Ильдар Хабибуллин

bhv®

Создание РАСПРЕДЕЛЕННЫХ ПРИЛОЖЕНИЙ на Java 2

- Графическая библиотека Swing
- Распределенные объекты RMI и CORBA
- Сервлеты, JSP и компоненты EJB
- Русификация распределенных приложений

MASTER

Ильдар Хабибуллин

Создание распределенных приложений на Java 2

Санкт-Петербург
«БХВ-Петербург»
2002

УДК 681.3.06
ББК 32.973.26-018.1
Х12

Хабибуллин И. Ш.

Х12 Создание распределенных приложений на Java 2. — СПб.: БХВ-Петербург, 2002. — 704 с.: ил.

ISBN 978-5-94157-106-2

Книга посвящена разработке многослойных приложений, отдельные части которых, возможно, распределены по сети, включая Internet, но работают как единое целое. В книге дано полное изложение стандартных средств Java 2 для создания распределенных приложений любой сложности, в том числе и Web-приложений. В качестве примера приведена система дистанционного обучения.

Для разработчиков распределенных приложений,
студентов старших курсов и специалистов по информационным технологиям,
интересующихся новейшими средствами разработки

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Анна Кузьмина*
Редактор	*Татьяна Коротяева*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 28.05.02.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 56,76.
Тираж 4000 экз. Заказ №
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

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

ISBN 978-5-94157-106-2

© Хабибуллин И. Ш., 2002
© Оформление, издательство "БХВ-Петербург", 2002

Содержание

Введение **1**

Структура книги 3

Часть I. Понятие распределенного приложения **7**

Глава 1. Распределенные приложения и Java **9**

Сетевые средства стандарта Java 9
Сокеты потокового типа TCP 13
Сокеты дейтаграммного типа UDP 15
Новые средства Java API 17
Механизм распределенных объектов RMI 18
Использование средств CORBA 22
Использование технологии WWW 27
Сервлеты 27
Java 2 Enterprise Edition 28
Сервер J2EE 29
Инсталляция J2EE SDK 29
Простейший сервлет 30
Установка сервлета в контейнер 32
Виртуальные каталоги 32
Работа сервлетов по протоколу HTTP 33
Язык JSP 35
Компоненты EJB 37
Система именования JNDI 41
Установка компонента EJB в контейнер 41
Распределенное приложение Java 43

Глава 2. Пример: система дистанционного обучения **46**

Объектно-ориентированное проектирование 47
Design Patterns (паттерны проектирования) 48
Пользователи СДО 49
Действия ученика 49
Регистрация 49
Учебный процесс 50
Оплата учебы 50
Действия учителя 50
Действия администратора 50
Графические интерфейсы 51

Начальная страница ... 51
Страницы ученика ... 51
Страница регистрации ... 51
Страница подтверждения регистрации ... 51
Страница учебной темы ... 51
Страница вопросов ... 52
Страница оценки ... 52
Страницы учителя ... 52
Начальная страница ... 52
Страница нового курса ... 52
Страница изменения курса ... 52
Утилита администратора ... 52
Функции и модули СДО ... 52
Структура СДО ... 53
Объекты СДО ... 53
Связи между объектами ... 54
Атрибуты объектов ... 55
Класс *Student* ... 55
Класс *StudentAccount* ... 57
Класс *Teacher* ... 58
Класс *Course* ... 60
Класс *Topic* ... 61
Класс *Task* ... 61
База данных SDO ... 63
Заключение ... 65

Часть II. Клиентская часть распределенного приложения Java ... 67

Глава 3. Оформление клиентского приложения компонентами Swing ... 69

Состав библиотеки Swing ... 70
Основные компоненты Swing ... 73
Класс *JComponent* ... 74
Схема MVC в компонентах Swing ... 75
Надпись *JLabel* ... 77
Кнопки ... 79
Кнопка *JButton* ... 82
Флажок *JToggleButton* ... 83
Флажок *JCheckBox* ... 84
Переключатель *JRadioButton* ... 84
Раскрывающийся список *JComboBox* ... 87
Список выбора *JList* ... 89
Визуализация элементов списков ... 90
Счетчик *JSpinner* ... 93
Линейка прокрутки *JScrollBar* ... 95
Ползунок *JSlider* ... 96
Индикатор *JProgressBar* ... 97
Дерево объектов *JTree* ... 98

Построение меню ... 103
Строка меню *JMenuBar* ... 103
Меню *JMenu* ... 103
Меню *JMenuItem* ... 105
Меню *JCheckBoxMenuItem* ... 106
Меню *JRadioButtonMenuItem* ... 106
Всплывающее меню *JPopupMenu* ... 107
Панель выбора цвета *JColorChooser* ... 108
Окно выбора файла *JFileChooser* ... 110
Фильтр файлов *FileFilter* ... 110
Как получить выбранный файл ... 112
Дополнительный компонент ... 113
Замена изображений ... 115
Русификация Swing ... 116

Глава 4. Создание форм с помощью текстовых компонентов ... 118

Компонент *JTextComponent* ... 118
Модель данных — документ ... 119
Строка символов *Segment* ... 119
Запись текста в документ ... 120
Атрибуты текста ... 120
Удаление текста из документа ... 121
Фильтрация документа ... 121
Внесение структуры в документ ... 122
События в документе ... 123
Реализации документа ... 123
Установка модели данных ... 124
Вид ... 124
Контроллер — редактор текста ... 126
Курсор ... 127
Ограничение перемещения курсора ... 127
Реализации редактора ... 129
Раскладка клавиатуры ... 129
Поле ввода *JTextField* ... 131
Поле ввода пароля *JPasswordField* ... 133
Редактор объектов *JFormattedTextField* ... 134
Область ввода *JTextArea* ... 135
Текстовый редактор *JEditorPane* ... 136
Редактор *JTextPane* ... 137

