.

Аналогичный метод `setFocusTraversalKeys(int, Set)` есть и в классе `Component`.

Определение того, какой компонент будет следующим при применении навигационных клавиш, делается абстрактным классом `FocusTraversalPolicy`. Этот класс содержит методы, возвращающие:

- `getComponentAfter(Container, Component)` — следующий компонент и его контейнер;
- `getComponentBefore(Container, Component)` — предыдущий компонент и его контейнер;
- `getFirstComponent(Container)` — первый компонент в контейнере;

- `getLastComponent(Container)` — последний компонент в контейнере;
- `getDefaultComponent(Container)` — компонент, на который передается фокус при активизации контейнера;
- `getInitialComponent(Window)` — компонент, на который передается фокус при активизации окна.

Два конкретных порядка следования компонентов определены двумя подклассами класса `FocusTraversalPolicy`.

Первый подкласс `ContainerOrderFocusTraversalPolicy` следует порядку, в котором компоненты расположены в массиве, возвращаемом методом `getComponents()` класса `Container`. Этот класс добавляет логический метод `accept(Component)`, проверяющий видимость и доступность компонента. Метод начинается так:

```
protected boolean accept(Component comp){
  if (!(comp.isVisible() && comp.isDisplayable() &&
        comp.isEnabled() && comp.isFocusable())) return false;
// Продолжение метода...
}
```

Как видно из этого начала, метод сразу отвергает компоненты, не выполняющие хотя бы одно из перечисленных четырех условий.

У класса `ContainerOrderFocusTraversalPolicy` есть расширение `DefaultFocusTraversalPolicy`, которое переопределяет метод `accept(Component)` таким образом, что четвертое условие, `isFocusable()`, учитывается, лишь если его значение отлично от значения по умолчанию.

Второй подкласс, `InternalFrameFocusTraversalPolicy`, относится уже к библиотеке Swing. Это абстрактный класс, расположенный в пакете `javax.swing`. Он задает первый компонент во внутреннем окне методом `getInitialComponent(JInternalFrame)`. Метод возвращает ссылку на объект класса `Component`.

Класс `InternalFrameFocusTraversalPolicy` расширен классом `SortingFocusTraversalPolicy`. Он пользуется порядком расположения компонентов, определенным некоторой реализацией интерфейса `Comparator`, сделанной разработчиком. Описание этого интерфейса и правила его реализации мы рассмотрели в *главе 6*. Конкретная реализация интерфейса задается в конструкторе класса `SortingFocusTraversalPolicy(Comparator)`.

Класс `SortingFocusTraversalPolicy`, в свою очередь, расширен классом `LayoutFocusTraversalPolicy`. Он определяет порядок расположения компонентов, основываясь на их размерах и положении в контейнере. Этот класс используется по умолчанию L&F класса `BasicLookAndFeel` и его расширениями.

После того как определен порядок следования компонентов, методом `setDefaultFocusTraversalPolicy(FocusTraversalPolicy)` полученный экземпляр класса `LayoutFocusTraversalPolicy` устанавливается в текущий экземпляр класса `KeyboardFocusManager`. Можно задать особенный порядок расположения компонентов в отдельном контейнере методом `setFocusTraversalPolicy(FocusTraversalPolicy)` класса `Container`.

После этого методами `focusNextComponent()` и `focusPreviousComponent()` класса `KeyboardFocusManager` можно переходить к следующему или предыдущему компоненту.

Для отдельного компонента или контейнера то же самое можно сделать методами `transferFocus()`, `transferFocusBackward()` класса `Component`.

В листинге 19.1 приведен простой пример передачи фокуса от текстового поля к кнопке и обратно. Рисунок 19.1 демонстрирует протокол этой программы.

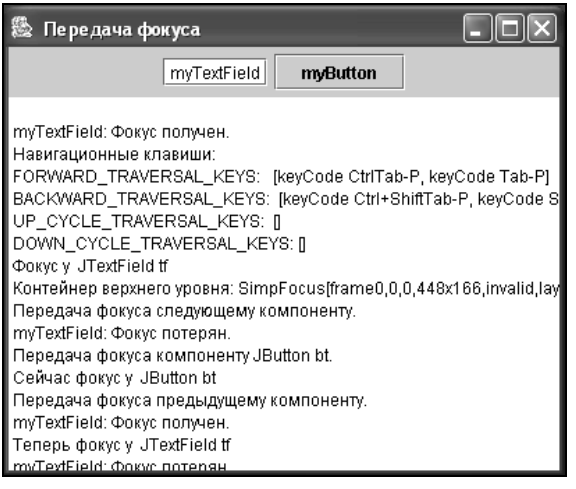


Рис. 19.1. Протокол передачи фокуса

Листинг 19.1. Передача фокуса ввода

```
import java.awt.*;
import java.awt.event.*;
   import javax.swing.*;

   public class SimpFocus extends JFrame implements FocusListener{

  DefaultKeyboardFocusManager myFocusmgr =
            new DefaultKeyboardFocusManager();
  JButton    bt;
  JTextField tf;
     JTextArea  ta;

  SimpFocus(){
    super(" Передача фокуса");
    JComponent c = (JComponent)getContentPane();
    JPanel p = new JPanel();
    c.add(p, BorderLayout.NORTH);
    bt = new JButton(" myButton ");
       bt.putClientProperty("Focus", "JButton bt");

    tf = new JTextField(" myTextField ");
    tf.putClientProperty("Focus", "JTextField tf");
```

```
    tf.addFocusListener(this);
    ta = new JTextArea(6, 40);
       ta.putClientProperty("Focus", "JTextArea ta");

    c.add(ta);
    p.add(tf);
       p.add(bt);

    pack();
    setDefaultCloseOperation(EXIT_ON_CLOSE);
       setVisible(true);

    printDefaultSettings(myFocusmgr);
    changeFocusOwner(myFocusmgr);
     }

  public void focusLost(FocusEvent e){
    ta.append("\n myTextField: Фокус потерян.");
    ta.append("\n Передача фокуса компоненту " +
       ((JComponent)e.getOppositeComponent()).
          getClientProperty("Focus")+".");
     }

  public void focusGained(FocusEvent e){
    ta.append("\n myTextField: Фокус получен.");
     }

  public void printDefaultSettings(
                DefaultKeyboardFocusManager fm){

    ta.append("\n Навигационные клавиши: ");
    ta.append("\n FORWARD_TRAVERSAL_KEYS:  " +
      fm.getDefaultFocusTraversalKeys(
        KeyboardFocusManager.FORWARD_TRAVERSAL_KEYS));
    ta.append("\n BACKWARD_TRAVERSAL_KEYS:  " +
      fm.getDefaultFocusTraversalKeys(
        KeyboardFocusManager.BACKWARD_TRAVERSAL_KEYS));
    ta.append("\n UP_CYCLE_TRAVERSAL_KEYS:  " +
      fm.getDefaultFocusTraversalKeys(
        KeyboardFocusManager.UP_CYCLE_TRAVERSAL_KEYS));
    ta.append("\n DOWN_CYCLE_TRAVERSAL_KEYS: " +
      fm.getDefaultFocusTraversalKeys(
        KeyboardFocusManager.DOWN_CYCLE_TRAVERSAL_KEYS));
     }

  public void changeFocusOwner(DefaultKeyboardFocusManager fm){
    ta.append("\n Фокус у  " +
       ((JComponent)fm.getFocusOwner()).getClientProperty("Focus"));
```

```
    ta.append("\n Контейнер верхнего уровня: " +
          (fm.getCurrentFocusCycleRoot()));

    ta.append("\n Передача фокуса следующему компоненту.");
    fm.focusNextComponent();
    ta.append("\n Сейчас фокус у  " +
           ((JComponent)fm.getFocusOwner()).getClientProperty("Focus"));

    ta.append("\n Передача фокуса предыдущему компоненту.");
    fm.focusPreviousComponent();
    ta.append("\n Теперь фокус у  " +
        ((JComponent)fm.getFocusOwner()).getClientProperty("Focus"));
     }

  public static void main(String[] args){
    new SimpFocus();
  }
}
```

Перенос данных Drag and Drop

В компоненты Swing `JColorChooser`, `JFileChooser`, `JList`, `JTable`, `JTree`, а также в компонент `JTextComponent` и в его потомки встроена возможность переноса данных с помощью мыши (Drag and Drop) или командных клавиш (Cut-Copy-Paste) через системный буфер обмена (clipboard). Обычно эта возможность выключена. Включается она методом `setDragEnabled(true)`.

Связь с системным буфером обмена происходит через объект класса `Clipboard` из пакета `java.awt.datatransfer`. Данные, предназначенные для переноса, должны удовлетворять интерфейсу `Transferable`, описывающему перенос данных, преобразованных в специально разработанный объект класса `DataFlavor`. Информация заносится в буфер методом

```
setContents(Transferable, ClipBoardOwner);
```

а извлекается методом `getContents(Object)` в виде объекта, реализующего интерфейс `Transferable`. Источник данных регистрируется в буфере как объект, реализующий интерфейс `ClipBoardOwner`.

Возможность переноса данных заложена в класс `TransferHandler`. Экземпляр этого класса установлен в перечисленные выше компоненты Swing. Экземпляр создается конструктором `TransferHandler(String)`, параметр которого определяет имя свойства JavaBean. Созданный экземпляр устанавливается в компонент методом `setTransferHandler(TransferHandler)` класса `JComponent`.

После создания экземпляра метод

```
exportAsDrag(JComponent comp, InputEvent evt, int action);
```

приводит в готовность механизм переноса данных из компонента `comp` при наступлении события `evt`. Последний параметр `action` принимает значение `COPY` или `MOVE`. Независимо

от значения этого параметра на большинстве платформ копирование происходит при нажатой клавише <Ctrl>, но это зависит от используемого L&F.

При перетаскивании данных работает сложный внутренний механизм, заимствованный из библиотеки AWT. Этот механизм реализован интерфейсами и классами двух пакетов: `java.awt.dnd` и `java.awt.datatransfer`. Он автоматически используется классом `TransferHandler`.

Листинг 19.2 показывает простой пример переноса выделенного текста из поля ввода класса `JTextField` в надпись класса `JLabel` перетаскиванием мышью.

Листинг 19.2. Перетаскивание текста из поля ввода в надпись

```
import java.awt.*;
import java.awt.event.*;
   import javax.swing.*;

   public class LabelDnD extends JFrame{

  public LabelDnD(){
       super(" Перенос текста в надпись JLabel");

    JTextField tf = new JTextField(100);
       tf.setDragEnabled(true);

    JLabel tl = new JLabel("Перетащи сюда", JLabel.LEFT);
    tl.setTransferHandler(new TransferHandler("text"));
    tl.addMouseListener(new MouseAdapter(){
       public void mousePressed(MouseEvent e){
         JComponent c = (JComponent)e.getSource();
         TransferHandler th = c.getTransferHandler();
         th.exportAsDrag(c, e, TransferHandler.COPY);
       }
       });

    getContentPane().add(tf, BorderLayout.NORTH);
       getContentPane().add(tl);

       tl.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(200, 100);
    setVisible(true);
      }

  public static void main(String[] args){
    new LabelDnD();
  }
}
```

Временная задержка *Timer*

Класс `Timer` играет роль будильника, генерирующего через заданное время событие класса `ActionEvent`. Задержка в миллисекундах и обработчик сгенерированного события задаются в единственном конструкторе класса

```
Timer(int millisec, ActionListener al);
```

Созданный этим конструктором "будильник" запускается методом `start()`. После этого стартует подпроцесс, в котором через заданное время `millisec` начнет генерироваться событие `ActionEvent`. Оно будет обрабатываться предварительно подготовленным объектом `al`.

Событие генерируется до тех пор, пока не будет выполнен метод `stop()`. Если надо сгенерировать событие только один раз, то следует обратиться к методу `setRepeats(false)`.

Задержку можно изменить методом `setDelay(int)`. Допустимо отдельно задать начальную задержку методом `setInitialDelay(int)`.

Если обратиться к методу `setLogTimers(true)`, то при каждом событии в стандартный вывод `System.out` будет выводиться протокол, а именно сообщение "Timer ringing:" со ссылкой на объект `Timer`.

В листинге 19.3 приведен простой пример использования класса `Timer`. Сообщение о событии выводится в область ввода. Нажатие кнопки прекращает работу "будильника".

Листинг 19.3. Методы класса `Timer`

```
import java.awt.*;
import java.util.*;
import java.awt.event.*;
   import javax.swing.*;

public class SimpTimer extends JFrame implements ActionListener{
  JButton           bt;
  JTextArea         ta;
     javax.swing.Timer t;

  SimpTimer(){
     super(" Передача фокуса");
     Container c = getContentPane();
     JPanel p = new JPanel();
        c.add(p, BorderLayout.NORTH);

     bt = new JButton("Останов");
     bt.addActionListener(new ActionListener(){
       public void actionPerformed(ActionEvent e){
          t.stop(); ta.append("\nStop");
       }
        });
```

```
        ta = new JTextArea(6, 30);

     c.add(ta);
        p.add(bt);

     t = new javax.swing.Timer(500, this);
     t.setInitialDelay(1000);
     t.setLogTimers(true);
        t.start();

     pack();
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     setVisible(true);
     }

  public void actionPerformed(ActionEvent e){
     ta.append("\n Время: " + new Date());
     }

   public static void main(String[] args){
      new SimpTimer();
   }
}
```

В листинге 19.3 использован класс `Timer` из пакета `javax.swing`. Еще один класс с таким же именем есть в пакете `java.util`. Он устроен сложнее и имеет больше возможностей, но класс `Timer` из пакета `javax.swing` оптимизирован под библиотеку Swing. Разработчики библиотеки Swing рекомендуют использовать этот класс `Timer`, а не его тезку из пакета `java.util`.



Глава 20

Изображения и звук

Как уже упоминалось в предыдущих главах, изображение в Java — это объект класса `Image`. В *главе 18* было показано, как в апплетах применяются методы `getImage()` для создания этих объектов из графических файлов.

Приложения 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 с именем `filename`;
- `createImage(URL address)` — создает изображение из содержимого графического файла, расположенного по адресу URL `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", которая требует более подробного объяснения.

Модель "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 графической библиотеке 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 `MemoryImageSource` — 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 20.1 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 20.1. Изображение, построенное по точкам

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;
public class InMemory extends JFrame{
  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);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
  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){
    new InMemory("  Изображение в памяти");
  }
}
```

В листинге 20.1 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 уже говорилось в *главе 18*, просто написав в нем оператор `return true`.

Рисунок 20.1 демонстрирует вывод этой программы. К сожалению, черно-белая печать не передает смешение цветов.

Интерфейс `ImageConsumer` описывает семь методов, самыми важными из которых являются два метода `setPixels()`. Сигнатура первого метода выглядит так:

```
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`, а не `byte`.

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).

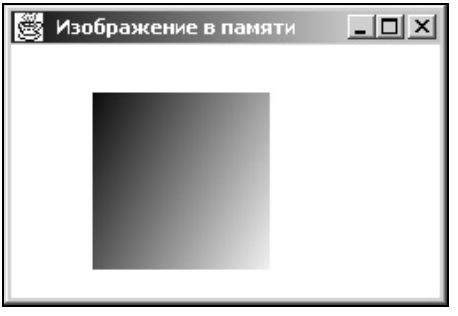


Рис. 20.1. Изображение, созданное по точкам

На рис. 20.2 показана иерархия классов, реализующих модель "поставщик-потребитель".

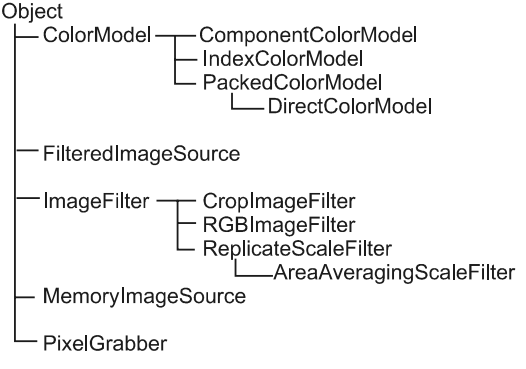


Рис. 20.2. Классы, реализующие модель "поставщик-потребитель"

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 защищенное (protected) поле consumer этого класса.

В пакете java.awt.image есть четыре готовых 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 20.2—20.4.

Как выделить фрагмент изображения

В листинге 20.2 выделяется фрагмент изображения и выводится на экран в увеличенном виде. Кроме того, ниже выводятся изображения, увеличенные с помощью классов `ReplicateScaleFilter` и `AreaAveragingScaleFilter`.

Листинг 20.2. Примеры масштабирования изображения

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;
public class CropTest extends JFrame{
  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);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
  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){
    new CropTest("  Масштабирование");
  }
}
```

На рис. 20.3 слева сверху показано исходное изображение, справа — увеличенный фрагмент, внизу — изображение, увеличенное двумя способами.

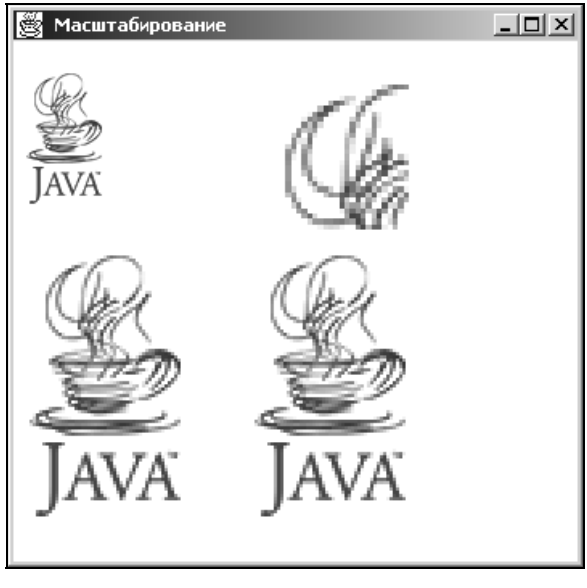


Рис. 20.3. Масштабированное изображение

Как изменить цвет изображения

В листинге 20.3 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 20.3. Изменение цвета всех пикселов

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;
public class RGBTest extends JFrame{
  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);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
  public void paint(Graphics g){
    g.drawImage(img, 10, 40, this);
    g.drawImage(newimg, 150,  40, this);
  }
  public static void main(String[] args){
    new RGBTest("  Изменение цвета");
  }
}
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 20.4 определяется преобразование пикселов изображения. Создается новый фильтр — расширение `ShiftFilter` класса `ImageFilter`, сдвигающее изображение цик-

лически вправо на указанное в конструкторе число пикселов. Все, что для этого нужно, — это переопределить метод `setPixels()`.

Листинг 20.4. Циклический сдвиг изображения

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;
public class ShiftImage extends JFrame{
     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);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
  public void paint(Graphics gr){
    gr.drawImage(img,     20, 40, this);
    gr.drawImage(newimg, 100, 40, this);
  }
  public static void main(String[] args){
    new ShiftImage("  Циклический сдвиг изображения");
  }
}
     // 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 20.4, 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.



Рис. 20.4. Перестановка пикселов изображения

На рис. 20.4 показан результат выполнения этой программы.

Вторая модель обработки изображения введена в Java 2D. Она названа моделью *прямого доступа* (immediate mode model).

Упражнения

1. Создайте "лупу", увеличивающую центр изображения.
2. Измените изображение, заменив цвет каждого пиксела дополнительным цветом.
3. Измените в листинге 20.4 метод `setPixels()` так, чтобы он давал зеркальное изображение, а не сдвиг.

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 находятся 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. 20.5 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`.

Применять эту систему приходится редко, только при создании своего способа преобразования изображения. Стандартные же преобразования выполняются очень просто.

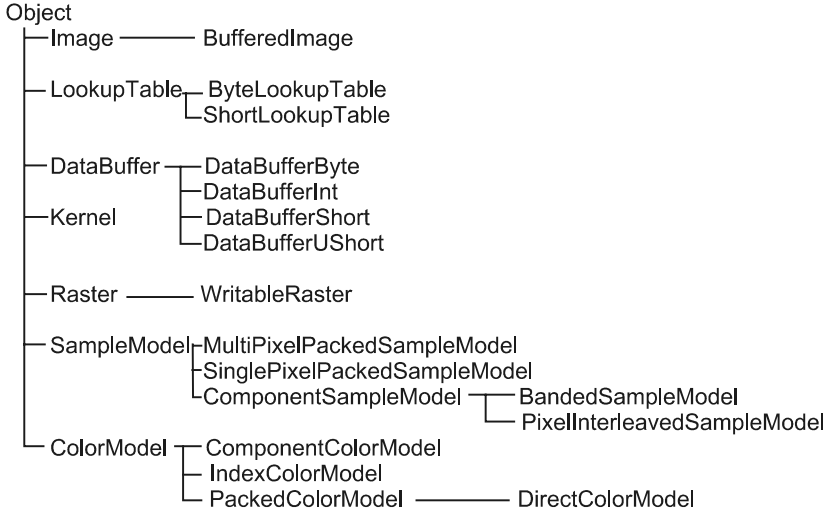


Рис. 20.5. Классы, реализующие модель прямого доступа

Преобразование изображения в Java 2D

Преобразование изображения `source`, хранящегося в объекте класса `BufferedImage`, в новое изображение `destination` выполняется методом

```
filter(BufferedImage source, BufferedImage destination);
```

описанным в интерфейсе `BufferedImageOp`. Указанный метод возвращает ссылку на новый, измененный объект `destination` класса `BufferedImage`, 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 `BufferedImageOp` 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. 20.6, 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. 20.6, посередине.

В листинге 20.5 показано, как задаются преобразования, представленные на рис. 20.6.

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 20.5. Аффинное преобразование изображения

```
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;
import java.awt.event.*;
   import javax.swing.*;

public class AffOp extends JFrame{
     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){
    JFrame f = new AffOp("  Аффинное преобразование");
    f.setSize(400, 200);
    f.setVisible(true);
    f.setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
}
```

На рис. 20.6 показано исходное изображение, преобразованная область и преобразованное внутри области изображение.



Рис. 20.6. 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. 20.7 интенсивность каждого цвета уменьшена вдвое, в результате белый фон стал серым, а цвета — темнее. Затем интенсивность увеличена на 70 единиц. В листинге 20.6 приведена программа, выполняющая это преобразование.

Листинг 20.6. Изменение интенсивности изображения

```
import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;
   import javax.swing.*;

public class Rescale extends JFrame{
     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){
     JFrame f = new Rescale("  Изменение интенсивности");
     f.setSize(300, 200);
     f.setVisible(true);
     f.setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
}
```



Рис. 20.7. 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 следует сформировать таблицу новых значений составляющих. В листинге 20.7 это двумерный массив `samples`.

Потом заполняем данный массив нужными значениями составляющих каждого цвета. В листинге 20.7 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 20.7 приведен только фрагмент программы. Для получения полной программы его надо вставить в листинг 20.6 вместо выделенного в нем фрагмента. Логотип Java будет нарисован ярко-красным цветом.

Листинг 20.7. Изменение составляющих цвета

```
//---------------- Вставить в листинг 20.6 ---------------
    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

В этом разделе мы рассмотрим методы создания различных цветовых 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 разные 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 20.8 записаны действия, необходимые для создания эффекта "размытости".

Листинг 20.8. Создание различных эффектов

```
//-------------- Вставить в листинг 20.6 ------------------------
  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 20.8 представлены слева направо исходное изображение и изображения, преобразованные весовыми матрицами `w1`, `w2` и `w3`, где матрица `w1` показана в листинге 20.8, 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 };
```

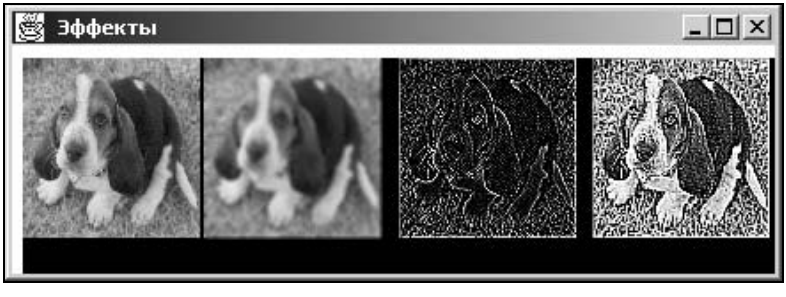


Рис. 20.8. Создание эффектов

Упражнения

4. Сделайте аффинное преобразование изображения, растягивающее его на все окно.
5. Измените изображение, максимально увеличив интенсивность каждой составляющей цвета каждого пиксела.
6. Замените в листинге 20.8 массивы из 9 элементов на массивы из 15 или 25 элементов. Как при этом изменится изображение?

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 20.9. 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 20.9. Простая анимация

```
import java.awt.*;
import java.awt.event.*;
   import javax.swing.*;

class SimpleAnim extends JFrame{
  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);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
     }

  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();
  }
}
```

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 выполнены 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 20.9 это переопределение заключено в 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. Затем 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 методу `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 20.10 и 20.11 разъясняют данный прием.

Листинг 20.10. 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 20.9 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 20.11. 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 20.12 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 20.12. Анимация рисованием

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.awt.image.*;
   import javax.swing.*;

class DrawAnim1 extends JFrame{
  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);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
     }

  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();
  }
}
```

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 20.13 показано применение этого способа. Квадрат, выведенный на экран, переливается разными цветами.

Листинг 20.13. Анимация с помощью `MemoryImageSource`

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
   import javax.swing.*;

class InMemory extends JFrame{
  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);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
     }

  public void paint(Graphics gr){
    gr.drawImage(img, 10, 30, this);
     }
```

```
  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();
  }
}
```

Вот и все средства для анимации, остальное — умелое их применение. Комбинируя рассмотренные способы, можно добиться удивительных эффектов. В документации, приложенной к Java SE, в каталогах $JAVA_HOME/demo/applets/ и $JAVA_HOME/demo/jfc/Java2D/src/, приведено много примеров апплетов и приложений с анимацией.

Упражнения

7. Создайте в каком-либо графическом редакторе или цифровым фотоаппаратом файлы run1.gif, run2.gif и т. д. и используйте их в программе листинга 20.9.
8. Средствами Java 2D сделайте небольшой рисованный фильм.

Звук

Как было указано в *главе 18*, 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 этого 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 не только в апплетах, но и в приложении, если, конечно, звуковая система компьютера уже настроена.

В листинге 20.14 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 20.14. 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, записанные без сжатия.

В состав виртуальной машины Java, входящей в Java SE 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

Проигрыватель звука, встроенный в JVM, рассчитан на два способа записи звука: моно- и стереооцифровку (digital audio) с частотой дискретизации (sample rate) от 8000 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, WAV 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`. По 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 битов), числа каналов `channels` (1 — моно, по умолчанию 2 — стерео); знания того, записаны числа 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 20.15. 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 20.16. 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 20.15.

Листинг 20.15. 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 20.16. 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 = "mamba.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 20.17. 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 20.17. 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 20.18.

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 из перечисленных выше конструкторов, 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, например число 60 обозначает ноту "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 20.18 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 20.17, 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. Саму же запись осуществляют статические методы:

```
write(Sequence in, int type, File out);
write(Sequence in, int type, OutputStream out);
```

Второй аргумент этих методов, `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 полученного массива. Все это показано в листинге 20.18.

Листинг 20.18. 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 Java SE в каталоге $JAVA_HOME/docs/technotes/guides/sound/programmer_guide/ есть подробное руководство программиста, а в каталоге $JAVA_HOME/demo/sound/src/ лежат исходные тексты синтезатора, использующего Java Sound API.

Упражнение

9. Средствами Java 2D и Java Sound API сделайте небольшой рисованный фильм со звуком.

Вопросы для самопроверки

1. Что понимается под изображением в Java?
2. Как создать объект класса `Image`?
3. Что такое модель "поставщик-потребитель"?
4. Как создать изображение по набору пикселов?
5. Какие способы фильтрации изображения предлагает Java?
6. Как реализуется модель обработки прямым доступом?
7. Какими средствами Java 2D можно изменить изображение?
8. Какие средства анимации предлагает Java?
9. Какие звуковые файлы можно проигрывать средствами Java?
10. Можно ли создать звуковой файл средствами Java?



ЧАСТЬ IV

НЕОБХОДИМЫЕ КОНСТРУКЦИИ JAVA



Глава 21

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 C или Pascal, и вы 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. Этот обработчик поступает очень просто: выводит на консоль сообщение о произошедшем исключении и прекращает выполнение программы.

Приведем пример. В программе листинга 21.1 может возникнуть деление на нуль, если запустить ее с аргументом 0. В программе нет никаких средств обработки такой исключительной ситуации. Посмотрите на рис. 21.1, какие сообщения выводит исполняющая система Java.

Листинг 21.1. 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");
  }
}
```

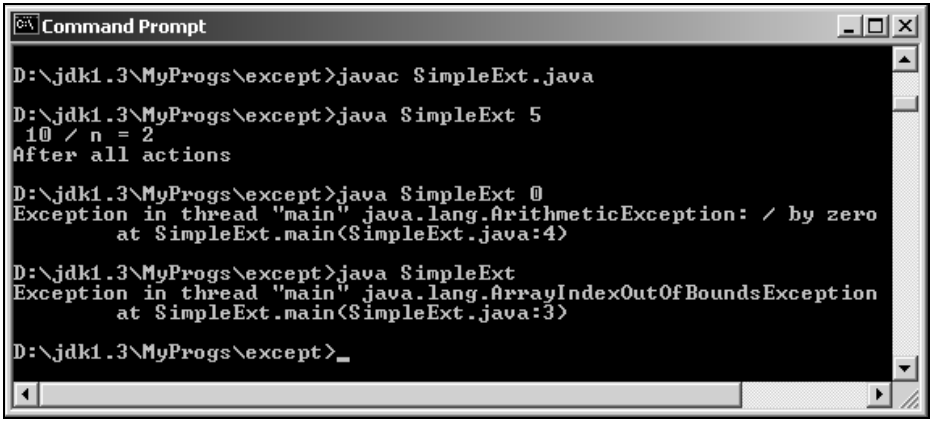


Рис. 21.1. Сообщения об исключительных ситуациях

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 на этом 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. Программа в одном или нескольких местах *пытается* (try) его *"перехватить"* (catch) и обработать. Обработку можно сделать полностью в одном месте, а можно частично 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 21.1 могут возникнуть исключения двух типов. Напишем блоки их обработки, как это сделано в листинге 21.2.

Листинг 21.2. 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 21.2 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, что бы 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{}` или из блока `catch(){}` осуществляется оператором `return`. В последнем случае оператор `return` выполняется после блока `finally{}`.

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 21.2 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. 21.2.

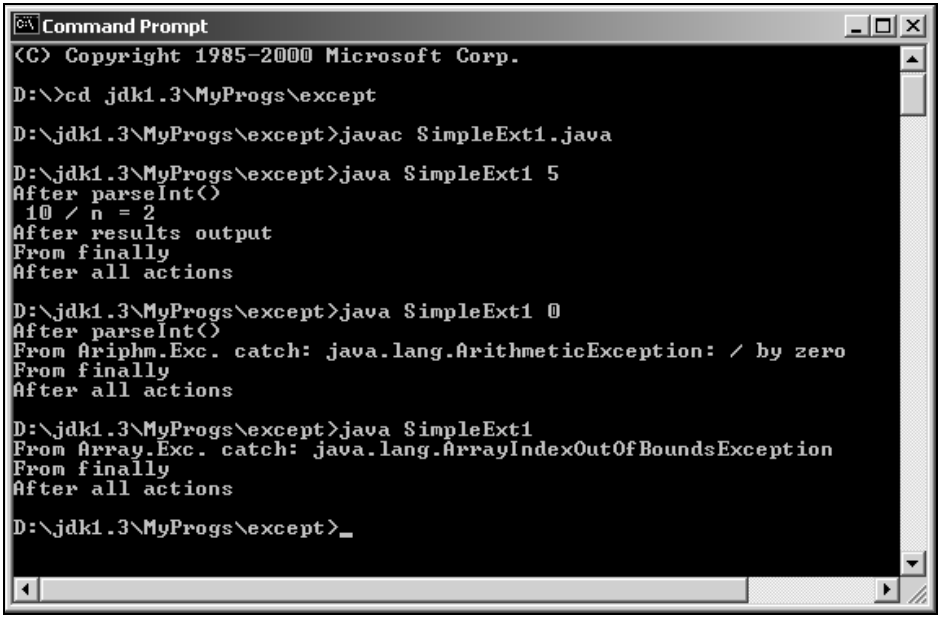


Рис. 21.2. Сообщения обработки исключений

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 21.1, 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. 21.2.

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 ранее было сказано, что выброшенное исключение "пролетает" через всю программу. Что это означает? Изменим программу листинга 21.2, вынеся деление в отдельный метод `f()`. Получим листинг 21.3.

Листинг 21.3. 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");
}
}
```

Скомпилировав и запустив программу листинга 21.3, убедимся, что вывод программы не изменился, он такой же, как на рис. 21.2. Исключение, возникшее при делении на нуль в методе `f()`, "пролетело" через этот метод, "вылетело" в метод `main()`, там перехвачено и обработано.

Упражнения

1. Просмотрите внимательно листинги предыдущих глав и подумайте, где в них требуется обработка исключительных ситуаций.
2. Вставьте в листинги предыдущих глав обработку исключительных ситуаций.

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 21.3? Дело в том, что спецификация JLS делит все исключения на *проверяемые* (checked), т. е. 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 21.1, 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 21.4. Результат показан на рис. 21.3.

Листинг 21.4. 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");
  }
}
```

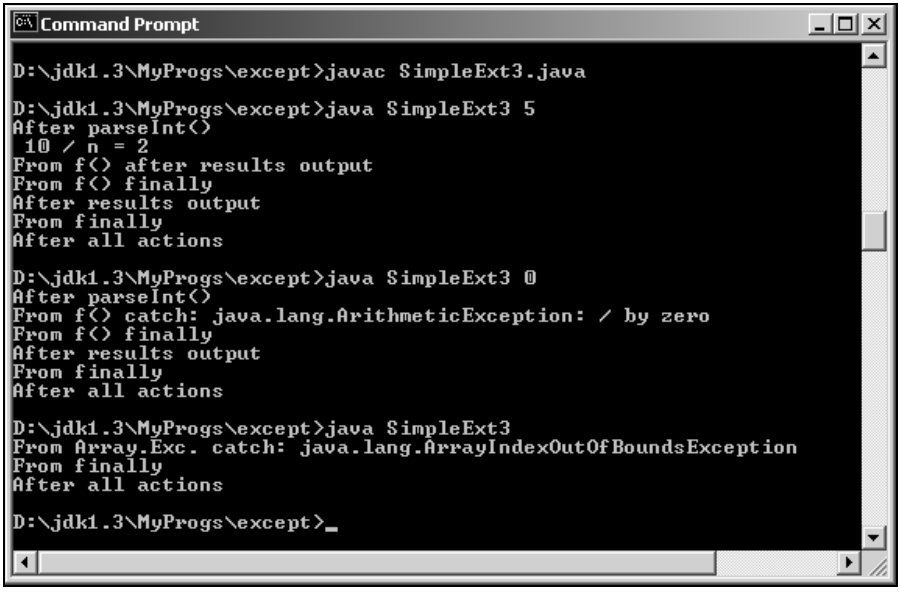


Рис. 21.3. Обработка исключения в методе

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, написанный в методе `main()`.

При необходимости исключение можно выбросить оператором `throw ae`. В листинге 21.4 этот оператор заключен в комментарий. Уберите символы комментария `//`, перекомпилируйте программу и посмотрите, как изменится ее вывод.

При переопределении в подклассах метода, выбрасывающего исключения, либо вообще не пишется часть заголовка `throws`, либо снова перечисляются те же классы исключений или их подклассы. Кроме того, можно перечислить не все классы.

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 21.2, получив листинг 21.5.

Листинг 21.5. 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");
  }
}
```

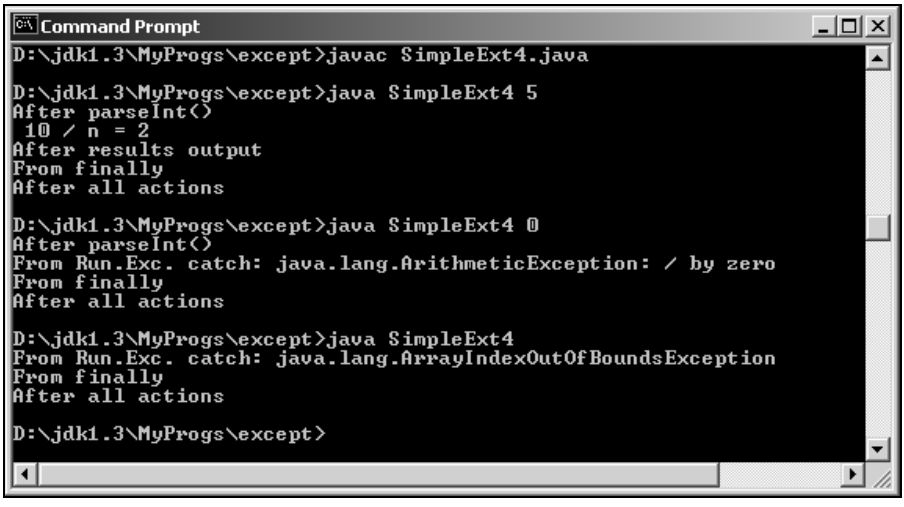


Рис. 21.4. Перехват нескольких типов исключений

В листинге 21.5 два блока `catch(){}` заменены одним блоком, перехватывающим исключение класса `RuntimeException`. Как видно на рис. 21.4, данный блок перехватывает оба исключения. Почему? Потому что это исключения подклассов класса `RuntimeException`.

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 J2SE JDK. 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 словом `throws`.

Имена классов-исключений, по соглашению, заканчиваются словом `Exception`.

Порядок обработки исключений

Блоки `catch(){}` перехватывают исключения в порядке написания этих блоков. Данное правило приводит к интересным результатам.

В листинге 21.2 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 21.4 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(){}`.

Упражнение

3. Обработайте все возможные исключительные ситуации, возникающие в предыдущих листингах.

Создание собственных исключений

Прежде всего, нужно четко определить ситуации, в которых будет возникать ваше собственное исключение, и подумать, не станет ли его перехват невольно причиной перехвата также и других, не учтенных вами исключений.

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, который назовем `CipherException`, прямо от класса `Exception`. Определим класс `CipherException`, как показано в листинге 21.6, и используем его в классе `ExceptDemo`. На рис. 21.5 продемонстрирован вывод этой программы.

Листинг 21.6. 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();
      }
  }
}
```

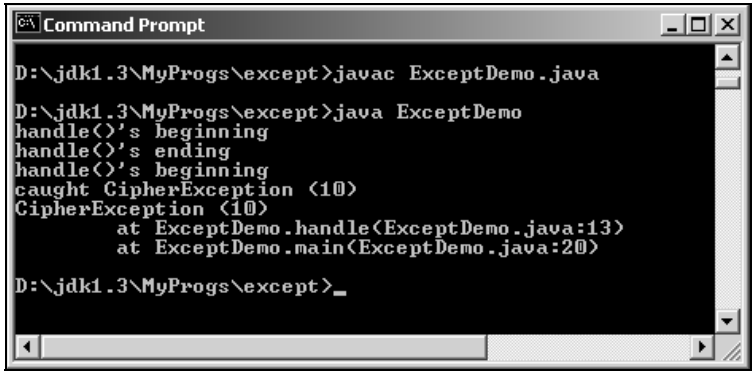


Рис. 21.5. Обработка собственного исключения

Заключение

Обработка исключительных ситуаций стала сейчас обязательной частью объектно-ориентированных программ. Применяя методы классов Java SE 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.

Вопросы для самопроверки

1. Почему в объектно-ориентированных языках принята модель обработки исключительных ситуаций, отличная от модели, принятой в процедурных языках?
2. Можно ли вкладывать друг в друга блоки обработки исключений?
3. Можно ли в блоках обработки исключений применить оператор `return`?

4. Можно ли в блоках обработки исключений снова вызвать исключение?
5. Можно ли в одном блоке обработки исключений написать несколько блоков `catch(){}`?
6. Можно ли в одном блоке обработки исключений написать несколько блоков `finally{}`?
7. В каких случаях в заголовке метода, выбрасывающего исключения, может отсутствовать часть `throws`?
8. С какой целью создана разветвленная иерархия классов исключений?
9. Можно ли создать свои собственные классы-исключения?



Глава 22

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, например, 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. Системы MS Windows 2000/Server 2003/XP/Vista 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 большим числом 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 light-weight 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 некоторые 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 может находиться в одном из пяти состояний, помеченных следующими константами класса-перечисления (enum) `Thread.State`, статически вложенного в класс `Thread`:

- `NEW` — подпроцесс создан, но еще не запущен;
- `RUNNABLE` — подпроцесс выполняется;
- `BLOCKED` — подпроцесс блокирован;
- `WAITING` — подпроцесс ждет окончания работы другого подпроцесса;
- `TIMED_WAITING` — подпроцесс ждет некоторое время окончания другого подпроцесса;
- `TERMINATED` — подпроцесс окончен.

Эти константы — объекты класса `Thread.State`.

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();
```

Состояние подпроцесса можно определить методом `getState()`, возвращающим одну из констант класса `Thread.State`.

Класс `Thread` реализует интерфейс `Runnable`, который 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` восемь конструкторов. Основной из них,

```
Thread(ThreadGroup group, Runnable target, String name, long stackSize);
```

создает подпроцесс с именем `name`, принадлежащий группе `group` и выполняющий метод `run()` объекта `target`. Последний параметр, `stackSize`, задает размер стека и сильно зависит от операционной системы.

Все остальные конструкторы 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, 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 способна отсчитывать наносекунды, то можно приостановить подпроцесс с точностью до наносекунд методом

```
sleep(long ms, int nanosec);
```

В листинге 22.1 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 22.1. 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. 22.1 показан результат двух запусков программы листинга 22.1. 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.

Уберем в листинге 22.1 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. 22.2 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.



Рис. 22.1. Два подпроцесса работают без задержки



Рис. 22.2. Подпроцессы работают с задержкой

Как же добиться согласованности, как говорят, *синхронизации* (synchronization) подпроцессов? Обсудим это позже, а пока покажем еще два варианта создания той же самой программы.

В листинге 22.2 приведен второй вариант программы: сам класс `TwoThreads2` является расширением класса `Thread`, а метод `run()` реализуется прямо в нем.

Листинг 22.2. 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 22.3. Здесь нельзя использовать методы класса `Thread`, но зато класс `TwoThreads3` может быть расширением другого класса. Например, можно сделать его аплетом, расширив класс `Applet` или `JApplet`.

Листинг 22.3. 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: 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 22.4 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 J2SE JDK. Главный подпроцесс в данном примере только создает объекты-подпроцессы, ждет 1 секунду и останавливает их.

Листинг 22.4. 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
rest += sum;          // Изменяем остаток на величину sum
myDep.setRest(rest); // 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. 22.1 и 22.2, 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 setRest() 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();
   rest += sum;
   myDep.setRest(rest);
}
```

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 void setRest(int rest){
   // Тело метода
}
```

В этом случае блокируется объект, выполняющий метод, т. е. объект 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 22.5 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 22.5 как комментарий. Результат выполнения программы представлен на рис. 22.3.