Глава 5. Помещение результата запроса в таблицу ... 138

Класс *JTable* ... 138
Модель данных таблицы ... 140
Модель ячеек таблицы ... 141
Свойства столбца таблицы *TableColumn* ... 147
Модель столбцов таблицы ... 147

Заголовки столбцов таблицы *JTableHeader* 147
Модель выделения ячеек 150
Визуализация ячеек таблицы 150
Редактор ячеек таблицы 155

Глава 6. Контейнеры 160

Панель *JPanel* 160
Панель прокрутки *JScrollPane* 161
Двойная панель *JSplitPane* 164
Панель со вкладками *JTabbedPane* 166
Линейная панель *Box* 168
Менеджер размещения *BoxLayout* 169
Компоненты-заполнители 169
Менеджер размещения *SpringLayout* 172
Размеры *Spring* 172
Промежутки *Constraints* 173
Размещение компонентов 174
Панель инструментальных кнопок *JToolBar* 176
Интерфейс *Action* 178
Слоеная панель *JLayeredPane* 180
Корневая панель *JRootPane* 181
Окно *JWindow* 183
Диалоговое окно *JDialog* 184
Окно верхнего уровня *JFrame* 186
Внутреннее окно *JInternalFrame* 187
Рабочий стол *JDesktopPane* 189
Стандартные диалоговые окна *JOptionPane* 191
Окно с индикатором *ProgressMonitor* 196
Апплет *JApplet* 198

Глава 7. Обработка событий в Swing 201

Событие изменения предка *AncestorEvent* 206
Событие изменения свойств *PropertyChangeEvent* 206
Управление колесиком мыши 208
Событие изменения состояния *ChangeEvent* 210

Глава 8. Оформление рамок 212

Пустая рамка *EmptyBorder* 214
Прямолинейная рамка *LineBorder* 215
Объемная рамка *BevelBorder* 215
Объемная рамка с закругленными краями *SoftBevelBorder* 216
Врезанная рамка *EtchedBorder* 216
Рамка с изображением *MatteBorder* 217
Рамка с надписью *TitledBorder* 218
Сдвоенная рамка *CompoundBorder* 221
Создание собственных рамок 223

Глава 9. Изменение внешнего вида компонента ... 231

Получение сведений о свойствах L&F ... 233
Задание стандартного стиля L&F ... 235
Дополнительные стили L&F ... 238
Смена всего стиля L&F ... 238
Замена отдельных свойств стиля L&F ... 241
Темы стиля Java L&F ... 244

Глава 10. Прочие свойства Swing ... 249

Индивидуальные свойства экземпляра компонента ... 249
Прокрутка содержимого компонента ... 250
Передача фокуса ввода ... 251
Перенос данных методом Drag and Drop ... 256
Временная задержка *Timer* ... 258
Интерфейс пользователя СДО ... 260

Часть III. Связь приложения Java с базами данных ... 271

Глава 11. Соединение приложения с базой данных ... 273

Загрузка драйверов JDBC ... 275
Соединение с базой данных ... 278
Использование JNDI для соединения ... 279
Интерфейс *DataSource* ... 279
Работа с источником данных ... 280
Пул соединений ... 282
Соединение для распределенной транзакции ... 284
Соединение *Connection* ... 285
Управление транзакцией ... 285
Уровни изоляции ... 286
Прочие методы ... 286
Свойства базы данных ... 287

Глава 12. Оформление запроса ... 290

Интерфейс *Statement* ... 290
Получение экземпляра *Statement* ... 291
Формирование и выполнение запроса ... 292
Изменение таблиц базы ... 299
Общий случай ... 300
Предварительно откомпилированные запросы ... 303
Параметры откомпилированного запроса ... 303
Выполнение откомпилированного запроса ... 304
Хранимые процедуры ... 306
Регистрация выходных параметров ... 307
Выполнение хранимой процедуры ... 307
Escape-синтаксис ... 308
Escape-синтаксис даты и времени ... 309

Escape-синтаксис внешних соединений ... 309
Escape-синтаксис экранирования ... 309
Сгенерированные ключи ... 310

Глава 13. Обработка результатов запроса ... 311

Интерфейс *ResultSet* ... 311
Расширенный просмотр выборки ... 312
Дополнительные свойства выборки ... 313
Интерфейс *ResultSetMetaData* ... 314
Обработка выборки ... 315
Обновляемые результаты ... 316
Удаление строки ... 318
Вставка строки ... 318
Выборка типа *RowSet* ... 319
Соединение с источником данных ... 320
Заполнение выборки ... 324
Обработка результатов ... 325
Обработка событий ... 330

Глава 14. Прочие свойства интерфейса JDBC ... 332

Пакетное изменение источника данных ... 332
Типы данных *BLOB* и *CLOB* ... 334
Интерфейсы *Blob* и *Clob* ... 334
Получение большого объекта ... 336
Занесение большого объекта в источник данных ... 338
Составные типы данных ... 338
Интерфейс *Array* ... 340
Интерфейс *Struct* ... 340
Ссылочный тип данных *REF* ... 340
Типы данных пользователя, UDT ... 341
Тип данных *DISTINCT* ... 343
Двухслойная схема СДО ... 343

Часть IV. Сетевая часть приложения Java ... 353

Глава 15. Обращение к удаленным методам ... 355

Структура системы RMI ... 356
Создание удаленного объекта ... 358
Удаленный интерфейс ... 358
Удаленный объект ... 359
Создание RMI-сервера ... 361
Создание RMI-клиента ... 362
Компиляция и запуск системы RMI ... 363
Аргументы и результат удаленного метода ... 368
Протоколы системы RMI ... 369
Протокол JRMP ... 370
RMI поверх HTTP ... 371

Активизация удаленных объектов ... 371
Группа активизации ... 375
Применение собственных сокетов ... 376
Использование RMI-реестра ... 380
Обратный вызов ... 381

Глава 16. Обмен сообщениями между частями распределенного приложения ... 387

Сообщение *Message* ... 388
Заголовок сообщения ... 389
Свойства сообщения ... 389
Тело сообщения ... 391
Соединение *Connection* ... 392
Сведения о соединении ... 394
Сеанс *Session* ... 394
Отбор сообщений ... 397
Адресат *Destination* ... 398
Отправители и получатели ... 399
Обмен сообщениями в СДО ... 402

Глава 17. Поиск распределенных объектов в сети ... 409

Имена ... 411
Имя класса *CompoundName* ... 412
Имя класса *CompositeName* ... 414
Имя в виде строки URL ... 416
Контекст ... 417
Начальный контекст *InitialContext* ... 418
Контекст каталога ... 420
Интерфейс *Attribute* ... 420
Класс *BasicAttribute* ... 421
Интерфейс *Attributes* ... 422
Класс *BasicAttributes* ... 422
Действия с атрибутами ... 423
Поиск объектов в каталоге контекста ... 425
Реализация контекста каталога ... 426
Ссылки на объекты ... 434
Поставщик услуг RMI ... 435
Поставщик услуг файловой системы ... 436
Поставщик услуг DNS ... 438
Поставщик услуг NIS ... 439
Поставщик услуг LDAP ... 440
Поставщик услуг COS Naming ... 441

Глава 18. Средства электронной почты ... 443

Замечания о кодировке ... 444
Письмо *Message* ... 447

Сеанс связи *Session* ... 450
Адрес *Address* ... 452
Связь *Service* ... 453
Хранилище *Store* ... 459
Получение писем ... 460
Пометки писем ... 462
Работа с присоединенной информацией ... 463
Отправка HTML-файла ... 466

Глава 19. Связь компонентов Java с архитектурой CORBA ... 469

Использование RMI-IIOP ... 471
Адаптер BOA ... 473
Использование COS Naming ... 476
Адаптер POA ... 478
Использование IDL ... 483
Долговременные объекты ... 488
Заключение ... 489

Часть V. Серверная часть распределенного приложения Java ... 491

Глава 20. Сервлеты и JSP ... 493

Интерфейс *Servlet* ... 494
Конфигурационный файл ... 495
Интерфейс *ServletConfig* ... 498
Контекст сервлета ... 501
Метод *Service* ... 501
Интерфейс *ServletRequest* ... 502
Интерфейс *ServletResponse* ... 503
Цикл работы сервлета ... 504
Класс *GenericServlet* ... 505
Работа по протоколу HTTP ... 506
Интерфейс *HttpServletRequest* ... 506
Интерфейс *HttpServletResponse* ... 508
Класс *HttpServlet* ... 509
Сеанс связи с сервлетом ... 516
Фильтры ... 520
Обращение к другим ресурсам ... 525
Страницы JSP ... 526
Теги JSP ... 530
Встроенные объекты JSP ... 533
Обращение к компоненту JavaBean ... 535
Выполнение апплета в браузере клиента ... 536
Передача управления ... 537
Заказные теги ... 538
Класс-обработчик заказного тега ... 541
Заказной тег с атрибутами ... 543

Заказной тег с телом ... 544
Обработка тела заказного тега ... 546
Обработка взаимодействующих тегов ... 549
Обработка исключений в заказных тегах ... 554
Стандартная библиотека тегов JSTL ... 555

Глава 21. Компоненты на сервере ... 557

Роли создателей компонентов EJB ... 559
Интерфейсы компонентов EJB ... 560
Связь с клиентом ... 561
Состав компонента EJB ... 563
Разновидности компонентов EJB ... 568
Контекст компонента EJB ... 569
Метаданные компонента ... 570
Идентификаторы компонента EJB ... 571
Клиентские заглушки ... 572
Переменные окружения компонента ... 573
Ограничения компонентов EJB ... 576

Глава 22. Создание Session и MessageDriven Beans ... 577

Создание и удаление session-компонента ... 577
Активизация SFSB-компонента ... 581
Пример SFSB-компонента ... 581
Клиенты компонента EJB ... 589
Отдельное приложение ... 590
Сервлет ... 591
Страница JSP ... 594
Другой компонент EJB ... 596
Клиент, связанный с приложением J2EE ... 596
Компоненты, управляемые сообщениями ... 597

Глава 23. Создание Entity Beans ... 605

Home-методы ... 606
Интерфейс *EntityBean* ... 607
Методы создания экземпляра ... 609
Методы поиска ... 612
Состояния экземпляра entity-компонента ... 614
CMP-компонент ... 615
Отношения, управляемые контейнером ... 621
Методы выбора ... 623
Язык запросов EJB QL ... 623
BMP-компонент ... 626

Глава 24. Транзакции ... 637

Стандарты распределенных транзакций ... 639
Интерфейсы управления транзакцией ... 640

Прикладной интерфейс 642
Интерфейсы управления ресурсами 643
Транзакции в апплетах и приложениях 645
Транзакции в Web-компонентах 645
Транзакции в EJB-компонентах 647
Транзакции, управляемые компонентом 647
Транзакции, управляемые контейнером 653