Листинг 22.5. 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();
  }
}
```

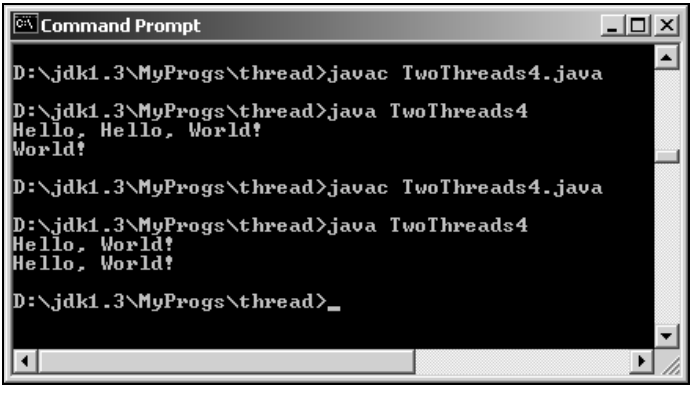


Рис. 22.3. Синхронизация метода

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 быстро-

ты ее выполнения критических участков должно быть как можно меньше, и они должны быть как можно короче.

Многие методы классов Java 2 JDK синхронизированы. Обратите внимание, что на рис. 22.1 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()`, которые позволяют приостановить работу подпроцесса с этим объектом, разрешают другому подпроцессу поработать с объектом, а затем уведомляют (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 пошлет 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 20*. 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 22.6 класс `Store` не обеспечивает согласования получения и выдачи информации. Результат работы показан на рис. 22.4.

Листинг 22.6. 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();
  }
}
```

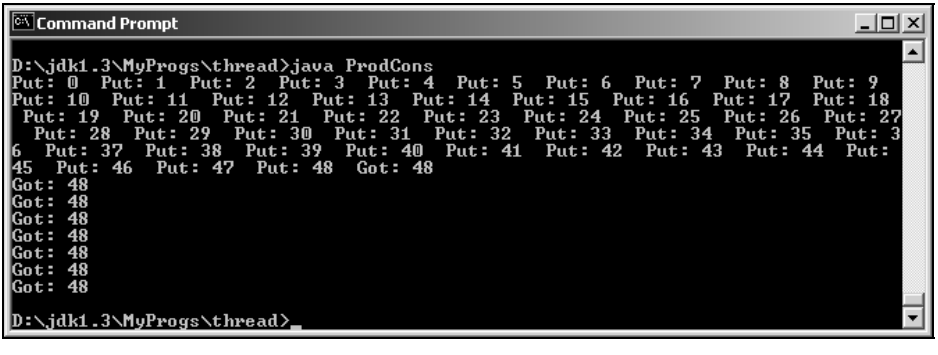


Рис. 22.4. Несогласованная работа двух подпроцессов

В листинге 22.7 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. 22.5.

Листинг 22.7. 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();
  }
}
```

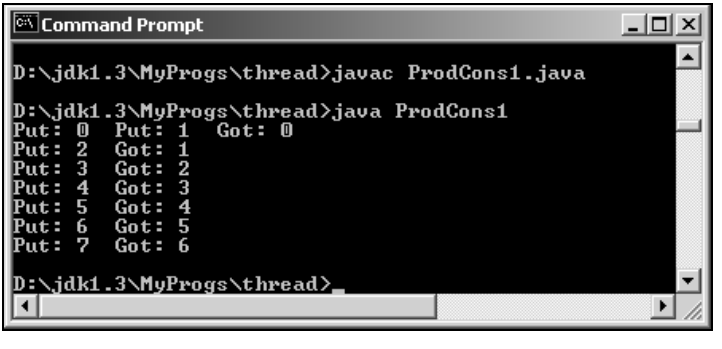


Рис. 22.5. Согласованная работа подпроцессов

Поскольку уведомление поставщика в методе `getInform()` должно происходить уже после отправки информации оператором `return inform`, оно включено в блок `finally{}`.

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 необходимо 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()`, который 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 собственные 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.

Вопросы для самопроверки

1. Что такое процесс и подпроцесс в современных операционных системах?
2. Почему языки программирования, как правило, не содержат средств управления подпроцессами?
3. Зачем в язык Java введены средства создания и управления подпроцессами?
4. Какими способами можно создать и запустить подпроцесс?
5. Когда подпроцесс заканчивает свою работу?
6. Как можно остановить подпроцесс?
7. Что такое "монитор" в теории операционных систем?
8. Каким образом монитор реализуется в Java?



Глава 23

Потоки ввода/вывода и печать

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()`, которые в классе `PrintStream` перегружены около двадцати раз 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, так называемые *log-файлы*, а не выводятся на кон-

соль. В Java есть средства переназначения потока. Можно, например, переназначить поток с консоли в файл.

Поток `in` — это экземпляр класса `InputStream`. Стандартно он назначен на клавиатурный ввод с консоли, который выполняется 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 исключение класса `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 несколько методов:

- `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`;
- `Writer append(char c)` — добавляет символ к выходному потоку, возвращая ссылку на этот поток;
- `Writer append(CharSequence seq)` — добавляет последовательность символов к выходному потоку;
- `Writer append(CharSequence seq, int offset, int len)` — добавляет к потоку `len` символов последовательности `seq`, 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()`, который сразу же выводит все содержимое буфера в поток.

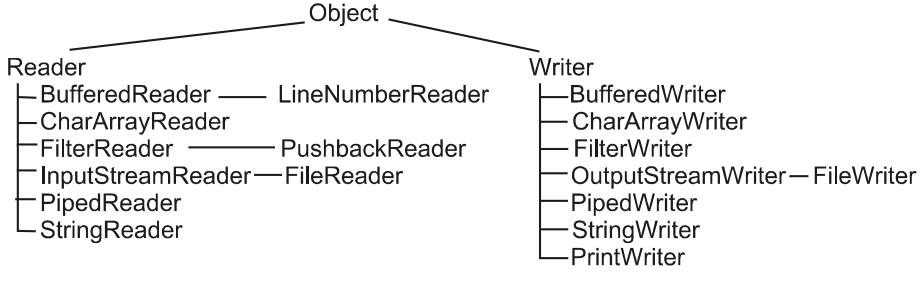


Рис. 23.1. Иерархия классов символьных потоков

Наконец, по окончании работы с потоком его необходимо закрыть методом `close()`.

Классы, входящие в иерархии потоков ввода/вывода, показаны на рис. 23.1 и 23.2.

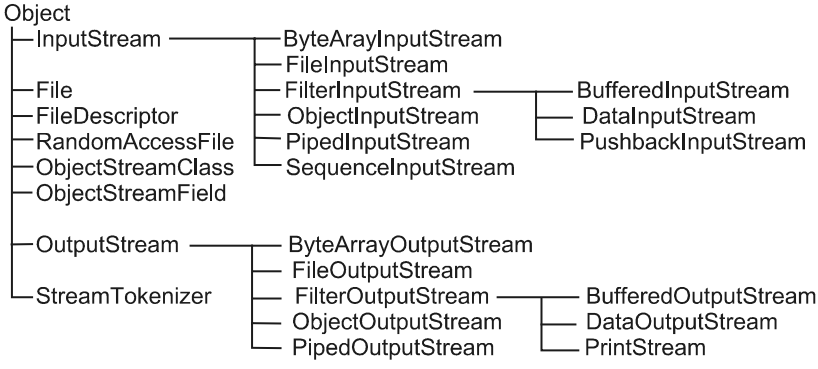


Рис. 23.2. Классы байтовых потоков

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
FileWriter                    FileOutputStream
```

- 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'`.

Особняком стоит класс `RandomAccessFile`, реализующий прямой доступ к файлу. Он не создает поток байтов или символов, а позволяет непосредственно обратиться к любому байту файла.

Еще один особенный класс `Console`, не создающий поток, выполняет ввод-вывод, связанный с консолью.

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, 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 уже 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/Server 2003. 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 выполнения метода print() буфер не сбрасывается! Для сброса буфера после каждого обращения к методу print() надо обращаться к методу flush().

> ***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 23.1 и на рис. 23.3.

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 23.1 приведена программа, иллюстрирующая перечисленные методы консольного ввода/вывода. На рис. 23.3 показан вывод этой программы.

Листинг 23.1. 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);
    }
  }
}
```

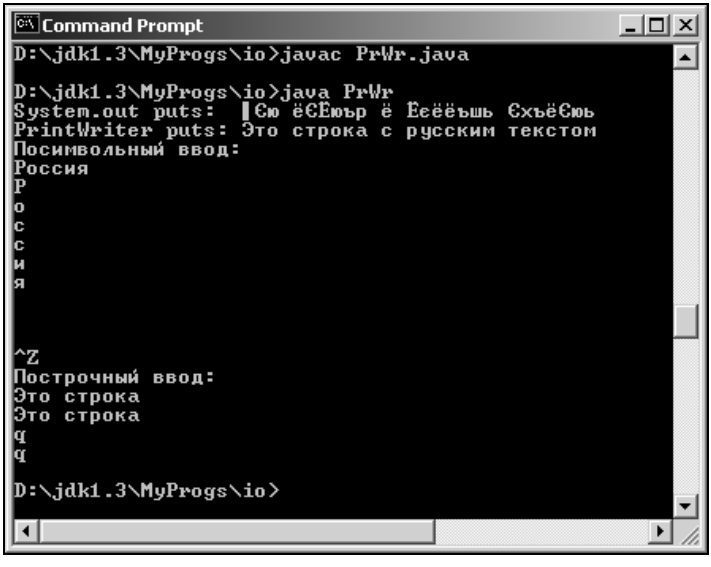


Рис. 23.3. Консольный ввод/вывод

Поясним рис. 23.3. 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 выведены 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>.

Форматированный вывод

На технологию Java традиционно переходит очень много программистов, прежде писавших программы на языке C. Им очень не хватает функции `printf()`, позволяющей самому программисту как-то оформить вывод информации: задать количество цифр при выводе вещественных чисел, точно указать количество пробелов между данными. Начиная с JDK 5.0 методы `printf()`, очень похожие на одноименные функции языка C, появились в классах `PrintStream` и `PrintWriter`. Кроме них, в эти классы введены методы `format()`, выполняющие те же действия. Последние методы заимствованы из класса `Formatter`, находящегося в пакете `java.util` и специально предназначенного для форматирования.

Заголовки методов форматированного ввода/вывода класса `PrintStream` выглядят так:

```
PrintStream format(Local l, String format, Object ... args);
PrintStream format(String format, Object ... args);
PrintStream printf(Local l, String format, Object ... args);
PrintStream printf(String format, Object ... args);
```

В классе `PrintWriter` такие же методы возвращают ссылку на свой экземпляр класса `PrintWriter`.

Как видите, при форматировании эти методы могут учесть локальные установки даты, времени, денежных единиц, взятые из объекта класса `Locale`. Если данный аргумент отсутствует, то соответствующие установки будут взяты из локали по умолчанию.

Строка символов `format` описывает шаблон для вывода данных, перечисленных в следующих аргументах метода, а также содержит надписи, которые должны появиться на консоли. Например, тот же самый вывод на консоль, который мы до сих пор делали методом

```
System.out.println("x = " + x + ", y = " + y);
```

можно сделать методом

```
System.out.printf("x = %d, y = %d\n", x, y);
```

В строке формата мы пишем поясняющий текст `"x = , y = \n"`, который будет просто выводиться на консоль. В текст вставляем *спецификации формата* `"%d"`. На место этих спецификаций во время вывода будут подставлены значения данных, перечисленных в следующих аргументах метода. Вместо первой спецификации появится значение первой переменной в списке, т. е. `x`, вместо второй — значение второй переменной, в нашем примере это переменная `y`, и т. д. Если знак процента надо вывести, а не понимать как начало спецификации, то его следует удвоить, например:

```
System.out.printf("Увеличение на %d%% процентов\n", x);
```

Если спецификаций окажется больше, чем данных в списке, то будет выброшено исключение класса `MissingFormatArgumentException`, если меньше, то последние данные, для которых "не хватило" спецификаций, просто не станут выводиться.

Если нужно изменить порядок вывода, то в спецификации можно явно написать порядковый номер выводимого аргумента, завершив его знаком доллара: `%1$d`, `%2$d`, `%3$d`. Например, если написать

```
System.out.printf("x = %2$d, y = %1$d\n", y = ", x, y);
```

то на консоль сначала будет выведено значение переменной `y`, а потом значение переменной `x`.

Можно несколько раз вывести одно и то же значение, например два раза вывести значение второй переменной:

```
System.out.printf("x = %2$d, y = %2$d\n", y = ", x, y);
```

Каждая спецификация начинается со знака процента, а заканчивается одной или двумя буквами a, A, b, B, c, C, d, e, E, f, g, G, h, H, n, o, s, S, t, T, x, X, показывающими тип выводимого данного. Буква d, использованная нами в предыдущем примере, показывает, что соответствующее этой спецификации данное следует выводить на консоль как целое число в десятичной системе счисления.

Как показано ранее, после знака процента можно указать порядковый номер аргумента, завершенный знаком доллара. Между знаком процента или доллара и буквой, обозначающей тип и завершающей спецификацию, могут находиться дополнительные символы, число и вид которых зависит от спецификации.

Спецификации вывода целых чисел

Для спецификации d можно задать количество позиций, выделяемых для выводимого данного, например %10d или %2$10d. Если целое число невелико и займет меньше места, то оно будет прижато к правому краю поля из 10 позиций, а слева останутся пробелы. Если оно велико, содержит больше 10 цифр, то поле будет расширено так, чтобы все число было выведено. В спецификации сразу же после знака процента или доллара можно поставить дефис: %-10d, %2$-10d, и тогда число будет прижиматься к левому краю отведенного для него поля, а пробелы останутся справа.

Если вместо пробелов вы хотите вывести слева незначащие нули, то напишите нуль в спецификации: %010d, %2$010d.

В больших числах группы по три цифры: тысячи, сотни тысяч, часто разделяются пробелами или запятыми. Такую разбивку можно указать запятой %,10d, %2$,10d, но этот элемент форматирования сильно зависит от локали, и надо проверить, как он действует на вашей машине.

Положительное число обычно выводится без начального плюса. Если поставить знак плюс в спецификации, %+10d, %2$+10d, то положительные числа будут выведены с плюсом, а отрицательные по-прежнему с минусом. Если вместо плюса оставить один пробел, то знак плюс у положительного числа не будет выводиться, но вместо него останется пробел. Это удобно для формирования колонок чисел.

Целое число выводится по спецификации d в десятичной системе счисления. Спецификация o выводит целое число в восьмеричной системе счисления, спецификации x и X — в шестнадцатеричной системе счисления. Если написана малая буква x, то шестнадцатеричные цифры будут записаны малыми буквами, например d2cf10, а если написана заглавная буква X — то заглавными, D2CF10.

В спецификациях o, x, X знак плюс и пробел можно указывать только для данных типа BigInteger, а запятую вообще нельзя писать. Зато в этих спецификациях можно написать "решетку" (#). Тогда восьмеричное число будет выведено с начальным нулем, а шестнадцатеричное — с начальными символами 0x или 0X, как это принято при записи констант Java.

Спецификации вывода вещественных чисел

Для вывода вещественных чисел предназначены спецификации `a`, `A`, `e`, `E`, `f`, `g`, `G`. Спецификация `f` выводит вещественное число в десятичной системе счисления с фиксированной точкой, спецификации `e` и `E` — с плавающей точкой. Две последние спецификации отличаются только регистром выводимой буквы `E`. Спецификации `g`, `G` универсальны — они выводят короткие числа с фиксированной точкой, а длинные — с плавающей точкой.

Во всех этих спецификациях можно использовать те же символы, что и при выводе целых чисел. Дополнительно можно записывать точность вывода числа — количество цифр в его дробной части. Точность записывается после точки в конце спецификации, например `%.8f`, `%2$.6e`, `%+10.5E`, `%2$-10.4g`, `%2.5G`. По умолчанию точность равна 6 цифрам.

Спецификации `a`, `A` выводят число в шестнадцатеричной системе счисления с плавающей точкой. В этих спецификациях точность записывать нельзя.

Спецификация вывода символов

Спецификация `c` выводит один символ в кодировке Unicode. В данной спецификации можно записывать только ширину поля и дефис для вывода символа в левой позиции этого поля. Например: `%c`, `%2$6c`, `%-10C`, `%2$-3c`.

Спецификации вывода строк

Спецификации `s`, `S` выводят строку символов. Соответствующий аргумент должен быть ссылкой на объект, которая преобразуется в строку своим методом `toString()`. Если строка пуста, то выводится слово `null`. В этих спецификациях можно записывать только ширину поля и дефис для вывода строки в левой позиции этого поля. Если спецификация записана заглавной буквой `S`, то символы строки преобразуются в верхний регистр. Например: `%s`, `%2$6S`, `%-10s`, `%2$-50S`.

Спецификации вывода логических значений

Спецификации `b`, `B` выводят логическое значение словом `true` или `false`. В данных спецификациях можно записывать только ширину поля и дефис для вывода слов в левой позиции этого поля. Если спецификация записана заглавной буквой `B`, то слова выводятся заглавными буквами: `TRUE`, `FALSE`. Например: `%b`, `%2$6B`, `%-10b`, `%2$-3b`.

Спецификации вывода хеш-кода объекта

Спецификации `h`, `H` выводят хеш-код объекта в шестнадцатеричной системе счисления. Если написана буква `h`, то шестнадцатеричные цифры выводятся малыми буквами, если написана буква `H`, то заглавными буквами. В этой спецификации можно задавать только ширину поля и дефис для вывода кода в левой позиции этого поля. Например: `%h`, `%2$6H`, `%-10h`, `%2$-3h`.

Спецификации вывода даты и времени

Спецификации `t`, `T` выводят дату и время по правилам заданной локали. Дата и время для вывода по этим спецификациям должны быть заданы в секундах и миллисекундах в виде

целого числа типа `long` или объекта класса `Long`, а также в виде объекта классов `Calendar` или `Date`.

После буквы `t` или буквы `T` обязательно должна быть записана еще одна буква, указывающая объем выводимой информации: дата и время, только дата, только время, только часы и так далее. Например, метод

```
System.out.printf("Местное время: %tT\n", Calendar.getInstance());
```

выведет только текущее время, без даты, в местной локали. В русской локали получим запись вида 12:36:14.

Метод

```
System.out.printf("Сейчас %tH часов\n", Calendar.getInstance());
```

выведет только часы, например 12. Спецификация `%tM` выведет только минуты, а спецификация `%tS` — только секунды.

Мы можем отформатировать время по-своему, написав, например, метод

```
System.out.printf("Местное время: %1$tH часов, %1$tM минут\n",
    Calendar.getInstance());
```

Кроме этого, по спецификации `%ts` можно получить время в секундах, начиная с 1 января 1970 года (Epoch), а по спецификации `%tQ` — в миллисекундах, отсчитанных от той же даты.

Дату в виде 10/24/08 можно получить по спецификации `%tD`, а в виде 2008-10-24 — по спецификации `%tF`. Ни та, ни другая форма не соответствует российским стандартам. Привычный для нас вид представления даты — 24.10.08 — можно получить методом

```
System.out.printf("Сегодня %1$td.%1$tm.%1$ty\n", Calendar.getInstance());
```

Спецификация `%td` дает день месяца всегда двумя цифрами, например первый день месяца будет записан с начальным нулем: 01. Если вы хотите записать первые дни месяца одной цифрой, то используйте спецификацию `%te`.

Спецификация `%ty` записывает только две последние цифры года. Полная запись года четырьмя цифрами получится по спецификации `%tY`.

Название месяца, записанное полным словом, можно получить по спецификации `%tB`, а записанное сокращением из первых трех букв — по спецификации `%tb` или `%th`.

День недели полным словом можно получить по спецификации `%tA`, а сокращением из первых трех букв — по спецификации `%ta`.

Наконец, день недели, дату и время полностью можно получить по спецификации `%tc`.

Класс *Console*

Как видно из приведенного ранее текста, связь с консолью средствами классов-потоков весьма сложна. Начиная с Java SE 6 в пакет `java.io` добавлен класс `Console`, облегчающий эту задачу.

Поскольку программа связывается с той консолью, в которой запущена виртуальная машина Java, единственный объект класса `Console` создается статическим методом `console()` класса `System`, например:

```
Console cons = System.console();
```

Метод возвращает `null`, если виртуальная машина Java запущена не из консоли, а каким-нибудь приложением. Поэтому обычный метод получения ссылки на объект `cons` таков:

```
Console cons;

if ((cons = System.console()) != null){

   // Работаем с консолью...

}
```

Чтение строки символов с консоли выполняется методами

```
public String readLine();
public String readLine(String format, Object... args);
```

Ввод пароля можно выполнить методами

```
public char[] readPassword();
public char[] readPassword(String format, Object... args);
```

Эти методы возвращают `null`, если ввод завершен, например, нажатием <Ctrl>+<D> в UNIX или <Ctrl>+<Z> в MS Windows.

Еще один метод,

```
Reader reader();
```

возвращает ссылку на объект типа `Reader`, с помощью которого можно выполнить потоковый ввод символов с консоли.

Для вывода информации на консоль есть метод

```
Console printf(String format, Object... args);
```

Кроме того, можно получить ссылку на объект класса `PrintWriter`:

```
PrintWriter writer();
```

Наконец, очистку буфера консоли выполняет метод

```
void flush();
```

Упражнения

1. Известная программа "Алиса" отвечает на задаваемые ей вопросы, просто переставляя порядок слов в вопросе. Напишите свой вариант этой программы для консоли.
2. Запишите метод вывода даты в виде "25 декабря 2007 года", записывающем название месяца в правильном падеже.

Файловый ввод/вывод

Поскольку файлы в большинстве современных операционных систем понимаются как последовательность байтов, для файлового ввода/вывода создаются байтовые потоки с помо-

щью 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("ch23.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 ранее.

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 — три символа: 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 заданным именем, поэтому после создания объекта класса `File` следует это проверить логическим методом `exists()`.

Класс `File` содержит около сорока методов, позволяющих узнать и изменить 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)` или `listFiles(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 года методом `lastModified()`. Если файл не существует, эти методы возвращают нуль. Время последней модификации можно изменить методом `setLastModified(long time)`.

Логические методы `canRead()`, `canWrite()`, `canExecute()` показывают права доступа к файлу или каталогу, а несколько методов — `setReadable()`, `setWritable()`, `setExecu-`

`table()` — позволяют установить их для всех пользователей или только для владельца файла или каталога.

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.

Методы `getTotalSpace()`, `getFreeSpace()`, `getUsableSpace()` возвращают сведения о размерах пространства на разделе диска в виде длинного числа типа `long`.

Наконец, метод `toURI()` возвращает путь к файлу в форме адреса URI.

В листинге 23.2 показан пример использования класса `File`, а на рис. 23.4 — начало вывода этой программы.

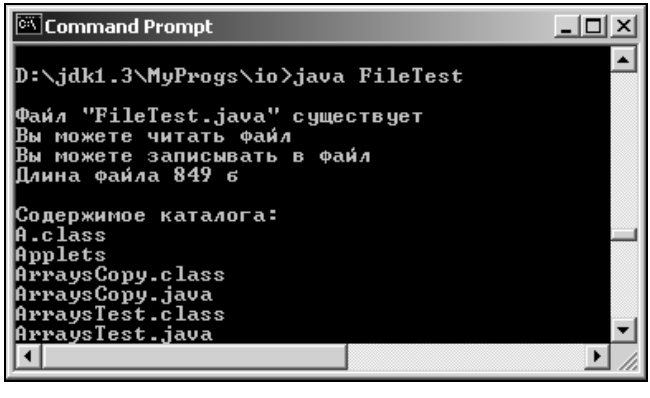


Рис. 23.4. Свойства файла и начало вывода каталога

Листинг 23.2. 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]);
  }
 }
}
```

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 ранее.

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 ранее.

Программа листинга 23.3 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 23.3. 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("Копирование окончено.");
 }
}
```

Каналы буферизованного ввода/вывода

Буферизация — очень мощное средство ускорения ввода и вывода данных. Поэтому во всех системах программирования ей уделяется особое внимание. Технология Java предлагает несколько пакетов `java.nio.*` (New Input-Output, NIO), обеспечивающих дополнительные средства работы с буферами ввода/вывода. Классы, собранные в эти пакеты, обеспечивают безопасную одновременную работу с буфером нескольких подпроцессов.

Основа классов, составляющих систему NIO, — это абстрактный класс `Buffer`. Он предлагает самые общие методы работы с любым буфером: метод вычисления его емкости `capacity()`, метод определения текущей позиции в буфере `position()`, методы изменения текущей позиции `reset()` и `rewind()` и другие методы.

Класс `Buffer` расширяется буферами для хранения данных простых типов — абстрактными классами `ByteBuffer`, `CharBuffer`, `ShortBuffer`, `IntBuffer`, `LongBuffer`, `FloatBuffer` и `DoubleBuffer`. Они уже могут предложить методы `get()` получения данных из буфера и методы `put()` занесения данных в буфер. Кроме этих общих для всех классов методов, каждый класс содержит методы, специфичные для своего типа данных.

Работа с буферами перечисленных классов обеспечивается интерфейсами и классами, собранными в пакеты `java.nio.channels` и `java.nio.channels.spi`. Их основу образует интерфейс `Channel`. Он описывает в самом общем виде открытую связь с источником данных. Такая связь представлена всего двумя методами: `isOpen()` и `close()`. Как видите, никакого канала еще не образуется.

Канал связи с буфером описывается интерфейсами `ReadableByteChannel`, `WritableByteChannel`, `GatheringByteChannel` и `ScatteringByteChannel`, расширяющими интерфейс `Channel`. Они содержат методы чтения, `read(ByteBuffer)`, и записи, `write(ByteBuffer)`, байтов в буфер. Еще один интерфейс, `ByteChannel`, объединяет методы интерфейсов `ReadableByteChannel` и `WritableByteChannel`.

Сам канал создается методами окончательного (final) класса `Channels`. В нем собраны методы создания потоков классов `InputStream`, `OutputStream`, `Reader` и `Writer`, соединенных с буфером каналами типов `ReadableByteChannel` и `WritableByteChannel`.

Канал класса `Channels` соединен с буфером общего типа, связанным с каким угодно источником данных. Он не учитывает специфику источника данных. Для работы с буфером файлового ввода/вывода специально создан абстрактный класс `FileChannel`. Его методы определены в классах `FileInputStream` и `FileOutputStream`, которые предоставляют ссылку на экземпляр класса `FileChannel` методом `getChannel()`.

Упражнения

3. Напишите программу, подсчитывающую количество символов, слов и строк в заданном текстовом файле.
4. Напишите программу, убирающую лишние пробелы из текстового файла.

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, точнее, 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 *DataOutputStream*

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 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 23.4 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. 23.5 показан вывод этой программы.

Листинг 23.4. 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);
      }
  }
}
```

```
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. 23.5. Ввод и вывод данных

Обратите внимание на то, что попытка чтения за концом файла выбрасывает исключение класса `IOException`, в листинге 23.4 его обработка заключается в закрытии файла и завершении программы.

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`. Этот класс не порождает поток байтов. Он может

читать файл или записывать в него информацию, начиная с любого байта файла, реализуя прямой доступ к файлу.

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, `"rw"` — открытие файла для чтения и записи, `"rwd"` — чтение и запись с немедленным обновлением источника данных и `"rws"` — чтение и запись с немедленным обновлением не только данных, но и метаданных.

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 с текущей позиции файла. При открытии файла конструктором указатель стоит на начале файла, в позиции 0. 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.

Текущую позицию файла можно узнать методом `getFilePointer()`.

Как и классы `FileInputStream` и `FileOutputStream`, класс `RandomAccessFile` предоставляет ссылку на экземпляр класса `FileChannel` методом `getChannel()`.

В классе нет методов преобразования символов в байты и обратно по кодовым таблицам, поэтому он не приспособлен для работы с кириллицей.

Упражнение

5. Выполните упражнение 4 с помощью прямого доступа к файлу.

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 23.5 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. 23.6 видна последовательность записи и чтения информации.

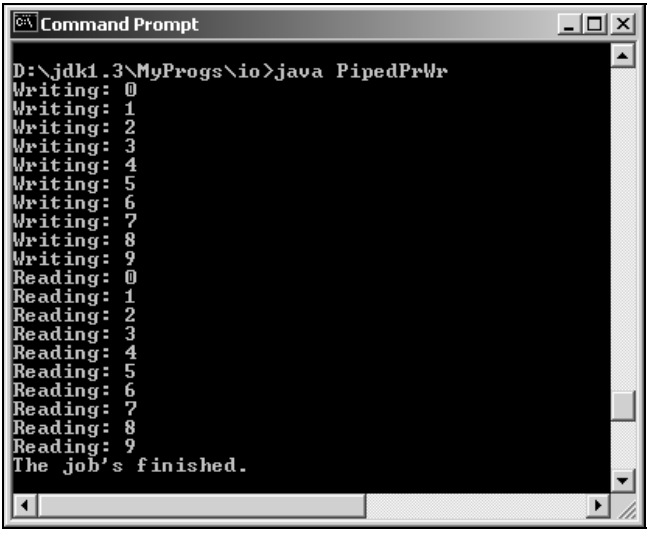


Рис. 23.6. Данные, передаваемые между подпроцессами

Листинг 23.5. 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();
  }
}
```

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 J2SE JDK 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 23.6 мы создаем объект класса `GregorianCalendar` с текущей датой и временем, сериализуем его в файл date.ser, через три секунды десериализуем обратно и сравниваем с текущим временем. Результат показан на рис. 23.7.

Листинг 23.6. 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));
  }
}
```

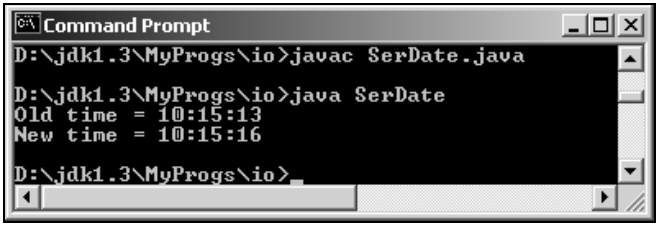


Рис. 23.7. Сериализация объекта

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 Java SE в каталоге docs/platform/serialization/spec/. В каталоге docs/technotes/guides/serialization/examples/ есть много примеров 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. Кроме того, в пакетах `javax.print.*` есть средства работы с сервером печати, но они выходят за рамки нашей книги.

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` определяет параметры страницы. Класс `PageProperties` тоже содержит пять подклассов со статическими константами, которые 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 разделе.

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 следует 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 23.7 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 23.7. 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 стан-

дартные 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 *Ox* направлена вправо, ось *Oy* — вниз;
- `LANDSCAPE` — начало координат в левом нижнем углу, ось *Ox* идет вверх, ось *Oy* — вправо;
- `REVERSE_LANDSCAPE` — начало координат в правом верхнем углу, ось *Ox* идет вниз, ось *Oy* — 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 23.9.

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** (Cancel), 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 определить 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** (Cancel).

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 сказанное вместе в листинге 23.8. В нем средствами Java 2D печатается то же самое, что и в листинге 23.7. Обратите внимание на п. 6. После окончания печати программа не заканчивается автоматически, для ее завершения мы обращаемся к методу `System.exit(0)`.

Листинг 23.8. 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 23.9 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 23.9. 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, 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 23.10 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 23.10. 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 pf1 = pj.defaultPage();
      pf1.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(), pf1);
          // 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;
   }
}
```

Вопросы для самопроверки

1. Что называется потоком (stream) данных?
2. Какие потоки ввода/вывода создаются исполняемой системой Java для каждой запущенной программы?
3. Как можно преобразовать поток ввода/вывода?
4. Как изменить кодировку символов в потоке?
5. Можно ли начать чтение файла не с его начала, а с произвольного места?
6. Можно ли вставить новую информацию в середину существующего файла?
7. Что такое буферизация ввода/вывода и для чего она нужна?
8. Что такое сериализация объекта?
9. Почему вопросы, относящиеся к печати, разобраны в этой главе, посвященной вводу/выводу?



Глава 24

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 иногда *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 клиента. 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 базируются 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 приемом и отправкой электронной почты, другая — пересылкой файлов, третья предоставляет Web-страницы. Для того чтобы различать программы-серверы, 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 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 телекомпания "Первый канал", на втором — "РТР" и т. д. Другой пример — поликлиника, в которой больных клиентов обслуживают врачи-серверы. Врачи сидят в своих кабинетах, различающихся по номерам.

Чтобы равномерно распределить нагрузку на сервер, часто несколько портов прослушиваются несколькими экземплярами программы-сервера одного типа. Web-сервер кроме порта с номером 80 может прослушивать порт 81, 8080, 8001 и еще какой-нибудь другой.

В процессе передачи сообщения используется несколько протоколов. Даже при отправлении обычного письма мы берем лист бумаги и 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 в почтовое отделение, 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 выполнено 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: сначала вынимается из самолета, затем вынимается 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 можем быть 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 отправителя 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 Интернета, 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, например: 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 Интернета большое распространение получила информационная система WWW (World Wide Web), основанная на прикладном протоколе HTTP (Hypertext Transfer Protocol). В ней используется 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/query?user_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/

https://132.192.5.10:448/public/some.html

ftp://guest:password@lenta.ru/users/local/pub

file:///C:/text/html/index.htm

Если какой-то из элементов адреса URL отсутствует, то берется его стандартное значение. Например, номер порта протокола HTTP по умолчанию равен 80, протокола HTTPS — 443, а протокола FTP — 21. Если отсутствует путь к имени файла path/file, то сервер отправляет клиенту файл, определяемый хостом, так называемый "welcome file". Чаще всего это файл с именем index.html, index.htm или index.jsp.

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 `getProtocol()`, `getAuthority()`, `getHost()`, `getPort()`, `getPath()`, `getFile()`, `getQuery()`, `getUserInfo()`, 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 24.1 показывает, как легко можно получить файл из Интернета, пользуясь методом `openStream()`.

Листинг 24.1. 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 = null;
       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 связи 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 многочисленными методами `get`*`Xxx`*`()`. Вот некоторые из них:

- `getContentType()` — возвращает строку типа `String`, показывающую MIME-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 24.1, — это на самом деле сокращение записи

```
bhv.openConnection().getInputStream();
```

В листинге 24.2 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 отыскивает и 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 программы, принимает результат ее работы и отправляет его обратно клиенту.

CGI-программу можно написать на любом языке: C, C++, Pascal, Perl, ASP, 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) и страниц JSP, которые мы рассмотрим в *главах 26 и 27*. CGI-программы обычно лежат на сервере в каталоге cgi-bin.

Листинг 24.2. 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);
     }
  }
}
```

Упражнения

1. Напишите программу, получающую файлы с известных вам сайтов Интернета.
2. Напишите программу, связывающуюся с известными вам CGI-программами Интернета и передающую им необходимую информацию.

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 электрической 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.net`. В классе семь 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. Протокол содержит следующие действия:

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 \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 24.3 и серверной программе листинга 24.4.

Листинг 24.3. 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. И наоборот, 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, приостанавливая выполнение серверной программы. Когда запрос получен, метод устанавливает соединение с клиентом и возвращает объект класса `Socket`, через который сервер будет обмениваться информацией с клиентом.

Листинг 24.4. 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 клиент-серверных приложений. Вы можете запускать клиент и сервер на одной машине, пользуясь этим адресом.

Работа с proxy-сервером

В настоящее время выход в Интернет часто осуществляется через proxy-сервер, кэширующий полученную информацию и дающий возможность нескольким машинам выходить в Интернет с одним и тем же IP-адресом. Для того чтобы создавать сокеты, работающие через proxy-сервер, начиная с JDK 5.0 в класс `Socket` добавлен конструктор

```
Socket(Proxy proxy);
```

Этот конструктор использует ссылку на объект абстрактного класса `Proxy`, устанавливающего связь с proxy-сервером. Объект создается конструктором

```
Proxy(Proxy.Type type, SocketAddress address);
```

в котором указывается тип proxy-сервера — одна из констант:

- `DIRECT` — соединение без proxy-сервера;
- `HTTP` — соединение с proxy-сервером, обслуживающим протокол HTTP или FTP;
- `SOCKS` — соединение через proxy-сервер, работающий по протоколу SOCKS4 или SOCKS5.

Класс `SocketAddress`, содержащий адрес и порт сервера, — это абстрактный класс. На практике при создании адреса используется его расширение `InetSocketAddress`.

Соберем все это вместе. Обычный способ создания сокета, работающего через proxy-сервер, выглядит так:

```
Socket sock = new Socket(new Proxy(Proxy.Type.SOCKS,
    new InetSocketAddress("socks.domain.com", 1080)));
```

Упражнения

3. Напишите программу, передающую файлы по упрощенному протоколу FTP.
4. Установите связь через proxy-сервер с известными вам сайтами Интернета.

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 24.5 показан пример класса `Sender`, посылающего сообщения, набираемые в командной строке, на `localhost`, порт номер 1050. Класс `Recipient`, описанный в листинге 24.6, принимает эти сообщения и выводит их в свой стандартный вывод.

Листинг 24.5. 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 24.6. 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);
        }
      }
}
```

Упражнение

5. Создайте аналог электронной почтовой связи с помощью дейтаграмм.

Вопросы для самопроверки

1. Что такое сетевая архитектура "клиент-сервер"?
2. Чем отличается клиентская часть приложения от серверной части?
3. Может ли клиентская часть приложения взаимодействовать сразу с несколькими серверами?
4. Что такое сетевой протокол?
5. Что такое стек протоколов?
6. Какие протоколы входят в стек протоколов TCP/IP?
7. Что такое IP-адрес?
8. Что такое номер порта?
9. Что такое дейтаграмма?
10. Что такое DNS-имя?
11. Что такое адрес URL?
12. Как установить связь через прокси-сервер?



ЧАСТЬ V

WEB-ТЕХНОЛОГИИ JAVA



Глава 25

Web-инструменты Java

В этой главе мы вкратце перечислим важные аспекты технологии Java, до сих пор не освещенные в книге, но необходимые для дальнейшего изложения.

Архиватор *jar*

Для упаковки нескольких файлов в один архивный файл, со сжатием или без сжатия, в технологии Java разработан формат архивирования 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 jar-файлы автоматически включается каталог META-INF, содержащий несколько файлов с информацией об упакованных в архив файлах.

Архивные файлы очень удобно использовать в апплетах, о чем уже говорилось в *главе 18*, 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 не только в jar-архив, но 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 позже.

При установке JDK на MS Windows автоматически создается ассоциация расширения имени файла jar с интерпретатором javaw, которая действует при двойном щелчке мыши по имени файла, а именно:

```
"C:\jre1.6.0_02\bin\javaw.exe" -jar "%1" %*
```

Если такой ассоциации нет, то ее легко создать средствами Windows.

Архивные файлы удобны и просты для компактного хранения всей необходимой для работы программы информации. Программа может работать с файлами архива 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 посредством утилиты командной строки `jar`.

Правила использования 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. 25.1.

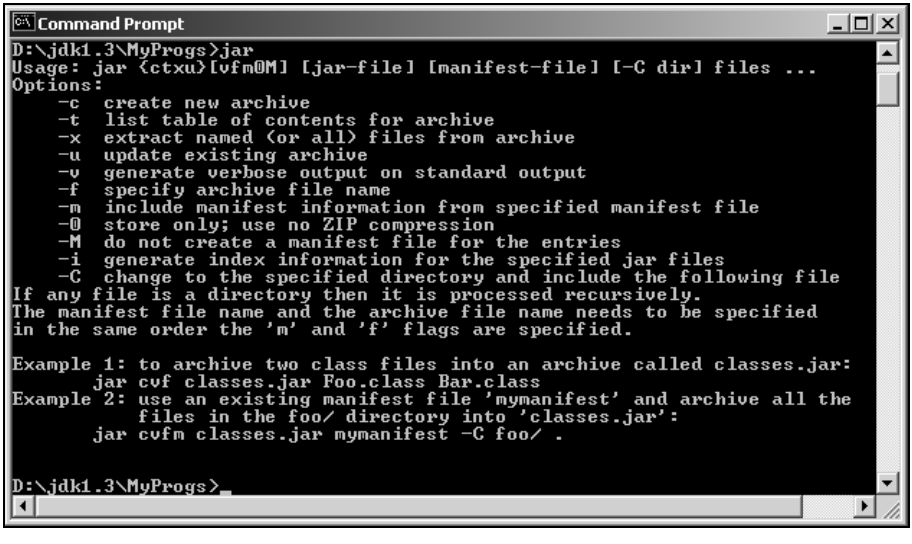


Рис. 25.1. Правила употребления утилиты `jar`

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) — направить 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.

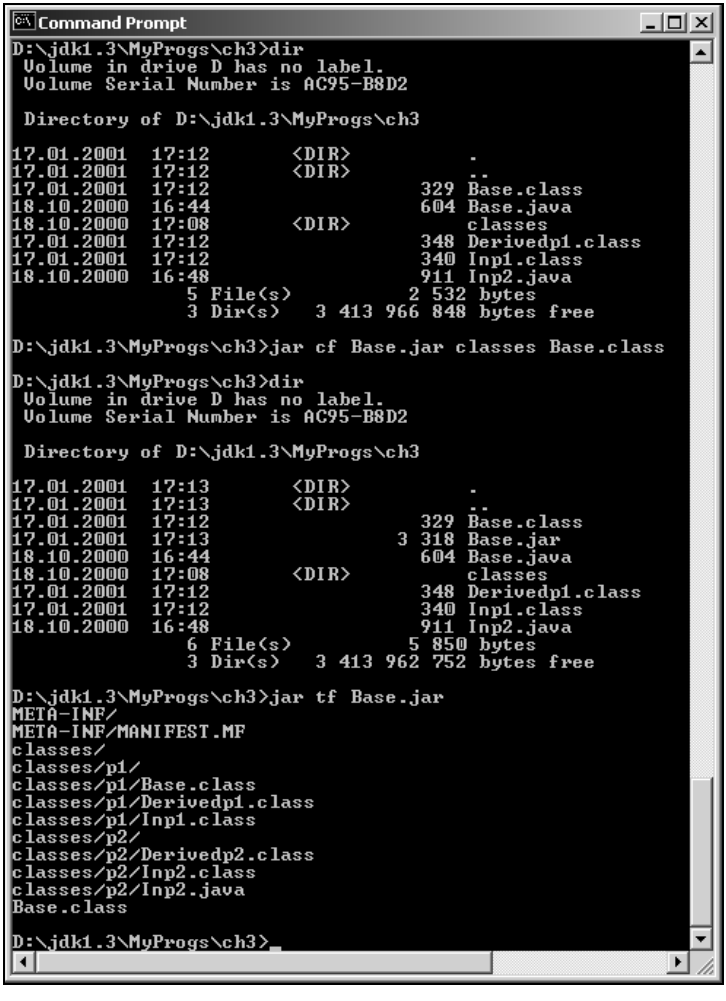


Рис. 25.2. Работа с утилитой `jar`

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;

- `e (entry)` — используется при создании архива. После буквы `e` записывается имя основного класса, содержащего метод `main()`, с которого начинается выполнение программы. Это имя будет занесено в создаваемый файл MANIFEST.MF (см. далее).

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.

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. 25.2 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;
- указания имени файла, содержащего изображение загрузочного окна;
- 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 SE, в файле docs/technotes/guides/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'`.

Имя файла, например name.gif, с изображением для загрузочного окна (splash screen) указывается строкой

```
SplashScreen-Image: name.gif
```

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 формирования 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. 25.3 представлено, как создается файл поиска и как выглядит содержимое архива после его создания.

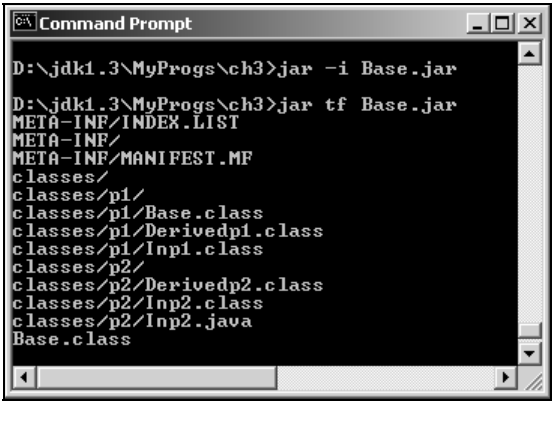


Рис. 25.3. Создание файла поиска

Компоненты JavaBeans

Многие программисты предпочитают разрабатывать приложения с графическим интерфейсом пользователя с помощью визуальных средств разработки IDE (Integrated Development Environment), таких как NetBeans, IntelliJ IDEA, Eclipse, JBuilder и др. Эти средства позволяют помещать компоненты в контейнер графически, с помощью мыши.

В окне приложения центральное место занимает форма, на которой размещаются компоненты. Сами компоненты показаны ярлыками на панели компонентов, расположенной обычно выше формы или сбоку от формы.

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.

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 обычно справа от формы. Окно свойств появляется чаще всего при выборе пункта меню Properties из контекстного меню, появляющегося при щелчке правой кнопкой мыши на компоненте. В левой колонке окна свойств перечислены имена свойств, в правую колонку надо записать их значения.