Глава 25. Технология Java и язык XML 655

Пространство имен тегов 659
Анализ документа XML 660
Анализ документов XML с помощью SAX2 API 661
Анализ документов XML с помощью DOM API 664
Интерфейс *Node* 665
Интерфейс *Document* 667
Интерфейс *Element* 668
Преобразование дерева объектов в XML 671
Таблицы стилей XSL 674
Преобразование документа XML в HTML 676

Глава 26. Упаковка и установка компонентов 678

Утилита упаковки 683
Утилита проверки 685
Утилита установки 685

Список литературы и Internet-источников информации 687

Предметный указатель 689

Введение

Сразу после выхода книги [15] я стал получать много отзывов о ней. Среди критики и указаний на многочисленные опечатки оказалось удивительно много предложений расширить приложение к книге, названное "Развитие Java", и подробно описать затронутые в нем вопросы, относящиеся к серверной части технологии Java.

Это непростая задача.

Технология Java быстро превратилась в мощную и развитую платформу для построения крупных информационных систем уровня предприятия. В ней есть все средства для создания рабочих мест клиента, оснащенных полноценным графическим интерфейсом, и серверов приложений, предоставляющих все необходимые клиентам услуги, а также для установления сетевой связи между различными удаленными частями информационной системы.

Воспользоваться средствами Java очень легко. Все необходимое для разработки распределенного приложения Java находится в двух самораспаковывающихся архивах, которые можно свободно загрузить с сайта фирмы Sun Microsystems. Они расположены по адресам **http://java.sun.com/j2se/** и **http://java.sun.com/j2ee/**. Надо только выбрать файл, соответствующий вашей операционной системе: MS Windows, Solaris, Linux, MacOS X. Я настойчиво рекомендую загрузить, кроме того, файл с документацией. Документация в формате HTML едина для всех операционных систем, поэтому она поставляется отдельным файлом.

Развернув первый архив на своем компьютере, вы получите набор J2SE SDK (Java 2 Standard Edition Software Development Kit), содержащий компилятор, интерпретатор, отладчик, набор базовых классов. Из второго архива вы получите J2EE SDK (Java 2 Enterprise Edition Software Development Kit), содержащий дополнительный набор классов для создания серверной части приложения, а также небольшой сервер приложений и утилиты для работы с ним. Остается только создать переменные окружения `JAVA_HOME`, `J2EE_HOME`, занести пути к исполняемым файлам в переменную `PATH`, как указано в прилагаемой инструкции по инсталляции, и можно работать.

Разумеется, лучше установить последние версии SDK. Ко времени написания книги это были версии J2SE SDK 1.4.0 и J2EE SDK 1.3.1.

Вы можете работать непосредственно с компилятором, отладчиком и интерпретатором из командной строки, но большинство разработчиков, особенно для совместной работы, выбирают какое-либо из многочисленных RAD-

средств разработки (Rapid Application Development). В России наиболее популярны следующие RAD-продукты:

- JBuilder фирмы Borland Software Corporation, **http://www.borland.com/**;
- Visual Age for Java фирмы IBM, **http://www.ibm.com/java/**;
- Kawa фирмы Allaire Corporation, **http://www.allaire.com/**;
- IDEA фирмы IntelliJ Software, **http://www.intellij.com/**;
- Together фирмы TogetherSoft Corporation, **http://www.togethersoft.com/**;
- Rational Rose фирмы Rational Software Corporation, **http://www.rational.com/**;
- если ваши данные хранятся в базе Oracle, то удобно использовать JDeveloper корпорации Oracle, **http://www.oracle.com/**.

Хотя средства разработки Java относительно компактны, сама технология Java настолько обширна, что книги, описывающие каждую ее часть, содержат по тысяче страниц. Для того чтобы книга, которую вы держите в руках, имела разумный объем, пришлось, во-первых, предположить, что читатель уже знаком со стандартными средствами Java, например, по книгам [1, 11, 15, 18], и с объектно-ориентированным программированием [3].

Во-вторых, пришлось ограничить книгу описанием средств создания только одного распределенного приложения, а не целой информационной системы. Особенностям создания разветвленных систем уровня предприятия посвящена книга [12].

Под распределенным приложением в этой книге понимается информационная система, компоненты которой работают на различных машинах, часто удаленных друг от друга на значительное расстояние. Компоненты могут функционировать в разных операционных средах, зачастую построенных по совершенно непохожим принципам. Для связи элементов распределенного приложения могут применяться самые различные сетевые средства, но в этих сложных условиях распределенное приложение должно выполняться как единое целое, как будто все его компоненты работают на одной машине.

В-третьих, книга адресована профессиональному разработчику. Это не ограничивает круг читателей, поскольку в России пока только профессионалы интересуются технологией Java, в отличие от США, где это сейчас повсеместно распространенная технология создания распределенных систем. Я называю профессионалом того, кто, кроме всего прочего:

- свободно читает по-английски технические тексты;
- обязательно загрузит с адресов **http://java.sun.com/docs/books/tutorial/** и **http://java.sun.com/j2ee/** и установит на своем компьютере не входящие в документацию учебники Java Tutorial и J2EE Tutorial;
- просмотрит сайт **http://java.sun.com/blueprints/**, содержащий пример разработки распределенного приложения Java Pet Store и относящиеся к нему документы;

- познакомится со всеми спецификациями, описывающими отдельные продукты Java;
- внимательно прочитает документы и исходные тексты примеров из каталога $JAVA_HOME/docs/guide/ и просмотрит демонстрационные материалы из каталога $JAVA_HOME/demo;
- распакует архив $JAVA_HOME/src.zip и будет обращаться к исходным текстам классов Java, чтобы узнать, как они работают;
- будет постоянно просматривать материалы, размещенные на сайтах **http://java.sun.com/**, **http://www.ibm.com/java/**, **http://www.theserverside.com/**, **http://www.javable.com/** и других многочисленных сайтах, посвященных технологии Java.