Потом можно задать обработку событий, открыв вторую страницу окна свойств или выбрав соответствующий пункт контекстного меню.

Для того чтобы компонент можно было применять в таком визуальном средстве разработки, как Eclipse, 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. Этими методами будет пользоваться IDE, чтобы определить свойства компонента.

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 III*, то вы тоже получаете свой JavaBean. Но для того чтобы не упустить каких-либо важных качеств JavaBeans, лучше использовать для их разработки специальные средства.

Компания Sun Microsystems поставляет набор необходимых утилит и классов BDK (Beans Development Kit) для разработки JavaBeans. Так же, как и JDK, 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 Интернете нужных компонентов и сборке из них приложения.

Связь с базами данных через JDBC

В основном информация 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 собственный 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 C. 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.

Компания 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`, а также пакет `javax.sql` и его подпакеты, входящие в Java SE.

Слово "JDBC" — это не аббревиатура, хотя иногда расшифровывается как "Java Database Connectivity".

Кроме классов с методами доступа к базам данных для каждой СУБД необходим драйвер JDBC — промежуточная программа, реализующая интерфейсы JDBC методами данной СУБД. Драйверы JDBC могут быть написаны разработчиками СУБД или независимыми фирмами. В настоящее время написано несколько сотен драйверов JDBC для разных СУБД под разные их версии и платформы. Их список можно посмотреть на странице **http://developers.sun.com/product/jdbc/drivers**.

Существуют четыре типа драйверов JDBC:

- драйвер, реализующий методы JDBC вызовами функций ODBC. Это так называемый *мост* (bridge) JDBC—ODBC. Непосредственную связь с базой при этом осуществляет драйвер ODBC, который должен быть установлен на той машине, на которой работает программа;
- драйвер, реализующий методы JDBC вызовами функций API самой СУБД. В этом случае на машине должен быть установлен клиент СУБД;
- драйвер, реализующий методы JDBC вызовами функций сетевого протокола, независимого от СУБД, например HTTP. Этот протокол должен быть, затем, реализован средствами СУБД;
- 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 25.1.

После того как драйвер установлен, можно 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 = "admin";
String password = "1nF4vb";
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.

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 полученной выборки — и извлечь данные многочисленными методами `get`*`Xxx`*`()` интерфейса `ResultSet`, например:

```
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 выборки методом `previous()`, к первому элементу — методом `first()`, к последнему — методом `last()`. Можно также модифицировать объект `rs` методами `updateXxx()` 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`, позволяющим создавать предварительно откомпилированный запрос, перед выполнением которого можно задавать аргументы методами `setXxx()`.

Интерфейс `PreparedStatement`, в свою очередь, расширен интерфейсом `CallableStatement`, в котором описаны методы выполнения хранимых процедур.

В листинге 25.1 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 25.1. 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 24* 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 по адресу: **http://java.sun.com/products/jdbc/**.

В документации Java SE, в каталоге docs/technotes/guides/jdbc/getstart/, есть пособие по использованию JDBC.

Вопросы для самопроверки

1. Почему в Java создан свой формат архивирования JAR?
2. Нужно ли распаковывать jar-архивы для использования классов, содержащихся в них?
3. Можно ли упаковывать апплеты в jar-архив?
4. Куда надо помещать jar-файлы для использования их в приложении?
5. Что такое JavaBeans: классы, интерфейсы, пакеты, правила оформления классов?
6. Что должно быть в классе, называемом JavaBean?
7. Где применяются JavaBeans?
8. Что такое JDBC?
9. Какие существуют типы драйверов JDBC?
10. Какие фирмы разрабатывают драйверы JDBC?



Глава 26

Сервлеты

Первоначально перед HTTP-серверами стояла простая задача: найти и отправить клиенту файл, указанный в полученном от клиента запросе. Запрос составлялся тоже очень просто по правилам протокола HTTP в специально придуманной форме URL.

Потом понадобилось сделать на сервере какую-либо небольшую предварительную обработку отправляемого файла. Появились включения на стороне сервера SSI (Server Side Include) и различные приемы динамической генерации страниц HTML. HTTP-сервер усложнился и стал называться Web-сервером.

Затем возникла необходимость выполнять на сервере процедуры. В запрос URL вставили возможность вызова процедур, а на сервере реализовали технологию CGI (Common Gateway Interface), о которой мы говорили в предыдущих главах. Теперь в запросе URL указывается процедура, которую надо выполнить на сервере, и записываются аргументы этой процедуры в виде пар "имя — значение", например:

http://some.firm.com/cgi-bin/mycgiprog.pl?name=Ivanov&age=27

Для составления таких запросов в язык HTML введен тег `<form>`.

Web-сервер, получив запрос, загружает и запускает CGI-процедуру (в предыдущем примере это процедура mycgiprog.pl), расположенную на сервере в каталоге cgi-bin, и передает ей значения "Ivanov" и "27" аргументов name и age. Процедура оформляет свой ответ в виде страницы HTML, которую Web-сервер отправляет клиенту.

Процедуру CGI можно написать на любом языке, лишь бы он воспринимал стандартный ввод и мог направить результат работы процедуры в стандартный вывод. Неожиданную популярность получил язык Perl. Оказалось, что на нем удобно писать CGI-программы. Возникли специальные языки: PHP, ASP, серверный вариант JavaScript.

Технология Java не могла пройти мимо такой насущной потребности и отозвалась на нее созданием сервлетов и языком JSP (JavaServer Pages).

Сервлеты (servlets) выполняются под управлением Web-сервера подобно тому, как апплеты выполняются под управлением браузера, откуда и произошло их название. Для слежения за работой сервлетов и управления ими создается специальный программный модуль, называемый *контейнером сервлетов* (servlet container). Слово "контейнер" в русском языке означает пассивную емкость стандартных размеров, но контейнер сервлетов активен, он загружает сервлеты, инициализирует их, передает им запросы клиентов, принимает ответы. Сервлеты не могут работать без контейнера, как апплеты не могут работать без браузера. Жаргонное выражение "сервлетный движок", происходящее от английского "servlet engine", лучше выражает суть дела, чем выражение "контейнер сервлетов".

Web-сервер, снабженный контейнером сервлетов и другими контейнерами, стал называться *сервером приложений* (application server, AS).

Чтобы сервлет мог работать, он должен быть зарегистрирован в контейнере, по терминологии спецификации "Java Servlet Specification" *установлен* (deploy) в него. *Установка* (deployment) сервлета в контейнер включает получение уникального имени и определение начальных параметров сервлета, запись их в конфигурационные файлы, создание каталогов для хранения всех файлов сервлета и другие операции. Процесс установки сильно зависит от контейнера. Одному контейнеру достаточно скопировать сервлет в определенный каталог, например autodeploy/ или webapps/, другому надо после этого перезапустить контейнер, для третьего надо воспользоваться утилитой установки. В стандартном контейнере Java EE SDK такая утилита называется deploytool.

Один контейнер может управлять работой нескольких установленных в него сервлетов. При этом один контейнер способен в одно и то же время работать в нескольких виртуальных машинах Java, образуя распределенное Web-приложение. Сами же виртуальные машины Java могут работать на одном компьютере или на разных компьютерах.

Контейнеры сервлетов создаются как часть Web-сервера или как встраиваемый в него модуль. Большую популярность получили встраиваемые контейнеры Tomcat, разработанный сообществом Apache Software Foundition в рамках проекта Jakarta, Resin фирмы Caucho, JRun фирмы Macromedia. Точное распределение обязанностей между Web-сервером и контейнером сервлетов выпадает на долю их производителей.

В стандартную поставку Java EE SDK входит контейнер сервлетов Tomcat как составная часть Java EE-сервера приложений Sun Java System Application Server. Можно скопировать Tomcat со страницы **http://tomcat.apache.org/** и установить отдельно. Для выполнения всех примеров этой и следующей главы понадобится Tomcat версии не меньше 5.0.

Web-приложение

Как правило, сервлет не выполняется один. Он работает в составе Web-приложения. *Web-приложение* (web application) составляют все ресурсы, написанные для обслуживания запросов клиента: сервлеты, JSP, страницы HTML, документы XML, другие документы, изображения и чертежи, музыкальные и видеофайлы. Спецификация “Java Servlet Specification” описывает структуру каталогов, содержащих все эти ресурсы. Она изображена на рис. 26.1.

Как видно из рисунка, все, что относится к данному Web-приложению, содержится в одном каталоге, имя которого будет именем Web-приложения. В этом каталоге обязательно должен быть каталог WEB-INF и необязательно другие каталоги и файлы. Здесь обычно располагаются так называемые Welcome-файлы, которые отправляются клиенту, если он не указал конкретный файл, а записал в запросе только имя Web-приложения. Все, что находится в каталоге WEB-INF и его подкаталогах, недоступно клиенту. Это "внутренняя кухня" Web-приложения. Все, что расположено вне каталога WEB-INF, доступно клиенту.

В каталоге WEB-INF обязательно должен быть конфигурационный XML-файл с именем web.xml, в котором описано Web-приложение: его ресурсы, их адреса и связи между ресурсами. Это минимальный обязательный состав Web-приложения — каталог с его именем, в нем подкаталог WEB-INF, а в нем файл web.xml.

Скомпилированные сервлеты обычно располагаются в подкаталогах каталога WEB-INF/classes в соответствии со своей структурой пакетов и подпакетов.

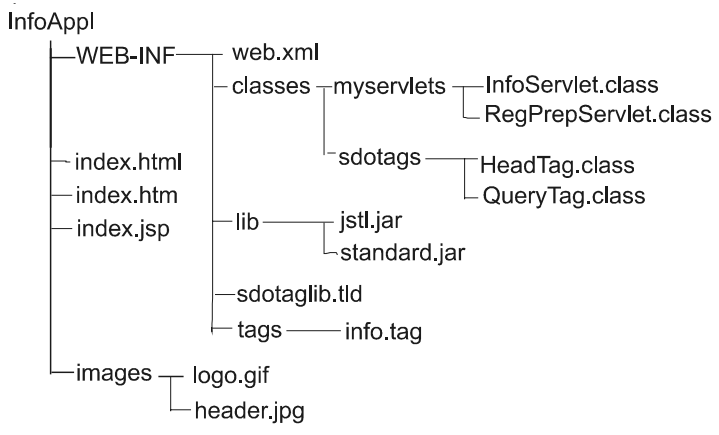


Рис. 26.1. Структура каталогов Web-приложения

Все Web-приложение целиком часто упаковывается в один файл по технологии JAR. Такой файл обычно получает расширение war (Web ARchive). Этот файл можно переносить с одного Web-сервера на другой, при этом многие контейнеры сервлетов могут запускать Web-приложение прямо из архива, не распаковывая его.

Интерфейс *Servlet*

Как водится в технологии Java, понятие "сервлет" описывается интерфейсом `Servlet`. Рассмотрим его подробнее.

Ранее уже говорилось о том, что сервлет выполняется в контейнере подобно тому, как апплет выполняется в браузере. Это сходство усиливается тем, что контейнер инициализирует сервлет методом `init()` так же, как браузер инициализирует апплет, но, в отличие от апплета, у метода `init()`, описанного интерфейсом `Servlet`, есть аргумент типа `ServletConfig`:

```
public void init(ServletConfig conf);
```

Объект, описанный интерфейсом `ServletConfig`, создается Web-приложением и передается контейнеру для инициализации сервлета. Информация, необходимая для создания объекта, содержится в конфигурационном файле Web-приложения.

Конфигурационный файл

Конфигурационный файл (deployment descriptor) описывает ресурсы, составляющие Web-приложение: сервлеты, страницы JSP, документы HTML и XML, изображения и документы других типов. Он формируется при создании Web-приложения и заполняется при установке сервлета и других ресурсов в контейнер. Конфигурационный файл записывается на языке XML и называется web.xml. Он располагается в каталоге WEB-INF, одном из каталогов Web-приложения, и создается вручную или утилитой установки сервлета в контейнер. Каждая фирма-производитель контейнера сервлетов предоставляет свою утилиту установки или make-файл, содержащий команды установки.

Надо заметить, что вместо построителя `make` в технологии Java используются другие построители, написанные на языке Java: построитель `ant`, разработанный Apache Software Foundation в рамках проекта Jakarta, построитель Maven — еще одна разработка Apache Software Foundation.

В стандартную поставку Java EE SDK входит графическая утилита установки Application Deployment Tool, вызываемая из командной строки набором команды `deploytool`. Утилита установки контейнера Tomcat запускается из браузера, она расположена на его странице /manager/html. Для того чтобы запустить утилиту, в браузере надо набрать примерно такую строку: **http://localhost:8080/manager/**.

В листинге 26.1 показан фрагмент конфигурационного файла web.xml, созданного для контейнера Tomcat. С языком XML мы познакомимся в *главе 28*, а пока смотрите разъяснения элементов XML в комментариях.

Листинг 26.1. Конфигурационный файл web.xml

```
   <?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
     "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

<web-app>
<!--
Если в запросе не указан ресурс, то вызывается сервлет по умолчанию.
Ниже записано его имя "default" и полное имя класса сервлета.
-->
    <servlet>
        <servlet-name>default</servlet-name>
        <servlet-class>
            org.apache.tomcat.servlets.DefaultServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
<!--
Если пришел запрос к сервлету, не описанному в данном файле,
то вызывается сервлет с именем "invoker".
-->
    <servlet>
        <servlet-name>invoker</servlet-name>
        <servlet-class>
            org.apache.tomcat.servlets.InvokerServlet
        </servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>0</param-value>
    </init-param>
```

```
    <load-on-startup>2</load-on-startup>
    </servlet>
<!--
Все запросы к страницам JSP вначале обрабатываются
сервлетом JspServlet. Ему дано имя "jsp".
-->
    <servlet>
        <servlet-name>jsp</servlet-name>
        <servlet-class>
            org.apache.jasper.runtime.JspServlet
           </servlet-class>

<!-- uncomment the following to use Jikes for JSP compilation
Уберите комментарий, если вы используете компилятор jikes.
        <init-param>
            <param-name>jspCompilerPlugin</param-name>
            <param-value>
              org.apache.jasper.compiler.JikesJavaCompiler
            </param-value>
        </init-param>
-->
        <load-on-startup>
            -2147483646
        </load-on-startup>
    </servlet>
<!--
Некоторым путям-псевдонимам сопоставляется сервлет, который
вызывается при указании в строке URL этого пути. Путь отсчитывается
относительно корневого каталога контейнера, чаще всего это
public_html или webapps.
Если в адресе URL указан каталог servlet, то вызывается сервлет
с именем "invoker", т. е. InvokerServlet.
-->
    <servlet-mapping>
        <servlet-name>invoker</servlet-name>
        <url-pattern>/servlet/*</url-pattern>
    </servlet-mapping>
<!--
Если в адресе указан путь к странице JSP, то вначале вызывается
сервлет с именем "jsp", т. е. сервлет JspServlet.
-->
    <servlet-mapping>
        <servlet-name>jsp</servlet-name>
        <url-pattern>*.jsp</url-pattern>
       </servlet-mapping>
```

```
    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
<!--
Ниже идет длинный список соответствий расширений имен файлов
и MIME-типов содержимого этих файлов.
-->
    <mime-mapping>
        <extension>txt</extension>
        <mime-type>text/plain</mime-type>
    </mime-mapping>
    <mime-mapping>
        <extension>html</extension>
        <mime-type>text/html</mime-type>
    </mime-mapping>
    <mime-mapping>
        <extension>htm</extension>
        <mime-type>text/html</mime-type>
    </mime-mapping>
    <mime-mapping>
        <extension>gif</extension>
        <mime-type>image/gif</mime-type>
    </mime-mapping>
<!--
И так далее.
Наконец, идет список файлов, которые посылаются клиенту при
обращении только к каталогу без указания ресурса.
-->
  <welcome-file-list>
    <welcome-file>index.jsp </welcome-file>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
  </welcome-file-list>
</web-app>
```

Интерфейс *ServletConfig*

Каждый объект типа `ServletConfig` содержит имя сервлета, извлеченное из элемента `<servlet-name>` конфигурационного файла, набор начальных параметров, взятых из элементов `<init-param>`, и контекст сервлета в виде объекта типа `ServletContext`. Эти конфигурационные параметры сервлет может получить методами

```
public String getServletName();
public Enumeration getInitParameterNames();
```

```
public String getInitParameter(String name);
public ServletContext getServletContext();
```

описанными в интерфейсе `ServletConfig`.

Начальные параметры записываются в конфигурационный файл web.xml во время установки сервлета вручную или с помощью утилиты установки. Механизм задания и чтения начальных параметров сервлета очень похож на механизм определения параметров апплета, записываемых в теги `<param>` и читаемых методами `getParameter()` апплета.

В листинге 26.2 приведен простейший сервлет, отправляющий клиенту свое имя и начальные параметры.

Листинг 26.2. Начальные параметры сервлета

```
package myservlets;
import java.io.*;
import java.util.*;
   import javax.servlet.*;

   public class InfoServlet extends GenericServlet{

     private ServletConfig sc;

  public void init(ServletConfig conf)
           throws ServletException{
    super.init(conf);
    sc = conf;
     }

  public void service(ServletRequest req, ServletResponse resp)
              throws ServletException, IOException{

       resp.setContentType("text/html; charset=windows-1251");

       PrintWriter pw = resp.getWriter();

    pw.println("<html><head>");
    pw.println("<title>Параметры сервлета</title>");
    pw.println("</head><body><h2>Сведения о сервлете</h2>");
       pw.println("Имя сервлета - " + sc.getServletName() + "<br>");

       pw.println("Параметры сервлета: <br>");

       Enumeration names = sc.getInitParameterNames();

    while (names.hasMoreElements()){
      String name = (String)names.nextElement();
      pw.print(name + ": ");
      pw.println(sc.getInitParameter(name) + "<br>");
       }
```

```
    pw.println("</body></html>");
    pw.flush();
    pw.close();
  }
  public void destroy(){
    sc = null;
  }
}
```

Полностью код листинга 26.2 будет подробно разъяснен позднее, а пока запомните два правила:

- переопределяя метод `init(ServletConfig)`, вызывайте метод `init(ServletConfig)` суперкласса;
- кодировку ответа устанавливайте методом `setContentType()` *перед* получением ссылки на выходной поток методом `getWriter()`. Это относится и к другим заголовкам ответа.

Код листинга 26.2 компилируется обычным образом, а затем полученный файл с сервлетом `InfoServlet.class` устанавливается в контейнер. Процедура установки выполняется соответствующей утилитой, входящей в состав контейнера сервлетов или сервера приложений, make-файлом или ant-файлом. По окончании процедуры установки сервлет можно вызвать из браузера, набрав в нем строку адреса вида

http://<servlethost>:8000/InfoAppl/servlet/InfoServlet

Браузер покажет страничку HTML, сформированную сервлетом. Для других контейнеров и серверов приложений номер порта и путь к сервлету будет, конечно, иным.

> ***Замечание по отладке***
>
> Для многих контейнеров и серверов приложений недостаточно просто перекомпилировать измененный код сервлета, чтобы контейнер воспринял обновленный сервлет. Надо еще сервлет переустановить. В утилите deploytool переустановка сервлета выполняется выбором пункта **Update and Redeploy** в меню **Tools**.

Многие серверы приложений, в их числе и последние версии Tomcat, не запускают сервлеты по отдельности, а только в составе Web-приложения. Обычно утилиты установки создают необходимую для Web-приложения структуру каталогов, но это можно сделать и вручную. Обращение к сервлету в составе Web-приложения выглядит проще:

http://<servlethost>:8000/InfoAppl/InfoServlet

Как видно из сигнатуры метода `getServletContext()`, контейнер получает доступ к еще одному объекту — объекту типа `ServletContext`, содержащему контекст сервлета.

Контекст сервлета

Для всех сервлетов, работающих в рамках одного Web-приложения, создается один контекст. *Контекст* (context) сервлетов составляют каталоги и файлы, описывающие Web-приложение. Они содержат, в частности, код сервлета, изображения, чертежи, конфигурационный файл web.xml, короче говоря, все относящееся к сервлету. При инициализации сервлета некоторые

сведения о его контексте заносятся в объект типа `ServletContext`. Методы этого интерфейса позволяют сервлету получить сведения, содержащиеся в контексте.

Метод `getServerInfo()` позволяет получить имя и версию Java EE SDK, методы `getMajorVersion()` и `getMinorVersion()` возвращают номер версии и модификации Servlet API.

В контексте можно определить начальные параметры, общие для всего Web-приложения. Они задаются при создании Web-приложения вручную или с помощью утилиты установки и хранятся в конфигурационном файле web.xml в элементах `<context-param>`. Их имена и значения можно получить методами

```
public Enumeration getInitParameterNames();
public String getInitParameter(String name);
```

Кроме строковых параметров в контексте допустимо определить атрибуты, значениями которых могут служить объекты любых типов Java. Их имена и значения можно получить методами

```
public Enumeration getAttributeNames();
public Object getAttribute(String name);
```

Установить и удалить атрибуты можно методами

```
public void setAttribute(String name, Object value);
public void removeAttribute(String name);
```

Атрибуты — это удобный способ сохранять объекты, общие для всего Web-приложения, разделяемые всеми сервлетами, входящими в Web-приложение, и независимые от отдельных запросов.

Метод *Service*

Основная работа сервлета заключена в методе

```
public void service(ServletRequest req, ServletResponse resp);
```

К этому методу контейнер обращается автоматически после завершения метода `init()` и передает ему объект `req` типа `ServletRequest`, содержащий всю информацию, находящуюся в запросе клиента. Созданием и заполнением объекта `req` тоже занимается контейнер сервлетов. Кроме того, контейнер создает и передает методу `service()` ссылку на пустой объект `resp` типа `ServletResponse`.

Метод `service()` обрабатывает сведения, содержащиеся в объекте `req`, и заносит результаты обработки в объект `resp`. Заполненный объект `resp` передается контейнеру, который через Web-сервер отправляет ответ клиенту. Все эти действия выполняются методами, описанными в интерфейсах `ServletRequest` и `ServletResponse`.

Интерфейс *ServletRequest*

В интерфейсе `ServletRequest`, который должен реализовать каждый контейнер сервлетов, описана масса методов `getXxx()`, возвращающих параметры запроса или `null`, если параметр неизвестен.

Методы `getRemoteAddr()`, `getRemoteHost()` и `getRemotePort()` возвращают IP-адрес, полное DNS-имя отправителя запроса или proxy-сервера и его номер порта, а методы `getServerName()` и `getServerPort()` возвращают имя и номер порта сервера, принявшего запрос.

Методы `getLocalAddr()`, `getLocalName()` и `getLocalPort()` возвращают IP-адрес, полное DNS-имя сетевого интерфейса, с которого получен запрос, и его номер порта.

Метод `getScheme()` возвращает схему запроса: `http:`, `https:`, `ftp:` и т. д., а метод `getProtocol()` — имя протокола в виде строки, например `"HTTP/1.1"`.

Методы `getContentType()` и `getCharacterEncoding()` возвращают MIME-тип и кодировку запроса, если они указаны в заголовке запроса, а метод `getContentLength()` — длину тела запроса в байтах, если она известна, или –1, если длина неизвестна.

Метод `setCharacterEncoding(String)` устанавливает кодировку, если она не определяется методом `getCharacterEncoding()`, или переписывает кодировку, указанную в запросе. К этому методу часто приходится обращаться для правильного преобразования параметров запроса, пришедших в байтовой кодировке, в строку типа `String`. Пример такого обращения приведен в листинге 26.5. Этот метод следует применять до разбора параметров запроса.

Имена и значения параметров, пришедших с запросом, можно получить методами

```
public Enumeration getParameterNames();
public String getParameter(String name);
public String[] getParameterValues(String name);
public Map getParameterMap();
```

Если у запроса есть какие-либо атрибуты, то их имена и значения можно получить методами

```
public Enumeration getAttributeNames();
public Object getAttribute(String name);
```

Наконец, интерфейс описывает два входных потока для получения тела запроса: байтовый и символьный. Байтовый поток реализуется специально разработанным классом `ServletInputStream`.

Класс `ServletInputStream` — это абстрактный класс, расширяющий класс `InputStream`. Он добавляет к методам своего суперкласса только один метод

```
public int readLine(byte[] buf, int offset, int length);
```

читающий строку тела запроса в заранее определенный буфер `buf`. Чтение начинается с байта с номером `offset` и продолжается до достижения символа перевода строки `'\n'` или до достижения количества прочитанных символов `length`. Метод возвращает число прочитанных байтов или –1, если входной поток уже исчерпан.

Получить байтовый поток из запроса `req` можно методом

```
public ServletInputStream getInputStream();
```

Второй поток — символьный — это поток класса `BufferedReader`, который мы рассмотрели в *главе 23*. Получить его можно методом

```
public BufferedReader getReader();
```

В пакете `javax.servlet` есть прямая реализация интерфейса `ServletRequest` — класс `ServletRequestWrapper`. Объект этого класса создается конструктором

```
public ServletRequestWrapper(ServletRequest req);
```

и обладает всеми методами интерфейса `ServletRequest`. Разработчики, желающие расширить возможности объекта, содержащего запрос, или создать фильтр, могут расширить класс `ServletRequestWrapper`.

Интерфейс *ServletResponse*

Результаты своей работы метод `service()` заносит в объект типа `ServletResponse`, ссылка на который предоставлена вторым аргументом метода `service()`.

Методы `setContentType(String)` и `setLocale(Locale)` устанавливают в заголовок ответа MIME-тип и локаль тела ответа, а метод `setContentLength(int)` записывает длину тела ответа. Если надо установить только кодировку символов в ответе, то можно воспользоваться методом `setCharacterEncoding(String)`.

Тело ответа передается контейнеру через байтовый или символьный выходной поток. Байтовый поток специально разработанного класса `ServletOutputStream` возвращает метод

```
public ServletOutputStream getOutputStream();
```

Абстрактный класс `ServletOutputStream` расширяет класс `OutputStream`, добавляя к нему методы `print(`*xxx*`)` для вывода типов `boolean`, `char`, `int`, `long`, `float`, `double`, `String` и методы `println(`*xxx*`)` для тех же типов, добавляющие к выводимым данным символы `"\r\n"`. Еще один метод `println()` без аргументов просто заносит в выходной поток символы `"\r\n"`.

Символьный поток можно получить методом

```
public PrintWriter getWriter();
```

Именно он использован в листинге 26.2.

В пакете `javax.servlet` есть прямая реализация интерфейса `ServletResponse` — класс `ServletResponseWrapper`. Объект этого класса создается конструктором

```
public ServletResponseWrapper(ServletResponse resp);
```

и обладает всеми методами интерфейса `ServletResponse`. Разработчики, желающие расширить возможности объекта, содержащего ответ, например для создания фильтра, могут расширить класс `ServletResponseWrapper`.

Цикл работы сервлета

Сервлет загружается контейнером, как правило, при первом запросе к нему или во время запуска контейнера. После выполнения запроса сервлет может быть оставлен в спящем состоянии, ожидая следующего запроса, или выгружен, предварительно выполнив метод `destroy()`. Это зависит от реализации контейнера сервлетов.

Работа сервлета начинается с метода `init()`, затем выполняется метод `service()`, который может создавать объекты, обращаться к их методам, связываться с базами данных и

удаленными объектами, выполняя обычную работу обычного класса Java. При этом надо учитывать, что сервлету может быть направлено сразу несколько запросов. Число одновременных запросов в промышленных системах достигает сотен и тысяч. Для обработки каждого запроса контейнер создает новый подпроцесс (thread), выполняющий метод `service()`. Поэтому выполняйте следующие правила:

- разрабатывая метод `service()`, постоянно имейте в виду, что он будет параллельно выполняться несколькими подпроцессами, и принимайте меры к синхронизации их работы;
- выносите создание объектов, общих для сервлета, определение параметров, пула соединений с базами данных и удаленными объектами, в поля класса и в метод `init()`;
- завершающие действия, такие как закрытие потоков, запись результатов на диск, закрытие соединений, выносите в метод `destroy()`, выполняющийся при закрытии сервлета.

Класс *GenericServlet*

Абстрактный класс `GenericServlet` реализует сразу интерфейсы `Servlet` и `ServletConfig`. Кроме реализации методов обоих интерфейсов в него введен пустой метод `init()` без аргументов. Этот метод выполняется автоматически после метода `init(ServletConfig)`, точнее говоря, последний метод реализован так:

```
public void init(ServletConfig config) throws ServletException{
   this.config = config;
   log("init");
   this.init();
}
```

Поэтому удобно всю инициализацию записывать в метод `init()` без аргументов, не заботясь о вызове `super.init(config)`.

Класс `GenericServlet` оставляет нереализованным только метод `service()`. Удобно создавать сервлеты, расширяя этот класс и переопределяя только метод `service()`.

Так и сделано в листинге 26.2. Можно записать его проще, не определяя метод `init()`, а прямо используя реализацию методов интерфейса `ServletConfig`, сделанную в классе `GenericServlet`. Этот вариант приведен в листинге 26.3. В него добавлено еще получение контекста сервлета.

Листинг 26.3. Упрощенное чтение начальных параметров сервлета

```
package myservlets;
import java.io.*;
import java.util.*;
   import javax.servlet.*;

   public class InfoServlet extends GenericServlet{

  public void service(ServletRequest req, ServletResponse resp)
              throws ServletException, IOException{

      ServletContext cont = getServletContext();
```

```
        resp.setContentType("text/html; charset=windows-1251");

        PrintWriter pw = resp.getWriter();

      pw.println("<html><head>");
      pw.println("<title>Параметры сервлета</title>");
      pw.println("</head><body><h2>Сведения о сервлете</h2>");
        pw.println("Имя сервлета - " + getServletName() + "<br>");

        pw.println("Параметры и контекст сервлета: <br>");

        Enumeration names = getInitParameterNames();

      while(names.hasMoreElements()){
        String name = (String)names.nextElement();
        pw.print(name + ": ");
        pw.println(getInitParameter(name) + "<br>");
         }

        pw.println("Сервер: " + cont.getServerInfo() +"<br>");

      pw.println("</body></html>");
      pw.flush();
      pw.close();
    }
}
```

Работа по протоколу HTTP

Большинство запросов к сервлетам происходит по протоколу HTTP, который в настоящее время реализуется по рекомендации RFC 2616. Для удобства работы с этим протоколом интерфейсы `ServletRequest` и `ServletResponse` расширены интерфейсами `HttpServletRequest` и `HttpServletResponse` соответственно. При расширении интерфейсов в них добавлены методы, характерные для протокола HTTP.

Интерфейс *HttpServletRequest*

Несколько методов интерфейса `HttpServletRequest` позволяют разобрать HTTP-запрос.

Первая строка запроса, выполненного по протоколу HTTP, состоит из метода передачи данных, адреса URI и версии протокола. Строка завершается символами CRLF. Элементы строки разделяются пробелами, поэтому внутри каждого элемента первой строки запроса пробелов быть не должно. Первая строка запроса выглядит примерно так:

```
GET http://some.firm.com/MyWebAppl/servlet/MyServlet?age=27 HTTP/1.1
```

Запрос HTTP начинается с одного из слов `GET`, `POST`, `HEAD` или другого слова, обозначающего метод передачи данных. Определить HTTP-метод передачи позволяет метод интерфейса

```
public String getMethod();
```

Далее в запросе, после пробела, идет адрес URI, который разбирается несколькими методами:

- `public String getRequestURL()` — возвращает адрес URL от названия схемы `http` до вопросительного знака;
- `public String getServletPath()` — возвращает часть этого адреса, показывающую путь к сервлету.

Часть пути, определяющая контекст сервлета, возвращается методом

```
public String getContextPath();
```

Часть URI после вопросительного знака возвращается методом

```
public String getQueryString();
```

После имени сервлета может идти дополнительный путь к какому-нибудь файлу, который можно получить методом

```
public String getPathInfo();
```

Этот же путь, дополненный до абсолютного пути к каталогу документов сервера, можно получить методом

```
public String getPathTranslated();
```

После первой строки запроса могут идти заголовки запроса: `Accept`, `Accept-Charset`, `Accept-Language`, `User-Agent` и прочие заголовки, описанные в рекомендации RFC 2616. Узнать заголовки запроса и их значения можно методами

```
public Enumeration getHeaderNames();
public Enumeration getHeaders(String name);
public String getHeader(String name);
public int getIntHeader(String name);
public long getDateHeader(String name);
```

Наконец, можно получить cookies, хранящиеся в браузере клиента, в виде массива объектов класса `Cookie` методом

```
public Cookie[] getCookies();
```

В пакете `javax.servlet.http` есть прямая реализация интерфейса `HttpServletRequest` — класс `HttpServletRequestWrapper`, расширяющий класс `ServletRequestWrapper`. Объект этого класса создается конструктором

```
public HttpServletRequestWrapper(HttpServletRequest req);
```

и обладает всеми методами интерфейса `HttpServletRequest`. Разработчики, желающие расширить возможности объекта, содержащего HTTP-запрос, например для написания фильтра, могут расширить класс `HttpServletRequestWrapper`.

Интерфейс *HttpServletResponse*

При составлении ответа по протоколу HTTP можно использовать дополнительные методы, включенные в интерфейс `HttpServletResponse`.

Метод

```
public void setHeader(String name, String value);
```

устанавливает заголовок ответа с именем `name` и значением `value`. Старое значение, если оно существовало, при этом стирается.

Если надо дать несколько значений заголовку с именем `name`, то следует воспользоваться методом

```
public void addHeader(String name, String value);
```

Для заголовков с целочисленными значениями то же самое делается методами

```
public void setIntHeader(String name, int value);
public void addIntHeader(String name, int value);
```

Заголовок с датой записывается методами

```
public void setDateHeader(String name, long date);
public void addIntHeader(String name, int value);
```

Код ответа (status code) устанавливается методом

```
public void setStatus(int sc);
```

Как и все заголовки, этот метод записывается перед получением потока класса `PrintWriter`. Аргумент метода `sc` — это одна из множества констант, например константа `SC_OK`, соответствующая коду ответа 200 — успешная обработка запроса, `SC_BAD_REQUEST` — код ответа 400 и т. д. Около сорока таких статических констант приведено в документации к интерфейсу `HttpServletRequest`. Метод `setStatus()` применяется для сообщений об успешной обработке с кодами 200—299.

Сообщения об ошибке посылаются методом

```
public void sendError(int sc, String message);
```

Обычно код ответа заносится в сообщение об ошибке `message`. Если надо послать стандартное сообщение об ошибке, например "404 Not Found", то применяется второй метод:

```
public void sendError(int sc);
```

Этот метод использован в листинге 26.6.

Сообщение методом `sendError()` посылается вместо результатов обработки запроса. Если попытаться послать после него ответ, то система выбросит исключение класса `IllegalStateException`.

Наконец, к запросу можно добавить cookie методом

```
public void addCookie(Cookie cookie);
```

В пакете `javax.servlet.http` есть прямая реализация интерфейса `HttpServletResponse` — класс `HttpServletResponseWrapper`, расширяющий класс `ServletResponseWrapper`. Объект этого класса создается конструктором

```
public HttpServletResponseWrapper(HttpServletResponse resp);
```

и обладает всеми методами интерфейса `HttpServletResponse`. Разработчики, желающие расширить возможности объекта, содержащего запрос, могут расширить класс `HttpServletResponseWrapper`.

Класс *HttpServlet*

Для использования особенностей протокола HTTP класс `GenericServlet` расширен абстрактным классом `HttpServlet`. Главная особенность этого класса заключается в том, что,

расширяя его, не надо переопределять метод `service()`. Он уже определен, причем реализован так, что служит диспетчером, вызывающим методы `doGet()`, `doPost()` и другие методы, обрабатывающие HTTP-запросы с конкретными методами передачи данных `GET`, `POST` и др.

Вначале метод

```
public void service(ServletRequest req, ServletResponse resp);
```

анализирует типы аргументов `req` и `resp`. Эти типы должны быть на самом деле `HttpServletRequest` и `HttpServletResponse`. Если это не так, то метод выбрасывает исключение класса `ServletException` и завершается.

Если аргументы `req` и `resp` подходящего типа, то методом `getMethod()` определяется HTTP-метод передачи данных и вызывается метод, соответствующий этому HTTP-методу, а именно один из методов

```
protected void doXxx(HttpServletRequest req, HttpServletResponse resp);
```

где *Xxx* означает `Get`, `Post`, `Head`, `Delete`, `Options`, `Put` или `Trace`.

Вот эти-то методы и надо переопределять, расширяя класс `HttpServlet`. Методы `doHead()`, `doOptions()` и `doTrace()` уже реализованы в соответствии с рекомендацией RFC 2616, их редко приходится переопределять. Остальные методы "реализованы" таким образом, что просто посылают сообщение о том, что они не реализованы. Чаще всего приходится переопределять методы `doGet()` и `doPost()`.

Приведем пример сервлета, осуществляющего регистрацию клиента Web-приложения — некоторой системы дистанционного обучения (СДО). Сервлет `RegPrepServlet` принимает запрос от HTML-формы, соединяется с базой данных, заносит в нее полученную информацию и отправляет клиенту подтверждение регистрации в виде страницы HTML, содержащей форму для выбора учебного курса. В листинге 26.4 приведена HTML-форма регистрации клиента. Ее вид показан на рис. 26.2.

Листинг 26.4. Форма регистрации клиента СДО

```
<html><head>
  <title>Регистрация</title>
  <META http-equiv=Content-Type
        content="text/html; charset=windows-1251">
      </head>
  <body><h2 align="center">Дистанционная система обучения СДО</h2>
    <p>Для регистрации занесите сведения о себе в следующие поля:</p>
    <br>
    <form method=post
          action=
          "http://some.firm.com:8000/WebAppl/servlet/RegPrepServlet">
      <pre>
   Фамилия: <input type="text" size="40" name="surname">
       Имя: <input type="text" size="40" name="name">
  Отчество: <input type="text" size="40" name="secname">
       E-mail: <input type="text" size="40" name="addr">
```

```
            <input type="submit" value="Зарегистрировать">
        </pre>
    </form>
  </body>
</html>
```

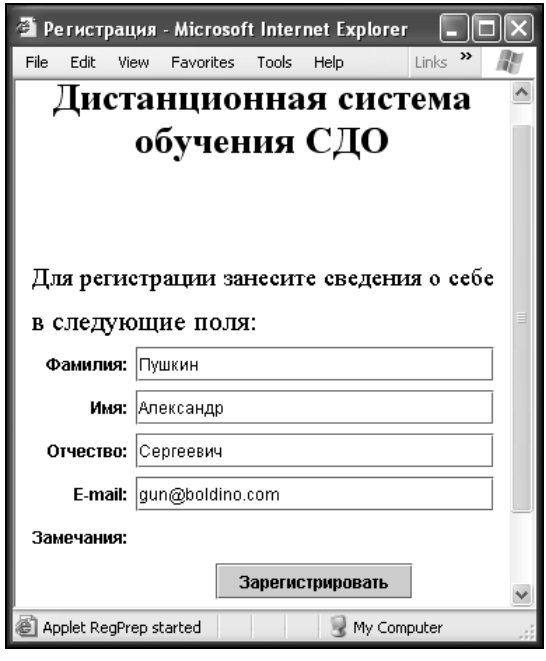


Рис. 26.2. Страница регистрации

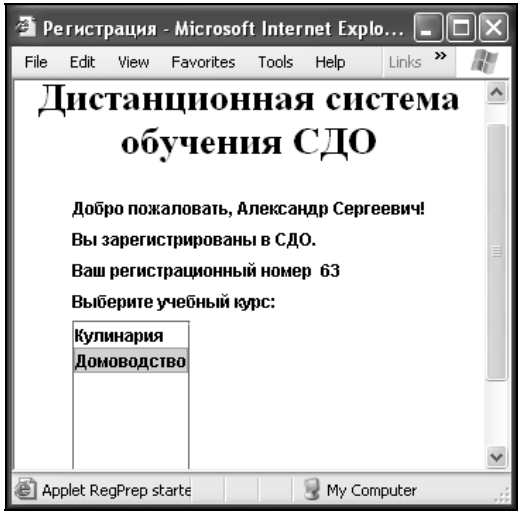


Рис. 26.3. Страница подтверждения регистрации

Форма посылает сервлету `RegPrepServlet` четыре параметра: `surname`, `name`, `secname` и `addr` по HTTP-методу `POST`. Сервлет должен принять их, обработать и послать клиенту результаты запроса или замечания по регистрации. Код сервлета приведен в листинге 26.5. Страница подтверждения регистрации показана на рис. 26.3.

Листинг 26.5. Сервлет регистрации клиента СДО

```
package myservlets;
import java.io.*;
import java.sql.*;
import java.util.*;
   import javax.servlet.http.*;

   public class RegPrepServlet extends HttpServlet{

  private String driver = "oracle.jdbc.driver.OracleDriver",
     url = "jdbc:oracle:thin:@homexp:1521:SDO",
        user = "sdoadmin", password = "sdoadmin";

  public void init(){
    try{
        Class.forName(driver);
    }catch(Exception e){
        System.err.println("From init(): " + e);
    }
     }

  public void doGet(HttpServletRequest req, HttpServletResponse resp){
     doPost(req, resp);
     }

  public void doPost(HttpServletRequest req, HttpServletResponse resp){
    try{
          req.setCharacterEncoding("Cp1251");

       String surname = req.getParameter("surname");
       String name    = req.getParameter("name");
       String secname = req.getParameter("secname");
          String addr    = req.getParameter("addr");

          resp.setContentType("text/html; charset=windows-1251");

          PrintWriter pw = resp.getWriter();

       if (surname.length() * name.length() *
             secname.length() * addr.length() == 0){

         pw.println("<html><head>");
         pw.println("<title>Продолжение регистрации</title>");
```

```
  pw.println("</head><body><h2 align=center>" +
        "Дистанционная система обучения СДО</h2>");
  pw.println("<h3>Замечание:</h3>");
  pw.println("Заполните, пожалуйста, все поля.<br>");
  pw.println("</body></html>");
  pw.flush();
  pw.close();
  return;
   }

String fullname = surname.trim() + " " + name.trim() +
     " " + secname.trim();

Connection con = DriverManager.getConnection(
       url, user, password);

PreparedStatement pst = con.prepareStatement(
    "INSERT INTO students (id, name, address) " +
    "VALUES(reg_seq.NEXTVAL, ?, ?)");
pst.setString(1, fullname);
pst.setString(2, addr);
int count = pst.executeUpdate();
   pst.close();

   Statement st = con.createStatement();

ResultSet rs = st.executeQuery(
      "SELECT id, name FROM students ORDER BY id DESC");

   rs.next();

int id = rs.getInt(1);
   fullname = rs.getString(2);

   rs.close();

StringTokenizer sttok = new StringTokenizer(fullname);
sttok.nextToken();
name = sttok.nextToken();
   secname = sttok.nextToken();

pw.println("<html><head>");
pw.println("<title>Регистрация</title>");
pw.println("</head><body><h2 align=center>" +
        "Дистанционная система обучения СДО</h2>");

pw.println("Добро пожаловать, " + name + " " +
           secname + "!<br>");
pw.println("Вы зарегистрированы в СДО.<br>");
pw.println("Ваш регистрационный номер  " + id + "<br>");
```

```
          pw.println("Выберите учебный курс:<br> ");

          rs = st.executeQuery("SELECT course_name FROM courses");

        pw.println("<form method=post action=" +
          "\"http://homexp:8000/InfoAppl/servlet/CoursesServlet\">");
          pw.println("<select size=5 name=courses>");

        while (rs.next())
              pw.println("<option>" + rs.getString(1));

        pw.println("</option></form></body></html>");
        pw.flush();
          pw.close();

        rs.close();
          con.close();

      }catch(Exception e){
         System.err.println(e);
      }
   }
}
```

В листинге 26.5 соединение с базой данных выполняется в методе `doPost()` и открывается заново каждым запросом. При большом количестве одновременных запросов это может снизить производительность системы и даже превысить допустимое число соединений. В таком случае при инициализации сервлета надо в методе `init()` создать пул соединений с тем, чтобы запросы брали соединения из этого пула и возвращали соединение в пул при своем завершении.

Следует заметить, что у системы управления базой данных Oracle, с которой соединяется сервлет `RegPrepServlet`, есть свой сервер приложений Oracle Application Server (OAS) с контейнером сервлетов Apache/JServ или Tomcat. Можно установить сервлет прямо в Oracle AS и использовать для соединения с базой серверный драйвер JDBC с именем kprb. Это резко повысит производительность сервлета.

Разумеется, сервлет может отправлять клиенту не только страницы HTML, но и изображения, тексты в разных форматах, например PDF, звуковые файлы, короче говоря, данные любых MIME-типов. В листинге 26.6 приведен пример сервлета, позволяющего клиенту просматривать изображения типа GIF и JPEG в каталоге, заданном начальным параметром сервлета.

Листинг 26.6. Сервлет, отправляющий изображения клиенту

```
package myservlets;
import java.io.*;
import java.util.*;
import javax.servlet.*;
   import javax.servlet.http.*;

   public class ImageServlet extends HttpServlet{
```

```
Vector imFiles = new Vector();
   int curIndex;

public void init() throws ServletException{
  File imDir = null;
     String imDirName = getInitParameter("imagedir");

  if (imDirName != null)
        imDir = new File(imDirName);

  if ((imDir != null) && imDir.exists() && imDir.isDirectory()){
        String[] files = imDir.list();

     for (int i = 0; i < files.length; i++)
        if (files[i].endsWith(".jpg") ||
               files[i].endsWith(".gif")){

           File curFile = new File(imDir, files[i]);
           imFiles.addElement(curFile);
           }

  }else log("Cannot find image dir: " + imDirName);
   }

public void doGet(HttpServletRequest req, HttpServletResponse resp)
       throws ServletException, IOException{
      int len = imFiles.size();

   if (len > 0){
      File curFile = (File)imFiles.elementAt(curIndex);
      String fileName = curFile.getName();
      ServletContext ctxt = getServletConfig().getServletContext();
         String ctype = ctxt.getMimeType(fileName);

      if (ctype == null) ctype = fileName.endsWith(".jpg") ?
              "image/jpeg" : "image/gif";

         resp.setContentType(ctype);

      try{
         BufferedInputStream bis = new BufferedInputStream(
               new FileInputStream(curFile));
            OutputStream os = resp.getOutputStream();

         int cur = 0;
            while ((cur = bis.read()) != -1) os.write(cur);

         os.close();
            bis.close();

      }catch(FileNotFoundException e){
         resp.sendError(HttpServletResponse.SC_NOT_FOUND);
```

```
        }catch(Exception e){
           resp.sendError(HttpServletResponse.SC_SERVICE_UNAVAILABLE);
        }
           curIndex= (curIndex + 1) % len;

      }else resp.sendError(HttpServletResponse.SC_SERVICE_UNAVAILABLE);
    }

  public long getLastModified(){
      return System.currentTimeMillis();
  }
}
```

Сеанс связи с сервлетом

Сервер HTTP не сохраняет информацию о клиенте, связавшемся с ним. Хотя TCP-соединение может сохраняться вплоть до его явного закрытия (persistent HTTP connection) и за это время можно передать несколько запросов и ответов, протокол HTTP не предполагает средств сохранения информации о клиенте.

Многие задачи, решаемые Web-приложениями, требуют знания сведений о клиенте. Например, клиент электронного магазина может сделать несколько заказов в течение дня или даже нескольких дней. Сервер должен знать, в чью "корзину" складывать заказанные покупки. Еще пример. В программе листинга 26.5 завязывается диалог. Клиент регистрируется и посылает сведения о себе на сервер СДО. Сервер предлагает клиенту подобрать себе учебный курс. Клиент выбирает курс и посылает его название серверу. Сервер должен занести имя курса в учетную карточку клиента, а для этого ему надо знать регистрационный номер клиента.

Для получения сведений о клиенте, приславшем запрос, приходится применять искусственные средства, не входящие в протокол HTTP. Наиболее часто используются три средства: cookie, параметр в строке URL и скрытое поле в HTML-форме.

Первое средство — cookie — действует так. Получив первый запрос, сервер составляет заголовок ответа `Set-Cookie`, в который заносит пару "имя = значение", обычно это идентификатор клиента, а также диапазон URL, для которого действует cookie, срок хранения этих сведений и другую информацию. Браузер, получив ответ с таким заголовком, создает небольшой, размером не более 4 Кбайт, cookie-файл с этими сведениями и сохраняет его у себя в каталоге. Посылая следующие запросы, браузер отыскивает у себя соответствующий cookie-файл и заносит в заголовок `Cookie` запроса пару "имя = значение". Сервер по этому заголовку "узнает" клиента.

В пакете `javax.servlet.http` есть класс `Cookie`, методы которого обеспечивают работу с cookie. Объект этого класса создается конструктором

```
public Cookie(String name, String value);
```

Методы `set`*`Xxx`*`()` позволяют добавить в объект остальные сведения, а методы `get`*`Xxx`*`()` — прочитать их.

После создания и формирования объекта cookie этот объект устанавливается в заголовок ответа методом

```
public void addCookie(Cookie cookie);
```

интерфейса `HttpServletResponse`.

Вот обычная последовательность действий по созданию cookie и отправке его клиенту:

```
String value = "" + id;
Cookie ck = new Cookie("studentid", value);
ck.setMaxAge(60*60*24*183);   // Cookie будет существовать полгода
resp.addCookie(ck);
```

Прочитать cookie из запроса клиента можно методом

```
public Cookie[] getCookies();
```

интерфейса `HttpServletRequest`.

Вот обычная последовательность действий по чтению cookie:

```
int id = 0;
   Cookie[] cks = req.getCookies();

if (cks != null)
   for (int i = 0; i < cks.length; i++)
      if (cks[i].getName().equals("studentid")){
         id = Integer.parseInt(cks[i].getValue());
         break;
      }
```

Использовать cookie удобно, но беда в том, что многие клиенты запрещают запись cookie-файлов, справедливо полагая, что нельзя записывать что-либо на диск без ведома хозяина.

Второе средство — параметр в строке URL (rewriting URL) — просто записывает в первую строку запроса после имени ресурса идентификатор клиента, например:

```
GET /some.com/InfoAppl/index.html?jsessionid=12345678 HTTP/1.1
```

Это тоже хороший способ, но клиент может сам формировать строку запроса, например, просто набирая ее в поле адреса браузера и забывая об идентификаторе.

Третье средство — скрытое поле HTML-формы — это поле

```
<input type="hidden" name="studentid" value="12345678">
```

в которое можно записать идентификатор клиента. Для применения данного средства надо на каждой странице HTML создавать форму.

Итак, какого-то одного универсального средства, для того чтобы создать сеанс связи с Web-сервером, нет. В пакет `javax.servlet.http` внесены интерфейсы и классы для облегчения распознавания клиента. Они автоматически переключаются с одного средства на другое. Если запрещены cookies, то формируется идентификатор клиента в строке URL и т. д.

Основу средств создания сеанса связи с клиентом составляет интерфейс `HttpSession`. Объект типа `HttpSession` формируется контейнером сервлета при получении запроса, а получить или создать его можно методом

```
public HttpSession getSession(boolean create);
```

описанным в интерфейсе `HttpServletRequest`. Метод возвращает объект типа `HttpSession`, если он существует. Если же такой объект отсутствует, то поведение метода

зависит от значения аргумента `create` — если он равен `false`, то метод возвращает `null`, если `true` — создает новый объект.

Второй метод того же интерфейса

```
public HttpSession getSession();
```

эквивалентен `getSession(true)`.

Логическим методом

```
public boolean isNew();
```

интерфейса `HttpSession` можно узнать, новый ли это, только что созданный сеанс (`true`) или продолжающийся, уже запрошенный клиентом (`false`).

В сеансе отмечается время его создания и время последнего запроса, которые можно получить методами

```
public long getCreationTime();
public long getLastAccessedTime();
```

Они возвращают время в миллисекундах, прошедших с полуночи 1 января 1970 года (дата рождения UNIX).

Создавая сеанс, контейнер дает ему уникальный идентификатор. Метод

```
public String getId();
```

возвращает идентификатор сеанса в виде строки.

У сеанса могут быть атрибуты, в качестве которых способны выступать любые объекты Java. Атрибут задается методом

```
public void setAttribute(String name, Object value);
```

Получить имена и значения атрибутов можно методами

```
public Enumeration getAttributeNames();
public Object getAttribute(String name);
```

Атрибуты — удобное средство хранения объектов, которые должны существовать на протяжении сеанса.

Атрибут удаляется методом

```
public void removeAttribute(String name);
```

Контейнер следит за событиями, происходящими во время сеанса. Создание или удаление атрибута, изменение его значения — события класса `HttpSessionBindingEvent`. Этот класс является подклассом класса `HttpSessionEvent`, экземпляр которого создается при всяком изменении в активных сеансах Web-приложения — создании сеанса, прекращении сеанса, истечении срока ожидания запроса.

Сеанс завершается методом `invalidate()` или по истечении времени ожидания очередного запроса. Это время, в секундах, задается методом

```
public void setMaxInactiveInterval(int secs);
```

Сервлет может узнать назначенное время ожидания методом

```
public int getMaxInactiveInterval();
```

Фильтры

Суть работы сервлета заключается в обработке полученного из объекта типа `ServletRequest` запроса и формировании ответа в виде объекта типа `ServletResponse`. Попутно сервлет может проделать массу работы, создавая объекты, обращаясь к их методам, загружая файлы, соединяясь с базами данных. Эти действия усложняют изначально простую и четкую структуру сервлета. Чтобы придать стройность и упорядоченность сервлету, можно организовать цепочку фильтров — объектов, последовательно пропускающих через себя информацию, идущую от запроса к ответу, и преобразующих ее.

Удобство фильтров заключается еще и в том, что один фильтр может использоваться несколькими сервлетами и даже всеми ресурсами Web-приложения. Разработчик может подготовить набор фильтров на все случаи жизни и применять их в своих сервлетах.

При работе с фильтрами сразу создается цепочка фильтров, даже если в нее входит всего один фильтр. Вся работа с цепочками фильтров описывается интерфейсами `Filter`, `FilterChain` и `FilterConfig`. Интерфейс `Filter` реализуется разработчиком приложения, остальные интерфейсы должен реализовать контейнер сервлетов.

Каждый фильтр в цепочке — это объект типа `Filter`. Структура этого объекта напоминает структуру сервлета. Интерфейс `Filter` описывает три метода:

```
public void init(FilterConfig cong);
public void doFilter(ServletRequest req, ServletResponse resp,
                     FilterChain chain);
public void destroy();
```

Как видно из этих описаний, фильтр может выполнить начальные действия методом `init()`, причем ему передается созданный контейнером объект типа `FilterConfig`, который очень похож на объект типа `ServletConfig`. Он также содержит начальные параметры, которые можно получить методами

```
public Enumeration getInitParameterNames();
public String getInitParameter(String name);
```

У него тоже есть имя, которое дается ему при установке фильтра в контейнер и записывается в конфигурационный файл web.xml в элемент `<filter-name>`. Имя класса-фильтра можно получить методом

```
public String getFilterName();
```

Наконец, он тоже возвращает ссылку на контекст методом

```
public ServletContext getServletContext(String name);
```

Вся фильтрация выполняется методом `doFilter()`, который получает объекты `req` и `resp`, изменяет их и передает управление следующему фильтру в цепочке с помощью аргумента `chain`.

Организация цепочки достается на долю контейнера, интерфейс `FilterChain` описывает только один метод

```
public void doFilter(ServletRequest req, ServletResponse resp);
```

Чтобы передать управление фильтру, следующему в цепочке, фильтр должен просто обратиться к этому методу, передав ему измененные объекты `req` и `resp`.

Приведем пример фильтра. Русскоязычному программисту постоянно приходится думать о правильной кодировке кириллицы. Параметры запроса идут от браузера чаще всего в MIME-типе `application/x-www-form-urlencoded`, использующем байтовую кодировку, принятую по умолчанию на машине клиента. Эта кодировка должна указываться в заголовке `Content-Type`, например:

```
Content-Type: application/x-www-form-urlencoded;charset=windows-1251
```

Но, как правило, браузер не посылает этот заголовок Web-серверу. В таком случае встает задача определить кодировку параметров запроса и заслать ее в метод `setCharacterEncoding(String)`, чтобы метод `getParameter(String)` правильно перевел значение параметра в Unicode. Эту задачу должен решать метод `getCharacterEncoding()`, но он чаще всего реализован так, что просто берет кодировку из заголовка `Content-Type`. Листинг 26.7 показывает схему такой реализации в одной из прежних версий контейнера сервлетов Tomcat. Оцените качество кодирования и посмотрите, почему, используя эту версию Tomcat, между словом `"charset"` и знаком равенства нельзя оставлять пробелы.

Кстати говоря, некоторые контейнеры сервлетов не воспринимают кодировку, если оставлены пробелы между точкой с запятой и словом `"charset"`.

Листинг 26.7. Фильтр определения кодировки параметров запроса

```
   // Copyright (c) 1999 The Apache Software Foundation.

import java.io.*;
   import javax.servlet.*;

   public class SetCharEncFilter implements Filter{

     protected String enc;

     protected FilterConfig fc;

     public void init(FilterConfig conf) throws ServletException{

    fc = conf;
    enc = conf.getInitParameter("encoding");
     }

  public void doFilter(ServletRequest req,
                       ServletResponse resp,
                       FilterChain chain)
                throws IOException, ServletException{

    String encoding = selectEncoding(req);
    if (encoding != null)
          req.setCharacterEncoding(encoding);
```

```
    chain.doFilter(req, resp);
     }

  protected String selectEncoding(ServletRequest req){
    String charEncoding =
       getCharsetFromContentType(req.getContentType());
    return (charEncoding == null) ? enc : charEncoding;
     }

   // From org.apache.tomcat.util.RequestUtil.java

  public static String getCharsetFromContentType(String type){
    if (type == null) {
      return null;
    }
    int semi = type.indexOf(";");
    if (semi == -1) {
       return null;
    }
    String afterSemi = type.substring(semi + 1);
    int charsetLocation = afterSemi.indexOf("charset=");
    if (charsetLocation == -1) {
       return null;
    }
    String afterCharset = afterSemi.substring(charsetLocation + 8);
    String encoding = afterCharset.trim();
    return encoding;
     }

  public void destroy(){
    enc = null;
    fc = null;
  }
}
```

Фильтр класса `SetCharEncFilter`, описанный в листинге 26.7, очень прост. Он извлекает из запроса `req` заголовок `Content-Type` методом `getContentType()`. Если такой заголовок есть, то он пытается извлечь из него кодировку. Если это не удается, то берет кодировку из своего начального параметра `"encoding"`. Затем он заносит кодировку в ответ `resp` методом `setCharacterEncoding(String)` и передает управление следующему фильтру.

Всякий разработчик, желающий улучшить определение кодировки, может переопределить метод `selectEncoding()`, извлекая информацию из других заголовков, например: `User-Agent`, `Accept-Language`, `Accept-Charset`, `Content-Language`.

В более сложных случаях понадобится расширить объекты `req` и `resp`. Вот тут-то и пригодятся классы `HttpServletRequestWrapper` и `HttpServletResponseWrapper`. Дополнительные свойства запроса и ответа можно занести в расширения этих классов-оболочек и использовать их в фильтре по такой схеме.

```
   public class MyRequestHandler extends HttpServletRequestWrapper{

   public MyRequestHandler(HttpServletRequest req){
      super(req);
      // ...
   }
   // ...
   }

   public class MyResponseHandler extends HttpServletResponseWrapper{

   public MyResponseHandler(HttpServletResponse resp){
      super(resp);
      // ...
   }
   // ...
   }

   public class MyFilter implements Filter{

   private MyRequestHandler mreq;
      private MyResponseHandler mresp;

   public void init(FilterConfig conf){
      // ...
      }

   public void doFilter(ServletRequest req,
                        ServletResponse resp,
                        FilterChain chain){
      mreq = new MyRequestHandler((HttpServletRequest)req);
         mresp = new MyResponseHandler((HttpServletResponse)resp);

         // Действия до перехода к следующему фильтру.

      chain.doFilter(mreq, mresp);
      // Действия после возврата из сервлета и фильтров.
      }

   public void destroy(){
      mreq = null;
      mresp = null;
   }
}
```

После того как класс-фильтр написан и скомпилирован, его надо установить в контейнер и приписать (map) к одному или нескольким сервлетам. Это выполняется утилитой установки, в которой указывается имя фильтра, его начальные параметры и сервлет, к которому приписывается фильтр. Утилита установки заносит сведения о фильтре в конфигурационный файл web.xml в элемент `<filter>`. Это можно сделать и вручную. Приписка фильтра к сервлету

отмечается внутри элемента <filter-mapping> парой вложенных элементов <filter-name> и <servlet-name>. Например:

```
<filter-mapping>
   <filter-name>MyFilter</filter-name>
   <servlet-name>RegServlet</servlet-name>
</filter-mapping>
```

Фильтр можно приписать не только сервлетам, но и другим ресурсам. Для этого записывается элемент <url-pattern>, например после

```
<filter-mapping>
   <filter-name>MyFilter</filter-name>
   <url-pattern>*.html</url-pattern>
</filter-mapping>
```

фильтр будет применен ко всем вызовам документов HTML.

Порядок фильтров в цепочке соответствует порядку элементов <filter-mapping> в конфигурационном файле web.xml. При обращении клиента к сервлету контейнер сначала отыскивает фильтры и последовательно выполняет их, а уж потом запускает сервлет. После работы сервлета его ответ проходит цепочку фильтров в обратном порядке.

Обращение к другим ресурсам

В некоторых случаях недостаточно вставить в сервлет фильтр или даже цепочку фильтров, а надо обратиться к другому сервлету, странице JSP, документу HTML, XML или иному ресурсу. Если требуемый ресурс находится в том же контексте, что и сервлет, который его вызывает, то для получения ресурса следует обратиться к методу

```
public RequestDispatcher getRequestDispatcher(String path);
```

описанному в интерфейсе ServletRequest. Здесь path — это путь к ресурсу относительно контекста. Например:

```
RequestDispatcher rd = req.getRequestDispatcher("CourseServlet");
```

Если же ресурс находится в другом контексте, то нужно сначала получить контекст методом

```
public ServletContext getContext(String uripath);
```

интерфейса ServletContext, а потом воспользоваться методом

```
public RequestDispatcher getRequestDispatcher(String path);
```

интерфейса ServletContext. Здесь путь path должен быть абсолютным, т. е. начинаться с наклонной черты /. Например:

```
RequestDispatcher rd = conf.getServletContext().
   getContext("/product").
   getRequestDispatcher("/product/servlet/CourseServlet");
```

Если требуемый ресурс — сервлет, помещенный в контекст под своим именем, то для его получения можно обратиться к методу

```
public RequestDispatcher getNamedDispatcher(String name);
```

интерфейса ServletContext.

Все три метода возвращают `null`, если ресурс недоступен или сервер не реализует интерфейс `RequestDispatcher`.

Как видно из описания методов, к ресурсу можно обратиться только через объект типа `RequestDispatcher`. Этот объект предлагает два метода обращения к ресурсу.

Первый метод,

```
public void forward(ServletRequest req, ServletResponse resp);
```

просто передает управление другому ресурсу, предоставив ему свои аргументы `req` и `resp`. Вызывающий сервлет выполняет предварительную обработку объектов `req` и `resp` и передает их вызванному сервлету или другому ресурсу, который окончательно формирует ответ `resp` и отправляет его клиенту или опять-таки вызывает другой ресурс. Например:

```
if (rd != null) rd.forward(req, resp);
else resp.sendError(HttpServletResponse.SC_NO_CONTENT);
```

Вызывающий сервлет не должен выполнять какую-либо отправку клиенту до обращения к методу `forward()`, иначе будет выброшено исключение класса `IllegalStateException`. Если же вызывающий сервлет уже что-то отправлял клиенту, то следует обратиться ко второму методу,

```
public void include(ServletRequest req, ServletResponse resp);
```

Этот метод вызывает ресурс, который на основании объекта `req` может изменить тело объекта `resp`. Но вызванный ресурс не может изменить заголовки и код ответа объекта `resp`. Это естественное ограничение, поскольку вызывающий сервлет мог уже отправить заголовки клиенту. Попытка вызванного ресурса изменить заголовок будет просто проигнорирована контейнером. Можно сказать, что метод `include()` выполняет такую же работу, как вставки на стороне сервера SSI (Server Side Include).

После выполнения метода `include()` управление возвращается в сервлет.

Вопросы для самопроверки

1. Какая разница между HTTP-сервером и Web-сервером?
2. Что такое сервер приложений?
3. Что такое сервлет?
4. Что такое контейнер сервлетов?
5. Что означает процедура установки сервлета?
6. Может ли сервлет отправить клиенту не страницу HTML, а другой документ?
7. Может ли сервлет обрабатывать параллельно несколько запросов?
8. Могут ли сервлеты, установленные в один контейнер, обмениваться информацией?
9. Может ли сервлет установить сеанс связи с клиентом?



Глава 27

Страницы JSP

Как видно из приведенных в предыдущей главе листингов, большую часть сервлета занимают операторы вывода в выходной поток тегов HTML, формирующих результат — страницу HTML. Эти операторы почти без изменений повторяются из сервлета в сервлет. Возникла идея не записывать теги HTML в операторах Java, а, наоборот, записывать операторы Java в коде HTML с помощью тегов специального вида. Затем обработать полученную страницу препроцессором, распознающим все теги и преобразующим их в код сервлета Java.

Так получился язык разметок JSP (JavaServer Pages), расширяющий язык HTML тегами вида `<% имя_тега атрибуты %>`. С помощью тегов можно не только записать описания, выражения и операторы Java, но и вставить в страницу файл с текстом или изображением, вызвать объект Java, компонент JavaBean или компонент EJB.

Программист может даже расширить язык JSP своими собственными, как говорят, *пользовательскими тегами* (custom tags), которые сейчас чаще называют *расширениями тегов* (tag extensions).

В листинге 27.1 приведен пример JSP-страницы "Hello, World!" с текущей датой.

Листинг 27.1. Простейшая страница JSP

```
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ page contentType="text/html;charset=windows-1251" %>
<%@ page import="java.util.*, java.text.*" %>
<html><head><title> Простейшая страница JSP </title>
<META http-equiv=Content-Type
      content="text/html; charset=windows-1251">
</head><body>
Hello, World!<p>
Сегодня <%= getFormattedDate() %>
</body></html>
<%!
  String getFormattedDate(){
    SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy hh:mm");
    return sdf.format(new Date());
  }
%>
```

Контейнер сервлетов расширили препроцессором, переводящим запись, подобную листингу 27.1, в сервлет. В контейнере сервлетов Tomcat такой препроцессор называется Jasper. Препроцессор срабатывает автоматически при первом обращении к странице JSP. Полученный в результате его работы сервлет тут же компилируется и выполняется. Откомпилированный сервлет затем хранится в контейнере, так же как и все сервлеты, и выполняется при следующих вызовах страницы JSP.

Для сервлетов и страниц JSP придумано общее название — *Web-компоненты* (Web Components). Контейнер сервлетов, расширенный средствами работы с JSP, называется *Web-контейнером* (Web Container или JSP Container). Приложение, составленное из сервлетов, страниц JSP, апплетов, документов HTML и XML, изображений и прочих документов, относящихся к приложению, называется *Web-приложением* (Web Application).

Весь статический текст HTML, называемый в документации JSP *шаблоном HTML* (template HTML), сразу направляется в выходной поток. Выходной поток страницы буферизуется. Буферизацию обеспечивает класс `JspWriter`, расширяющий класс `Writer`. Размер буфера по умолчанию 8 Кбайт, его можно изменить атрибутом `buffer` тега `<%@ page>`. Наличие буфера позволяет заносить заголовки ответа в выходной поток вперемешку с выводимым текстом. В буфере заголовки будут размещены перед текстом.

Таким образом, программисту достаточно написать страницу JSP, записать ее в файл с расширением jsp и установить файл в контейнер, подобно странице HTML, не заботясь о компиляции. При установке можно задать начальные параметры страницы JSP так же, как и начальные параметры сервлета.

В то время как создавался и развивался язык JSP, широкое распространение получил язык XML, который мы рассмотрим в *главе 28*. Для совместимости с ним все теги JSP продублированы элементами XML с именами из пространства имен **http://java.sun.com/JSP/Page**, получающего, как правило, префикс `jsp:`. Теги JSP, записанные в форме XML, сейчас называют *действиями* (actions) JSP.

Например, страницу JSP, представленную в листинге 27.1, можно написать в форме XML так, как это сделано в листинге 27.2.

Листинг 27.2. Простейшая страница JSP в форме XML

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
  <jsp:directive.page
     contentType="text/html;charset=windows-1251" />
  <jsp:directive.page
       import="java.util.Date, java.text.SimpleDateFormat" />

  <jsp:text>
    <![CDATA[
    <html><head><title> Простейшая страница JSP </title>
    <META http-equiv=Content-Type
          content="text/html; charset=windows-1251">
    </head><body>
       Hello, World!<p>
       Сегодня  ]]>
  </jsp:text>
```

```
  <jsp:expression>getFormattedDate()</jsp:expression>
  <jsp:text>
     <![CDATA[
     </body></html> ]]>
  </jsp:text>
  <jsp:declaration>
    String getFormattedDate(){
      SimpleDateFormat sdf = new SimpleDateFormat("dd-MMMM-yyyy hh:mm");
      return sdf.format(new Date());
    }
  </jsp:declaration>
</jsp:root>
```

Файл со страницей JSP, записанной в форме XML, обычно получает расширение jspx.

Некоторые теги — `<jsp:forward>`, `<jsp:include>`, `<jsp:plugin>`, `<jsp:useBean>`, `<jsp:getProperty>`, `<jsp:setProperty>` — всегда записывались в форме XML.

Вы уже знаете, что язык XML различает регистр букв. Теги XML записываются, как правило, строчными буквами. Значения атрибутов тегов обязательно записываются в кавычках или апострофах. У большинства элементов XML есть *открывающий тег* (start tag) с необязательными атрибутами, *тело* (body) и *закрывающий тег* (end tag), они имеют вид:

```
<jsp:тег атрибуты_тега >
   Тело элемента или вложенные элементы XML
</jsp:тег>
```

Тело элемента может быть пустым, тогда элемент выглядит так:

```
<jsp:тег атрибуты_тега ></jsp:тег>
```

> ***Внимание***
>
> Если между открывающим и закрывающим тегом есть хотя бы один пробел, то тело элемента уже не пусто.

Наконец, тело может отсутствовать, тогда закрывающий тег не пишется, а в открывающем теге перед закрывающей угловой скобкой ставится наклонная черта:

```
<jsp:тег атрибуты_тега />
```

Если у XML-элемента есть тело, то его открывающий тег заканчивается просто угловой скобкой, без наклонной черты.

Спецификация "JavaServer Pages Specification" не рекомендует смешивать в одном документе теги вида JSP с элементами XML. Страницу HTML, в которую вставлены теги вида `<%...%>`, она официально называет *страницей JSP* (JSP Page), а документ XML с тегами вида `<jsp:.../>` называет *документом JSP* (JSP Document). Файл со страницей JSP обычно получает имя с расширением jsp, а файл с документом JSP — имя с расширением jspx.

Документ JSP проходит еще одну стадию предварительной обработки, на которой он приводится в полное соответствие с синтаксисом XML. Это приведение включает в себя запись корневого элемента с пространством имен **http://java.sun.com/JSP/Page** и вставку элемен-

тов CDATA. После приведения получается документ XML, официально называемый *представлением XML* (XML View).

Стандартные действия (теги) JSP

Набор стандартных тегов JSP довольно прост. При их написании следует помнить три правила:

- язык JSP различает регистр букв, как и язык Java;
- при записи атрибутов, после знака равенства, отделяющего имя атрибута от его значения, нельзя оставлять пробелы;
- значения атрибутов можно заносить не только в кавычки, но и в апострофы.

Будем записывать теги и в старой форме JSP, и в новой форме элементов XML.

Комментарий на страницах JSP отмечается тегом

```
<%-- Комментарий --%>
```

или тегом

```
<!-- Комментарий -->
```

Комментарий первого вида не передается клиенту. Все, что написано внутри него, не обрабатывается препроцессором. Комментарий второго вида переносится в формируемую HTML-страницу. Все JSP-теги, записанные внутри такого комментария, интерпретируются.

Объявления полей и методов Java записываются в теге

```
<%! Объявления %>
<jsp:declaration> Объявления </jsp:declaration>
```

После обработки препроцессором они будут полями и методами сервлета.

Выражение Java записывается в теге

```
<%= Выражение %>
<jsp:expression> Выражение </jsp:expression>
```

Выражение вычисляется, результат подставляется на место тега. Учтите, что в конце выражения не надо ставить точку с запятой, поскольку выражение, завершающееся точкой с запятой, — это уже оператор.

Фрагмент кода Java, называемый в JSP *скриптлетом* (scriptlet), который может включать в себя не только операторы, но и определения, записывается в теге

```
<% Скриптлет %>
<jsp:scriplet> Скриптлет </jsp:scriptlet>
```

Такой фрагмент после обработки препроцессором попадет в метод _jspService() создаваемого сервлета, являющийся оболочкой метода service().

Включение файла во время компиляции производится тегом

```
<%@ include file="URL файла относительно контекста" %>
<jsp:directive.include file="URL файла относительно контекста" />
```

Общие свойства страницы JSP задаются тегом

```
<%@ page Атрибуты %>
<jsp:directive.page Атрибуты />
```

Все атрибуты здесь необязательны. В листингах 27.1 и 27.2 уже использованы атрибуты `contentType` и `import` этого тега. Другие атрибуты:

- `pageEncoding="Кодировка"` — по умолчанию `"ISO 8859-1"`;
- `extends="Полное имя расширяемого класса"` — суперкласс того класса, в который компилируется страница JSP;
- `session="true или false"` — создавать ли сеанс связи с клиентом, по умолчанию `"true"`;
- `buffer="Nkb или none"` — размер выходного буфера, по умолчанию `"8kb"`;
- `autoFlush="true или false"` — автоматическая очистка буфера по его заполнении, по умолчанию `"true"`; если значение этого атрибута равно `"false"`, то при переполнении буфера выбрасывается исключение;
- `isThreadSafe="true или false"` — одновременный доступ к странице нескольких клиентов, по умолчанию `"true"`; если этот атрибут равен `"false"`, то полученный после компиляции сервлет реализует устаревший и не используемый сейчас интерфейс `SingleThreadModel`;
- `info="какой-то текст"` — сообщение, которое можно будет прочитать методом `getServletInfo()`;
- `errorPage="URL относительно контекста"` — адрес страницы JSP, которой посылается исключение и которая показывает сообщения, посылаемые исключением;
- `isErrorPage="true или false"` — может ли данная страница JSP использовать объект-исключение `exception`, или он будет переслан другой странице, по умолчанию `"false"`.

Два последних атрибута требуют разъяснения. Дело в том, что если страница JSP не обрабатывает исключение, а выбрасывает его дальше, то Web-контейнер формирует страницу HTML с сообщениями об исключении и посылает ее клиенту. Это мешает работе клиента. Атрибут `errorPage` позволяет вместо страницы HTML с сообщениями передать встроенный объект-исключение `exception` для обработки странице JSP, которую предварительно создал разработчик. Атрибут `isErrorPage` страницы-обработчика исключения должен равняться `"true"`.

Пусть, например, имеется страница JSP:

```
<html><body>
<%@ page errorPage="myerrpage.jsp" %>
<%
   String str = <%= request.getParameter("name") %>;
   int n = str.length();
%>
</body></html>
```

Ссылка `str` может оказаться равной `null`, тогда метод `length()` выбросит исключение. Контейнер создаст встроенный объект `exception` и передаст его странице myerrpage.jsp. Она может выглядеть так:

```
<html><body>
   <%@ page isErrorPage="true" %>
   При выполнении страницы JSP выброшено исключение:
   <%= exception %>
</body></html>
```

На этом набор тегов вида `<%...%>` заканчивается. Остальные теги записываются только в форме элементов XML.

Включение файла на этапе выполнения или включение результата выполнения сервлета, если этот результат представлен страницей JSP, выполняется элементом

```
<jsp:include Атрибуты />
```

В этом теге может быть один или два атрибута. Обязательный атрибут

```
page="относительный URL или выражение JSP"
```

задает адрес включаемого ресурса. Здесь "выражение JSP" записывается в форме JSP `<%= выражение %>` или в форме XML

```
<jsp:expression> выражение </jsp:expression>
```

В результате выражения должен получиться адрес URL. Чаще всего в качестве выражения используется обращение к методу `getParameter(String)`.

Второй, необязательный, атрибут

```
flush="true или false"
```

указывает, очищать ли выходной буфер перед включением ресурса. Его значение по умолчанию `"false"`.

Вторая форма элемента, `include`, содержит теги `<jsp:param>` в теле элемента (обратите внимание на отсутствие наклонной черты в конце открывающего тега):

```
<jsp:include Атрибуты >
   Здесь записываются теги вида <jsp:param>
</jsp:include>
```

В теле элемента можно записывать произвольные параметры. Они имеют вид

```
<jsp:param name="имя параметра"
           value="значение параметра или выражение JSP" />
```

"Выражение JSP" записывается так же, как и в заголовке тега. Параметры передаются включаемого ресурсу как его начальные параметры, и их имена, разумеется, должны совпадать с именами начальных параметров ресурса.

Включаемый JSP-файл может быть оформлен не полностью, а содержать только отдельный фрагмент кода JSP. В таком случае его имя записывают обычно с расширением jspf (JSP Fragment).

Следующий стандартный тег <jsp:element> позволяет динамически определить какой-нибудь элемент XML. У него один обязательный атрибут name — имя создаваемого элемента XML. В его теле можно определить атрибуты создаваемого элемента XML с помощью элемента <jsp:attribute> и тело создаваемого элемента XML с помощью элемента <jsp:body>. Например:

```
<jsp:element name="${myElem}"
             xmlns:jsp="http://java.sun.com/JSP/Page">
   <jsp:attribute name="lang">${content.lang}</jsp:attribute>
   <jsp:body>${content.body}</jsp:body>
</jsp:element>
```

Еще один простой стандартный тег <jsp:text> не содержит атрибутов. Его тело без всяких изменений передается в выходной поток. Например:

```
<jsp:text>
   while(k < 10) {a[k]++; b[k++] = $1;}
</jsp:text>
```

Вы, наверное, заметили, что теги JSP не создают никакого кода инициализации сервлета, того, что обычно записывается в метод init() сервлета. Такой код при необходимости как-то инициализировать полученный после компиляции сервлет надо записать в метод jspInit() по следующей схеме:

```
<jsp:declaration>
   public void jspInit(){
      // Записываем код инициализации
   }
</jsp:declaration>
```

Аналогично, завершающие действия сервлета можно записать в метод jspDestroy() по такой схеме:

```
<jsp:declaration>
   public void jspDestroy(){
      // Записываем код завершения
   }
</jsp:declaration>
```

Язык записи выражений EL

Хотя на странице JSP можно записать любое выражение языка Java в теге <%=...%> или в элементе XML <jsp:expression>...</jsp:expression>, в JSP, начиная с версии 2.0, введен язык EL (Expression Language), предназначенный для записи выражений. Язык JSP EL появился в русле тенденции к изгнанию со страниц JSP чужеродного кода и замены его своими "родными" конструкциями.

В языке JSP EL выражения окружаются символами `${...}`, например `${2 + 2}`. Это окружение дает сигнал к вычислению заключенного в них выражения. Выражение, записанное без этих символов, не будет вычисляться и воспримется как простой набор символов.

В выражениях можно использовать данные типов `boolean`, `long`, `float` и строковые константы, заключенные в кавычки или апострофы. Значение `null` считается отдельным типом.

С данными этих типов можно выполнять арифметические операции `+`, `-`, `*`, `/`, `%`, сравнения `==`, `!=`, `<`, `>`, `<=`, `>=`, логические операции `!`, `&&`, `||` и условную операцию `?:`. Интересно, что сравнения "равно", "не равно", "меньше", "больше", "меньше или равно", "больше или равно" можно записать не только специальными символами, но и сокращениями `eq`, `ne`, `lt`, `gt`, `le`, `ge` слов "equal", "not equal", "less than", "greater", "less than or equal", "greater or equal". Это позволяет оставить знаки "больше" и "меньше" только для записи тегов XML. По аналогии операцию деления можно записать словом `div`, операцию взятия остатка от деления — словом `mod`, а логические операции — словами `not`, `and` и `or`.

В выражениях можно обращаться к переменным, например `${name}`, полям и методам объектов, например `${pageContext.request.requestURI}`. В выражениях языка JSP EL можно использовать следующие предопределенные объекты:

- `pageContext` — объект типа `PageContext`;
- `pageScope` — объект типа `Map`, содержащий атрибуты страницы и их значения;
- `requestScope` — объект типа `Map`, содержащий атрибуты запроса и их значения;
- `sessionScope` — объект типа `Map`, содержащий атрибуты сеанса и их значения;
- `applicationScope` — объект типа `Map`, содержащий атрибуты приложения и их значения;
- `param` — объект типа `Map`, содержащий параметры запроса, получаемые в сервлетах методом `ServletRequest.getParameter(String name)`;
- `paramValues` — объект типа `Map`, содержащий параметры запроса, получаемые в сервлетах методом `ServletRequest.getParameterValues(String name)`;
- `header` — объект типа `Map`, содержащий заголовки запроса, получаемые в сервлетах методом `ServletRequest.getHeader(String name)`;
- `headerValues` — объект типа `Map`, содержащий заголовки запроса, получаемые в сервлетах методом `ServletRequest.getHeaders(String name)`;
- `initParam` — объект типа `Map`, содержащий параметры инициализации контекста, получаемые в сервлетах методом `ServletContext.getInitParameter(String name)`;
- `cookie` — объект типа `Map`, содержащий имена и объекты типа `Cookie`.

Наконец, в выражениях языка JSP EL можно записывать вызовы функций.

Встроенные объекты JSP

Каждая страница JSP может содержать в выражениях и скриптлетах девять готовых встроенных объектов, создаваемых контейнером JSP при выполнении сервлета, полученного после компиляции страницы JSP. Мы уже использовали объекты request и exception. У этих

объектов заданы определенные имена и типы. В большинстве случаев заданы не точные типы объектов, а их суперклассы и интерфейсы:

- `request` — объект типа `ServletRequest`, чаще всего это объект типа `HttpServletRequest`;
- `response` — объект типа `ServletResponse`, обычно это объект типа `HttpServletResponse`;
- `config` — объект типа `ServletConfig`;
- `application` — объект типа `ServletContext`;
- `session` — объект типа `HttpSession`;
- `pageContext` — объект типа `PageContext`;
- `out` — выходной поток типа `JspWriter`;
- `page` — произвольный объект класса `Object`;
- `exception` — исключение класса `Throwable`.

В листинге 27.3 приведена страница JSP, использующая скриптлеты и встроенные объекты для организации запроса к базе СДО, описанной в предыдущей главе.

Листинг 27.3. Страница JSP, использующая скриптлеты

```
<%@ page import="java.sql.*"
       contentType="text/html;charset=windows-1251" %>

<html><head><title>Запрос к базе СДО</title></head>
<body>
<h2> Здравствуйте
   <%= (request.getRemoteUser() != null ? ", " +
        request.getRemoteUser() : "") %>!
</h2><hr><p>
<%
try{
   Connection conn =
     DriverManager.getConnection(
          (String)session.getValue("connStr"), "sdoadmin", "sdoadmin");

      Statement st = conn.createStatement ();

   ResultSet rs = st.executeQuery ("SELECT name, mark " +
         "FROM students ORDER BY name");

   if (rs.next()){
   %>

<table border=1>
<tr>
  <th width=200> <i>Ученик</i> </th>
  <th width=100> <i>Оценка</i> </th>
</tr>
```

```
<tr>
  <td> <%= rs.getString(1) %> </td>
  <td> <%= rs.getInt(2) %> </td>
   </tr>

<%
  while (rs.next()){
   %>

<tr>
  <td> <%= rs.getString(1) %> </td>
  <td> <%= rs.getInt(2) %> </td>
   </tr>

<%
  }
   %>

   </table>

<%
  }else{
   %>

   <p> Извините, но сведений нет! </p>

<%
   }
   rs.close();
   st.close();
}catch(SQLException e){
   out.println("<p>Ошибка при выполнении запроса:");
   out.println ("<pre>" + e + "</pre>");
}
%>
</body></html>
```

Обращение к компоненту JavaBean

Из страницы JSP можно обращаться к объектам Java, оформленным как компоненты JavaBeans. Это выполняется тегом

```
<jsp:useBean id="имя экземпляра компонента"
   [ scope="page или request или session или application" ]
    Класс компонента
/>
```

"Имя экземпляра компонента" `id` определяет имя JavaBean, уникальное в заданной атрибутом `scope` области. По умолчанию принимается область `page` — текущая страница JSP и включенные в нее страницы.

Компонент хранится как атрибут контекста указанной атрибутом `scope` области и вызывается методом `getAttribute()` соответствующего контекста.

- Если атрибут `scope` равен "`page`", то компонент хранится как один из атрибутов объекта класса `PageContext`.
- Если атрибут `scope` равен "`request`", то компонент хранится как атрибут объекта типа `ServletRequest`.
- Если атрибут `scope` равен "`session`", то компонент будет атрибутом объекта типа `HttpSession`.
- Наконец, если атрибут `scope` равен "`application`", то компонент станет атрибутом типа `ServletContext`.

Определенное в атрибуте `id` имя используется при обращении к свойствам и методам компонента JavaBean. Обязательный атрибут "класс компонента" описывается одним из трех способов:

- `class="`*`полное имя класса`*`" [ type="`*`полное имя суперкласса`*`" ]`
- `beanName="`*`полное имя класса или выражение JSP`*`"`
  `type="`*`полное имя суперкласса`*`"`
- `type="`*`полное имя суперкласса`*`"`

При обращении к компоненту JavaBean в теле элемента можно задавать и другие элементы.

Свойство (property) уже вызванного тегом `<jsp:useBean>` компонента JavaBean с именем `id="myBean"` устанавливается тегом

```
<jsp:setProperty name="myBean"
   property="имя" value="строка или выражение JSP" />
```

или тегом

```
<jsp:setProperty name="myBean"
   property="имя" param="имя параметра запроса" />
```

Во втором случае свойству компонента JavaBean дается значение, определенное параметром запроса, имя которого указано атрибутом `param`.

Третья форма этого тега

```
<jsp:setProperty name="myBean" property="*" />
```

применяется в тех случаях, когда имена всех свойств компонента JavaBean совпадают с именами параметров запроса вплоть до совпадения регистров букв.

Для получения свойств уже вызванного компонента JavaBean с именем "`myBean`" существует тег

```
<jsp:getProperty name="myBean" property="имя свойства" />
```

В его атрибуте `property` уже нельзя записывать звездочку.

Выполнение апплета в браузере клиента

Если в браузере клиента установлен Java Plug-in, то в нем можно организовать выполнение апплета или компонента с помощью элемента

```
<jsp:plugin type="bean или applet"
   [code="имя класса апплета"]
   [codebase= "каталог апплета"]
   Прочие параметры заголовка тега
>
   Здесь записываются необязательные параметры
</jsp:plugin>
```

Как видно из этого описания, элемент `<jsp:plugin>` очень похож на тег `<applet>` языка HTML. Похожи и его атрибуты `code` и `codebase`, только в имени класса апплета или компонента надо обязательно указывать его расширение `.class`. Если атрибут `codebase` не указан, то по умолчанию понимается каталог, в котором лежит страница JSP. Прочие атрибуты заголовка тоже подобны параметрам тега `<applet>`:

- `name="имя экземпляра"`;
- `archive="список адресов URL архивов апплета"`;
- `align="bottom или top, или middle, или left, или right"`;
- `height="высота в пикселах или выражение JSP"`;
- `width="ширина в пикселах или выражение JSP"`;
- `hspace="горизонтальные поля в пикселах"`;
- `vspace="вертикальные поля в пикселах"`;
- `jreversion="версия JRE, по умолчанию 1.2"`;
- `nspluginurl="полный адрес URL, с которого можно загрузить Java Plug-in для Netscape Communicator"`;
- `iepluginurl="полный адрес URL, с которого можно загрузить Java Plug-in для Internet Explorer"`.