Эти профессиональные качества читателя позволяют уменьшить объем книги до разумного предела. Книга не содержит полных описаний классов Java. Они приведены в документации. В книге минимум схем и рисунков, потому что лучше, чем в Java Tutorial, их не сделаешь. В книге нет описания работы с графическими утилитами и RAD-продуктами — профессионал осваивает их по встроенной в продукт справке (Help). В книге нет пространных рассуждений, оценок и сравнений продуктов — у каждого профессионала есть свое твердое мнение на этот счет.

Что же есть в книге?

Книга представляет собой полное руководство по использованию Java для создания распределенных приложений. В ней описано ЧТО, какие средства, есть в технологии Java для создания распределенных приложений, и КАК ими пользоваться в повседневной практической работе. Подробно рассматриваются стандартные классы, из которых составляется приложение, и методы, которыми они обеспечивают работу приложения. Рассказано, как применять эти классы и методы для решения своих задач. Приведены примеры "из жизни", показывающие работу классов Java и их методов в распределенных приложениях. На протяжении всей книги разрабатывается гипотетическая распределенная система дистанционного обучения СДО.

Структура книги

Книга состоит из двадцати шести глав, сгруппированных в пять частей.

Часть I содержит две главы.

В *главе 1* рассматриваются базовые понятия распределенных приложений, их строение, приемы и средства Java для их разработки. Приведен краткий обзор стандартных пакетов классов и интерфейсов Java, необходимых для разработки распределенных приложений, даны примеры их использования. Эта глава поможет составить общее представление о структуре распределенного приложения Java.

В *главе 2* рассказывается о современных средствах проектирования и разработки сложных приложений. Они использованы для проектирования сквозного примера книги — системы дистанционного обучения, СДО. Рассмотрена многослойная структура распределенного приложения.

В *части II* подробно описывается клиентская часть распределенного приложения и основное средство для ее создания — графическая библиотека Swing. Для знакомства с ней потребовалось восемь глав.

В *главе 3* рассматриваются основные графические компоненты библиотеки Swing: метки, кнопки, списки, ползунки, меню, деревья и другие компоненты, их возможности и правила применения.

В *главе 4* описаны текстовые компоненты Swing и работа с текстом с их помощью: редактирование текста, замена цвета и шрифта, вставка компонентов в текст.

Создание таблиц и работа с ними объясняются в *главе 5*. Описаны приемы заполнения ячеек таблицы, их оформления цветом и шрифтом, выделение заголовков, отдельных строк и столбцов.

В *главе 6* показано, как разместить компоненты в контейнерах, какие виды контейнеров есть в библиотеке Swing и как их использовать для придания графическому интерфейсу пользователя удобного и красивого вида.

О том, как обработать события, вызываемые действиями пользователя, рассказывается в *главе 7*. Библиотека Swing содержит удобные и гибкие средства реагирования на подобные события, они подробно разобраны в этой главе.

Глава 8 посвящена рассмотрению различного вида рамок, обрамляющих графические компоненты и контейнеры.

Библиотека Swing предоставляет интересную возможность изменять внешний вид графического интерфейса в зависимости от платформы, на которой работает графическое приложение Java или, наоборот, сделать внешний вид приложения независимым от платформы. Эта возможность описана в *главе 9*.

Наконец, *глава 10* посвящена прочим возможностям библиотеки Swing, не описанным в предыдущих главах.

В *части III*, состоящей из четырех глав, рассматриваются средства Java для связи с базами данных.

В *главе 11* рассказывается о том, что такое JDBC, как выбрать драйвер JDBC, загрузить его и установить соединение с базой данных.

В *главе 12* рассматриваются различные виды запросов к базе данных: выборки, включение строк в таблицы, их изменение и удаление, создание таблиц и прочие действия с базами данных. В этой главе показано, как оформить и выполнить запрос средствами JDBC.

О том, как обработать результаты запроса, в частности, поместить их в ячейки таблицы Swing, рассказано в *главе 13*.

Глава 14 посвящена рассмотрению прочих свойств JDBC, не освещенных в предыдущих главах, в частности, работе с курсорами.

В *IV части* книги, состоящей из пяти глав, описаны сетевые средства Java, составляющие сердцевину всякого распределенного приложения.

В *главе 15* подробно объясняется механизм RMI и его использование для обращения к методам распределенных объектов, а также применение IIOP — транспортного протокола CORBA.

В *главе 16* рассмотрена система асинхронного обмена сообщениями JMS — реализация MOM средствами Java. Показано, как организовать очереди сообщений и разделы подписки, прием и отправку сообщений.

В *главе 17* рассказывается о средствах JNDI именования и поиска распределенных объектов в сети — распределенной службе каталогов Java.

В *главе 18* показано, каким образом с помощью почтовой службы JavaMail распределенное приложение можно включить во всемирную систему электронной почты, принимать и отправлять почтовые сообщения.

Наконец, в *главе 19* рассматривается язык Java IDL — средство включения объектов Java в архитектуру CORBA.

V часть книги посвящена серверной части распределенного приложения. Она состоит из семи глав.

В *главе 20* рассказывается о сервлетах и страницах JSP — основных средствах Java, предназначенных для расширения возможностей Web-сервера. С их помощью можно легко включить в Web-сервер дополнительные средства обработки информации.