В теле элемента можно поместить любые параметры вида

```
<jsp:params>
   <jsp:param name="имя" value="значение или выражение JSP" />
</jsp:params>
```

которые будут переданы апплету или компоненту. Кроме того, в теле элемента допустимо указывать сообщение, которое появится в окне браузера, если апплет или компонент не удалось загрузить. Для этого используется элемент

```
<jsp:fallback> Текст сообщения </jsp:fallback>
```

Передача управления

Страница JSP имеет возможность передать управление другому ресурсу: странице JSP, сервлету или странице HTML. Это выполняется тегом

```
<jsp:forward page="адрес URL относительно контекста" />
```

содержащим адрес объекта, которому передается управление. Адрес может быть получен как результат вычисления выражения JSP. Управление не возвращается, и строки, следующие за тегом `<jsp:forward>`, не будут выполняться.

Ресурсу можно передать один или несколько параметров в теле элемента:

```
<jsp:forward page="адрес URL относительно контекста" >
   <jsp:param name="имя" value="значение или выражение JSP" />
</jsp:forward>
```

Пользовательские теги

Разработчик страниц JSP может расширить набор стандартных действий (тегов) JSP, создав свои собственные, как говорят, *пользовательские теги* (custom tags). Пользовательские теги организуются в виде целой библиотеки, даже если в нее входит только один тег. Описание каждой библиотеки хранится в отдельном XML-файле с расширением tld, называемым *описателем библиотеки тегов* TLD (Tag Library Descriptor). Этот файл хранится в каталоге WEB-INF данного Web-приложения или в его подкаталоге. Если Web-приложение упаковано в JAR-архив, то TLD-файлы, описывающие библиотеки пользовательских тегов этого приложения, находятся в каталоге META-INF архива.

В листинге 27.4 приведен пример простого TLD-файла, описывающего библиотеку пользовательских тегов с одним тегом `head`, реализуемым классом `HeadTag`. Полное объяснение использованных в нем элементов XML вы получите в следующей главе.

Листинг 27.4. Описатель TLD библиотеки тегов

```
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd"
  version="2.0">
  <tlib-version>1.0</tlib-version>
  <short-name></short-name>
  <uri>/sdo</uri>
  <tag>
    <name>head</name>
    <tag-class>sdotags.HeadTag</tag-class>
    <body-content>JSP</body-content>
    <attribute>
      <name>size</name>
      <required>false</required>
```

```
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
</taglib>
```

На странице JSP перед применением пользовательских тегов следует сослаться на библиотеку тегом

```
<%@ taglib uri="адрес URI библиотеки"
           prefix="префикс тегов библиотеки" %>
```

Например, если на странице JSP написан тег

```
<%@ taglib uri="/WEB-INF/sdotaglib.tld" prefix="sdo" %>
```

то на ней можно использовать теги вида `<sdo:head />`.

У этого тега нет прямого XML-эквивалента. Префикс тегов библиотеки и ее адрес определяются в форме XML как пространство имен атрибутом `xmlns` в элементе `<jsp:root>`. Например, библиотека с префиксом тегов `sdo` и адресом `/WEB-INF/sdotaglib.tld` описывается так:

```
<jsp:root
   xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:sdo="/WEB-INF/sdotaglib.tld"
   version="2.0"
>
   Страница JSP
</jsp:root>
```

Необязательный элемент `<jsp:root>`, если он присутствует, должен быть корневым элементом документа XML.

Как видите, к каждой странице JSP всегда подключается библиотека с префиксом тегов `jsp`. Стандартные (core) действия JSP входят в эту библиотеку. Префиксы `jsp`, `jspx`, `java`, `javax`, `servlet`, `sun`, `sunw` зарезервированы корпорацией Sun Microsystems, их нельзя употреблять в библиотеках пользовательских тегов.

В конфигурационном файле web.xml можно создать псевдонимы адреса URI библиотеки с помощью элемента `<taglib>`, например:

```
<taglib>
   <taglib-uri>/sdo</taglib-uri>
   <taglib-location>/WEB-INF/sdotaglib.tld</taglib-location>
</taglib>
```

После этого на странице JSP можно написать ссылку на библиотеку так:

```
<%@ taglib uri="/sdo" prefix="sdo" %>
```

Псевдоним может быть указан и в TLD-файле, в элементе `<uri>`, как показано в листинге 27.4. В этом случае контейнер, обнаружив на странице JSP тег `<%@ taglib %>` с псевдонимом, просматривает TLD-файлы в поисках этого псевдонима в их элементах `<uri>`. Найдя псевдоним, контейнер связывает его с путем к TLD-файлу, содержащему псевдоним.

Поиск TLD-файлов происходит только в подкаталогах каталога WEB-INF и во всех JAR-архивах, а именно в каталогах META-INF, находящихся в JAR-архивах.

Каждый тег создаваемой библиотеки реализуется классом Java, называемым в документации *обработчиком тега* (tag handler). Обработчик тега должен реализовать интерфейс `Tag`, а если у тега есть тело, которое надо выполнить несколько раз, то нужно реализовать его расширение — интерфейс `IterationTag`. Если же тело пользовательского тега требует предварительной обработки, то следует использовать расширение интерфейса `IterationTag` — интерфейс `BodyTag`. Эти интерфейсы собраны в пакет `javax.servlet.jsp.tagext`. В нем есть и готовые реализации указанных интерфейсов — класс `TagSupport`, реализующий интерфейс `IterationTag`, и его расширение — класс `BodyTagSupport`, реализующий интерфейс `BodyTag`.

Итак, для создания пользовательского тега без тела или с телом, но не требующим предварительной обработки, удобно расширить класс `TagSupport`, а для создания тега с телом, которое надо предварительно преобразовать, — класс `BodyTagSupport`.

Классы, реализующие библиотеку тегов, хранятся в каталоге WEB-INF/classes, а если они упакованы в JAR-архив — то в каталоге WEB-INF/lib своего Web-приложения. Если библиотека разделяется несколькими Web-приложениями, то она хранится в каком-нибудь общем каталоге, например common/lib. Соответствие между именами тегов и классами, реализующими их, указывается в TLD-файле, описывающем библиотеку, в XML-элементе `<tag>`. В этом же элементе, во вложенном XML-элементе `<attribute>`, описываются атрибуты открывающего тега. Например:

```
<tag>
   <name>reg</name>
   <tag-class>sdotags.RegTag</tag-class>
   <body-content>EMPTY</body-content>
   <attribute>
      <name>name</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
      <type>java.lang.String</type>
   </attribute>
</tag>
```

Если после этого на странице JSP написать пользовательский тег `<sdo:reg>`, например:

```
<sdo:reg name="%=request.getParameter(\"name\") %" />
```

или с помощью языка JSP EL:

```
<sdo:reg name="${param.name}" />
```

то для обработки данного тега будет создан объект класса `RegTag` из пакета `sdotags`.

Класс-обработчик пользовательского тега

Как уже было сказано ранее, класс-обработчик элемента, не имеющего тела, или тело которого не требует преобразований, должен реализовать интерфейс `Tag` или расширить класс

TagSupport. Если тело элемента перед посылкой клиенту надо преобразовать, то классу-обработчику следует реализовать интерфейс BodyTag или расширить класс BodyTagSupport. В случае простых преобразований можно реализовать интерфейс SimpleTag или расширить класс SimpleTagSupport.

Основной метод интерфейса Tag

```
public int doStartTag();
```

выполняет действия, предписанные открывающим тегом. К нему сервлет обращается автоматически, начиная обработку элемента. Метод должен вернуть одну из двух констант интерфейса Tag, указывающих на дальнейшие действия:

- EVAL_BODY_INCLUDE — обрабатывать тело элемента;
- SKIP_BODY — не обрабатывать тело элемента.

После завершения этого метода сервлет обращается к методу

```
public int doEndTag();
```

который выполняет действия, завершающие обработку пользовательского тега. Метод возвращает одну из двух констант интерфейса Tag:

- EVAL_PAGE — продолжать обработку страницы JSP;
- SKIP_PAGE — завершить на этом обработку страницы JSP.

Интерфейс IterationTag добавляет метод

```
public int doAfterTag();
```

позволяющий обработать повторно тело пользовательского тега. Он будет выполняться перед методом doEndTag(). Если метод doAfterTag() возвращает константу EVAL_BODY_AGAIN интерфейса IterationTag, то тело элемента будет обработано еще раз, если константу SKIP_BODY — обработка тела не станет повторяться.

Интерфейс BodyTag позволяет буферизовать выполнение тела элемента. Буферизация производится, если метод doStartTag() возвращает константу EVAL_BODY_BUFFERED интерфейса BodyTag. В таком случае перед обработкой тела тега контейнер обращается к методу

```
public void doInitBody();
```

который может выполнить различные предварительные действия.

У этих методов нет аргументов, они получают информацию из объекта класса PageContext, который всегда создается Web-контейнером для выполнения любой страницы JSP. При реализации интерфейса Tag или BodyTag данный объект можно получить методом getPageContext() класса JspFactory, предварительно получив объект класса JspFactory его собственным статическим методом getDefaultFactory(). Сложность таких манипуляций — еще один аргумент для того, чтобы не реализовывать интерфейсы, а расширять класс TagSupport или класс BodyTagSupport.

Итак, для создания пользовательского тега без тела или с телом, не требующим обработки, удобнее всего расширить класс TagSupport, а для создания пользовательского тега с обработкой тела — расширить класс BodyTagSupport. При этом разработчик получает в свое распоряжение объект класса PageContext просто в виде защищенного поля pageContext.

Описанные ранее методы реализованы в классе `TagSupport` очень просто:

```
public int doStartTag() throws JspException{
   return SKIP_BODY;
}
public int doEndTag() throws JspException{
   return EVAL_PAGE;
}
public int doAfterBody() throws JspException{
   return SKIP_BODY;
}
```

В подклассе `BodyTagSupport` реализация метода `doStartTag()` немного изменена:

```
public int doStartTag() throws JspException{
  return EVAL_BODY_BUFFERED;
}
```

Метод `doInitBody()` оставлен пустым.

Итак, в самом простом случае достаточно расширить класс `TagSupport`, переопределив метод `doStartTag()`. Пусть, например, определен пользовательский тег без аргументов и без тела, всего лишь отправляющий клиенту сообщение:

```
<sdo:info />
```

Реализующий его класс может выглядеть так, как показано в листинге 27.5.

Листинг 27.5. Класс простейшего пользовательского тега

```
package sdotags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class InfoTag extends TagSupport{
   public int doStartTag() throws JspException{
      pageContext.getOut().print("Библиотека тегов СДО.");
      return SKIP_BODY;
   }
}
```

Исходный текст листинга 27.5 надо откомпилировать обычным образом и установить в контейнер так же, как устанавливается сервлет. Проследите за правильным соответствием пакетов Java и каталогов файловой системы: в каталоге WEB-INF/classes должен быть подкаталог sdotags с файлом InfoTag.class.

Еще проще эти действия выполняются с помощью метода `doTag()` интерфейса `SimpleTag`, реализованного в классе `SimpleTagSupport`. У данного метода нет аргументов и возвращаемого значения, он объединяет действия, обычно выполняемые методами `doStartTag()` и `doEndTag()`.

Пользовательский тег с атрибутами

Для каждого атрибута открывающего тега надо определить свойство JavaBean, т. е. поле с именем, совпадающим с именем атрибута, и методы доступа `get`*`Xxx`*`()` и `set`*`Xxx`*`()`. Например, немного ранее *(см. разд. "Пользовательские теги" данной главы)* мы определили пользовательский тег

```
<sdo:reg name="имя" />
```

с одним атрибутом `name`. Класс `RegTag`, содержащийся в листинге 27.6, реализует этот тег.

Листинг 27.6. Пользовательский тег с атрибутом

```
   package sdotags;

import javax.servlet.jsp.*;
   import javax.servlet.jsp.tagext.*;

   public class RegTag extends TagSupport{

      private String name;

   public String getName(){
      return name;
      }

   public void setName(String name){
      this.name = name;
      }

   public int doStartTag() throws JspException{
      if (name == null)
        name = pageContext.getRequest().getParameter("name");
      registrate(name);
      return SKIP_BODY;
   }
}
```

Пользовательский тег с телом

Если у пользовательского тега есть тело, то при описании тега в TLD-файле в элементе `<body-content>` вместо слова `EMPTY` следует написать слово `JSP` или вообще не писать этот элемент, поскольку его значение `JSP` принимается по умолчанию.

У тела элемента `<body-content>` могут быть еще два значения. Значение `tagdependent` применяется, если содержимое тела тега написано не на языке JSP, а на каком-то другом языке, например это запрос на языке SQL. Значение `scriptless` показывает, что в теге нет скриптлетов.

Если содержимое тела тега не нужно обрабатывать, а надо только отправить клиенту, то при создании его обработчика достаточно реализовать интерфейс `Tag` или расширить класс

TagSupport. Если метод doStartTag() обработчика вернет значение EVAL_BODY_INCLUDE, то все тело тега будет автоматически отправлено в выходной поток.

Пусть, например, в файле sdotaglib.tld определен пользовательский тег head:

```
<tag>
   <name>head</name>
   <tag-class>sdotags.HeadTag</tag-class>
   <body-content>JSP</body-content>
   <attribute>
      <name>size</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
   </attribute>
</tag>
```

Этот тег реализован классом HeadTag, описанным в листинге 27.7.

Листинг 27.7. Пользовательский тег с простым телом

```
   package sdotags;

import javax.servlet.jsp.*;
   import javax.servlet.jsp.tagext.*;

   public class HeadTag extends TagSupport{

      private String size = "4";

   public String getSize(){
      return size;
      }

   public void setSize(String size){
      this.size = size;
      }

   public int doStartTag(){
      try{
         JspWriter out = pageContext.getOut();
         out.print("<font size=\"" + size + "\">");
      }catch(Exception e){
         System.err.println(e);
      }
      return EVAL_BODY_INCLUDE;
      }

   public int doEndTag(){
      try{
         JspWriter out = pageContext.getOut();
```

```
            out.print("</font>");
         }catch(Exception e){
            System.err.println(e);
         }
         return EVAL_PAGE;
      }
}
```

После этого на странице JSP можно писать пользовательский тег

```
<sdo:head size = "2" >
   Сегодня <jsp:expression>new java.util.Date()</jsp:expression>
</sdo:head>
```

Текст, написанный в его теле, будет выведен у клиента шрифтом указанного размера.

Обработка тела пользовательского тега

Если тело пользовательского тега требует обработки, то его класс-обработчик должен реализовать интерфейс `BodyTag` или расширить класс `BodyTagSupport`. Метод `doStartTag()` должен вернуть значение `EVAL_BODY_BUFFERED`. После завершения метода `doStartTag()`, если тело тега не пусто, контейнер вызовет метод `doInitBody()`, который может выполнить предварительные действия перед обработкой содержимого тела пользовательского тега. Далее контейнер обратится к методу `doAfterBody()`, в котором и надо проделать обработку тела тега, поскольку к этому моменту тело тега будет прочитано и занесено в объект класса `BodyContent`.

Класс `BodyContent` расширяет класс `JspWriter`, значит, формально является выходным потоком. Однако его удобнее рассматривать как хранилище информации, полученной из тела тега.

Объект класса `BodyContent` создается после каждой итерации метода `doAfterBody()`, и все эти объекты хранятся в стеке.

Ссылку на объект класса `BodyContent` можно получить двумя способами: методом

```
public BodyContent getBodyContent();
```

класса `BodyTagSupport` или, используя объект `pageContext`, следующим образом:

```
BodyContent bc = pageContext.pushBody();
```

Содержимое тела тега можно прочитать из объекта класса `BodyContent` тоже двумя способами: или получить ссылку на символьный входной поток методом

```
public Reader getReader();
```

или представить содержимое объекта в виде строки методом

```
public String getString();
```

После обработки прочитанного содержимого его надо отправить в выходной поток `out` методом

```
public void writeOut(Writer out);
```

Выходной поток `out`, аргумент этого метода, выводит информацию в стек объектов класса `BodyContent`. Поэтому его можно получить двумя способами: методом

```
public JspWriter getPreviousOut();
```

класса `BodyTagSupport` или методом

```
public JspWriter getEnclosingWriter();
```

класса `BodyContent`.

Приведем пример. Пусть тег `query`, описанный в TLD-файле sdotaglib.tld следующим образом:

```
<tag>
   <name>query</name>
   <tag-class>sdotags.QueryTag</tag-class>
   <body-content>tagdependent</body-content>
   <attribute>
      <name>size</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
   </attribute>
</tag>
```

содержит в своем теле SQL-запросы, например:

```
<sdo:query>
   SELECT * FROM students
</sdo:query>
```

В листинге 27.8 приведен фрагмент обработчика этого тега.

Листинг 27.8. Пользовательский тег обработки SQL-запросов

```
   package sdotags;

import java.sql.*;
import javax.servlet.jsp.*;
   import javax.servlet.jsp.tagext.*;

   public class QueryTag extends BodyTagSupport{

   private Connection conn;
      private ResultSet rs;

   public int doStartTag(){
      // . . .
      return EVAL_BODY_BUFFERED;
      }

   public void doInitBody(){
         conn = DriverManager.getConnection(. . .);
```

```
        // Проверка соединения
      }

   public int doAfterBody(){
      BodyContent bc = getBodyContent();
         if (bc == null) return SKIP_BODY;

         String query = bc.getString();

      try{
         Statement st = conn.createStatement();
            rs = st.executeQuery(query);

            // Обработка результата запроса

            JspWriter out = bc.getEnclosingWriter();

         out.print("Вывод результатов");
      }catch(Exception e){
         System.err.println(e);
      }
      return SKIP_BODY;
      }

   public int doEndTag(){
      conn = null;
      return EVAL_PAGE;
   }
}
```

Обработка взаимодействующих тегов

Часто пользовательские теги, расположенные на одной странице JSP, должны взаимодействовать друг с другом. Например, тело тега может содержать другие, вложенные, теги JSP. Они могут быть самостоятельными по отношению к внешнему тегу или зависеть от него. Так, например, в языке HTML тег `<tr>` может появиться только внутри тега `<table>`. При этом атрибуты тега `<table>` могут использоваться в теге `<tr>`, а также могут быть переопределены внутри него. Таких примеров много в языке XML.

В языке JSP тоже могут появиться теги, зависящие друг от друга. Например, мы можем определить тег

```
<sdo:connection source="источник данных" >
```

устанавливающий соединение с базой данных, указанной в атрибуте `source`. Внутри этого тега допустимо обращение к базе данных, например:

```
<sdo:connection source="SDO" >
   <sdo:query >
      SELECT * FROM students
   </sdo:query>
```

```
   <sdo:insert>
      INSERT INTO students (name) VALUES ('Иванов')
   </sdo:insert>
</sdo:connection>
```

Конечно, вложенные теги можно реализовать вложенными классами-обработчиками или расширениями внешних классов. Но в классах-обработчиках пользовательских тегов есть свои средства. Внешний и вложенный теги реализуются отдельными классами, расширяющими классы `TagSupport` или `BodyTagSupport`. Описания тегов в TLD-файле тоже не вложены, они записываются независимо друг от друга, например:

```
<tag>
   <name>connection</name>
   <tag-class>sdotags.ConnectionTag</tag-class>
</tag>
<tag>
   <name>query</name>
   <tag-class>sdotags.QueryTag</tag-class>
</tag>
<tag>
   <name>insert</name>
   <tag-class>sdotags.InsertTag</tag-class>
</tag>
```

Для связи обработчиков тегов используются их методы. Сначала обработчик вложенного тега задает себе внешний, "родительский" тег `parent` методом

```
public void setParent(Tag parent);
```

Затем обработчик вложенного тега может обратиться к обработчику внешнего тега методом

```
public Tag getParent();
```

Более общий метод

```
public static final Tag findAncestorWithClass(Tag from, Class class);
```

позволяет обратиться к обработчику тега, не обязательно непосредственно окружающего данный тег. Первый аргумент этого метода чаще всего просто `this`, а второй аргумент должен реализовать интерфейс `Tag` или его расширения.

Итак, все вложенные теги могут обратиться к полям и методам внешних тегов и взаимодействовать с их помощью.

Допустим, класс `ConnectionTag`, реализующий пользовательский тег `connection`, устанавливает соединение с источником данных, как показано в листинге 27.9.

Листинг 27.9. Реализация тега соединения с базой данных

```
  public class ConnectionTag extends TagSupport{

    private Connection conn;

 public Connection getConnection(){
```

```
    return conn;
     }
  public int doStartTag(){
       Connection conn = DriverManager.getConnection(url, user, password);
    if (conn == null) return SKIP_BODY;
    return EVAL_BODY_INCLUDE;
     }
  public int doEndTag(){
    conn = null;
    return EVAL_BODY;
  }
}
```

Класс `QueryTag`, реализующий тег `query`, может воспользоваться объектом `conn`, как показано в листинге 27.10.

Листинг 27.10. Реализация связанного тега

```
 public class QueryTag extends BodyTagSupport{
   private ConnectionTag parent;
   public int doStartTag(){
    parent = (ConnectionTag)findAncestorWithClass(this,
           ConnectionTag.class);
    if (parent == null) return SKIP_BODY;
    return EVAL_BODY_INCLUDE;
     }
  public int doAfterBody(){
       Connection conn = parent.getConnection();
       // Прочие действия
    return SKIP_BODY;
  }
}
```

Другой способ сделать объект `obj` доступным для нескольких тегов — указать его в виде атрибута какого-нибудь контекста. Это выполняется методом

```
public void setAttribute(String name, Object obj);
```

класса `PageContext`. Контекст этого атрибута — страница, на которой он определен. Если область действия атрибута надо расширить, то используется метод

```
public void setAttribute(String name, Object obj, int scope);
```

того же класса. Третий аргумент данного метода задает область действия атрибута — это одна из констант: PAGE_SCOPE, APPLICATION_SCOPE, REQUEST_SCOPE, SESSION_SCOPE.

Значение атрибута всегда можно получить методами

```
public Object getAttribute(String name, int scope);
public Object getAttribute(String name);
```

класса PageContext.

Для удобства работы с атрибутом обработчик тега может создать переменную, известную в области действия атрибута и доступную для других тегов в этой области (scripting variable). Она будет содержать ссылку на созданный атрибут. Переменную можно определить не только по атрибуту контекста, но и по атрибуту открывающего тега. Для определения переменной есть два способа.

Первый способ — указать в TLD-файл элемент <variable>, описывающий переменную. В этот элемент вкладывается хотя бы один из двух элементов:

- <name-given> — постоянное имя переменной;
- <name-from-attribute> — имя атрибута, значение которого будет именем переменной.

Остальные вложенные элементы необязательны:

- <variable-class> — класс переменной, по умолчанию String;
- <declare> — создавать ли новый объект при обращении к переменной, по умолчанию true;
- <scope> — область действия переменной, одна из трех констант:
  - NESTED — между открывающим и закрывающим тегами, т. е. в методах doStartTag(), doInitBody(), doEndBody(); принимается по умолчанию;
  - AT_BEGIN — от открывающего тега до конца страницы;
  - AT_END — после закрывающего тега до конца страницы.

Например:

```
<tag>
   <name>tagname</name>
   <tag-class>sdotags.SomeTag</tag-class>
   <variable>
      <name-given>varname</name-given>
      <variable-class>java.lang.String</variable-class>
      <declare>true</declare>
      <scope>AT_END</scope>
   </variable>
</tag>
```

Второй способ — определить объект, содержащий ту же самую информацию, что и элемент <variable>. Данный объект создается как расширение класса TagExtraInfo.

Вот как выглядит это расширение для примера, приведенного ранее:

```
public class MyTei extends TagExtraInfo{
  public VariableInfo[] getVariableInfo(TagData data){
    return new VariableInfo[]{
      new VariableInfo(data.getAttributeString("id"),
                       "java.lang.String", true, VariableInfo.AT_END)
    };
  }
}
```

Класс `MyTei` описывается в TLD-файле в элементе `<tei-class>` следующим образом:

```
<tag>
   <name>tagname</name>
   <tag-class>sdotags.SomeTag</tag-class>
   <tei-class>sdotags.MyTei</tei-class>
   <bodycontent>JSP</bodycontent>
</tag>
```

У этого способа больше возможностей, чем у элемента `<variable>`. Класс `TagExtraInfo` имеет в своем распоряжении экземпляр класса `TagData`, содержащий сведения о теге, собранные из TLD-файла. Для удобства использования этого экземпляра в классе `TagExtraInfo` есть логический метод

```
public boolean isValid(TagData info);
```

Его можно применять для проверки атрибутов тега на этапе компиляции страницы JSP.

Обработка исключений в пользовательских тегах

Ранее уже говорилось о том, что обработку исключения, возникшего на странице JSP, можно перенести на другую страницу, указанную атрибутом `errorPage` тега `<%@ page %>`. В пользовательских тегах можно облегчить обработку исключения, реализовав интерфейс `TryCatchFinally`.

Интерфейс `TryCatchFinally` описывает всего два метода:

```
public void doCatch(Throwable thr);
public void doFinally();
```

Метод `doCatch()` вызывается автоматически контейнером при возникновении исключения в одном из методов `doStartTag()`, `doInitBody()`, `doAfterTag()`, `doEndTag()` обработчика, реализующего интерфейс `TryCatchFinally`. Методу `doCatch()` передается созданный объект-исключение. Метод `doCatch()` сам может выбросить исключение.

Метод `doFinally()` выполняется во всех случаях после метода `doEndTag()`. Он уже не может выбрасывать исключения.

Например, при реализации тега <sdo:connection> допускается использование методов интерфейса TryCatchFinally для отката транзакции следующим образом:

```
 public class ConnectionTag extends TagSupport implements TryCatchFinally{

   private Connection conn;

   // Прочие поля и методы класса

 public void doCatch(Throwable t) throws Throwable{
   conn.rollback();
   throw t;
    }

 public void doFinally(){
   conn.close();
 }
}
```

Обработка тегов средствами JSP

Уже упоминавшаяся тенденция к изгнанию со страниц JSP всякого чужеродного кода, даже кода Java привела к тому, что для обработки пользовательского тега вы можете вместо класса Java написать страницу JSP, описав на ней действия тега. Имя файла с такой страницей-обработчиком JSP получает расширение tag, а если она оформлена как документ XML, то расширение tagx. Файл, содержащий отдельный фрагмент страницы-обработчика, получает имя с расширением tagf.

Например, тег <sdo:info />, действие которого мы показали в листинге 27.5, можно обработать следующей, очень простой, страницей JSP, записанной в tag-файл с именем info.tag:

```
<jsp:root
      xmlns:jsp="http://java.sun.com/JSP/Page"
      version="2.0" >
   Библиотека тегов СДО.
</jsp:root>
```

Поскольку элемент <jsp:root> необязателен, весь файл info.tag может состоять только из одной строчки:

```
Библиотека тегов СДО.
```

На странице-обработчике пользовательских тегов можно записывать любые теги JSP, кроме директивы page. Вместо нее указывается специально введенная для этой цели в язык JSP директива tag. Например:

```
<%@ tag body-content="scriptless">
```

Второе отличие заключается в том, что в директиве <%@ taglib %> следует записывать не атрибут uri, а атрибут tagdir, в котором указывается каталог с tag-файлами, например:

```
<%@ taglib tagdir="/WEB-INF/tags" prefix="sdo" %>
```

Атрибуты обрабатываемого пользовательского тега описываются еще одной специально введенной директивой `attribute`, аналогичной элементу `<attribute>` TLD-файла. Пусть, например, на странице JSP применяется пользовательский тег с двумя атрибутами:

```
<sdo:reg name="Иванов" age="25"/>
```

В файле reg.tag, обрабатывающем этот тег, можно написать

```
<%@ attribute name="name"
   required="true"
   fragment="false"
   rtexprvalue="false"
%>
<%@ attribute name="age"
   type="java.lang.Integer"
%>
```

Если тип `type` атрибута не указан, то по умолчанию он считается строкой класса `java.lang.String`.

Типом атрибута может быть фрагмент JSP, т. е. объект класса `javax.servlet.jsp.tagext.JspFragment`. В таком случае атрибуты `type` и `rtexprvalue` не указываются, а атрибут `fragment="true"`.

После этого описания значения атрибутов, введенные в обрабатываемый тег, в примере они равны `"Иванов"` и `"25"`, можно использовать в файле reg.tag как `${name}` и `${age}`.

Третья, специально введенная директива `variable`, аналогичная элементу `<variable>` TLD-файла, позволяет описать переменную на странице-обработчике. Например:

```
<%@ variable name-given="x"
   variable-class="java.lang.Integer"
   scope="NESTED"
   declare="true"
%>
```

Область действия переменной определяется атрибутом `scope`, таким же, как описанный в *разд. "Обработка взаимодействующих тегов" данной главы* одноименный элемент TLD-файла.

Значения атрибутов, являющихся фрагментами страниц JSP, и тело пользовательского тега не вычисляются Web-контейнером, а передаются tag-файлу.

Для того чтобы заставить Web-контейнер выполнить значение атрибута, являющегося фрагментом страницы JSP, в tag-файле нужно использовать стандартный тег `<jsp:invoke>`. Пусть, например, атрибут `price` описан следующим образом:

```
<%@ attribute name="price"
              fragment="true"
%>
```

Для вычисления его значения Web-контейнером, в tag-файле следует написать:

```
<jsp:invoke fragment="price"/>
```

Если же надо заставить Web-контейнер выполнить не фрагмент, а все тело пользовательского тега, то в tag-файле записывается стандартный тег `<jsp:doBody/>`.

Tag-файлы не требуют никакого TLD-описания, если файл, содержащий страницу-обработчик пользовательского тега, хранится в каталоге WEB-INF/tags/ или в его подкаталогах. Другое место хранения tag-файлов должно быть описано в TLD-файле элементом `<tag-file>`. Например:

```
<taglib>
   <tag-file>
      <name>info</name>
      <path>/WEB-INF/sdo/tags/info.tag</path>
   </tag-file>
</taglib>
```

Если Web-приложение упаковано в JAR-архив, то tag-файлы должны храниться в каталоге META_INF/tags/ архива или в его подкаталогах.

Стандартные библиотеки тегов JSTL

Несмотря на недолгую историю языка JSP, его возможность создания пользовательских тегов была использована многими фирмами и отдельными разработчиками. Уже создано множество библиотек пользовательских тегов. Их обзор можно посмотреть, например, на сайте **http://www.servletsuite.com/jsp.htm**. В рамках уже не раз упоминавшегося проекта Jakarta создана мощная и широко применяемая библиотека пользовательских тегов Struts, которую можно скопировать с сайта **http://struts.apache.org/**. К сожалению, рамки нашей книги не позволяют дать ее описание.

Корпорация Sun Microsystems создает стандартные библиотеки пользовательских тегов, носящие общее название JSTL (JSP Standard Tag Library). Их реализации составляют пакет `javax.servlet.jsp.jstl` и его подпакеты. Справку о том, где найти самую последнюю реализацию, можно получить по адресу **http://java.sun.com/products/jsp/jstl/**. Кроме того, библиотеки JSTL входят в стандартную поставку сервера приложений Sun Java System Application Server, входящего в состав Java EE SDK. Всю библиотеку JSTL составляют два JAR-архива: `jstl-1.2.jar` и `standard.jar`. Номер версии JSTL, конечно, может быть другим.

В пакет JSTL входят пять библиотек пользовательских тегов: core, xml, fmt, sql и fn.

Библиотека core

Библиотека core описывается на странице JSP тегом

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

В нее входят теги, создающие подобие языка программирования. Они призваны заменить операторы Java, тем самым устраняя скриптлеты со страниц JSP.

Тег `<c:out>` выводит значение своего обязательного атрибута `value` в выходной поток `out`. Если оно пусто, то будет выведено значение атрибута `default`, если, конечно, этот атрибут написан.

Например:

```
Здравствуйте, уважаемый <c:out value="${name}"
   default="господин" />!
```

Тег <c:set> устанавливает значение переменной, имя которой задается атрибутом var, а область действия — атрибутом scope. Например:

```
<c:set var="name" scope="session" value="${param.name}"/>
```

или, что то же самое:

```
<c:set var="name" scope="session">
   ${param.name}
</c:set>
```

Тег <c:remove> удаляет ранее определенные переменные, например:

```
<c:remove var="name" scope="session"/>
```

Тег <c:url> тоже устанавливает переменную, но ее значение должно быть адресом, записанным в форме URL. Например:

```
<c:url var="bhv" value="http://www.bhv.ru"/>
```

или, что то же самое:

```
<c:url var="name">
   http://www.bhv.ru
</c:url>
```

После этого переменную bhv можно использовать для создания сеанса связи с пользователем, как это делалось в предыдущей главе, или записывать в гиперссылках HTML:

```
<a href="${bhv}">Издательство</a>
```

К создаваемому адресу URL можно добавить параметры вложенным тегом <c:param>, например:

```
<c:url var="bhv" value="http://www.bhv.ru">
   <c:param name="id" value="${book.id}"/>
   <c:param name="author" value="${book.author}"/>
</c:url>
```

Тег <c:if> проверяет условие, записанное в его атрибуте test логическим выражением. Если условие истинно, то выполняется тело тега. Например:

```
<c:if test="${age <= 0}">
   Уважаемый ${name}! Укажите, пожалуйста, свой возраст.
</c:if>
```

Как видите, этот тег не реализует полностью условие "if-then-else". Такое разветвление можно организовать тегом <c:choose>.

Тег <c:choose> со вложенными в него элементами <c:when> и <c:otherwise> реализует выбор одного из вариантов.

Например:

```
<c:choose>
   <c:when test="${balance < 0}" >
        На вашем счету отрицательный остаток.
   </c:when>
   <c:when test="${balance == 0}" >
        На вашем счету нулевой остаток.
   </c:when>
   <c:when test="${balance > 0}" >
        На вашем счету положительный остаток.
   </c:when>
   <c:otherwise>
        Нет сведений о вашем остатке.
   </c:otherwise>
</c:choose>
```

В примере листинга 27.11 показано, как можно организовать разветвление с помощью тега `<c:choose>`. В нем проверяется, известны ли уже сведения о клиенте. Если они известны, клиенту посылается приветствие, если нет — форма для ввода имени и пароля.

Листинг 27.11. Ответ, сформированный с помощью JSTL

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
  <head><title>Проверка ввода имени</title></head>
  <body>
    <c:choose>
      <c:when test="${user != null}">
        Здравствуйте, ${user.name}!
      </c:when>
      <c:otherwise>
        <form method="POST" action="/sdo/jsp/user.jsp">
          Имя:    <input type="text" name="name">
          Пароль: <input type="password" name="pswd">
          <input type="submit" name="ok" value="Отправить">
        </form>
      </c:otherwise>
    </c:choose>
  </body>
</html>
```

У тега `<c:forEach>` две формы. Первая создает цикл, пробегающий коллекцию типа `Collection` или `Map`, указанную атрибутом `items`. Ссылка на текущий элемент коллекции заносится в переменную, определенную атрибутом `var`. Ее можно использовать в теле тега.

Например:

```
<c:forEach var="item" items="${sessionScope.cart.items}">
   <td>
       ${item.quantity}
   </td>
</c:forEach>
```

Вторая форма тега <c:forEach> создает цикл с перечислением, в котором переменная цикла, определяемая атрибутом var, пробегает от начального значения, задаваемого значением атрибута begin, до конечного значения, задаваемого значением атрибута end, с шагом — значением атрибута step. Например:

```
<c:forEach var="k" begin="0" step="1" end="${n}" >
   <td>
       ${k}-й столбец
   </td>
</c:forEach>
```

Тег <c:forTokens> разбивает строку символов, заданную атрибутом items на слова подобно классу StringTokenizer, рассмотренному в *главе 5*. Разделители слов перечисляются в атрибуте delims. Текущее слово заносится в переменную, определенную атрибутом var. Например:

```
<c:forTokens var="word" items="${text}" delims=" \n\r\t:;,.?!">
   <c:out value="${word}"/>
</c:forTokens>
```

Тег <c:import> включает на страницу JSP ресурсы по их адресу URL. Например:

```
<c:import url="/html/intro.html"
          var="intro"
          scope="session"
          charEncoding="windows-1251"
/>
```

Переменную, определенную атрибутом var, можно использовать в своей области действия, определенной атрибутом scope (по умолчанию, page). Атрибут charEncoding показывает кодировку символов включаемого ресурса. По умолчанию это кодировка ISO 8859-1, которая плохо подходит для кириллицы.

Тег <c:redirect> прекращает обработку страницы и посылает HTTP-ответ redirect клиенту с указанием адреса, записанного в атрибуте url. Браузер клиента сделает новый запрос по этому адресу. В соответствующем сервлете метод doEndTag() возвращает константу SKIP_PAGE. Например:

```
<c:redirect url="/books/list.html" context="/lib" />
```

Необязательный атрибут context устанавливает контекст для нового запроса.

В тег <c:redirect>, как и в тег <c:url>, можно вложить теги <c:param>, задающие параметры нового запроса.

Библиотека xml

Библиотека xml, описываемая тегом

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/xml" prefix="x" %>
```

содержит теги `<x:out>`, `<x:set>`, `<x:forEach>`, `<x:if>`, `<x:choose>`, `<x:when>`, `<x:otherwise>`, аналогичные соответствующим тегам библиотеки core, но выбирающие нужный элемент XML из интерпретируемого документа атрибутом `select`, а также теги `<x:parse>` и `<x:transform>`, интерпретирующие и преобразующие документ XML.

Работа с библиотекой xml основана на адресации элементов документа XML средствами языка XPath, что выходит за рамки нашей книги.

Библиотека fmt

Библиотека fmt содержит теги, помогающие в интернационализации страниц JSP. Она описывается так:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
```

В нее входят теги `<fmt:setLocale>`, `<fmt:timeZone>`, `<fmt:formatDate>`, `<fmt:parseDate>`, `<fmt:formatNumber>`, `<fmt:parseNumber>` и другие теги, делающие локальные установки.

Например, тег

```
<fmt:formatNumber var="formattedAmount" pattern="0.00" value="${amount}" />
```

запишет в переменную `formattedAmount` типа String количество `amount` с двумя цифрами в дробной части, а тег

```
<fmt:formatDate var="ruDate" pattern="dd.MM.yyyy" value="${today}" />
```

запишет в переменную `ruDate` типа String дату `today` в виде 08.12.2007.

Теги

```
<fmt:parseNumber var="n" pattern="0.00" value="${amount}" />
```

```
<fmt:formatDate var="d" pattern="dd.MM.yyyy" value="${today}" />
```

выполняют обратное преобразование строки символов, заданной атрибутом `value`, в объекты типа Number и Date соответственно, записанные в переменные `n` и `d` в соответствии с шаблоном `pattern`.

Эти теги реализованы классами `DecimalFormat` и `SimpleDateFormat` из пакета java.text, и преобразуют данные по правилам этих классов, которые можно посмотреть в документации Java SE.

Библиотека sql

Четвертая библиотека, sql, описываемая тегом

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>
```

содержит теги связи и работы с базами данных: <sql:setDataSource>, <sql:query>, <sql:update> и <sql:transaction>. Их действие построено на JDBC и работа с ними очень похожа на работу с базами данных через JDBC, что можно понять из следующего примера:

```
<sql:setDataSource driver="oracle.jdbc.driver.OracleDriver"
   url="jdbc:oracle:thin:@localhost:1521:ORCL"
   user="scott"
   password="tiger"
/>
<sql:query var="depts" >
   select * from DEPT
</sql:query>
<sql:transaction>
   <sql:update>
       insert into DEPT values(50, 'XXX', 'YYY')
   </sql:update>
</sql:transaction>
```

Библиотека fn

Пятая библиотека, fn, содержит функции для обработки строк. Она описывается тегом

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
```

и содержит теги <fn:substring>, <fn:substringBefore>, <fn:substringAfter>, <fn:indexOf>, <fn:toUpperCase> <fn:toLowerCase>, <fn:startsWith>, <fn:endsWith>, <fn:contains>, <fn:containsIgnoreCase>, <fn:trim>, <fn:replace>, <fn:split>, <fn:join>, <fn:escapeXml>. Вы, несомненно, узнаете в названиях этих тегов привычные методы обработки строк, которые мы изучили в *главе 5*.

Кроме тегов обработки строк, библиотека fn содержит функцию <fn:length>, вычисляющую как длину строки, так и длину коллекции, переданной ей в качестве аргумента. Например:

```
<c:if test="${fn:length(param.username) > 0}" >
   <%@include file="response.jsp" %>
</c:if>
```

Frameworks

Приступая к разработке Web-приложения, каждая команда решает вопрос о его архитектуре. Чаще всего ответ находится в схеме MVC (Model-View-Controller) *(см. главу 3)*. Один или несколько сервлетов, принимающих и обрабатывающих запросы клиента, составляют *Контроллер*. Подготовка ответа, связь с базой данных, отбор информации, все то, что называется бизнес-логикой или бизнес-процессами, выполняется классами Java, образующими *Модель*. Страницы HTML и JSP, заполненные информацией, полученной от *Модели*, составляют *Вид*.

В ходе обсуждения и реализации проекта каждая из трех частей схемы MVC конкретизируется, описывается интерфейсами и абстрактными классами. Иногда на этом этапе получается наполовину реализованная конструктивная схема, подходящая для выполнения целого класса типовых проектов. Для завершения проекта Web-приложения остается реализовать несколько интерфейсов, дописать при необходимости свои собственные классы и определить под свой проект параметры приложения, записав их в конфигурационные файлы.

Наиболее удачные из таких шаблонов готового приложения становятся достоянием всего сообщества программистов, активно обсуждаются, улучшаются, модернизируются и получают широкое распространение. Такие общепризнанные шаблоны, или, по-другому, каркасы программного продукта получили название *Frameworks*.

В технологии Java уже есть десятки таких каркасов, в их числе JSF, Seam, Struts, Facelets, Tales, Shale, Spring, Velocity, Tapestry, Jena, Stripes, Trails, RIFE, WebWork и множество других. В состав Java EE SDK входит каркас JSF. Познакомимся с ним подробнее.

JavaServer Faces

Framework под названием JSF (JavaServer Faces) вырос из простой библиотеки тегов, расширяющей возможности тегов HTML. Он входит в стандартную поставку Java EE, хотя всеми возможностями JSF можно воспользоваться, и не имея у себя на компьютере Java EE. Есть много других реализаций интерфейсов JSF.

Эталонную реализацию, которую предоставляет проект GlassFish, можно загрузить с сайта **http://javaserverfaces.dev.java.net/**. Из других реализаций следует выделить очень популярный продукт Apache MyFaces, **http://myfaces.apache.org/**.

Все необходимое для работы с JSF собрано в одном архивном файле. Достаточно распаковать его в какой-нибудь каталог, и JSF готов к работе. В этом каталоге вы найдете подкаталоги с документацией и подкаталог с подробными примерами. Основу реализации JSF составляют два JAR-файла `jsf-api.jar` и `jsf-impl.jar` из подкаталога lib, которые надо скопировать в ваше Web-приложение или в ваш сервер приложений или записать пути к ним в переменную `CLASSPATH`. Для работы JSF понадобится библиотека JSTL, проследите за тем, чтобы у вашего Web-приложения был доступ к ее JAR-архиву, например файлу `jstl-1.2.jar`.

Каркас JSF построен по схеме MVC (Model-View-Controller), которую мы обсуждали в *главе 3*.

Роль Контроллера в JSF играет сервлет под названием `FacesServlet`. Как и все сервлеты, он должен быть описан в конфигурационном файле web.xml. Это описание выглядит так:

```
<web-app>
 . . . . .
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    </servlet>
     . . . . .
```

```
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.faces</url-pattern>
    </servlet-mapping>
 . . . . .
</web-app>
```

После этого описания мы можем обращаться к JSF с запросами вида

`http://localhost:8080/MyAppl/index.faces`,
`http://localhost:8080/MyAppl/login.faces`

Часто элемент `<servlet-mapping>` записывают по-другому:

```
<servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>/faces/*</url-pattern>
 </servlet-mapping>
```

После такого описания запросы к JSF должны выглядеть так:

`http://localhost:8080/MyAppl/faces/index.jsp`,
`http://localhost:8080/MyAppl/faces/login.jsp`

И в первом, и во втором случае запрашиваются файлы index.jsp и login.jsp. Расширение имени файла faces в запросах первого вида сделано только для JSF, на сервере файлов с такими именами нет. Увидев в запросе имя index.faces, JSF будет искать файл index.jsp.

Для организации Вида в JSF разрабатываются библиотеки тегов. В состав JSF сейчас входят две библиотеки: core и html. Они обычно описываются на странице JSP директивами

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
```

Все теги JSF вкладываются в элемент `<f:view>`. Компилятор JSF просматривает и компилирует только теги, вложенные в `<f:view>`. Теги включаемого файла должны быть вложены в элемент `<f:subview>`. Это можно сделать так, как показано в листинге 27.12.