В *главах 21*, *22* и *23* подробно описываются компоненты EJB — переносимые компоненты, работающие на сервере и легко встраиваемые в любые J2EE-серверы приложений.

В *главе 24* показано, как можно средствами JTA объединить действия по обработке информации в транзакцию, управлять транзакциями и включать в транзакцию компоненты распределенного приложения и источники данных.

Все большее распространение получает язык XML. В *главе 25* изучаются средства пакета JAXP для обработки документов XML.

В *главе 26*, последней главе книги, описывается процедура упаковки компонентов распределенного приложения для переноса их на другой сервер и процедура установки компонентов на другом J2EE-сервере.

Часть I

Понятие распределенного приложения

Глава 1

Распределенные приложения и Java

В этой главе дан обзор средств, которые предлагает технология Java для создания распределенных приложений, и которыми мы будем пользоваться на протяжении всей книги. Для того чтобы читатель сразу получил представление о том, как реализуются эти средства, приведены листинги простейших примеров их использования.

Сетевые средства стандарта Java

Фирма Sun Microsystems, девиз которой: "Сеть — это компьютер", активно развивает сетевые технологии, стремясь в то же время упростить создание распределенных приложений. (К слову, у фирмы Sun с некоторых пор появился новый лозунг: "We are the dot in .com". Фирма Sun хочет сказать, что она полностью обеспечивает инструментами Internet-бизнес и ставит точку в умножении их числа.) Основную роль в упрощении разработки сетевых приложений фирма Sun Microsystems отводит технологии Java.

Уже в первый выпуск инструментального набора классов и утилит JDK 1.0 был включен пакет java.net, содержащий набор классов для связи отдельных частей приложения, работающих на разных машинах, по протоколу TCP посредством сокетов, для пересылки дейтаграмм UDP и для обмена информацией по протоколам, работающим посредством адресации URL. Эти классы предельно автоматизировали установку соединения клиента с сервером. Достаточно написать пять-семь строк кода на языке Java, и программа готова. Например, если отвлечься от проверок и обработки исключений, то, для того, чтобы получить домашнюю страничку сайта и отправить ее в стандартный вывод программы, необходимо написать только:

```
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);
```

Конечно, упрощение программирования, активно продвигаемое фирмой Sun, достигается за счет инкапсуляции всей сложности сетевого взаимодействия в классы Java. Такое "крупноблочное" программирование требует четкого взаимодействия классов на всех уровнях сетевых протоколов. Малейшее расхождение в реализации протокола приведет к сбоям в работе сетевой программы, а настроить ее невозможно, все параметры реализации протокола скрыты внутри классов. Можно только заменить один блок программы другим блоком. Много времени и сил уходит на подгонку программ разных производителей, подбор совместимых версий, наложение заплаток-патчей. Но все-таки мы перешли от программирования в машинных кодах к языку Ассемблера, а от него к языкам высокого уровня. Сейчас мы манипулируем целыми объектами и уже внедряем компонентное программирование, одним оператором объектно-ориентированного языка программирования вставляя в свое приложение электронную таблицу или окно открытия файла. Тенденцию к укрупнению "строительных блоков" не остановить, и фирма Sun только следует ей.

Впрочем, вместо класса `URL` можно применить класс `URLConnection` или его расширение, введенное в JDK 1.1, — класс `HttpURLConnection`, использующий особенности протокола HTTP. Методы этих классов позволяют задать параметры соединения, например, MIME-тип сообщения, его кодировку, метод передачи сообщения `GET` или `POST`, определить заголовок сообщения. При получении ответа от Web-сервера можно просмотреть код ответа, заголовок ответа, узнать MIME-тип ответа и его кодировку. Все это позволяет отрегулировать приложение, настроить взаимодействие клиентской и серверной частей программы.

В листинге 1.1 показано, как можно послать параметры *CGI-программе* (CGI — Common Gateway Interface, общедоступный интерфейс шлюза) методом `POST` протокола HTTP, и получить от нее ответ. Для правильной передачи символов кириллицы строка параметров, записанная в программе в кодировке Unicode, перед отправкой по сети преобразуется в байтовую кодировку KOI8-R.

Листинг 1.1. Посылка параметров CGI-программе

```
import java.net.*;
import java.io.*;

public class UsingCGI{
  public static void main(String[] args){
     String req = "Параметры, посылаемые CGI-программе";
     try{
        // Преобразуем строку Unicode в массив
```

```
// байтовых символов в кодировке KOI8-R.
byte[] data = req.getBytes("KOI8-R");

URL url =
   new URL("http://www.bhv.ru/cgi-bin/javastart.bat");

// Получаем экземпляр класса URLConnection.
URLConnection uc = url.openConnection();

// Устанавливаем свойства соединения
uc.setDoOutput(true);   // Будем отправлять
uc.setDoInput(true);    // и получать информацию.
uc.setUseCaches(false); // Кэш нам не нужен.
// Определяем MIME-тип и подтип посылаемого сообщения.
uc.setRequestProperty("content-type",
              "application/octet-stream");
// Для метода POST нужна длина сообщения.
uc.setRequestProperty("content-length",
              ""+req.length());

// Устанавливаем соединение.
uc.connect();

// Открываем выходной поток
DataOutputStream dos =
     new DataOutputStream(uc.getOutputStream());

// и выводим в него массив параметров.
dos.write(data, 0, data.length);

dos.close();

// Открываем входной поток, предполагая, что
// он записан в кодировке KOI8-R.
BufferedReader br = new BufferedReader(
      new InputStreamReader(uc.getInputStream(), "KOI8-R"));

String res = null;
// Читаем ответ строка за строкой.
```

```
        while ((res = br.readLine()) != null)
          System.out.println(res);

        br.close();

      }catch(Exception e){}
    }
}
```

CGI-программу тоже можно написать на языке Java. В листинге 1.2 приведен пример простой CGI-программы.

Листинг 1.2. CGI-программа, написанная на языке Java

```
import java.io.*;

public class JavaCGI{
   public static void main(String[] args){
      try{
        BufferedReader br = new BufferedReader(
          new InputStreamReader(System.in, "KOI8-R"));

        String s = br.readLine();

        // Здесь обработка полученной строки параметров...

        PrintWriter pr = new PrintWriter(
          new OutputStreamWriter(System.out, "KOI8-R"), true);
        pr.println("Ответ: " + s);

      }catch(Exception e){}
   }
}
```

Эта программа просто отправляет обратно первую строку полученного сообщения. Для правильной передачи символов кириллицы стандартный ввод и вывод программы принимают и посылают поток байтов в кодировке KOI8-R.

Файл JavaCGI.class, полученный после компиляции программы, приведенной в листинге 1.2, надо поместить в каталог cgi-bin или какой-то другой

каталог, в котором хранятся CGI-программы. Расположение этого каталога зависит от используемого Web-сервера. Кроме того, надо написать командный файл, содержащий всего одну строку:

```
java JavaCGI
```

и поместить его в тот же каталог. В листинге 1.1 этот командный файл назван javastart.bat. Именно он указан в URL-адресе CGI-программы. Web-сервер запустит соответствующий интерпретатор командной оболочки (shell), который, выполняя командный файл, запустит *виртуальную машину Java* (JVM — Java Virtual Machine). Она обратится к методу `main()` класса `JavaCGI`, передав на стандартный ввод программы полученную по сети строку параметров. После того как программа направит результаты своей работы в стандартный вывод, Web-сервер извлечет их оттуда и отправит клиенту.

Конечно, в производственной системе запускать новый экземпляр JVM при каждом обращении к CGI-программе нельзя. Это приведет к громадному расходу ресурсов. JVM надо запускать один раз при первом запросе клиента, а затем для обработки запросов создавать подпроцессы. Для реализации такого подхода в технологии Java разработана спецификация *сервлетов* (servlets) и реализованы специальные программы (servlets engine), запускающие сервлеты и следящие за их выполнением.

В состав пакета java.net включены классы, которые позволяют спуститься с прикладного уровня сетевого соединения на транспортный уровень и организовать соединения с помощью сокетов дейтаграммного типа по протоколу UDP или сокетов потокового типа по протоколу TCP.

Сокеты потокового типа TCP

Вся сложность установления связи по протоколу TCP спрятана в двух классах: `SocketServer` и `Socket`. На стороне клиента создается *сокет* — объект класса `Socket`. При создании объекта в него заносится адрес и номер порта сервера. На стороне сервера работает экземпляр класса `SocketServer`, "прослушивающий" назначенный ему порт. Сокет клиента устанавливает соединение с сервером и открывает входной и выходной потоки методами `getInputStream()` и `getOutputStream()` для обмена информацией с сервером. В листинге 1.3 показана схема работы клиента.

Листинг 1.3. Схема TCP-клиента

```
import java.net.*;
import java.io.*;

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

```
    try{
      Socket sock = new Socket("www.bhv.ru", 1666);

      PrintWriter pw = new PrintWriter(
         new OutputStreamWriter(sock.getOutputStream()), true);

      String req = "Запрос клиента";
      pw.println(req);

      BufferedReader br = new BufferedReader(
         new InputStreamReader(sock.getInputStream()));

      String res = null;
      while ((res = br.readLine()) != null)
         System.out.println(res);

      // Обмен продолжается...

      // По окончании работы потоки и сокет закрываются.
      pw.close(); br.close();
      sock.close();

    }catch(Exception e){}
  }
}
```

На серверной машине сервер — экземпляр класса `ServerSocket` — ждет запрос от сокета клиента, слушая назначенный ему порт. Получив запрос, он соединяется с клиентом методом `accept()`, создав для связи свой сокет — экземпляр класса `Socket`. После этого сервер продолжает слушать порт, обратившись снова к методу `accept()`, и ждет запрос от другого клиента. Сокет сервера открывает входной и выходной потоки для обмена информацией с сокетом клиента. По окончании работы сокеты закрываются. Листинг 1.4 показывает схему работы сервера.

Листинг 1.4. Схема TCP-сервера

```
import java.net.*;
import java.io.*;
```

```
class TCPServer{
  public static void main(String[] args){
    try{
      ServerSocket ss = new ServerSocket(1666); // 1666 — номер порта.

      while (true){
        Socket cl = ss.accept();

        // Для работы сокета cl лучше создать новый подпроцесс.

        BufferedReader br = new BufferedReader(
          new InputStreamReader(cl.getInputStream()));
        String req = br.readLine();

        // Здесь обработка запроса.

        PrintWriter pw = new PrintWriter(
          new OutputStreamWriter(cl.getOutputStream()), true);
        pw.println("Ответ: " + req);

        // Обмен данными продолжается...