Чаще всего на странице JSP формируется пользовательский интерфейс, который будет отображен браузером клиента. В нем размещаются формы с полями ввода, списками выбора, кнопками и прочими компонентами, текст с гиперссылками, панели, вкладки, таблицы.

Создание классической страницы HTML уже не удовлетворяет ни разработчиков, ни клиентов. Набор тегов HTML невелик и фиксирован, их возможности весьма ограничены. Уже давно придумываются разные способы оживления страниц HTML: таблицы стилей CSS, динамический HTML, апплеты. Библиотека тегов html в первую очередь призвана усилить возможности тегов HTML.

Для каждого тега HTML в JSF есть соответствующий тег, обладающий дополнительными возможностями. В листинге 27.12 показана форма с полями ввода имени и пароля и кнопкой типа Submit.

Листинг 27.12. Страница JSP с тегами библиотеки JSF

```
<html><head>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
</head><body>
   <f:view>

   <f:subview id="myNestedPage">
       <jsp:include page="theNestedPage.jsp" />
      </f:subview>

   <h:form>
      <h:inputText id="name" size="50"
                   value="#{cashier.name}"
                   required="true">
          <f:valueChangeListener type="listeners.NameChanged" />
      </h:inputText>
      <h:inputSecret id="pswd" size="50"
                   value="#{cashier.pswd}"
                   required="true">
          <f:valueChangeListener type="listeners.NameChanged" />
        </h:inputSecret>

      <h:commandButton type="submit" value="Отправить"
                       action="#{cashier.submit}"/>

      </h:form>

</f:view>
</body></html>
```

В библиотеке html есть более двух десятков тегов, соответствующих графическим компонентам пользовательского интерфейса. Кроме того, разработчик может легко создать свои, *пользовательские*, компоненты (custom components), расширив классы JSF. Уже создано много библиотек тегов для JSF, свободно распространяемых или коммерческих.

Главная особенность библиотеки html заключается в том, что ее составляют не просто теги, а целые компоненты. Компоненты JSF реагируют на события мыши и клавиатуры, которые могут быть обработаны обычными средствами JavaScript или специальными средствами JSF. Для этого введены теги-обработчики событий. Один такой тег, `<v:valueChangeListener>`, показан в листинге 27.12. Надо сказать, что разработчики тегов библиотеки html сознательно ориентировались на компоненты Swing, стараясь, по мере возможности, наделить теги такими же свойствами. В частности, компоненты JSF можно размещать на форме, пользуясь подходящим IDE. Это "умеет" делать, например, Java Studio Creator.

Форма, записанная в листинге 27.12, посылает на сервер имя и пароль, связанные с полями `cashier.name` и `cashier.pswd` атрибутом `value`. Что это за поля и какому объекту `cashier` они принадлежат?

Данные, полученные от HTML-формы, будут храниться в объекте, класс которого должен написать разработчик Web-приложения. Этот класс оформляется как JavaBean, чтобы JSF мог заполнять и читать его поля методами доступа. Он будет частью Модели в схеме MVC. Класс для хранения имени и пароля, полученных от формы листинга 27.12, показан в листинге 27.13.

Листинг 27.13. Класс с данными HTML-формы

```
package myjsf;
public class Cashier{
    private String name;
    private String pswd;
    public String getName(){ return name; }
    public void setName(String name){ this.name = name; }
    public String getPswd(){ return pswd; }
    public void setPswd(String pswd){ this.pswd = pswd; }
    public String submit(){
       if ("Cashier".equalsIgnoreCase(name) && "rT34?x_D".equals(pswd))
          return "success";
       else
          return "failure";
    }
}
```

Как видите, в этом же классе записан метод обработки щелчка по кнопке **Отправить**.

Теперь надо каким-то образом указать JSF этот класс. Сведения о нем записываются в конфигурационный XML-файл faces-config.xml, который должен храниться в каталоге WEB-INF вместе с файлом web.xml. Это файл для нашего примера записан в листинге 27.14.

Листинг 27.14. Конфигурационный файл JSF

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
              http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
              version="1.2">
    <managed-bean>
        <description>
            Класс-обработчик регистрации кассира
        </description>
        <managed-bean-name>cashier</managed-bean-name>
        <managed-bean-class>myjsf.Cashier</managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
    </managed-bean>
    <navigation-rule>
```

```
        <from-view-id>/index.jsp</from-view-id>
        <navigation-case>
            <description>
                После удачной проверки переход на страницу welcome.jsp
            </description>
            <from-outcome>success</from-outcome>
            <to-view-id>/welcome.jsp</to-view-id>
        </navigation-case>
        <navigation-case>
            <description>
                После неудачной проверки возврат на страницу index.jsp
            </description>
            <from-outcome>failure</from-outcome>
            <to-view-id>/index.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>
```

Элемент `<managed-bean>` связывает имя класса `cashier`, используемое в листинге 27.12, с полным именем класса `myjsf.Cashier`. Элемент `<navigation-rule>` показывает, откуда (`<from-view-id>`) и куда (`<to-view-id>`) следует перейти после обработки данных методом `submit()` объекта `cashier`, а также при каком событии (`<from-outcome>`) сделать тот или иной переход.

Для полноты осталось написать файл welcome.jsp. Он может выглядеть так, как написано в листинге 27.15.

Листинг 27.15. Страница приветствия с тегами библиотеки JSF

```
<html><head>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
</head><body>
   <f:view>

        Здравствуйте, ${cashier.name}!

</f:view>
</body></html>
```

Обзор всех возможностей JSF выходит за рамки нашей книги. Вы можете ознакомиться с ними на сайте разработчиков JSF **http://jsfcentral.com/**. Множество статей и учебников по JSF собрано на сайте **http://www.jsftutorials.net/**. Там вы можете найти дальнейшие ссылки.

Вопросы для самопроверки

1. Для чего придуман язык JSP?
2. Можно ли смешивать код JSP и код HTML на одной странице?

3. Можно ли записывать код Java на страницах JSP?
4. Можно ли включать в страницу JSP другие файлы?
5. Можно ли передавать управление из страницы JSP другим ресурсам?
6. Можно ли расширить набор стандартных тегов JSP?
7. Можно ли написать несколько классов Java, по-разному обрабатывающих один и тот же пользовательский тег?
8. Можно ли обработать пользовательский тег не классом Java, а страницей JSP?
9. Как подключить библиотеку пользовательских тегов к странице JSP?



Глава 28

Связь Java с технологией XML

В развитии Web-технологии огромную роль сыграл язык HTML (HyperText Markup Language) — язык разметки гипертекста. Любой человек, совсем не знакомый с программированием, мог за полчаса понять принцип разметки текста и за пару дней изучить теги HTML. Пользуясь простейшим текстовым редактором, он мог написать свою страничку HTML, тут же посмотреть ее в своем браузере, испытать чувство глубокого удовлетворения и гордо выставить в Интернете свой шедевр.

Замечательно! Не надо месяцами изучать запутанные языки программирования, явно предназначенные только для яйцеголовых "ботаников", осваивать сложные алгоритмы, возиться с компиляторами и отладчиками, размножать свое творение на дисках. Очень скоро появились текстовые редакторы, размечающие обычный "плоский" текст тегами HTML. Разработчику оставалось только поправлять готовую страницу HTML, созданную таким редактором.

Простота языка HTML привела к взрывному росту числа сайтов, пользователей Интернета и авторов многочисленных Web-страничек. Обычные пользователи компьютеров ощутили себя творцами, получили возможность заявить о себе, высказать собственные мысли и чувства, найти в Интернете единомышленников.

Ограниченные возможности языка HTML быстро перестали удовлетворять поднаторевших разработчиков, почувствовавших себя "профи". Набор тегов языка HTML строго определен и должен одинаково пониматься всеми браузерами. Нельзя ввести дополнительные теги или указать браузеру, как следует отображать на экране содержимое того или иного тега. Введение *таблиц стилей* CSS (Cascading Style Sheet) и *включений на стороне сервера* SSI (Server Side Include) лишь ненадолго уменьшило недовольство разработчиков. Профессионалу всегда не хватает средств разработки, он постоянно испытывает потребность добавить к ним какое-то свое средство, позволяющее реализовать все его фантазии.

Такая возможность есть. Еще в 1986 году стал стандартом язык создания языков разметки SGML (Standard Generalized Markup Language), с помощью которого и был создан язык HTML. Основная особенность языка SGML заключается в том, что он позволяет сформировать новый язык разметок, определив его набор тегов. Каждый конкретный набор тегов, созданный по правилам SGML, снабжается описанием DTD (Document Type Definition) — *определением типа документа*, разъясняющим связь тегов между собой и правила их применения. Специальная программа — драйвер принтера или SGML-браузер — руководствуется этим описанием для печати или отображения документа на экране дисплея.

В это же время выявилась еще одна, самая важная область применения языков разметки — поиск и выборка информации. В настоящее время подавляющее большинство информации хранится в реляционных базах данных. Они удобны для хранения и поиска сведений, представимых в виде таблиц: анкет, ведомостей, списков и т. п., но неудобны для хранения различных документов, планов, отчетов, статей, книг, не представимых в виде таблицы. Тегами языка разметки можно задать структурную, а не визуальную разметку документа, разбить документ на главы, параграфы и абзацы или на какие-то другие элементы, выделить важные для поиска участки документа. Легко написать программу, анализирующую размеченный такими тегами документ и извлекающую из него нужную информацию.

Язык SGML оказался слишком сложным, требующим тщательного и объемистого описания элементов создаваемого с его помощью языка. Он применяется только в крупных проектах, например для создания единой системы документооборота крупной фирмы. Скажем, man-страницы Solaris Operational Environment написаны на специально сделанной реализации языка SGML.

Золотой серединой между языками SGML и HTML стал язык XML (eXtensible Markup Language) — расширяемый язык разметок. Это подмножество языка SGML, избавленное от излишней сложности, но позволяющее разработчику Web-страниц создавать собственные теги. Язык XML достаточно широк, чтобы можно было создать все нужные теги, и достаточно прост, чтобы можно было быстро их описать.

Организуя описание документа на языке XML, надо прежде всего продумать структуру документа. Приведем пример. Пусть мы решили, наконец, упорядочить записную книжку с адресами и телефонами. В ней записаны фамилии, имена и отчества родственников, сослуживцев и знакомых, дни их рождения, адреса, состоящие из почтового индекса, города, улицы, дома и квартиры, и телефоны, если они есть: рабочие и домашние. Мы придумываем теги для выделения каждого из этих элементов, продумываем вложенность элементов и получаем структуру, показанную в листинге 28.1.

Листинг 28.1. Пример XML-документа

```
<?xml version="1.0" encoding="Windows-1251"?>
   <!DOCTYPE notebook SYSTEM "ntb.dtd">

   <notebook>

     <person>

    <name>
      <first-name>Иван</first-name>
      <second-name>Петрович</second-name>
      <surname>Сидоров</surname>
       </name>

       <birthday>25.03.1977</birthday>

    <address>
      <street>Садовая, 23-15</street>
      <city>Урюпинск</city>
```

```
      <zip>123456</zip>
       </address>

    <phone-list>
      <work-phone>2654321</work-phone>
      <work-phone>2654023</work-phone>
      <home-phone>3456781</home-phone>
       </phone-list>

     </person>

     <person>

    <name>
      <first-name>Мария</first-name>
      <second-name>Петровна</second-name>
      <surname>Сидорова</surname>
       </name>

       <birthday>17.05.1969</birthday>

    <address>
      <street>Ягодная, 17</street>
      <city>Жмеринка</city>
      <zip>234561</zip>
       </address>

    <phone-list>
      <home-phone>2334455</home-phone>
       </phone-list>

     </person>

</notebook>
```

Документ XML начинается с необязательного пролога, состоящего из двух частей.

В первой части пролога — *объявлении XML* (XML declaration), — записанной в первой строке листинга 28.1, указывается версия языка XML, необязательная кодировка документа и отмечается, зависит ли этот документ от других документов XML (`standalone="yes"/"no"`). По умолчанию принимается кодировка UTF-8.

Все элементы документа XML обязательно должны содержаться в *корневом элементе* (root element), в листинге 28.1 это элемент `<notebook>`. Имя корневого элемента считается именем всего документа и указывается во второй части пролога, называемой *объявлением типа документа* (document type declaration). (Не путайте с определением типа документа DTD!) Имя документа записывается после слова `DOCTYPE`. Объявление типа документа записано во второй строке листинга 28.1. В этой части пролога после слова `DOCTYPE` и имени документа в квадратных скобках идет описание DTD:

```
<!DOCYPE notebook [ Сюда заносится описание DTD ]>
```

Очень часто описание DTD составляется сразу для нескольких документов XML. В таком случае его удобно записать отдельно от документа. Если описание DTD отделено от документа, то во второй части пролога вместо квадратных скобок указывается одно из слов: `SYSTEM` или `PUBLIC`. За словом `SYSTEM` идет URI файла с описанием DTD, а за словом `PUBLIC`, кроме того, можно записать дополнительную информацию.

Документ XML состоит из *элементов*. Элемент начинается *открывающим тегом*, далее идет необязательное *тело элемента*, потом — *закрывающий тег*:

```
<Открывающий тег>Тело элемента</Закрывающий тег>
```

Закрывающий тег содержит наклонную черту, после которой повторяется имя открывающего тега.

Язык XML, в отличие от языка HTML, требует обязательно записывать закрывающие теги. Если у элемента нет тела и закрывающего тега (empty — *пустой элемент*), то его открывающий тег должен заканчиваться символами "`/>`", например:

```
<br />
```

> ***Внимание***
>
> Сразу надо сказать, что язык XML, в отличие от HTML, различает регистры букв.

Из листинга 28.1 видно, что элементы документа XML могут быть вложены друг в друга. Надо следить за тем, чтобы элементы не пересекались, а полностью вкладывались друг в друга. Как уже говорилось ранее, все элементы, составляющие документ, вложены в корневой элемент этого документа. Тем самым документ наделяется структурой дерева вложенных элементов. На рис. 28.1 показана структура адресной книжки, описанной в листинге 28.1.

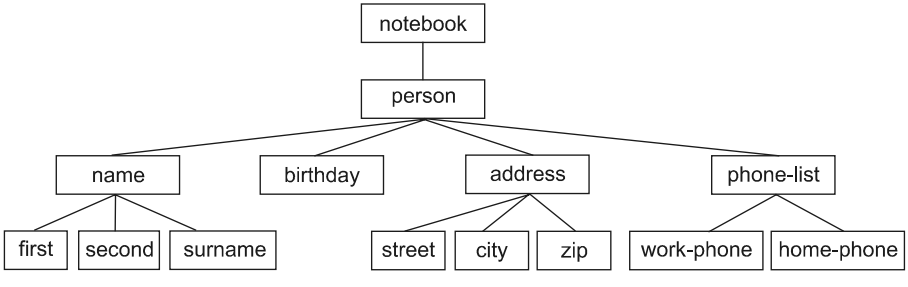


Рис. 28.1. Дерево элементов документа XML

У открывающих тегов XML могут быть *атрибуты*. Например, имя, отчество и фамилию можно записать как атрибуты `first`, `second` и `surname` тега `<name>`:

```
<name first="Иван" second="Петрович" surname="Сидоров" />
```

В отличие от языка HTML в языке XML значения атрибутов обязательно надо заключать в кавычки или в апострофы.

Атрибуты удобны для описания простых значений. У каждого гражданина России, уважающего паспортный режим, обязательно есть одно имя, одно отчество и одна фамилия. Их

лучше записывать атрибутами. Но у гражданина России может быть несколько телефонов, поэтому их номера удобнее оформить как элементы <work-phone> и <home-phone>, вложенные в элемент <phone-list>, а не атрибуты открывающего тега <phone-list>. Заметьте, что элемент <name> с атрибутами пустой, у него нет тела, следовательно, не нужен закрывающий тег. Поэтому тег <name> с атрибутами завершается символами "/>". В листинге 28.2 приведена измененная адресная книжка.

Листинг 28.2. Пример XML-документа с атрибутами в открывающем теге

```
<?xml version="1.0" encoding="Windows-1251"?>
   <!DOCTYPE notebook SYSTEM "ntb.dtd">

   <notebook>

     <person>

       <name first="Иван" second="Петрович" surname="Сидоров" />

       <birthday>25.03.1977</birthday>

    <address>
      <street>Садовая, 23-15</street>
      <city>Урюпинск</city>
      <zip>123456</zip>
       </address>

    <phone-list>
      <work-phone>2654321</work-phone>
      <work-phone>2654023</work-phone>
      <home-phone>3456781</home-phone>
       </phone-list>

     </person>

     <person>

       <name first="Мария" second="Петровна" surname="Сидорова" />

       <birthday>17.05.1969</birthday>

    <address>
      <street>Ягодная, 17</street>
      <city>Жмеринка</city>
      <zip>234561</zip>
       </address>

    <phone-list>
      <home-phone>2334455</home-phone>
       </phone-list>

     </person>

</notebook>
```

Атрибуты открывающего тега удобны и для указания типа элемента. Например, мы не уточняем, в городе живет наш родственник, в поселке или в деревне. Можно ввести в тег <city> атрибут type, принимающий одно из значений: город, поселок, деревня. Например:

```
<city type="город">Москва</city>
```

Для описания адресной книжки нам понадобились открывающие теги <notebook>, <person>, <name>, <address>, <street>, <city>, <zip>, <phone-list>, <work-phone>, <home-phone> и соответствующие им закрывающие теги, помеченные наклонной чертой. Теперь необходимо дать их описание. В описании указываются только самые общие признаки логической взаимосвязи элементов и их тип.

- Элемент <notebook> может содержать в себе ни одного, один или более одного элемента <person> и больше ничего.
- Элемент <person> содержит ровно один элемент <name>, ни одного, один или несколько элементов <address> и ни одного или только один элемент <phone-list>.
- Элемент <name> пустой.
- В открывающем теге <name> три атрибута first, second, surname, значения которых — строки символов.
- Элемент <address> содержит по одному элементу <street>, <city> и <zip>.
- Элементы <street> и <city> имеют по одной текстовой строке.
- Элемент <zip> содержит одно целое число.
- У открывающего тега <city> есть один необязательный атрибут type, принимающий одно из трех значений: город, поселок или деревня. Значение по умолчанию — город.
- Необязательный элемент <phone-list> содержит ни одного, один или несколько элементов <work-phone> и <home-phone>.
- Элементы <work-phone> и <home-phone> содержат по одной строке, состоящей только из цифр.

Это словесное описание, называемое *схемой* документа XML, формализуется несколькими способами. Наиболее распространены два способа: можно сделать описание DTD, пришедшее в XML из SGML, или описать схему на языке XSD (XML Schema Definition Language).

Описание DTD

Описание DTD нашей адресной книжки записано в листинге 28.3.

Листинг 28.3. Описание DTD документа XML

```
<!ELEMENT notebook (person)*>
<!ELEMENT person (name, birthday?, address*, phone-list?)>
<!ELEMENT name EMPTY>
<!ATTLIST name
   first   CDATA #IMPLIED
```

```
   second  CDATA #IMPLIED
   surname CDATA #REQUIRED>
<!ELEMENT birthday (#PCDATA)>
<!ELEMENT address (street, city, zip)?>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ATTLIST city
   type (город | поселок | деревня) "город">
<!ELEMENT zip (#PCDATA)>
<!ELEMENT phone-list (work-phone*, home-phone*)>
<!ELEMENT work-phone (#PCDATA)>
<!ELEMENT home-phone (#PCDATA)>
```

Как видите, описание DTD почти очевидно. Оно повторяет приведенное ранее словесное описание. Первое слово `ELEMENT` означает, что элемент может содержать тело с вложенными элементами. Вложенные элементы перечисляются в круглых скобках. Порядок перечисления вложенных элементов в скобках должен соответствовать порядку их появления в документе. Слово `EMPTY` в третьей строке листинга 28.3 означает пустой элемент.

Слово `ATTLIST` начинает описание списка атрибутов элемента. Для каждого атрибута указывается имя, тип и обязательность присутствия атрибута. Типов атрибута всего девять, но чаще всего употребляется тип `CDATA` (Character DATA), означающий произвольную строку символов Unicode, или перечисляются значения типа. Так сделано в описании атрибута `type` тега `<city>`, принимающего одно из трех значений: `город`, `поселок` или `деревня`. В кавычках показано значение по умолчанию — `город`.

Обязательность указания атрибута отмечается одним из трех слов:

- `#REQUIRED` — атрибут обязателен;
- `#IMPLIED` — атрибут необязателен;
- `#FIXED` — значение атрибута фиксировано, оно задается в DTD.

Первым словом могут быть, кроме слов `ELEMENT` или `ATTLIST`, слова `ANY`, `MIXED` или `ENTITY`. Слова `ANY` и `MIXED` означают, что элемент может содержать и простые данные и/или вложенные элементы. Слово `ENTITY` служит для обозначения или адреса данных, приведенного в описании DTD, так называемой *сущности*.

После имени элемента в скобках записываются вложенные элементы или тип данных, содержащихся в теле элемента. Тип `PCDATA` (Parsed Character DATA) означает строку символов Unicode, которую надо интерпретировать. Тип `CDATA` — строку символов Unicode, которую не следует интерпретировать.

Звездочка, записанная после имени элемента, означает "нуль или более вхождений" элемента, после которого она стоит, а плюс — "одно или более вхождений". Вопросительный знак означает "нуль или один раз". Если эти символы относятся ко всем вложенным элементам, то их можно указать после круглой скобки, закрывающей список вложенных элементов.

Описание DTD можно занести в отдельный файл, например ntb.dtd, указав его имя во второй части пролога, как показано во второй строке листингов 28.1 и 28.2. Можно включить описание во вторую часть пролога XML-файла, заключив его в квадратные скобки:

```
<!DOCTYPE notebook [ Описание DTD ]>
```

После того как создано описание DTD нашей реализации XML и написан документ, размеченный тегами этой реализации, следует проверить правильность их написания. Для этого есть специальные программы — *проверяющие анализаторы* (validating parsers). Все фирмы, разрабатывающие средства для работы с XML, выпускают бесплатные или коммерческие проверяющие анализаторы.

Проверяющий анализатор корпорации Sun Microsystems содержится в пакете классов JAXP (Java API for XML Processing), входящем в состав Java SE. Кроме того, этот пакет можно загрузить отдельно с адреса **http://java.sun.com/xml/**.

Корпорация Microsoft поставляет проверяющий анализатор MSXML (Microsoft XML Parser), доступный по адресу **http://msdn.microsoft.com/xml/**.

Есть еще множество проверяющих анализаторов, но лидером среди них является, пожалуй, Apache Xerces2, входящий во многие средства обработки документов XML, выпускаемые другими фирмами. Он свободно доступен по адресу **http://xerces.apache.org/xerces2-j/**.

Ограниченные средства DTD не позволяют полностью описать структуру документа XML. В частности, описание DTD не указывает точное количество повторений вложенных элементов, оно не задает точный тип тела элемента. Например, в листинге 28.3 из описания DTD не видно, что в элементе `<birthday>` содержится дата рождения. Эти недостатки DTD привели к появлению других схем описания документов XML. Наиболее развитое описание дает язык XSD. Мы будем называть описание на этом языке просто *схемой XML* (XML Schema).

Посмотрим, как создаются схемы XML, но сначала познакомимся еще с одним понятием XML — пространством имен.

Пространства имен XML

Поскольку в разных языках разметок — реализациях XML — могут встретиться одни и те же имена тегов и их атрибутов, имеющие совершенно разный смысл, а в документе XML их часто приходится смешивать, анализатору надо дать возможность их как-то различать. В языке Java мы имена полей класса уточняем именем класса, а имена классов уточняем указанием пакета, что позволяет назвать поле просто именем `a`, применяя при необходимости полное имя, что-нибудь вроде `com.mydomain.myhost.mypack.MyClass.a`. При этом рекомендуется пакет для уникальности называть доменным именем сайта разработчика, записывая его справа налево.

В языке XML принята подобная схема: локальное имя уточняется идентификатором. Идентификатор должен быть уникальным во всем Интернете, поэтому им должна быть строка символов, имеющая форму адреса URI. Адрес может соответствовать или не соответствовать какому-то реальному сайту Интернета, это не важно. Важно, чтобы он не повторялся в Интернете, можно записать, например, строку `http://some.firm.com/2008/ntbml`. Все имена с одним и тем же идентификатором образуют одно *пространство имен* (namespace).

Поскольку идентификатор пространства имен получается весьма длинным, было бы очень неудобно всегда записывать его перед локальным именем. В языке Java для сокращения записи имен мы используем оператор `import`. В XML принят другой подход. В каждом документе идентификатор пространства имен связывается с некоторым коротким префиксом, отделяемым от локального имени двоеточием. Префикс определяется атрибутом `xmlns` следующим образом:

```
<ntb:notebook xmlns:ntb = "http://some.firm.com/2008/ntbml">
```

Как видите, префикс `ntb` только что определен, но его уже можно использовать в имени `ntb:notebook`.

После такого описания имена тегов и атрибутов, которые мы хотим отнести к пространству имен http://some.firm.com/2008/ntbml, снабжаются префиксом `ntb`, например:

```
<ntb:city ntb:type="поселок">Горелово</ntb:city>
```

Имя вместе с префиксом, например `ntb:city`, называется *расширенным* или *уточненным именем* (Qualified Name, QName).

Хотя идентификатор пространства имен записывается в форме адреса URI, такого как `http://some.firm.com/2008/ntbml`, анализатор документа XML и другие программы, использующие документ, не будут обращаться по этому адресу. Там даже может не быть никакой Web-странички вообще. Просто идентификатор пространства имен должен быть уникальным во всем Интернете, и разработчики рекомендации по применению пространства имен, которую можно посмотреть по адресу **http://www.w3.org/TR/REC-xml-names**, справедливо решили, что будет удобно использовать для него DNS-имя сайта, на котором размещено определение пространства имен. Смотрите на URI просто как на строку символов, идентифицирующую пространство имен. Обычно указывается URL фирмы, создавшей данную реализацию XML, или имя файла с описанием схемы XML.

Поскольку идентификатор — это строка символов, то и сравниваются они как строки, с учетом регистра символов. Например, идентификатор `http://some.firm.com/2008/Ntbml` будет считаться XML-анализаторами отличным от идентификатора `http://some.firm.com/2008/ntbml`, введенного нами ранее, и будет определять другое пространство имен.

По правилам SGML и XML двоеточие может применяться в именах как обычный символ, поэтому имя с префиксом — это просто фокус, анализатор рассматривает его как обычное имя. Отсюда следует, что в описании DTD нельзя опускать префиксы имен. Некоторым анализаторам надо специально указать необходимость учета пространства имен. Например, при работе с анализатором Xerces следует применить метод `setNamespaceAware(true)`.

Атрибут `xmlns`, определяющий префикс имен, может появиться в любом элементе XML, а не только в корневом элементе. Определенный им префикс можно применять в том элементе, в котором записан атрибут `xmlns`, и во всех вложенных в него элементах. Больше того, в одном элементе можно определить несколько пространств имен:

```
<ntb:notebook xmlns:ntb = "http://some.firm.com/2008/ntbml"
              xmlns:bk  = "http://some.firm.com/2008/bookml">
```

Появление имени тега без префикса в документе, использующем пространство имен, означает, что имя принадлежит *пространству имен по умолчанию* (default namespace). Например, язык XHTML допускает применение тегов HTML и тегов XML в одном документе. Допустим, мы определили тег с именем `title`. Чтобы анализатор не принял его за один из тегов HTML, поступаем следующим образом:

```
<html xmlns = "http://www.w3.org/1999/xhtml"
         xmlns:ntb = "http://some.firm.com/2002/ntbml">

   <head>
      <title>Моя библиотека</title>
      </head>
```

```
   <body>
      <ntb:book>
          <ntb:title>Создание Java Web Services</ntb:title>
      </ntb:book>
   </body>
</html>
```

В этом примере пространством имен по умолчанию становится пространство имен XHTML, имеющее общеизвестный идентификатор `http://www.w3.org/1999/xhtml`, и теги, относящиеся к этому пространству имен, записываются без префикса.

Атрибуты никогда не входят в пространство имен по умолчанию. Если имя атрибута записано без префикса, то это означает, что атрибут не относится ни к одному пространству имен.

Префикс имени не относится к идентификатору пространства имен и может быть разным в разных документах. Например, в каком-нибудь другом документе мы можем написать корневой элемент

```
<nb:notebook xmlns:nb = "http://some.firm.com/2008/ntbml">
```

и записывать элементы с префиксом `nb`:

<nb:city nb:type="поселок">Горелово</nb:city>

Более того, можно связать несколько префиксов с одним и тем же идентификатором пространства имен даже в одном документе, но это может привести к путанице и применяется редко.

Теперь, после того как мы ввели понятие пространства имен, можно обратиться к схеме XML.

Схема XML

В мае 2001 года консорциум W3C рекомендовал описывать структуру документов XML на языке описания схем XSD (XML Schema Definition Language). На этом языке составляются *схемы XML* (XML Schema), описывающие элементы документов XML.

Схема XML сама записывается как документ XML. Его элементы называют *компонентами* (components), чтобы отличить их от элементов описываемого документа XML. Корневой компонент схемы носит имя `<schema>`. Компоненты схемы описывают элементы XML и определяют различные типы элементов. Рекомендация схемы XML, которую можно найти по ссылкам, записанным по адресу **http://www.w3.org/XML/Schema.html**, перечисляет 13 типов компонентов, но наиболее важны компоненты, определяющие простые и сложные типы элементов, сами элементы и их атрибуты.

Язык XSD различает простые и сложные элементы XML. *Простыми* (simple) элементами описываемого документа XML считаются элементы, не содержащие атрибутов и вложенных элементов. Соответственно, *сложные* (complex) элементы содержат атрибуты и/или вложенные элементы. Схема XML описывает *простые типы* — типы простых элементов, и *сложные типы* — типы сложных элементов.

Язык описания схем содержит много встроенных простых типов. Они перечислены в следующем разделе.

Встроенные простые типы XSD

Встроенные типы языка описания схем XSD позволяют записывать двоичные и десятичные целые числа, вещественные числа, дату и время, строки символов, логические значения, адреса URI. Рассмотрим их по порядку.

Вещественные числа

Вещественные числа в языке XSD разделены на три типа: `decimal`, `float` и `double`.

Тип `decimal` составляют вещественные числа, записанные с фиксированной точкой: 123.45, –0.1234567689345 и т. д. Фактически хранятся два целых числа: мантисса и порядок. Спецификация языка XSD не ограничивает количество цифр в мантиссе, но требует, чтобы можно было записать не менее 18 цифр. Этот тип легко реализуется классом `java.math.BigDecimal`, описанным в *главе 4*.

Типы `float` и `double` соответствуют стандарту IEEE754-85 и одноименным типам Java. Они записываются с фиксированной или с плавающей десятичной точкой.

Целые числа

Основной целый тип `integer` понимается как подтип типа `decimal`, содержащий числа с нулевым порядком. Это целые числа с любым количеством десятичных цифр: –34567, 123456789012345 и т. д. Данный тип легко реализуется классом `java.math.BigInteger`, описанным в *главе 4*.

Типы `long`, `int`, `short` и `byte` полностью соответствуют одноименным типам Java. Они понимаются как подтипы типа `integer`, типы более коротких чисел считаются подтипами более длинных чисел: тип `byte` — это подтип типа `short`, оба они подтипы типа `int` и т. д.

Типы `nonPositiveInteger` и `negativeInteger` — подтипы типа `integer` — составлены из неположительных и отрицательных чисел соответственно с любым количеством цифр.

Типы `nonNegativeInteger` и `positiveInteger` — подтипы типа `integer` — составлены из неотрицательных и положительных чисел соответственно с любым количеством цифр.

У типа `nonNegativeInteger` есть подтипы беззнаковых целых чисел `unsignedLong`, `unsignedInt`, `unsignedShort` и `unsignedByte`.

Строки символов

Основной символьный тип `string` описывает произвольную строку символов Unicode. Его можно реализовать классом `java.lang.String`.

Тип `normalizedString` — подтип типа `string` — это строки, не содержащие символы перевода строки `'\n'`, возврата каретки `'\r'` и символы горизонтальной табуляции `'\t'`.

В строках типа `token` — подтипа типа `normalizedString` — нет, кроме того, начальных и завершающих пробелов и нет нескольких подряд идущих пробелов.

В типе `token` выделены три подтипа. Подтип `language` определен для записи названия языка согласно RFC 1766, например: `ru`, `en`, `de`, `fr`. Подтип `NMTOKEN` используется только в атрибутах для записи их перечисляемых значений. Подтип `name` составляют *имена XML* —

последовательности букв, цифр, дефисов, точек, двоеточий, знаков подчеркивания, начинающиеся с буквы (кроме зарезервированной последовательности букв `X`, `x`, `M`, `m`, `L`, `l` в любом сочетании регистров) или знака подчеркивания. Двоеточие в значениях типа `name` используется для выделения префикса пространства имен.

Из типа `name` выделен подтип `NCName` (Non-Colonized Name) имен, не содержащих двоеточия, в котором, в свою очередь, определены три подтипа: `ID`, `ENTITY`, `IDREF`, описывающие идентификаторы XML, сущности и перекрестные ссылки.

Дата и время

Тип `duration` описывает промежуток времени, например запись `P1Y2M3DT10H30M45S` означает один год (`1Y`), два месяца (`2M`), три дня (`3D`), десять часов (`10H`), тридцать минут (`30M`) и сорок пять секунд (`45S`). Запись может быть сокращенной, например `P120M` означает 120 месяцев, а `T120M` — 120 минут.

Тип `dateTime` содержит дату и время в формате `CCYY-MM-DDThh:mm:ss`, например `2008-04-25T09:30:05`. Остальные типы выделяют какую-либо часть даты или времени.

Тип `time` содержит время в обычном формате `hh:mm:ss`.

Тип `date` содержит дату в формате `CCYY-MM-DD`.

Тип `gYearMonth` выделяет год и месяц в формате `CCYY-MM`.

Тип `gMonthDay` содержит месяц и день месяца в формате `-MM-DD`.

Тип `gYear` означает год в формате `CCYY`, тип `gMonth` — месяц в формате `-MM-`, тип `gDay` — день месяца в формате `-DD`.

Двоичные типы

Двоичные целые числа записываются либо в шестнадцатеричной форме без всяких дополнительных символов: `0B2F`, `356C0A` и т. д., это тип `hexBinary`, либо в кодировке Base64, это тип `base64Binary`.

Прочие встроенные простые типы

Еще три встроенных простых типа описывают значения, часто используемые в документах XML.

Адреса URI относятся к типу `anyURI`.

Расширенное имя тега или атрибута (qualified name), т. е. имя вместе с префиксом, отделенным от имени двоеточием, — это тип `QName`.

Элемент `NOTATION` описания DTD выделен как отдельный простой тип схемы XML. Его используют для записи математических, химических и других символов, нот, азбуки Бройля и прочих обозначений.

Определение простых типов

В схемах XML с помощью встроенных типов можно тремя способами определить новые типы простых элементов. Они вводятся как *сужение* (restriction) встроенного или ранее определенного простого типа, *список* (list) или *объединение* (union) простых типов.

Простой тип определяется компонентом схемы <simpleType>, имеющим вид:

```
<xsd:simpleType name="имя типа">Определение типа</xsd:simpleType>
```

Сужение

Сужение простого типа определяется компонентом <restriction>, в котором атрибут base указывает сужаемый простой тип, а в теле задаются ограничения, выделяющие определяемый простой тип. Например, почтовый индекс zip можно задать как шесть арабских цифр следующим образом:

```
  <xsd:simpleType name="zip">
 <xsd:restriction base="xsd:string">
   <xsd:pattern value="[0-9]{6}" />
    </xsd:restriction>
</xsd:simpleType>
```

Можно дать другое определение простого типа zip как целого положительного числа, находящегося в диапазоне от 100 000 до 999 999:

```
  <xsd:simpleType name="zip">
 <xsd:restriction base="xsd:positiveInteger">
   <xsd:minInclusive value="100000" />
   <xsd:maxInclusive value="999999" />
    </xsd:restriction>
</xsd:simpleType>
```

Теги <pattern>, <maxInclusive> и другие теги, задающие ограничения, называются *фасетками* (facets). Вот их список:

- <maxExclusive> — наибольшее значение, оно уже не входит в определяемый тип;
- <maxInclusive> — наибольшее значение определяемого типа;
- <minExclusive> — наименьшее значение, уже не входящее в определяемый тип;
- <minInclusive> — наименьшее значение определяемого типа;
- <totalDigits> — общее количество цифр в определяемом числовом типе — сужении типа decimal;
- <fractionDigits> — количество цифр в дробной части числа;
- <length> — длина значений определяемого типа;
- <maxLength> — наибольшая длина значений определяемого типа;
- <minLength> — наименьшая длина значений определяемого типа;
- <enumeration> — одно из перечисляемых значений;
- <pattern> — регулярное выражение [8];

- <whiteSpace> — применяется при сужении типа string и определяет способ преобразования пробельных символов '\n', '\r', '\t'. Атрибут value этого тега принимает одно из трех значений:
  - preserve — не убирать пробельные символы;
  - replace — заменить пробельные символы пробелами;
  - collapse — после замены пробельных символов пробелами убрать начальные и конечные пробелы, а из нескольких подряд идущих пробелов оставить только один.

В тегах-фасетках можно записывать следующие атрибуты, называемые *базисными фасетками* (fundamental facets):

- ordered — задает упорядоченность определяемого типа, принимает одно из трех значений:
  - false — тип неупорядочен;
  - partial — тип частично упорядочен;
  - total — тип полностью упорядочен;
- bounded — задает ограниченность или неограниченность типа значениями true или false;
- cardinality — задает конечность или бесконечность типа значениями finite или countably infinite;
- numeric — показывает, числовой этот тип или нет, значениями true или false.

Как видно из приведенных ранее и далее примеров, в одном сужении может быть несколько ограничений-фасеток. При этом фасетки <pattern> и <enumeration> задают независимые друг от друга ограничения, их можно мысленно объединить союзом "или". Остальные фасетки задают общие, совместно накладываемые ограничения, их можно мысленно объединить союзом "и".

Список

Простой тип-список — это тип элементов, в теле которых записывается, через пробел, несколько значений одного и того же простого типа. Например, в документе XML может встретиться такой элемент, содержащий список целых чисел:

```
<days>21 34 55 46</days>
```

Список определяется компонентом <list>, в котором атрибутом itemType указывается тип элементов определяемого списка. Тип элементов списка можно указать и в теле элемента <list>. Например, показанный ранее элемент документа XML <days> можно определить в схеме так:

```
<xsd:element name="days" type="listOfInteger" />
```

а использованный при его определении тип listOfInteger задать как список не более чем из пяти целых чисел следующим образом:

```
<xsd:simpleType name="listOfInteger">
  <xsd:restriction>
```

```
    <xsd:simpleType>
      <xsd:list itemType="xsd:integer" />
       </xsd:simpleType>

       <xsd:maxLength value="5" />

     </xsd:restriction>

</xsd:simpleType>
```

При определении списка можно применять фасетки <length>, <minLength>, <maxLength>, <enumeration>, <pattern>. В приведенном примере список — тело элемента <days> — не может содержать более пяти чисел.

Объединение

Простой тип-объединение определяется компонентом <union>, в котором атрибутом memberTypes можно указать имена объединяемых типов. Например:

```
<xsd:union memberTypes="xsd:string xsd:integer listOfInteger" />
```

Другой способ — записать в теле компонента <union> определения простых типов, входящих в объединение. Например:

```
   <xsd:attribute name="size">

  <xsd:simpleType>
      <xsd:union>

        <xsd:simpleType>

       <xsd:restriction base="xsd:positiveInteger">
         <xsd:minInclusive value="8"/>
         <xsd:maxInclusive value="72"/>
          </xsd:restriction>

        </xsd:simpleType>

        <xsd:simpleType>

       <xsd:restriction base="xsd:NMTOKEN">
         <xsd:enumeration value="small"/>
         <xsd:enumeration value="medium"/>
         <xsd:enumeration value="large"/>
          </xsd:restriction>

        </xsd:simpleType>

    </xsd:union>
     </xsd:simpleType>

</xsd:attribute>
```

После этого атрибут `size` можно использовать, например, так:

```
<font size='large'>Глава 28</font>
<font size='12'>Простой текст</font>
```

Описание элементов и их атрибутов

Элементы, которые будут применяться в документе XML, описываются в схеме компонентом `<element>`:

```
<xsd:element name="имя элемента" type="тип элемента"
  minOccurs="наименьшее число появлений элемента в документе"
  maxOccurs="наибольшее число появлений" />
```

Значение по умолчанию необязательных атрибутов `minOccurs` и `maxOccurs` равно 1. Это означает, что если эти атрибуты отсутствуют, то элемент должен появиться в документе XML ровно один раз. Определение типа элемента можно вынести в тело элемента `<element>`:

```
<xsd:element name="имя элемента" >
   Определение типа элемента
</xsd:element>
```

Описание атрибута элемента тоже несложно:

```
<xsd:attribute name="имя атрибута" type="тип атрибута"
  use="обязательность атрибута" default="значение по умолчанию" />
```

Необязательный атрибут `use` принимает три значения:

- `optional` — описываемый атрибут необязателен (это значение по умолчанию);
- `required` — описываемый атрибут обязателен;
- `prohibited` — описываемый атрибут неприменим. Это значение полезно при определении подтипа, чтобы отменить некоторые атрибуты базового типа.

Если описываемый атрибут необязателен, то атрибутом `default` можно задать его значение по умолчанию.

Определение типа атрибута, — а это должен быть простой тип, — можно вынести в тело элемента `<attribute>`:

```
<xsd:attribute name="имя атрибута">
   Тип атрибута
</xsd:attribute>
```

Определение сложных типов

Напомним, что тип элемента называется сложным, если в элемент вложены другие элементы и/или в открывающем теге элемента есть атрибуты.

Сложный тип определяется компонентом `<complexType>`, имеющим вид:

```
<xsd:complexType name="имя типа" >Определение типа</xsd:complexType>
```

Необязательный атрибут name задает имя типа, а в теле компонента <complexType> описываются элементы, входящие в сложный тип, и/или атрибуты открывающего тега.

Определение сложного типа можно разделить на определение типа пустого элемента, элемента с простым телом, и элемента, содержащего вложенные элементы. Рассмотрим эти определения подробнее.

Определение типа пустого элемента

Проще всего определяется тип пустого элемента — элемента, не имеющего тела, а содержащего только атрибуты в открывающем теге. Таков, например, элемент <name> из листинга 28.2. Каждый атрибут описывается одним компонентом <attribute>, например:

```
<xsd:complexType name="imageType">
  <xsd:attribute name="href" type="xsd:anyURI" />
</xsd:complexType>
```

После этого определения можно в схеме описать элемент <image> типа imageType:

```
<xsd:element name="image" type="imageType" />
```

а в документе XML использовать это описание:

```
<image href="http://some.com/images/myface.gif" />
```

Определение типа элемента с простым телом

Немного сложнее описание элемента, содержащего тело простого типа и атрибуты в открывающем теге. Этот тип отличается от простого типа только наличием атрибутов и определяется компонентом <simpleContent>. В теле данного компонента должен быть либо компонент <restriction>, либо компонент <extension>, атрибутом base задающий тип (простой) тела описываемого элемента.

В компоненте <extension> описываются атрибуты открывающего тега описываемого элемента. Все вместе выглядит так, как в следующем примере:

```
   <xsd:complexType name="calcResultType">

   <xsd:simpleContent>
      <xsd:extension base="xsd:decimal">
         <xsd:attribute name="unit" type="xsd:string" />
         <xsd:attribute name="precision"
                        type="xsd:nonNegativeInteger" />
      </xsd:extension>
      </xsd:simpleContent>

</xsd:complexType>
```

Эту конструкцию можно описать словами так: "Определяется тип calcResultType элемента, тело которого содержит значения встроенного простого типа xsd:decimal. Простой тип расширяется тем, что к нему добавляются атрибуты unit и precision".

Если в схеме описать элемент <result> этого типа следующим образом:

```
<xsd:element name="result" type="calcResultType" />
```

то в документе XML можно написать

```
<result unit="см" precision="2">123.25</result>
```

В компоненте <restriction> кроме атрибутов описывается простой тип тела элемента и/или фасетки, ограничивающие тип, заданный атрибутом base. Например:

```
   <xsd:complexType name="calcResultType">

   <xsd:simpleContent>
      <xsd:restriction base="xsd:decimal">
         <xsd:totalDigits value="8" />
         <xsd:attribute name="unit" type="xsd:string" />
         <xsd:attribute name="precision"
                        type="xsd:nonNegativeInteger" />
      </xsd:restriction>
      </xsd:simpleContent>

</xsd:complexType>
```

Определение типа вложенных элементов

Если значениями определяемого сложного типа будут элементы, содержащие вложенные элементы, как, например, элементы <address>, <phone-list> листинга 28.2, то перед перечислением описания вложенных элементов надо выбрать *модель группы* (model group) вложенных элементов. Дело в том, что вложенные элементы, составляющие определяемый тип, могут появляться или в определенном порядке, или в произвольном порядке, кроме того, можно выбирать только один из перечисленных элементов. Эта возможность и называется *моделью группы* элементов. Она определяется одним из трех компонентов: <sequence>, <all> или <choice>.

Компонент <sequence> применяется в том случае, когда перечисляемые элементы должны записываться в документе в конкретном порядке. Пусть, например, мы описываем книгу. Сначала определяем тип:

```
   <xsd:complexType name="bookType">

  <xsd:sequence maxOccurs="unbounded">
    <xsd:element name="author" type="xsd:normalizedString"
                 minOccurs="0" />
    <xsd:element name="title" type="xsd:normalizedString" />
    <xsd:element name="pages" type="xsd:positiveInteger"
                 minOccurs="0" />
    <xsd:element name="publisher" type="xsd:normalizedString"
                 minOccurs="0" />
     </xsd:sequence>

</xsd:complexType>
```

Потом описываем элемент:

```
<xsd:element name="book" type="bookType" />
```

Элементы <author>, <title>, <pages> и <publisher> должны входить в элемент <book> именно в таком порядке. В документе XML надо писать:

```
<book>
  <author>И. Ильф, Е. Петров</author>
  <title>Золотой теленок</title>
  <publisher>Детская литература</publisher>
</book>
```

Если же вместо компонента <xsd:sequence> записать компонент <xsd:all>, то элементы <author>, <title>, <pages> и <publisher> можно перечислять в любом порядке.

Компонент <choice> применяется в том случае, когда надо выбрать один из нескольких элементов. Например, при описании журнала вместо издательства, описываемого элементом <publisher>, следует записать название журнала. Это можно определить так:

```
   <xsd:complexType name="bookType">

     <xsd:sequence maxOccurs="unbounded">

    <xsd:element name="author" type="xsd:normalizedString"
                 minOccurs="0" />
    <xsd:element name="title" type="xsd:normalizedString" />
    <xsd:element name="pages" type="xsd:positiveInteger"
                   minOccurs="0" />

    <xsd:choice>
      <xsd:element name="publisher" type="xsd:normalizedString"
                   minOccurs="0" />
      <xsd:element name="magazine" type="xsd:normalizedString"
                   minOccurs="0" />
       </xsd:choice>

     </xsd:sequence>

</xsd:complexType>
```

Как видно из приведенного примера, компонент <choice> можно вложить в компонент <sequence>. Можно, наоборот, вложить компонент <sequence> в компонент <choice>. Такие вложения допустимо проделывать сколько угодно раз. Кроме того, каждая группа в этих моделях может появиться сколько угодно раз, т. е. в компоненте <choice> тоже разрешено записать атрибут maxOccurs="unbounded".

Модель группы <all> отличается в этом от моделей <sequence> и <choice>. В компоненте <all> не допускается применение компонентов <sequence> и <choice>. Обратно, в компонентах <sequence> и <choice> нельзя применять компонент <all>. Каждый элемент, входящий в группу модели <all>, может появиться не более одного раза, т. е. атрибут maxOccurs этого элемента может равняться только единице.

Определение типа со сложным телом

При определении сложного типа можно воспользоваться уже определенным, *базовым*, сложным типом, расширив его дополнительными элементами или, наоборот, удалив из него некоторые элементы. Для этого надо применить компонент `<complexContent>`. В этом компоненте, так же как и в компоненте `<simpleContent>`, записывается либо компонент `<extension>`, если нужно расширить базовый тип, либо компонент `<restriction>`, если необходимо сузить базовый тип. Базовый тип указывается атрибутом `base`, так же как и при записи компонента `<simpleContent>`, но теперь это должен быть сложный, а не простой тип!

Расширим, например, определенный ранее тип `bookType`, добавив год издания — элемент `<year>`:

```
   <xsd:complexType name="newBookType">

  <xsd:complexContent>
    <xsd:extension base="bookType">
      <xsd:sequence>
         <xsd:element name="year" type="xsd:gYear">
      </xsd:sequence>
    </xsd:extension>
     </xsd:complexContent>

</xsd:complexType>
```

При сужении базового типа компонентом `<restriction>` надо перечислить те элементы, которые останутся после сужения. Например, оставим в типе `newbookType` только автора и название книги из типа `bookType`:

```
   <xsd:complexType name="newBookType">

     <xsd:complexContent>

    <xsd:restriction base="bookType">
      <xsd:sequence>
         <xsd:element name="author" type="xsd:normalizedString"
                      minOccurs="0" />
         <xsd:element name="title" type="xsd:normalizedString" />
      </xsd:sequence>
       </xsd:restriction>

     </xsd:complexContent>

</xsd:complexType>
```

Это описание выглядит странно. Почему надо заново описывать все элементы, остающиеся после сужения? Не проще ли определить новый тип?

Дело в том, что в язык XSD внесены элементы объектно-ориентированного программирования, которых мы не будем касаться. Расширенный и суженный типы связаны со своим базовым типом отношением наследования, и к ним можно применить операцию подстановки. У всех типов языка XSD есть общий предок — базовый тип `anyType`. От него наследуются

все сложные типы. Это подобно тому, как у всех классов Java есть общий предок — класс `Object`, а все массивы наследуются от него. От базового типа `anyType` наследуется и тип `anySimpleType` — общий предок всех простых типов.

Таким образом, сложные типы определяются как сужение типа `anyType`. Если строго подходить к определению сложного типа, то определение типа `bookType`, сделанное в начале предыдущего раздела, надо записать так:

```
  <xsd:complexType name="bookType">

    <xsd:complexContent>

      <xsd:restriction base="xsd:anyType">

     <xsd:sequence maxOccurs="unbounded">
       <xsd:element name="author" type="xsd:normalizedString"
                    minOccurs="0" />
       <xsd:element name="title" type="xsd:normalizedString" />
       <xsd:element name="pages" type="xsd:positiveInteger"
                    minOccurs="0" />
       <xsd:element name="publisher" type="xsd:normalizedString"
                    minOccurs="0" />
        </xsd:sequence>

      </xsd:restriction>

    </xsd:complexContent>

</xsd:complexType>
```

Рекомендация языка XSD позволяет сократить эту запись, что мы и сделали в предыдущем разделе. Это подобно тому, как в Java мы опускаем слова "`extends Object`" в заголовке описания класса.

Закончим на этом описание языка XSD и перейдем к примерам.

Пример: схема адресной книги

В листинге 28.4 записана схема документа, приведенного в листинге 28.2.

Листинг 28.4. Схема документа XML

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:ntb="http://some.firm.com/2008/ntbNames"
               targetNamespace="http://some.firm.com/2008/ntbNames">

   <xsd:element name="notebook" type="ntb:notebookType" />

<xsd:complexType name="notebookType">
    <xsd:element name="person" type="ntb:personType"
```

```
                minOccurs="0" maxOccurs="unbounded" />
   </xsd:complexType>

<xsd:complexType name="personType">
   <xsd:sequence>

  <xsd:element name="name">
    <xsd:complexType>
      <xsd:attribute name="first"   type="xsd:string" use="optional" />
      <xsd:attribute name="second"  type="xsd:string" use="optional" />
      <xsd:attribute name="surname" type="xsd:string" use="required" />
    </xsd:complexType>
     </xsd:element>

     <xsd:element name="birthday" type="ntb:ruDate" minOccurs="0" />

  <xsd:element name="address" type="ntb:addressType"
                minOccurs="0" maxOccurs="unbounded" />

     <xsd:element name="phone-list" type="ntb:phone-listType" minOccurs="0"
   />

</xsd:sequence>
   </xsd:complexType>

<xsd:complexType name="addressType" >
<xsd:sequence>
  <xsd:element name="street" type="xsd:string" />
  <xsd:element name="city"   type="ntb:cityType" />
  <xsd:element name="zip"    type="xsd:positiveInteger" />
</xsd:sequence>
   </xsd:complexType>

<xsd:complexType name='cityType'>
  <xsd:simpleContent>
     <xsd:extension base='xsd:string' >
       <xsd:attribute name='type' type='ntb:placeType' default='город' />
     </xsd:extension>
  </xsd:simpleContent>
   </xsd:complexType>

<xsd:simpleType name="placeType">
  <xsd:restriction base = "xsd:string">
    <xsd:enumeration value="город" />
    <xsd:enumeration value="поселок" />
    <xsd:enumeration value="деревня" />
  </xsd:restriction>
   </xsd:simpleType>
```

```
<xsd:complexType name="phone-listType">
    <xsd:element name="work-phone"  type="xsd:string"
                 minOccurs="0" maxOccurs="unbounded" />
    <xsd:element name="home-phone"  type="xsd:string"
                 minOccurs="0" maxOccurs="unbounded" />
   </xsd:complexType>

<xsd:simpleType name="ruDate">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[0-9]{2}.[0-9]{2}.[0-9]{4}" />
  </xsd:restriction>
   </xsd:simpleType>

</xsd:schema>
```

Листинг 28.4, как обычный документ XML, начинается с пролога, показывающего версию XML и определяющего стандартное пространство имен схемы XML с идентификатором `http://www.w3.org/2001/XMLSchema`. Этому идентификатору дан префикс `xsd`. Конечно, префикс может быть другим, часто пишут префикс `xs`.

Еще не встречавшийся нам атрибут `targetNamespace` определяет идентификатор пространства имен, в которое попадут определяемые в этом документе имена типов, элементов и атрибутов, так называемое *целевое пространство имен* (target namespace). Этот идентификатор сразу же связывается с префиксом `ntb`, который тут же используется для уточнения только что определенных имен в ссылках на них.

Все описание схемы нашей адресной книжки заключено в одной третьей строке, в которой указано, что адресная книга состоит из одного элемента с именем `notebook`, имеющего сложный тип `notebookType`. Этот элемент должен появиться в документе ровно один раз. Остаток листинга 28.4 посвящен описанию типа этого элемента и других типов.

Описание сложного типа `notebookType` несложно (простите за каламбур). Оно занимает три строки листинга, не считая открывающего и закрывающего тега, и просто говорит о том, что данный тип составляют несколько элементов `person` типа `personType`.

Описание типа `personType` немногим сложнее. Оно говорит, что этот тип составляют четыре элемента: `name`, `birthday`, `address` и `phone-list`. Для элемента `name` сразу же указаны необязательные атрибуты `first` и `second` простого типа `string`, определенного в пространстве имен `xsd`. Тип обязательного атрибута `surname` — тоже `string`.

Далее в листинге 28.4 определяются оставшиеся типы: `addressType`, `phone-listType` и `ruDate`. Необходимость определения простого типа `ruDate` возникает потому, что встроенный в схему XML тип `date` предписывает задавать дату в виде 2004-10-22, а в России принят формат 22.10.2004. Тип `ruDate` определяется как *сужение* (restriction) типа `string` с помощью шаблона. *Шаблон* (pattern) для записи даты в виде `ДД.ММ.ГГГГ` задается регулярным выражением.

Безымянные типы

Все описанные в листинге 28.4 типы используются только один раз. Поэтому необязательно давать типу имя. Схема XML, как говорилось ранее, позволяет определять безымянные ти-

пы. Такое определение дается внутри описания элемента. Именно так в листинге 28.4 описаны атрибуты элемента `name`. В листинге 28.5 показано упрощенное описание схемы адресной книги.

Листинг 28.5. Схема документа XML с безымянными типами

```
<?xml version='1.0'?>
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'
               targetNamespace='http://some.firm.com/2008/ntbNames'>

<xsd:element name='notebook'>
<xsd:complexType>
   <xsd:sequence>

  <xsd:element name='person' maxOccurs='unbounded'>
  <xsd:complexType>
     <xsd:sequence>

    <xsd:element name='name'>
    <xsd:complexType>
       <xsd:attribute name='first'   type='xsd:string' use='optional' />
       <xsd:attribute name='second'  type='xsd:string' use='optional' />
       <xsd:attribute name='surname' type='xsd:string' use='required' />
    </xsd:complexType>
       </xsd:element>

    <xsd:element name='birthday'>
    <xsd:simpleType>
        <xsd:restriction base='xsd:string'>
           <xsd:pattern value='[0-9]{2}.[0-9]{2}.[0-9]{4}' />
        </xsd:restriction>
    </xsd:simpleType>
       </xsd:element>

    <xsd:element name='address' maxOccurs='unbounded'>
    <xsd:complexType>
    <xsd:sequence>
        <xsd:element name='street' type='xsd:string' />
        <xsd:element name='city'>
        <xsd:complexType>
           <xsd:simpleContent>

           <xsd:extension base='xsd:string'>
               <xsd:attribute name='type' type='xsd:string'
                              use='optional' default='gorod' />
              </xsd:extension>

        </xsd:simpleContent>
```

```
        </xsd:complexType>
           </xsd:element>

        <xsd:element name='zip' type='xsd:positiveInteger' />
    </xsd:sequence>
    </xsd:complexType>
       </xsd:element>

    <xsd:element name='phone-list'>
    <xsd:complexType>
    <xsd:sequence>
       <xsd:element name='work-phone' type='xsd:string'
                       minOccurs='0' maxOccurs='unbounded' />

       <xsd:element name='home-phone' type='xsd:string'
                    minOccurs='0' maxOccurs='unbounded' />
    </xsd:sequence>
    </xsd:complexType>
       </xsd:element>

  </xsd:sequence>
  </xsd:complexType>
     </xsd:element>

  </xsd:sequence>
     </xsd:complexType>

   </xsd:element>

</xsd:schema>
```

Еще одно упрощение можно сделать, используя пространство имен по умолчанию. Посмотрим, какие пространства имен применяются в схемах XML.

Пространства имен языка XSD

Имена элементов и атрибутов, используемые при записи схем, определены в пространстве имен с идентификатором `http://www.w3.org/2001/XMLSchema`. Префикс имен, относящихся к этому пространству, часто называют `xs` или `xsd`, как в листингах 28.4 и 28.5. Каждый анализатор "знает" это пространство имен и "понимает" имена из этого пространства.

Можно сделать это пространство имен пространством по умолчанию, но тогда надо обязательно определить префикс идентификатора целевого пространства имен для определяемых в схеме типов и элементов.

В листинге 28.6 для упрощения записи стандартное пространство имен схемы XML с идентификатором `http://www.w3.org/2001/XMLSchema` сделано пространством имен по умолчанию. Имена, относящиеся к целевому пространству имен, снабжены префиксом `ntb`, чтобы они не попали в пространство имен по умолчанию.

Листинг 28.6. Схема документа XML с целевым пространством имен

```
<?xml version='1.0'?>
<schema xmlns='http://www.w3.org/2001/XMLSchema'
   targetNamespace='http://some.firm.com/2008/ntbNames'
      xmlns:ntb='http://some.firm.com/2008/ntbNames'>

   <element name='notebook'>

  <complexType>
       <sequence>

      <element name='person' maxOccurs='unbounded'>
        <complexType>
             <sequence>

          <element name='name'>
            <complexType>
              <attribute name='first'   type='string' use='optional' />
              <attribute name='second'  type='string' use='optional' />
              <attribute name='surname' type='string' use='required' />
            </complexType>
             </element>

          <element name='birthday'>
            <simpleType>
               <restriction base='string'>
                  <pattern value='[0-9]{2}.[0-9]{2}.[0-9]{4}' />
               </restriction>
            </simpleType>
             </element>

          <element name='address' maxOccurs='unbounded'>
             <complexType>
                <sequence>
                  <element name='street' type='string' />
                  <element name='city'   type='string' />
                  <element name='zip'    type='positiveInteger' />
                </sequence>
             </complexType>
              </element>

           <element name='phone-list'>
              <complexType>
                 <sequence>
                    <element name='work-phone' type='string'
                             minOccurs='0' maxOccurs='unbounded'/>
                    <element name='home-phone' type='string'
```

```
                                    minOccurs='0' maxOccurs='unbounded'/>
                        </sequence>
                    </complexType>
                    </element>

            </sequence>
        </complexType>
         </element>

     </sequence>
      </complexType>

   </element>

</schema>
```

Поскольку в листинге 28.6 пространством имен по умолчанию сделано пространство `http://www.w3.org/2001/XMLSchema`, префикс `xsd` не нужен.

Следует заметить, что в целевое пространство имен попадают только *глобальные имена*, чьи описания непосредственно вложены в элемент `<schema>`. Это естественно, потому что лишь глобальными именами можно воспользоваться далее в этой или другой схеме. В листинге 28.6 только одно глобальное имя — `<notebook>`. Вложенные имена `name`, `address` и другие всего-навсего *ассоциированы* с глобальными именами.

В схемах и документах XML часто применяется еще одно стандартное пространство имен. Рекомендация языка XSD определяет несколько атрибутов: `type`, `nil`, `schemaLocation`, `noNamespaceSchemaLocation`, которые применяются не только в схемах, но и непосредственно в описываемых этими схемами документах XML, называемых *экземплярами схем* (XML schema instance). Имена этих атрибутов относятся к пространству имен `http://www.w3.org/2001/XMLSchema-instance`. Данному пространству имен чаще всего приписывают префикс `xsi`, например:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

Включение файлов схемы в другую схему

В создаваемую схему можно включить файлы, содержащие другие схемы. Для этого есть два элемента схемы: `<include>` и `<import>`. Например:

```
<xsd:include xsi:schemaLocation="names.xsd" />
```

Включаемый файл задается атрибутом `xsi:schemaLocation`. В примере он использован для того, чтобы включить в создаваемую схему содержимое файла names.xsd. Файл должен содержать схему с описаниями и определениями из того же пространства имен, что и в создаваемой схеме, или без пространства имен, т. е. в нем не использован атрибут `targetNamespace`. Это удобно, если мы хотим добавить к создаваемой схеме определения схемы names.xsd или просто хотим разбить большую схему на два файла. Можно представить себе результат включения так, как будто содержимое файла names.xsd просто записано на месте элемента `<include>`.

Перед включением файла можно изменить некоторые определения, приведенные в нем. Для этого используется элемент <redefine>, например:

```
   <xsd:redefine schemaLocation="names.xsd">

  <xsd:simpleType name="nameType">
    <xsd:restriction base="xsd:string">
      <xsd:maxLength value="40"/>
    </xsd:restriction>
     </xsd:simpleType>

</xsd:redefine>
```

Если же включаемый файл содержит имена из другого пространства имен, то надо воспользоваться элементом схемы <import>. Например, пусть файл A.xsd начинается со следующих определений:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://some.firm.com/someNames">
```

а файл B.xsd начинается с определений

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://some.firm.com/anotherNames">
```

Мы решили включить эти файлы в новый файл C.xsd. Это делается так:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://some.firm.com/yetAnotherNames"
    xmlns:pr1="http://some.firm.com/someNames"
       xmlns:pr2="http://some.firm.com/anotherNames">

<xsd:import namespace="http://some.firm.com/someNames"
       xsi:schemaLocation="A.xsd" />

<xsd:import namespace="http://some.firm.com/anotherNames"
    xsi:schemaLocation="B.xsd" />
```

После этого в файле C.xsd можно использовать имена, определенные в файлах A.xsd и B.xsd, снабжая их префиксами pr1 и pr2 соответственно.

Элементы <include> и <import> следует располагать перед всеми определениями схемы.

Значение атрибута xsi:schemaLocation — строка URI, поэтому файл с включаемой схемой может располагаться в любом месте Интернета.

Связь документа XML со своей схемой

Программе-анализатору, проверяющей соответствие документа XML его схеме, надо как-то указать файлы (один или несколько), содержащие схему документа. Это можно сделать разными способами. Во-первых, можно подать необходимые файлы на вход анализатора. Так делает, например, проверяющий анализатор XSV (XML Schema Validator) — **ftp://ftp.cogsci.ed.ac.uk/pub/XSV/**:

```
$ xsv ntb.xml ntb1.xsd ntb2.xsd
```

Во-вторых, можно задать файлы со схемой как свойство анализатора, устанавливаемое методом `setProperty()`, или значение переменной окружения анализатора. Так делает, например, проверяющий анализатор Xerces.

Эти способы удобны, когда документ в разных случаях нужно связать с различными схемами. Если же схема документа фиксирована, то ее удобнее указать прямо в документе XML. Это делается одним из двух способов.

- Если элементы документа не принадлежат никакому пространству имен и записаны без префикса, то в корневом элементе документа записывается атрибут `noNamespaceSchemaLocation`, указывающий расположение файла со схемой в форме URI:

  ```
  <notebook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="ntb.xsd">
  ```

  В этом случае в схеме не должно быть целевого пространства имен, т. е. не следует использовать атрибут `targetNamespace`.

- Если же элементы документа относятся к некоторому пространству имен, то применяется атрибут `schemaLocation`, в котором через пробел парами перечисляются пространства имен и расположение файла со схемой, описывающей это пространство имен. Продолжая пример предыдущего раздела, можно написать:

  ```
  <notebook xmlns="http://some.firm.com/2003/ntbNames"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation=
         "http://some.firm.com/someNames A.xsd
          http://some.firm.com/anotherNames B.xsd"
      xmlns:pr1="http://some.firm.com/someNames"
      xmlns:pr2="http://some.firm.com/anotherNames">
  ```

  После этого в документе можно использовать имена, определенные в схемах A.xsd и B.xsd, снабжая их префиксами `pr1` и `pr2` соответственно.

Другие языки описания схем

Даже из приведенного ранее краткого описания языка XSD видно, что он получился весьма сложным и запутанным. Есть уже несколько книг, полностью посвященных этому языку. Их объем ничуть не меньше объема этой книги.

Существуют и другие, более простые языки описания схемы документа XML. Наибольшее распространение получили следующие языки:

- Schematron — **http://www.ascc.net/xml/resource/schematron/**;
- RELAX NG (Regular Language Description for XML, New Generation), этот язык возник как слияние языков Relax и TREX — **http://www.oasis-open.org/committees/relax-ng/**;
- Relax — **http://www.xml.gr.jp/relax/**;
- TREX (Tree Regular Expressions for XML) — **http://www.thaiopensource.com/trex/**;
- DDML (Document Definition Markup Language), известный еще как XSchema — **http://purl.oclc.org/NET/ddml/**.

Менее распространены языки DCD (Document Content Description), SOX (One's Schema for Object-Oriented XML), XDR (XML-Data Reduced).

Все перечисленные языки позволяют более или менее полно описывать схему документа. Возможно, они вытеснят язык XSD, возможно, будут существовать совместно.

Инструкции по обработке

Упомянем еще одну конструкцию языка XML — *инструкции по обработке* (processing instructions). Она позволяет передать анализатору или другой программе-обработчику документа дополнительные сведения для обработки. Инструкция по обработке выглядит так:

```
<? сведения для анализатора ?>
```

Первая часть пролога документа XML — первая строка XML-файла — это как раз инструкция по обработке. Она передает анализатору документа версию языка XML и кодировку символов, которыми записан документ.

Первая часть работы закончена. Документы XML и их схемы написаны. Теперь надо подумать о том, каким образом они будут отображаться на экране дисплея, на листе бумаги, на экране сотового телефона, т. е. нужно подумать о визуализации документа XML.

Прежде всего, документ следует разобрать, проанализировать (parse) его структуру.

Анализ документа XML

На первом этапе разбора проводится *лексический анализ* (lexical parsing) документа XML. Документ разбивается на отдельные *неделимые элементы* (tokens), которыми являются теги, служебные слова, разделители, текстовые константы. Проводится проверка полученных элементов и их связей между собой. Лексический анализ выполняют специальные программы — *сканеры* (scanners). Простейшие сканеры — это классы `java.util.StringTokenizer` и `java.io.StreamTokenizer` из стандартной поставки Java SE JDK, которые мы рассматривали в предыдущих главах.

Затем осуществляется *грамматический анализ* (grammar parsing). При этом анализируется логическая структура документа, составляются выражения, выражения объединяются в блоки, блоки — в модули, которыми могут являться абзацы, параграфы, пункты, главы. Грамматический анализ проводят программы-анализаторы, так называемые *парсеры* (parsers).

Создание сканеров и парсеров — любимое развлечение программистов. За недолгую историю XML написаны десятки, если не сотни XML-парсеров. Многие из них написаны на языке Java. Все парсеры можно разделить на две группы.

В первую группу входят парсеры, проводящие анализ, основываясь на структуре дерева, отражающего вложенность элементов документа (tree-based parsing). Такие парсеры проще реализовать, но создание дерева требует большого объема оперативной памяти, ведь размер документов XML не ограничен. Необходимость частого просмотра дерева замедляет работу парсера.

Во вторую группу входят парсеры, проводящие анализ, основываясь на событиях (event-based parsing). Событием считается появление какого-либо элемента XML: открывающего или закрывающего тега, текста, содержащегося в теле элемента. При возникновении события вызывается какой-нибудь метод его обработки: `startElement()`, `endElement()`, `characters()` и т. д. Такие парсеры сложнее в реализации, зато они не строят дерево в оперативной памяти и могут анализировать не весь документ, а его отдельные элементы вместе с вложенными в них элементами. Фактическим стандартом здесь стал свободно распространяемый набор классов и интерфейсов SAX (Simple API for XML), созданный Давидом Меггинсоном (David Megginson). Основной сайт данного проекта — **http://www.saxproject.org/**. Сейчас применяется второе поколение этого набора, называемое SAX2. Набор SAX2 входит во многие парсеры, например Xerces2.

В стандартную поставку Java и Standard Edition и Enterprise Edition входит набор интерфейсов и классов для создания парсеров и преобразования документов XML, называемый JAXP (Java API for XML Processing). С помощью одной из частей этого набора, называемой DOM API (Document Object Model API), можно создавать парсеры первого типа, формирующие дерево объектов, а с помощью второй части набора JAXP, называемой SAX API, можно создавать SAX-парсеры. Интерфейсы и классы SAX2 собраны в пакеты `org.xml.sax`, `org.xml.sax.ext`, `org.xml.sax.helpers`, `javax.xml.parsers`. Рассмотрим их подробнее.

Анализ документов XML с помощью SAX2

Основу SAX2 составляет интерфейс `org.xml.sax.ContentHandler`, описывающий методы обработки событий: начала документа, появления открывающего тега, появление тела элемента, появление закрывающего тега, окончание документа. При возникновении такого события SAX2 обращается к методу-обработчику события, передавая ему аргументы, содержащие информацию о событии. Дело разработчика — реализовать эти методы, обеспечив правильный анализ документа.

В начале обработки документа вызывается метод

```
public void startDocument();
```

В нем можно задать начальные действия по обработке документа.

При появлении символа "<", начинающего открывающий тег, вызывается метод

```
public void startElement(String uri, String name,
                         String qname, Attributes attrs);
```

В метод передаются три имени, два из которых связаны с пространством имен: идентификатор пространства имен `uri`, локальное имя тега без префикса `name` и расширенное имя с префиксом `qname`, а также атрибуты открывающего тега элемента `attrs`, если они есть. Если пространство имен не определено, то значения первого и второго аргументов равны `null`. Если нет атрибутов, то передается ссылка на пустой объект `attrs`.

При появлении символов "</", начинающих закрывающий тег, вызывается метод

```
public void endElement(String uri, String name, String qname);
```

При появлении строки символов вызывается метод

```
public void characters(char[] ch, int start, int length);
```

В него передается массив символов `ch`, индекс начала строки символов `start` в этом массиве и количество символов `length`.

При появлении в тексте документа инструкции по обработке вызывается метод

```
public void processingInstruction(String target, String data);
```

В метод передается имя программы-обработчика `target` и дополнительные сведения `data`.

При появлении пробельных символов, которые должны быть пропущены, вызывается метод

```
public void ignorableWhitespace(char[] ch, int start, int length);
```

В него передается массив `ch` идущих подряд пробельных символов, индекс начала символов в массиве `start` и количество символов `length`.

Интерфейс `org.xml.sax.ContentHandler` реализован классом `org.xml.sax.helpers.DefaultHandler`. В нем сделана пустая реализация всех методов. Разработчику остается реализовать только те методы, которые ему нужны.

Применим методы SAX2 для обработки нашей адресной книжки. Запись документа на языке XML удобна для выявления структуры документа, но неудобна для работы с документом в объектно-ориентированной среде. Поэтому чаще всего содержимое документа XML представляется в виде одного или нескольких объектов, называемых *объектами данных* JDO (Java Data Objects). Эта операция называется *связыванием данных* (data binding) с объектами JDO.

Свяжем содержимое нашей адресной книжки с объектами Java. Для этого сначала опишем классы Java (листинги 28.7 и 28.8), которые представят содержимое адресной книги.

Листинг 28.7. Класс, описывающий адрес

```
public class Address{

  private String street, city, zip, type = "город";

  public Address(){}

public String getStreet(){ return street; }
  public void setStreet(String street){ this.street = street; }

public String getCity(){ return city; }
  public void setCity(String city){ this.city = city; }
```

```
  public String getZip(){ return zip; }
     public void setZip(String zip){ this.zip = zip; }

  public String getType(){ return type; }
     public void setType(String type){ this.type = type; }

  public String toString(){
    return "Address: " + street + " " +  city + " " + zip;
  }
}
```

Листинг 28.8. Класс, описывающий запись адресной книжки

```
   public class Person{

  private String firstName, secondName, surname, birthday;
  private Vector address;
  private Vector workPhone;
     private Vector homePhone;

     public Person(){}

  public Person(String firstName, String secondName, String surname){
    this.firstName = firstName;
    this.secondName = secondName;
    this.surname = surname;
     }

  public String getFirstName(){ return firstName; }
  public void setFirstName(String firstName){
     this.firstName = firstName;
     }

  public String getSecondName(){ return secondName; }
  public void setSecondName(String secondName){
     this.secondName = secondName;
     }

  public String getSurname(){ return surname; }
  public void setSurname(String surname){
      this.surname = surname;
     }

  public String getBirthday(){ return birthday; }
  public void setBirthday(String birthday){
      this.birthday = birthday;
     }
```

```
  public void addAddress(Address addr){
    if (address == null) address = new Vector();
    address.add(addr);
  }
     public Vector getAddress(){ return address; }

  public void removeAddress(Address addr){
    if (address != null) address.remove(addr);
     }

  public void addWorkPhone(String phone){
    if (workPhone == null) workPhone = new Vector();
    workPhone.add(new Integer(phone));
  }
     public Vector getWorkPhone(){ return workPhone; }

  public void removeWorkPhone(String phone){
    if (workPhone != null) workPhone.remove(new Integer(phone));
     }

  public void addHomePhone(String phone){
    if (homePhone == null) homePhone = new Vector();
    homePhone.add(new Integer(phone));
  }
     public Vector getHomePhone(){ return homePhone; }

  public void removeHomePhone(String phone){
    if (homePhone != null) homePhone.remove(new Integer(phone));
     }

  public String toString(){
    return "Person: " + surname;
  }
}
```

После определения классов Java, в экземпляры которых будет занесено содержимое адресной книжки, напишем программу, читающую адресную книжку и связывающую ее с объектами Java.

В листинге 28.9 приведен пример класса-обработчика `NotebookHandler` для адресной книжки, описанной в листинге 28.2. Методы класса `NotebookHandler` анализируют содержимое адресной книжки и помещают его в вектор, составленный из объектов класса `Person`, описанного в листинге 28.8.

Листинг 28.9. Класс-обработчик документа XML

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import javax.xml.parsers.*;
```

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

   public class NotebookHandler extends DefaultHandler{

   static final String JAXP_SCHEMA_LANGUAGE =
        "http://java.sun.com/xml/jaxp/properties/schemaLanguage";

   static final String W3C_XML_SCHEMA =
        "http://www.w3.org/2001/XMLSchema";

   Person person;
   Address address;
   static Vector pers = new Vector();
   boolean inBirthday, inStreet, inCity, inZip,
              inWorkPhone, inHomePhone;

  public void startElement(String uri, String name,
                           String qname, Attributes attrs)
         throws SAXException{
    if (qname.equals("name"))
       person = new Person(attrs.getValue("first"),
          attrs.getValue("second"), attrs.getValue("surname"));
    else if (qname.equals("birthday"))
       inBirthday = true;
    else if (qname.equals("address"))
       address = new Address();
    else if (qname.equals("street"))
       inStreet = true;
    else if (qname.equals("city")){
       inCity = true;
       if (attrs != null) address.setType(attrs.getValue("type"));
    }else if (qname.equals("zip"))
       inZip = true;
    else if (qname.equals("work"))
       inWorkPhone = true;
    else if (qname.equals("home"))
       inHomePhone = true;
     }

  public void characters(char[] buf, int offset, int len)
         throws SAXException{
       String s = new String(buf, offset, len);

    if (inBirthday){
       person.setBirthday(s);
       inBirthday = false;
    }else if (inStreet){
```

```
      address.setStreet(s);
      inStreet = false;
   }else if (inCity){
      address.setCity(s);
      inCity = false;
   }else if (inZip){
      address.setZip(s);
      inZip = false;
   }else if (inWorkPhone){
      person.addWorkPhone(s);
      inWorkPhone = false;
   }else if (inHomePhone){
      person.addHomePhone(s);
      inHomePhone = false;
   }
    }

public void endElement(String uri, String name, String qname)
       throws SAXException{
   if (qname.equals("address")){
       person.addAddress(address);
       address = null;
   }else if (qname.equals("person")){
       pers.add(person);
       person = null;
   }
    }

public static void main(String[] args){
   if (args.length < 1){
      System.err.println("Usage: java Notebook ntb.xml");
      System.exit(1);
   }
   try{
        NotebookHandler handler = new NotebookHandler();

        SAXParserFactory fact = SAXParserFactory.newInstance();

     fact.setNamespaceAware(true);
     fact.setValidating(true);


        SAXParser saxParser = fact.newSAXParser();

        saxParser.setProperty(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);

        File f = new File(args[0]);

        saxParser.parse(f, handler);
```

```
      for (int k = 0; k < pers.size(); k++)
           System.out.println(((Person)pers.get(k)).getSurname());

         }catch(SAXNotRecognizedException x){

         System.err.println("Неизвестное свойство: " +
                            JAXP_SCHEMA_LANGUAGE);
            System.exit(1);

       }catch(Exception ex){

      System.err.println(ex);
    }
     }

  public void warning(SAXParseException ex){
    System.err.println("Warning: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
                       "  col = " + ex.getColumnNumber());
     }

  public void error(SAXParseException ex){
    System.err.println("Error: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
                       "  col = " + ex.getColumnNumber());
     }

  public void fatalError(SAXParseException ex){
    System.err.println("Fatal error: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
                       "  col = " + ex.getColumnNumber());
  }
}
```

После того как класс-обработчик написан, проанализировать документ очень легко. Стандартные действия приведены в методе `main()` программы листинга 28.9.

Поскольку реализация парсера сильно зависит от его программного окружения, SAX-парсер — объект класса `SAXParser` — создается не конструктором, а фабричным методом `newSAXParser()`.

Объект-фабрика, в свою очередь, формируется методом `newInstance()`. Далее можно методом

```
void setFeature(String name, boolean value);
```

установить свойства парсеров, создаваемых этой фабрикой. Например, после

```
fact.setFeature("http://xml.org/sax/features/namespace-prefixes", true);
```

парсеры, создаваемые фабрикой `fact`, будут учитывать префиксы имен тегов и атрибутов.

Список таких свойств можно посмотреть в документации Java API в описании пакета `org.xml.sax` или на сайте проекта SAX **http://www.saxproject.org/**. Следует учитывать, что не все парсеры полностью выполняют эти свойства.

Если к объекту-фабрике применить метод

```
void setValidating(true);
```

как это сделано в листинге 28.9, то она будет производить парсеры, проверяющие структуру документа. Если применить метод

```
void setNamespaceAware(true);
```

то объект-фабрика будет производить парсеры, учитывающие пространства имен.

После того как объект-парсер создан, остается только применить метод `parse()`, передав ему имя анализируемого файла и экземпляр класса-обработчика событий.

В классе `javax.xml.parsers.SAXParser` есть десяток методов `parse()`. Кроме метода `parse(File, DefaultHandler)`, использованного в листинге 28.9, существуют еще методы, позволяющие извлечь документ из входного потока класса `InputStream`, объекта класса `InputSource`, адреса URI или из специально созданного источника класса `InputSource`.

Методом `setProperty()` можно задать различные свойства парсера. В листинге 28.9 этот метод использован для того, чтобы парсер проверял правильность документа с помощью схемы XSD. Данная возможность включена в JAXP, начиная с версии JAXP 1.2.

Если парсер выполняет проверки, т. е. применен метод `setValidating(true)`, то имеет смысл сделать развернутые сообщения об ошибках. Это предусмотрено интерфейсом `ErrorHandler`. Он различает предупреждения, ошибки и фатальные ошибки и описывает три метода, которые автоматически выполняются при появлении ошибки соответствующего вида:

```
public void warning(SAXParserException ex);
public void error(SAXParserException ex);
public void fatalError(SAXParserException ex);
```

Класс `DefaultHandler` делает пустую реализацию этого интерфейса. При расширении данного класса можно сделать реализацию одного или всех методов интерфейса `ErrorHandler`. Пример такой реализации приведен в листинге 28.9. Класс `SAXParserException` хранит номер строки и столбца проверяемого документа, в котором замечена ошибка. Их можно получить методами `getLineNumber()` и `getColumnNumber()`, как сделано в листинге 28.9.

Связывание данных XML с объектами Java

В приведенном примере мы сами создали классы `Address` и `Person`, представляющие документ XML. Поскольку структура документа XML четко определена, можно разработать стандартные правила связывания данных XML с объектами Java и создать программные средства для их реализации.

Корпорация Sun Microsystems разрабатывает пакет интерфейсов и классов JAXB (Java Architecture for XML Binding), облегчающих связывание данных. Он входит в стандартную поставку Java SE, а также может быть скопирован с сайта **http://jaxb.dev.java.net/**. Для ра-

боты с пакетом JAXB анализируемый документ XML обязательно должен быть снабжен описанием на языке XSD.

В состав пакета JAXB входит компилятор `xjc` (XML-Java Compiler). Он просматривает схему XSD и строит по нему объекты Java в оперативной памяти, а также создает исходные файлы объектов Java. Например, после выполнения команды

```
$ xjc -roots notebook ntb.xsd -d sources
```

в которой ntb.xsd — файл листинга 28.4 — в каталоге sources (по умолчанию в текущем каталоге) будут созданы файлы Address.java, Name.java, Notebook.java, Person.java, PhoneList.java с описаниями объектов Java.

Флаг `-roots` показывает один или несколько корневых элементов, разделенных запятыми.

Созданные компилятором `xjc` исходные файлы обычным образом, с помощью компилятора `javac`, компилируются в классы Java.

Получив объекты данных, можно перенести в них содержимое документа XML методом `unmarshal()`, который создает дерево объектов, или, наоборот, записать объекты Java в документы XML методом `marshal()`. Эти методы уже записаны в созданный компилятором `xjc` класс корневого элемента, в примере это класс `Notebook`.

Объекты данных JDO

Задачу связывания данных естественно обобщить — связывать объекты Java не только с документами XML, но и с текстовыми файлами, реляционными или объектными базами данных, другими хранилищами данных.

Корпорация Sun Microsystems опубликовала спецификацию JDO, сейчас уже версии 2.1, и разработала интерфейсы для работы с JDO. Спецификация JDO рассматривает более широкую задачу связывания данных, полученных не только из документа XML, но и из любого источника данных, называемого *информационной системой предприятия* (Enterprise Information System, EIS). Спецификация описывает два набора классов и интерфейсов:

- JDO SPI (JDO Service Provider Interface) — вспомогательные классы и интерфейсы, которые следует реализовать в сервере приложений для обращения к источнику данных, создания объектов, обеспечения их сохранности, выполнения транзакций, проверки прав доступа к объектам; эти классы и интерфейсы составляют пакет `javax.jdo.spi`;
- JDO API (JDO Application Programming Interface) — интерфейсы, предоставляемые пользователю для доступа к объектам, управления транзакциями, создания и удаления объектов; эти интерфейсы собраны в пакет `javax.jdo`.

Есть много коммерческих и свободно распространяемых реализаций интерфейсов JDO.

Сообщество Apache Software Foundition назвало свою реализацию Apache JDO. Эту разработку можно посмотреть по адресу **http://db.apache.org/jdo/index.html**.

Еще одна популярная, свободно распространяемая реализация, — JPOX (Java Persistent Objects), **http://www.jpox.org**/.

Фирма Prism Technologies, **http://www.prismtechnologies.com/**, выпускает продукт OpenFusion JDO.

Компания SolarMetric, **http://www.solarmetric.com/**, выпускает свою реализацию спецификации JDO под названием Kodo JDO. Ее можно встроить в серверы приложений WebLogic, WebSphere, JBoss.

Некоторые фирмы, не дожидаясь выхода спецификации JDO, разработали свои реализации JDO, более или менее соответствующие спецификации. Наиболее известна свободно распространяемая разработка, названная Castor. Ее можно посмотреть по адресу **http://www.castor.org/**.

С помощью Castor можно предельно упростить связывание данных. Например, создание объекта Java из простого документа XML, если отвлечься от проверок и обработки исключительных ситуаций, выполняется одним действием:

```
Person person = (Person)Unmarshaller.unmarshal(
                    Person.class, new FileReader("person.xml"));
```

Обратно, сохранение объекта Java в виде документа XML в самом простом случае выглядит так:

```
Marshaller.marshall(person, new FileWriter("person.xml"));
```

В более сложных случаях надо написать файл XML, аналогичный схеме XSD, с указаниями по *связыванию данных* (mapping file).

Анализ документов XML с помощью DOM API

Как видно из предыдущих разделов, SAX-парсер читает документ только один раз, отмечая появляющиеся по ходу чтения открывающие теги, содержимое элементов и закрывающие теги. Этого достаточно для связывания данных, но неудобно для редактирования документа.

Консорциум W3C разработал спецификации и набор интерфейсов DOM (Document Object Model), которые можно посмотреть на сайте этого проекта **http://www.w3.org/DOM/**. Методами этих интерфейсов документ XML можно загрузить в оперативную память в виде дерева объектов. Это позволяет не только анализировать документ анализаторами, основанными на структуре дерева, но и менять дерево, добавляя или удаляя объекты из дерева. Кроме того, можно обращаться непосредственно к каждому объекту в дереве и не только читать, но и изменять информацию, хранящуюся в нем. Но все это требует большого объема оперативной памяти для загрузки большого дерева.

Корпорация Sun Microsystems реализовала интерфейсы DOM в пакетах `javax.xml.parsers` и `org.w3c.dom`, входящих в состав пакета JAXP. Воспользоваться этой реализацией очень легко:

```
DocumentBuilderFactory fact =
        DocumentBuilderFactory.newInstance();
DocumentBuilder builder = fact.newDocumentBuilder();
Document doc = builder.parse("ntb.xml");
```

Метод `parse()` строит дерево объектов и возвращает ссылку на него в виде объекта типа `Document`. В классе `DocumentBuilder` есть несколько методов `parse()`, позволяющих за-

грузить файл с адреса URL, из входного потока, как объект класса `File` или из источника класса `InputSource`.

Интерфейс `Document`, расширяющий интерфейс `Node`, описывает дерево объектов документа в целом. Интерфейс `Node` — основной интерфейс в описании структуры дерева — описывает узел дерева. У него есть еще один наследник — интерфейс `Element`, описывающий лист дерева, соответствующий элементу документа XML. Как видно из структуры наследования этих интерфейсов, и само дерево, и его лист считаются узлами дерева. Каждый атрибут элемента дерева описывается интерфейсом `Attr`. Еще несколько интерфейсов — `CDATASection`, `Comment`, `Text`, `Entity`, `EntityReference`, `ProcessingInstruction`, `Notation` — описывают разные типы элементов XML.

На рис. 28.2 показано начало дерева объектов, построенного по документу, приведенному в листинге 28.2. Обратите внимание на то, что текст, находящийся в теле элемента, хранится в отдельном узле дерева — потомке узла элемента. Для каждого атрибута открывающего тега тоже создается отдельный узел.

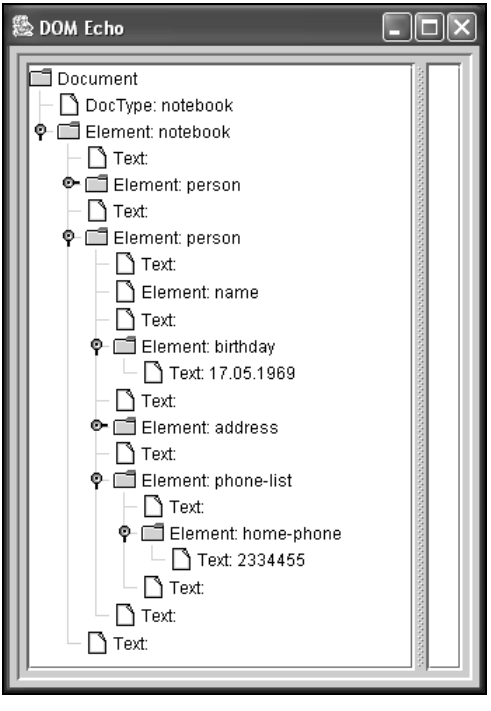


Рис. 28.2. Дерево объектов документа XML

Интерфейс *Node*

Интерфейс `Node` описывает тип узла одной из следующих констант:

- `ATTRIBUTE_NODE` — узел типа `Attr`, содержит атрибут элемента;
- `CDATA_SECTION_NODE` — узел типа `CDADASection`, содержит данные типа `CDATA`;

- COMMENT_NODE — узел типа Comment, содержит комментарий;
- DOCUMENT_FRAGMENT_NODE — в узле типа DocumentFragment находится фрагмент документа;
- DOCUMENT_NODE — корневой узел типа Document;
- DOCUMENT_TYPE_NODE — узел типа Document;
- ELEMENT_NODE — узел является листом дерева типа Element;
- ENTITY_NODE — в узле типа Entity хранится сущность ENTITY;
- ENTITY_REFERENCE_NODE — в узле типа EntityReference хранится ссылка на сущность;
- NOTATION_NODE — в узле хранится нотация типа Notation;
- PROCESSING_INSTRUCTION_NODE — узел типа ProcessingInstruction, содержит инструкцию по обработке;
- TEXT_NODE — в узле типа Text хранится текст.

Методы интерфейса Node описывают действия с узлом дерева:

- public short getNodeType() — возвращает тип узла;
- public String getNodeName() — возвращает имя узла;
- public String getNodeValue() — возвращает значение, хранящееся в узле;
- public boolean hasAttributes() — выполняет проверку существования атрибутов у элемента XML, хранящегося в узле в виде объекта типа NamedNodeMap, если это узел типа Element;
- public NamedNodeMap getAttributes() — возвращает атрибуты; метод возвращает null, если у элемента нет атрибутов;
- public boolean hasChildNodes() — проверяет, есть ли у данного узла узлы-потомки;
- public NodeList getChildNodes() — возвращает список узлов-потомков в виде объекта типа NodeList;
- public Node getFirstChild() — возвращает первый узел в списке узлов-потомков;
- public Node getLastChild() — возвращает последний узел в списке узлов-потомков;
- public Node getParentNode() — возвращает родительский узел;
- public Node getPreviousSibling() — возвращает предыдущий узел, имеющий того же предка, что и данный узел;
- public Node getNextSibling() — возвращает следующий узел, имеющий того же предка, что и данный узел;
- public Document getOwnerDocument() — возвращает ссылку на весь документ.

Следующие методы позволяют изменить дерево объектов:

- public Node appendChild(Node newChild) — добавляет новый узел-потомок newChild;

- `public Node insertBefore(Node newChild, Node refChild)` — вставляет новый узел-потомок `newChild` перед существующим потомком `refChild`;
- `public Node replaceChild(Node newChild, Node oldChild)` — заменяет один узел-потомок `oldChild` новым узлом `newChild`;
- `public Node removeChild(Node child)` — удаляет узел-потомок.

Интерфейс *Document*

Интерфейс `Document` добавляет к методам своего предка `Node` методы работы с документом в целом:

- `public DocumentType getDocType()` — возвращает общие сведения о документе в виде объекта типа `DocumentType`;
- `getName()`, `getEntitied()`, `getNotations()` и другие методы интерфейса `DocumentType` возвращают конкретные сведения о документе;
- `public Element getDocumentElement()` — возвращает корневой элемент дерева объектов;
- `public NodeList getElementsByTagName(String name);`

  `public NodeList getElementsByTagNameNS(String uri, String qname);`

  `public Element getElementById(String id);`

  возвращают все элементы с указанным именем `tag` без префикса или с префиксом, а также элемент, определяемый значением атрибута с именем `ID`.

Несколько методов позволяют изменить структуру и содержимое дерева объектов:

- `public Element createElement(String name)` — создает новый пустой элемент по его имени;
- `public Element createElementNS(String uri, String name)` — создает новый пустой элемент по имени с префиксом;
- `public CDATASection createCDATASection(String name)` — создает узел типа `CDATA_SECTION_NODE`;
- `public EntityReference createEntityReference(String name)` — создает узел типа `ENTITY_REFERENCE_NODE`;
- `public ProcessingInstruction createProcessingInstruction(String name)` — создает узел типа `PROCESSING_INSTRUCTION_NODE`;
- `public TextNode createTextNode(String name)` — создает узел типа `TEXT_NODE`;
- `public Attr createAttribute(String name)` — создает узел-атрибут с именем `name`;
- `public Attr createAttributeNS(String uri, String name)` — аналогично;
- `public Comment createComment(String comment)` — создает узел-комментарий;
- `public DocumentFragment createDocumentFragment()` — создает пустой документ — фрагмент данного документа с целью его дальнейшего заполнения;

- `public Node importNode(Node importedNode, boolean deep)` — вставляет созданный узел, а значит, и все его поддерево, в дерево документа. Этим методом можно соединить два дерева объектов. Если второй аргумент равен `true`, то рекурсивно вставляется все поддерево.

Интерфейс *Element*

Интерфейс `Element` добавляет к методам своего предка `Node` методы работы с атрибутами открывающего тега элемента XML и методы, позволяющие обратиться к вложенным элементам. Только один метод

```
public String getTagName();
```

дает сведения о самом элементе, а именно имя элемента.

Прежде чем получить значение атрибута с именем `name`, надо проверить его наличие методами

```
public boolean hasAttribute(String name);
public boolean hasAttributeNS(String uri, String name);
```

Второй из этих методов учитывает пространство имен с именем `uri`, записанным в виде строки URI; имя `name` должно быть полным, с префиксом.

Получить атрибут в виде объекта типа `Attr` или его значение в виде строки по имени `name` с учетом префикса или без него можно методами

```
public Attr getAttributeNode(String name);
public Attr getAttributeNodeNS(String uri, String name);
public String getAttribute(String name);
public String getAttributeNS(String uri, String name);
```

Удалить атрибут можно методами

```
public Attr removeAttributeNode(Attr name);
public void removeAttribute(String name);
public void removeAttributeNS(String uri, String name);
```

Установить значение атрибута можно методами

```
public void setAttribute(String name, String value);
public void setAttributeNS(String uri, String name, String value);
```

Добавить атрибут в качестве потомка можно методами

```
public Attr setAttributeNode(String name);
public Attr setAttributeNodeNS(Attr name);
```

Два метода позволяют получить список узлов-потомков:

```
public NodeList getElrmentsByTagName(String name);
public NodeList getElrmentsByTagNameNS(String uri, String name);
```

Итак, методы перечисленных интерфейсов позволяют перемещаться по дереву, менять структуру дерева, просматривать информацию, хранящуюся в узлах и листьях дерева, и изменять ее.

Приведем пример работы с деревом объектов, построенным по документу XML. Добавим в адресную книжку листинга 28.2 новый рабочий или домашний телефон Сидоровой. Это действие записано в листинге 28.10.

Листинг 28.10. Анализ адресной книжки с помощью DOM API

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
class ErrHand implements ErrorHandler{
  public void warning(SAXParseException ex){
    System.err.println("Warning: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
                       "  col = " + ex.getColumnNumber());
     }

  public void error(SAXParseException ex){
    System.err.println("Error: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
                       "  col = " + ex.getColumnNumber());
     }

  public void fatalError(SAXParseException ex){
    System.err.println("Fatal error: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
                       "  col = " + ex.getColumnNumber());
  }
   }

   public class TreeProcessDOM{

  static final String JAXP_SCHEMA_LANGUAGE =
         "http://java.sun.com/xml/jaxp/properties/schemaLanguage";

  static final String W3C_XML_SCHEMA =
         "http://www.w3.org/2001/XMLSchema";

  public static void main(String[] args) throws Exception{
    if (args.length != 3){
      System.err.println("Usage: java TreeProcessDOM " +
            "<file-name>.xml {work|home} <phone>");
      System.exit(-1);
       }

       DocumentBuilderFactory fact = DocumentBuilderFactory.newInstance();
```

```
    fact.setNamespaceAware(true);

    fact.setValidating(true);

 try{
    fact.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
 }catch(IllegalArgumentException x){
    System.err.println("Неизвестное свойство: " +
                       JAXP_SCHEMA_LANGUAGE);
    System.exit(-1);
    }

    DocumentBuilder builder = fact.newDocumentBuilder();

    builder.setErrorHandler(new ErrHand());

    Document doc = builder.parse(args[0]);

    NodeList list = doc.getElementsByTagName("notebook");

    int n = list.getLength();

 if (n == 0){
   System.err.println("Документ пуст.");
   System.exit(-1);
    }

    Node thisNode = null;

 for (int k = 0; k < n; k++){
   thisNode = list.item(k);
   String elemName = null;
   if (thisNode.getFirstChild() instanceof Element){
      elemName = (thisNode.getFirstChild()).getNodeName();
   if (elemName.equals("name")){
      if (!thisNode.hasAttributes()){
        System.err.println("Атрибуты отсутствуют " + elemName);
        System.exit(1);
      }
      NamedNodeMap attrs = thisNode.getAttributes();
      Node attr = attrs.getNamedItem("surname");
      if (attr instanceof Attr)
         if (((Attr)attr).getValue().equals("Сидорова")) break;
   }
  }
   }

NodeList topics = ((Element)thisNode)
          .getElementsByTagName("phone-list");
   Node newNode;
```

```
    if (args[1].equals("work"))
        newNode = doc.createElement("work-phone");
      else newNode = doc.createElement("home-phone");

      Text textNode = doc.createTextNode(args[2]);

      newNode.appendChild(textNode);

    thisNode.appendChild(newNode);
   }
}
```

Дерево объектов можно вывести на экран дисплея, например, как объект класса `JTree` — одного из компонентов графической библиотеки Swing. Именно так сделано на рис. 28.2. Для вывода применена программа DomEcho из электронного учебника "Web Services Tutorial". Исходный текст программы слишком велик, чтобы приводить его здесь, но его можно посмотреть по адресу **http://java.sun.com/webservices/tutorial.html**. В состав парсера Xerces в качестве примера анализа документа в раздел samples/ui/ входит программа `TreeView`, которая тоже показывает дерево объектов в виде дерева `JTree` библиотеки Swing.

Другие DOM-парсеры

Модель дерева объектов DOM была первоначально разработана группой OMG (Object Management Group) в рамках языка IDL (Interface Definition Language) без учета особенностей Java. Только потом она была переведена на Java консорциумом W3C в виде интерфейсов и классов, составивших пакет `org.w3c.dom`. Этим объясняется, в частности, широкое применение в DOM API интерфейсов и фабричных методов вместо классов и конструкторов.

Данное неудобство привело к появлению других разработок.

Участники общественного проекта JDOM не стали реализовать модель DOM, а разработали свою модель дерева объектов, получившую название JDOM. Они выпускают одноименный свободно распространяемый программный продукт, с которым можно ознакомиться на сайте проекта **http://www.jdom.org/**. Этот продукт широко используется для обработки документов XML средствами Java.

Участники другого общественного проекта — dom4j — приняли модель W3C DOM, но упростили и упорядочили DOM API. С их одноименным продуктом dom4j можно ознакомиться на сайте **http://www.dom4j.org/**.

Преобразование дерева объектов в XML

Итак, дерево объектов DOM построено надлежащим образом. Теперь надо его преобразовать в документ XML, страничку HTML, документ PDF или объект другого типа. Средства для выполнения такого преобразования составляют третью часть набора JAXP — пакеты `javax.xml.transform`, `javax.xml.transform.dom`, `javax.xml.transform.sax`, `javax.xml.transform.stream`, которые представляют собой реализацию языка описания таблиц стилей для преобразований XSLT (XML Stylesheet Language for Transformations) средствами Java.

Язык XSLT разработан консорциумом W3 как одна из трех частей, составляющих язык записи таблиц стилей XSL (XML Stylesheet Language). Все материалы по XSL можно посмотреть на сайте проекта по адресу **http://www.w3.org/Style/XSL/**.

Интерфейсы и классы, входящие в пакеты `javax.xml.transform.*`, управляют процессором XSLT, в качестве которого выбран процессор Xalan, разработанный в рамках проекта Apache Software Foundation, **http://xml.apache.org/xalan-j/**.

Исходный объект преобразования должен иметь тип `Source`. Интерфейс `Source` определяет всего два метода доступа к идентификатору объекта:

```
public String getSystemId();
public void setSystemId(String id);
```

У интерфейса `Source` есть три реализации. Класс `DOMSource` подготавливает к преобразованию дерево объектов DOM, класс `SAXSource` подготавливает SAX-объект, класс `StreamSource` — подготавливает простой поток данных. В конструкторы этих классов заносится ссылка на исходный объект — для конструктора класса `DOMSource` это узел дерева, для конструктора класса `SAXSource` — имя файла, для конструктора класса `StreamSource` — входной поток. Методы перечисленных классов позволяют задать дополнительные свойства исходных объектов преобразования.

Результат преобразования описывается интерфейсом `Result`. Он тоже определяет точно такие же методы доступа к идентификатору объекта-результата, как и интерфейс `Source`. У него тоже есть три реализации — классы `DOMResult`, `SAXResult` и `StreamResult`. В конструкторы этих классов заносится ссылка на выходной объект. В первом случае это узел дерева, во втором — объект типа `ContentHandler`, в третьем — файл, в который будет занесен результат преобразования, или выходной поток.

Само преобразование выполняется объектом класса `Transformer`. Вот стандартная схема преобразования дерева объектов DOM в документ XML, записываемый в файл.

```
TransformerFactory transFactory = TransformerFactory.newInstance();
Transformer transformer = transFactory.newTransformer();
DOMSource source = new DOMSource(document);
File newXMLFile = new File("ntb1.xml");
FileOutputStream fos = new FileOutputStream(newXMLFile);
StreamResult result = new StreamResult(fos);
transformer.transform(source, result);
```

Вначале методом `newInstance()` создается экземпляр `transFactory` фабрики объектов-преобразователей. Методом

```
public void setAttrbute(String name, String value);
```

класса `TransformerFactory` можно установить некоторые атрибуты этого экземпляра. Имена и значения атрибутов зависят от реализации фабрики.

С помощью фабрики преобразователей создается объект-преобразователь класса `Transformer`. При формировании этого объекта в него можно занести объект, содержащий правила преобразования, например таблицу стилей XSL.

В созданный объект класса `Transformer` методом

```
public void setParameter(String name, String value);
```

можно занести параметры преобразования, а методами

```
public void setOutputProperties(Properties out);
public void setOutputProperty(String name, String value);
```

легко определить свойства преобразованного объекта. Имена свойств `name` задаются константами, которые собраны в специально определенный класс `OutputKeys`, содержащий только эти константы. Вот их список:

- `CDATA_SECTION_ELEMENTS` — список имен секций `CDATA` через пробел;
- `DOCTYPE_PUBLIC` — открытый идентификатор `PUBLIC` преобразованного документа;
- `DOCTYPE_SYSTEM` — системный идентификатор `SYSTEM` преобразованного документа;
- `ENCODING` — кодировка символов преобразованного документа, значение атрибута `encoding` объявления XML;
- `INDENT` — делать ли отступы в тексте преобразованного документа. Значения этого свойства `"yes"` или `"no"`;
- `MEDIA_TYPE` — MIME-тип содержимого преобразованного документа;
- `METHOD` — метод вывода, одно из значений: `"xml"`, `"html"` или `"text"`;
- `OMIT_XML_DECLARATION` — не включать объявление XML. Значения `"yes"` или `"no"`;
- `STANDALONE` — отдельный или вложенный документ, значение атрибута `standalone` объявления XML. Значения `"yes"` или `"no"`;
- `VERSION` — номер версии XML для атрибута `version` объявления XML.

Например, можно задать кодировку символов преобразованного документа следующим методом:

```
transformer.setOutputProperty(OutputKeys.ENCODING, "Windows-1251");
```

Затем в приведенном примере по дереву объектов `document` типа `Node` создается объект класса `DOMSource` — упаковка дерева объектов для последующего преобразования. Тип аргумента конструктора этого класса `Node`, откуда видно, что можно преобразовать не все дерево, а какое-либо его поддерево, записав в конструкторе класса `DOMSource` корневой узел поддерева.

Наконец, определяется результирующий объект `result`, связанный с файлом newCourses.xml, и осуществляется преобразование методом `transform()`.

Более сложные преобразования выполняются с помощью таблицы стилей XSL.

Таблицы стилей XSL

В документах HTML часто применяются таблицы стилей CSS (Cascading Style Sheet), задающие общие правила оформления документов HTML: цвет, шрифт, заголовки. Выполнение этих правил придает документам единый стиль оформления.

Для документов XML, в которых вообще не определяются правила визуализации, идея применить таблицы стилей оказалась весьма плодотворной. Таблицы стилей для документов XML записываются на специально сделанной реализации языка XML, названной XSL (XML Stylesheet Language). Все теги документов XSL относятся к пространству имен с идентификатором `http://www.w3.org/1999/XSL/Transform`. Обычно они записываются с пре-

фиксом `xsl`. Если принят этот префикс, то корневой элемент таблицы стилей XSL будет называться `<xsl:stylesheet>`.

Простейшая таблица стилей выглядит так, как записано в листинге 28.11.

Листинг 28.11. Простейшая таблица стилей XSL

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet  version="1.0"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="text" encoding="CP866"/>
</xsl:stylesheet>
```

Здесь только определяется префикс пространства имен `xsl` и правила вывода, а именно выводится "плоский" текст, на что показывает значение `text` (другие значения — `html` и `xml`), в кодировке CP866. Такая кодировка выбрана для вывода кириллицы на консоль MS Windows. Объект класса `Transformer`, руководствуясь таблицей стилей листинга 28.11, просто выводит тела элементов так, как они записаны в документе XML, т. е. просто удаляет теги вместе с атрибутами, оставляя их содержимое.

Эту таблицу стилей записываем в файл, например, simple.xsl. Ссылку на таблицу стилей можно поместить в документ XML как одну из инструкций по обработке:

```
<?xml version="1.0" encoding="Windows-1251"?>
<?xml-stylesheet type="text/xsl" href="simple.xsl"?>
<notebook>
<!-- Продолжение адресной книжки -->
```

После этого XML-парсер, если он, кроме того, является XSLT-процессором, выполнит преобразование, заданное в файле simple.xsl.

Другой путь — использовать таблицу стилей при создании объекта-преобразователя, например, так, как записано в листинге 28.12.

Листинг 28.12. Консольная программа преобразования документа XML

```
import java.io.*;
import javax.xml.transform.*;
   import javax.xml.transform.stream.*;

   public class SimpleTransform{

   public static void main(String[] args) throws TransformerException{
     if (args.length != 2){
        System.out.println("Usage: " +
            "java SimpleTransform xmlFileName xsltFileName");
        System.exit(1);
        }

     File xmlFile = new File(args[0]);
        File xsltFile = new File(args[1]);
```

```
    Source xmlSource = new StreamSource(xmlFile);
    Source xsltSource = new StreamSource(xsltFile);
       Result result = new StreamResult(System.out);

    TransformerFactory transFact = TransformerFactory.newInstance();
    Transformer trans = transFact.newTransformer(xsltSource);
    trans.transform(xmlSource, result);
  }
}
```

После компиляции набираем в командной строке

```
java SimpleTransform ntb.xml simple.xsl
```

и получаем на экране дисплея содержимое тегов документа ntb.xml.

В более сложных случаях просто изменяем таблицу стилей. Например, если мы хотим получить содержимое документа на консоли, то таблицу стилей для объекта-преобразователя класса `Transformer` надо записать так, как показано в листинге 28.13.

Листинг 28.13. Таблица стилей для показа адресной книжки

```
   <?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet  version="1.0"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

     <xsl:output method="text" encoding="CP866" />

  <xsl:template match="person">
       <xsl:apply-templates />
     </xsl:template>

  <xsl:template match="name">
     <xsl:value-of select="@first" />  <xsl:text> </xsl:text>
     <xsl:value-of select="@second" /> <xsl:text> </xsl:text>
     <xsl:value-of select="@surname" />
     </xsl:template>

  <xsl:template match="address">
     <xsl:value-of select="street" /> <xsl:text> </xsl:text>
     <xsl:value-of select="city" />   <xsl:text> </xsl:text>
     <xsl:value-of select="zip" />
     </xsl:template>

  <xsl:template match="phone-list">
     <xsl:value-of select="work-phone" /> <xsl:text>&#xA;</xsl:text>
     <xsl:value-of select="home-phone" /> <xsl:text>&#xA;</xsl:text>
     </xsl:template>

</xsl:stylesheet>
```

Мы не будем в этой книге заниматься языком XSL — одно его описание будет толще всей этой книги. На русский язык переведена "библия" XSLT [20]. Ее автор Майкл Кэй (Michael H. Kay) создал и свободно распространяет популярный XSLT-процессор Saxon, **http://saxon.sourceforge.net/**.

Преобразование документа XML в HTML

С помощью языка XSL и методов класса `Transformer` можно легко преобразовать документ XML в документ HTML. Достаточно написать соответствующую таблицу стилей. В листинге 28.14 показано, как это можно сделать для адресной книжки листинга 28.2.

Листинг 28.14. Таблица стилей для преобразования XML в HTML

```
   <?xml version="1.0" encoding="Windows-1251"?>

<xsl:stylesheet  version="1.0"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

     <xsl:output method="html" encoding="Windows-1251"/>

  <xsl:template match="/">
    <html><head><title>Адресная книжка</title></head>
    <body><h2>Фамилии, адреса и телефоны</h2>
       <xsl:apply-templates />
    </body></html>
     </xsl:template>

  <xsl:template match="name">
    <p />
    <xsl:value-of select="@first" /> <xsl:text> </xsl:text>
    <xsl:value-of select="@second" /> <xsl:text> </xsl:text>
    <xsl:value-of select="@surname" /> <br />
     </xsl:template>

  <xsl:template match="address">
     <br />
     <xsl:value-of select="street" /> <xsl:text> </xsl:text>
     <xsl:value-of select="city" /> <xsl:text> </xsl:text>
     <xsl:value-of select="zip" /> <br />
     </xsl:template>

  <xsl:template match="phone-list">
     Рабочий: <xsl:value-of select="work-phone" /> <br />
     Домашний: <xsl:value-of select="home-phone" /> <br />
     </xsl:template>

</xsl:stylesheet>
```

Эту таблицу стилей можно записать в файл, например, ntb.xsl, и сослаться на него в документе XML, описывающем адресную книжку:

```
<?xml version="1.0" encoding="Windows-1251"?>
<?xml-stylesheet type="text/xsl" href="ntb.xsl"?>
<notebook>
<!-- Содержание адресной книжки -->
```

После этого любой браузер, "понимающий" XML и XSLT, например Mozilla Firefox или Internet Explorer, покажет содержимое адресной книжки, как предписано листингом 28.14.

Если набрать в командной строке

```
$ java SimpleTransform ntb.xml ntb.xsl > ntb.html
```

то в текущем каталоге будет записан преобразованный HTML-файл с именем ntb.html.

Вопросы для самопроверки

1. Зачем понадобился новый язык разметки XML?
2. Какой основной цели служат элементы XML?
3. Каким образом выявляется смысл элементов XML?
4. Обязателен ли корневой элемент в документе XML?
5. Почему документ XML должен быть снабжен описанием DTD, схемой XSD или описанием на каком-то другом языке?
6. Как выполняется синтаксический анализ документа XML?
7. В каких случаях удобнее проводить синтаксический анализ документа XML, основываясь на событиях, а в каких — построением дерева?
8. Каким образом можно преобразовать один документ XML в другой документ?
9. Почему в технологии XML предпочитают использовать таблицы стилей XSL, а не CSS?
10. Какими способами можно перебрать только узлы-элементы?
11. Преобразуются ли узлы, для которых не написаны правила преобразования?

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.

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. Флэнаган Д. Java. Справочник.: Пер. с англ. — СПб.: Символ-Плюс, 2004. — 1040 с.
2. Эккель Брюс. Философия Java. Библиотека программиста. 3-е изд. — СПб.: Питер, 2003. — 971 с.
3. Бадд Т. Объектно-ориентированное программирование в действии: Пер. с англ. — СПб.: Питер, 1997. — 464 с.
4. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++: Пер. с англ. — 2-е изд. — М.: Бином, СПб.: Невский диалект, 1998. — 560 с.
5. Коуд П., Норт Д., Мейфилд М. Объектные модели. Стратегии, шаблоны и приложения: Пер. с англ. — М.: Лори, 1999.
6. Страуструп Б. Язык программирования C++: Пер. с англ. — 3-е изд. — М.: Бином, СПб.: Невский диалект, 1999. — 991 с.
7. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2001. — 368 с.
8. Стелтинг С., Маассен О. Применение шаблонов Java. Библиотека профессионала: Пер. с англ. — М.: Вильямс, 2002. — 576 с.
9. Bruce Eckel. Thinking in Patterns with Java. — 2004. **http://mindview.net/Books/TIPatterns/**.
10. Ларман К. Применение UML и шаблонов проектирования: Пер. с англ. — 2-е изд. — М.: Вильямс, 2002. — 624 с.
11. Вирт Н. Алгоритмы и структуры данных: Пер. с англ. — М.: Невский диалект, 2001. — 352 с.
12. Ярмола Ю. А. Компьютерные шрифты. — СПб.: BHV — Санкт-Петербург, 1994. — 208 с.
13. Блох Дж. Java. Эффективное программирование: Пер. с англ. — М.: Лори, 2002. — 224 с.

14. Хабибуллин И. Ш. Создание распределенных приложений на Java 2. — СПб.: БХВ-Петербург, 2002. — 704 с.

15. Холл М. Сервлеты и JavaServer Pages. Библиотека программиста. — СПб.: Питер, 2001. — 496 с.

16. Хабибуллин И. Ш. Самоучитель XML. — СПб.: БХВ-Петербург, 2003. — 336 с.

17. Мак-Лахлин Б. Java и XML: Пер. с англ. — СПб.: Символ-Плюс, 2002. — 544 с.

18. Иванова Е. Б., Вершинин М. М. Java 2. Enterprise Edition. Технологии проектирования и разработка. — СПб.: БХВ-Петербург, 2003. — 1088 с.

19. Будилов В. Интернет-программирование на Java. — СПб.: БХВ-Петербург, 2003. — 694 с.

20. Кэй М. XSLT. Справочник программиста. — СПб.: Символ-Плюс, 2002. — 1016 с.

Предметный указатель

A

Abstract method 69
Abstract Window Toolkit (AWT) 192, 274
Access methods 61
Accessibility 274
Anonymous classes 82
Applet 451
Application 451
Arguments 18
Array 26
Ascent 210
Associative array 166

B

Baseline 206
Bitwise 32
Bridge 607
Bytecodes 6

C

Callback 396
Capacity 145, 150
Caret 228, 245
Cascading Style Sheet (CSS) 460, 683
Castor 722
CIEXYZ 202
Class 18, 26, 56
  body 18
  constructor 71
  fields 56
  methods 56, 74
  modifiers 63
  variables 73
Clipboard 325
CMYK 202
Code Conventions 19
Color model 202
Common Desktop Environment (CDE) 436
Component 193
Components 692
Compound assignment operators 35
Concatenation 138
Container 193
Content pane 380
Context 620
  menu 270
Contract 60
Controller 105
Convolution 496
Cookie 634
Core API 8
Critical section 538
CSS 731
Custom tags 643

D

Daemon 544
Dangling else 40
Data binding 714
Data processing 549
Data sink 549
Daylight Savings Time (DST) 183
DCD 712
DDML 712
Declaration 47
Default access 88
Default constructor 71
Default namespace 691
Delegation 279, 397
Deployment 614
Deployment descriptor 615
Deprecated 3
Descent 211
Deserialization 570
Design pattern 291
Design patterns 104
Dictionary 166
Document Object Model (DOM) 722
Document type declaration 685
Document Type Definition (DTD) 683
DOM API 713
Dom4j 729
Double buffering 363, 500
Drop-down menu 264
DTD 688
Dynamic binding 6

E

EIS 721
Empty element 686
Enumeration 100
Epoch 182
Event 395
Event source 395
Exception 517
Expression 36
  statement 39
Extension 59

F

Facets 695
Fall through labels 46
FAQ 1
Feel 436
First In First Out (FIFO) 163
Floating-point types 26
Font face name 207
Font height 211
Font name 207
Fully qualified name 88
Fundamental facets 696

G

Gap 354
Get method 61
Glass pane 380
Glue 370
Glyph 206
Grammar parsing 712
Graphical User Interface (GUI) 274
Graphics context 201

H

Handlers 396
Heavy component 192
High cohesion 61

Hint 230
HSB 203
HyperText Markup Language (HTML) 683

I

Identifiers 25
Identifier 396
Immediate mode model 488
Implementation 97
Incapsulation 53
Inheritance 59
Initialization 47
Inner class 81
Input focus 237
Input Method Framework 274
Input stream 546
Instance 57
  initialization 81
  methods 74
  variables 73
Instantation 47
Integral types 26
Interface 26, 97
Item 241

J

Java 2 8
Java 2 Standard Edition (J2SE) 19
Java 2D 274
Java Foundation Classes (JFC) 274
Java L&F 276
Java Look and Feel 276
Java Plug-in 12, 274
Java XML Pack 690
JavaServer Pages (JSP) 643
JAXB 720
JAXP 690, 713, 720, 722, 729
JBoss 722
JDK 7
JDO 714, 721
JDOM 729
JFC 192
JIT-компилятор 6
JLS 17
JRE 9
JSP Container 644
JSP Document 645
JSP Page 645
JSP Standard Tag Library (JSTL) 671
JVM 5

K

Kodo JDO 722

L

L&F 276
Last In First Out (LIFO) 154
Layered pane 378
Layout manager 237, 353
Leading 210
Lexical parsing 712
Lightweight component 192
List 694
Listener 396
Local class 82
Locale 134
Location 315
Logical font names 206
Look 436
Look and Feel 276
  defaults 437
Low coupling 61

M

Map 165
Mapping file 722
Member classes 82
Menu bar 264
Message 59
Method 18
Modal 382
  window 256
Model 105
  group 700
Model-View-Controller 278
Modifiers 18
Modularity 60
MSXML 690
Multiple inheritance 96
MVC 105, 278

N

Names 25
Namespace 88, 690
Narrowing 31
Native methods 7
Nested class 81
Nested top-level classes 82
New Input-Output (NIO) 563
Numeric 26

O

Object-oriented programming 54
Offset 315
OOD 54
OOP 54
OpenFusion 721
Output stream 546
Overloading 66
Overriding 67

P

Package 88
Parser 148, 712
Parsing 148
Peer-интерфейс 192
Pen 214
Pipe 547, 568
PL&F 276, 438
PLAF 276, 438
Pluggable Look and Feel 276, 438
Polymorphism 60
Pop-up menu 270, 304
Position 315
Primitive types 26
Private 70
Process 530
Processing instructions 712
Producer-consumer 481
Promotion 30
Protocol 583
Pure Java 12

Q

QName 691
Qualified Name 25, 691, 694

R

Radio buttons 239
Reference 47
Reference types 26

Relax 712
RELAX NG 712
Rendering 230
Responsibility 60
Restriction 694
RGB 201, 202
Rigid area 370
Root element 685
Root pane 380
Runtime 517

S

Sandbox 465
SAX 713, 720
SAX API 713
SAX2 713
Schematron 712
Scope 80
Scriptlet 646
Sequence 161
Serialization 547, 570
Server Side Include (SSI) 642, 683
Servlet 613
  container 613
Session 583
Set 162
Set method 61
Signature 66
Simple assignment operator 35
Size 150
Slider 248
Socket 590
SOX 712
Splash window 381
Stack 154
Standard Generalized Markup Language (SGML) 683
Statement 38
Status bar 454
stderr 546
stdin 546
stdout 546
Stream 546
strut 370
Style 316
Style sheets 319
Subclass 59
Subpackage 88
Subprotocol 608
Superclass 59
Swing 274
System defaults 436

T

Tag 452
Tear-off menu 266
Theme 447
Thread 531
Throws 522
Time zone 183
TLD 655
Token 712
TREX 712
Trusted applet 466

U

Union 694
Unnamed package 89
User defaults 437

V

Validating parser 690
View 105

Virtual key codes 408
VMS 6

W

Web Application 644
Web Components 644
Web Container 644
WebLogic 722
WebSphere 722
Web-компоненты 644
Web-контейнер 644
Web-приложение 644
Widening 31
Window manager 451
Wrapper 111

X

Xalan 730
XDR 712
Xerces 690, 691, 711, 729
XHTML 691
XML 684
  declaration 685
  Schema 690, 692
  Schema Definition Language (XSD) 688
  Schema instance 709
  View 646
XSchema 712
XSD 702
XSL 730, 731
XSLT 729
XSV 711

А

Абстрактный класс 69
Абстракция 55
Альфа 202
Апплет 451
Арифметический сдвиг 33
Ассоциативный массив 166
Ассоциированное имя 709
Атрибут 686

Б

Базовая линия 206
Базовый тип 702
Байт-коды 6
Беззнаковый сдвиг вправо 33
Безымянный класс 82
Безымянный пакет 89
Блок инициализации экземпляра 81
Блок статической инициализации 75
Буква Java 25
Буфер обмена 325

В

Вектор:
  добавление элемента 151
  емкость 150, 152
  замена элемента 152
  индекс элемента 152
  размер 150, 152
  создание 151
  удаление элемента 153
Вещественные типы 34

Визуализация 230
Виртуальная машина Java 5
Виртуальные коды клавиш 408
Вложенный класс 81
  верхнего уровня 82
Внутренний класс 81
Всплывающее меню 270, 304
Вставка подстроки 146
Выборка символов строки 139
Выборка полстроки 140
Вывод текста 225
Выпадающее меню 264
Выражение 36

Г

Глобальное имя 709
Грамматический анализ 712
Графический контекст 201

Д

Двойная буферизация 363, 500
Делегирование 279, 397
Демон 544
Десериализация 570
Дизъюнкция 27
Динамическая компоновка 6
Длина массива 47
Длина строки 139
Добавление подстроки 146
Доверенный апплет 466
Документ 314
  JSP 645
Дополнение 32
Дополнительный код 29
Драйвер ODBC 607

Е

Емкость 145

З

Загрузочный модуль 5
Заказные теги 643
Закон Деметра 61
Закрытый член класса 60
Замена подстроки 144
Замена регистра букв 143
Замена символа 144
Заполнитель:
  "надувная подушка" 370
  невидимая "распорка" 370
  невидимая разделительная область 370
Зацепление 61
Защищенный член класса 62

И

Идентификатор 25
  события 396
Иерархия 58
Имя 25
  XML 693
  семейства шрифтов 207
  составное 25
Инициализация массива 47
Инкапсуляция 53
Инструкция по обработке 712
Интерлиньяж 210
Интерфейс 96
Информационная система
  предприятия 721
Исключающее ИЛИ 27

Исключение:
непроверяемое 522
проверяемое 522
Исключительная ситуация 517
Источник события 395
Исходный модуль 5

К

Канал 547, 568
Каскадная таблица стилей 460
Класс 18, 56
Класс-оболочка 111
Класс-обработчик события 396
Класс-слушатель события 396
Классы-члены 82
Комментарий 20
Компонент 193
вид 436
поведение 436
Компонент AWT
группа 239
кнопка 238
кнопка выбора 239
линейка прокрутки 248
метка 238
поле ввода 246
раскрывающийся список 241
список 242
строка ввода 245
Компонент Swing:
дерево объектов 297
индикатор 296
кнопка 284
кнопка выбора 285
надпись 280
ползунок 295
полосы прокрутки 294
радиокнопка 286
раскрывающийся список 288
список выбора 289
счетчик 293
Компоненты 692
Конвертирование строки 148
Константа 22, 70
действительная 23
целая 22
Конструктор класса 71
Конструктор по умолчанию 71
Контейнер 193, 237
сервлетов 613
Контекст 620
Контекстное меню 270
Контракт 60
Конфигурационный файл 615
Конъюнкция 27
Корневой элемент 685
Критический участок 538
Курсор 228, 321
создание 235
форма 235

Л

"Легкий" компонент 192, 262
Лексический анализ 712
Летнее время 183
Логические имена шрифтов 206
Логические операции 27
Логический сдвиг 33
Локаль 134, 181
Локальный класс 82

М

Массив 47
копирование 186

Менеджер размещения 237, 353
BorderLayout 355
BoxLayout 369, 371
CardLayout 359
FlowLayout 353
GridBagLayout 360
GridLayout 357
Метка 45
Метод 18
аргумент 18
параметр 18
тело 18
Метод main 78
Метод класса 56, 74
Метод экземпляра 74
Методы доступа 61
Множественное наследование 96
Множество 162
упорядоченное 162, 167, 175
Модель группы 700
Модель прямого доступа 488
Модель-Вид-Контроллер 278
Модификатор 18
Модификаторы класса 63
Модульное программирование 53
Модульность 60
Мост 607

Н

Надкласс 59
Наследование 59

О

Область видимости 80
Обратный вызов 396
Объединение 694
Объект данных 714
Объектно-ориентированное программирование 54
Объектно-ориентированное проектирование 54
Объектно-ориентированный анализ 54
Объектный модуль 5
Объявление XML 685
Объявление массива 47
Объявление типа документа 685
Окно:
"родительское" 381
внутреннее 384
модальное 256, 382
немодальное 382
предварительное 381
с индикатором 392
Оконный менеджер 451
Окончательный метод 70
ООА 54
ООП 54
Оператор 38
break 45
continue 44
варианта 46
присваивания 39
цикла 42
Операция присваивания 35
Операция свертки 496
Определение массива 47
Ответственность 60
Открытый член класса 60
Отображение 165
Отрицание 27
Отсоединяемое меню 266
Очередь 163

П

Пакет 88
Пакетный доступ 88

Панель:
корневая 380
прозрачная 380
слоеная 378
содержимого 380
Парсер 148, 712
Парсинг 148
Перегрузка метода 66
Переменные класса 73
Переменные экземпляра 73
Переопределение метода 67
Перечисления 100
Перо 214
окончание линии 214
сопряжение линий 214
тип линии 214
толщина 214
Песочница 465
Побитовые операции 32
дизъюнкция 32
исключающее ИЛИ 32
конъюнкция 32
Повышение типа 30
Подкласс 59
Подпакет 88
Подпротокол 608
Подпроцесс 531
главный 531
пользовательский 544
Позиция 315
Поиск подстроки 143
Поиск символа в строке 142
Поле класса 56
Ползунок 248
Полиморфизм 60, 98
Полное имя класса 88
Помеченный блок 45
Помеченный оператор 45
Последовательность 161
Поставщик-потребитель 481
Поток 546
входной 546
выходной 546
Представитель 279
Представление:
XML 646
даты и времени 185
Преобразование координат 214
Приложение
консольное 191
Принцип KISS 62, 528
Проверяющий анализатор 690
Промежуток 354
Простой тип 692
Простой элемент 692
Пространство имен 88
Пространство имен по умолчанию 691
Протокол 583
Процедурное программирование 52
Процесс 530
Пустой элемент 686

P

Рабочий стол 386
Радиокнопка 239
Разбиение строки 140
Раскладка клавиатуры 323, 324
Расширение класса 59
Расширение типа 31
Расширенное имя 691
Реализация интерфейса 97
Рисование фигур 218
Родные методы 7
Русификация Swing 312

C

Сведения:
платформенные 437
пользовательские 437
системные 436

Связность 61
Связывание данных 714
Сдвиг влево 33
Сдвиг вправо 33
Сеанс связи 583
Сервлет 613
Сериализация 547, 570
Сессия 583
Сеть:
  одноранговая 583
  с выделенным сервером 583
Сигнатура метода 66
Символ 23
  начертание 206
Синтаксический разбор строки 148
Скриптлет 646
Словарь 166
Сложный тип 692
Сложный элемент 692
Случайное число 186
Слушатель 396
Событие 395
Создание строки 133
Сокет 590
Сокращенная дизъюнкция 28
Сокращенная конъюнкция 28
Составные операции присваивания 35
Спецификация виртуальной
  машины Java 6
Список 171, 694
Сравнение строк 141
Ссылка 47
Стандартный ввод 546
Стандартный вывод 546
Стандартный вывод сообщений 546
Статическая переменная 74
Статический метод 74
Стек 154
Стиль 316
Страница JSP 645
Строка 24
  меню 264
  состояния 454
Структурное программирование 53
Сужение 694
  типа 31
Суперкласс 59
Сущность 689
Схема 688
  XML 690, 692
Схема проектирования MVC 105
  Вид 105
  Контроллер 105
  Модель 105
Сцепление строк 138

Т

"Тяжелый" компонент 192
Таблица стилей 319
Таблицы 332
Тег 452
  закрывающий 686
  открывающий 686
Текстовый редактор 329
Тело класса 18
Тело элемента 686
Тема 447
Типы данных:
  примитивные 26
    логический 26
    числовые 26
      вещественные 26
      целые 26
  ссылочные 26
    интерфейсы 26
    классы 26
    массивы 26

У

Удаление подстроки 147
Удаление символа 147

Условная операция 36
Условный оператор 39
Установка 614
Уточненное имя 691

Ф

Фасетки 695
  базисные 696
Физическое имя шрифта 207
Фокус ввода 237

Ц

Цветовая модель 202
Целое деление 30

Ч

Часовой пояс 183
Члены класса 56

Э

Экземпляр класса 57
Экземпляр схемы 709
Элемент XML 686

Я

Язык
  Expression Language (EL) 649