        // По окончании обмена потоки и сокет закрываются.
        br.close(); pw.close(); cl.close();

        // Сервер продолжает слушать порт...
      }

    }catch(Exception e){}
  }
}
```

Классы `Socket` и `ServerSocket` содержат большое число методов, с помощью которых можно задать и получить различные параметры сокетов для настройки характеристик связи.

Сокеты дейтаграммного типа UDP

Обмен дейтаграммами с использованием классов пакета java.net происходит через сокеты дейтаграммного типа — экземпляры класса `DatagramSocket`.

Посылка *дейтаграмм* осуществляется методом `send()`, прием — методом `receive()`. Параметром этих методов служит дейтаграмма — экземпляр класса `DatagramPacket`. После получения дейтаграммы, из нее нужно извлечь сообщение методом `getData()`.

В листинге 1.5 дана схема клиента, посылающего дейтаграмму.

Листинг 1.5. Схема UDP-клиента

```
import java.net.*;
import java.io.*;

public class Sender{
   public static void main(String[] args){
      try{
         String message = "Привет!";

         // Преобразуем строку Unicode в массив байтов Cp866.
         byte[] data = message.getBytes("Cp866");

         InetAddress addr = InetAddress.getByName("www.bhv.ru");

         DatagramPacket pack =
            new DatagramPacket(data, data.length, addr, 1234);

         DatagramSocket ds = new DatagramSocket();

         ds.send(pack);

         ds.close();

      }catch(Exception e){
         System.out.println("From client: " + e);
      }
   }
}
```

В листинге 1.6 показана схема сервера, который ждет дейтаграмму, прослушивая порт 1234. После получения дейтаграммы сервер извлекает из нее сообщение и отправляет его в свой стандартный вывод.

Листинг 1.6. Схема UDP-сервера

```
import java.net.*;
import java.io.*;

class Recipient{
   public static void main(String[] args){
      try{
         DatagramSocket ds = new DatagramSocket(1234);

         while(true){
            DatagramPacket pack =
               new DatagramPacket(new byte[2048], 2048);

            ds.receive(pack);

            System.out.println(new String(pack.getData(), "Cp866"));

         }
      }catch(Exception e){
         System.out.println("From server: " + e);
      }
   }
}
```

Новые средства Java API

В следующем выпуске JDK, его версия 1.1, число сетевых средств сильно возросло. Была разработана спецификация компонентов JavaBeans, которые могут совершенно самостоятельно "путешествовать" по сети и встраиваться в приложения, работающие на удаленной машине, как "родные" компоненты. Эта совместимость и взаимозаменяемость достигается четко описанным стандартным интерфейсом, который должен предоставлять каждый JavaBean. Описание содержится в спецификации JavaBeans API Specification, которая доступна по адресу **http://java.sun.com/products/javabeans/docs/spec.html**. В настоящее время все графические компоненты библиотек AWT и Swing, многие классы и целые приложения оформляются как JavaBeans. Классы, помогающие создавать JavaBeans, составили пакет java.beans.

Для того чтобы передавать уже существующие объекты Java по сети, было введено понятие *сериализации* — записи объекта в виде последовательности

байтов. Эта последовательность передается по сети или сохраняется на диске. По мере надобности она преобразуется снова в объект на удаленной машине. Процедуры сериализации и *десериализации* опять-таки предельно упрощены:

```
// Создаем объект
SomeClass object = new SomeClass();


// Работаем с объектом, меняем его состояние...


// Потом, в какой-то момент, сериализуем его.
// Для этого создаем выходной поток в файл
ObjectOutputStream oos = new ObjectOutputStream
   (new FileOutputStream("object.ser"));


// и заносим в него объект
oos.writeObject(object);
```

Все, объект `object` сериализован и записан в файл object.ser. Обратная процедура десериализации выполняется еще проще:

```
// Создаем входной поток
ObjectInputStream ois = new ObjectInputStream
   (new FileInputStream("object.ser"));


// Получаем объект из потока
SomeClass sc = (SomeClass)ois.readObject();
```

Механизм распределенных объектов RMI

В JDK 1.1 был введен также механизм RMI (Remote Method Invocation) обращения к методам объектов, созданных на удаленной виртуальной машине Java. Этот механизм расширяет старый метод вызова удаленных процедур RPC (Remote Procedure Call), позволяя взаимодействовать не только процедурам, но и объектам. RMI использует архитектуру клиент-сервер. Сервер, работающий на удаленной машине, создает объект и ждет обращения к его методам со стороны клиента, предоставляя интерфейс этого объекта. Клиент получает реализацию этого интерфейса в виде *объекта-заглушки* (stub) и обращается к методам заглушки как к методам обычного локального объекта. Взаимодействие заглушки с сервером RMI происходит по специально разработанному протоколу JRMP (Java Remote Method Protocol). Поиск сервера *(lookup)*, соединение с ним, преобразование параметров вызова метода

(marshalling) для пересылки их на удаленную машину, обратная пересылка клиенту результатов выполнения метода, все это скрыто в механизме RMI. Создание программ с применением RMI, конечно, не так просто, как разработка программ, использующих URL, но позволяет отвлечься от массы деталей реализации.

Приведем простейший пример распределенного объекта, выдающего строку по запросу от клиента.

Сначала описываем в интерфейсе методы, которые будет предоставлять распределенный объект, как показано в листинге 1.7. Пусть наш удаленный объект предоставит запросам от клиента всего один метод `getResult()`.

Листинг 1.7. Интерфейс методов распределенного объекта

```
import java.rmi.*;
public interface SimpleRMI extends Remote{
    String getResult() throws RemoteException;
}
```

Интерфейс, описывающий методы, предоставляемые распределенным объектом, должен быть открытым (public) и расширять интерфейс `Remote`. Интерфейс `Remote` не содержит ни одного метода, его не надо реализовывать. Слово Remote служит пометкой, относящей интерфейс к разряду интерфейсов RMI. Каждый метод такого интерфейса должен выбрасывать исключение класса `RemoteException` — суперкласса целого семейства исключений, возникающих при взаимодействии распределенных объектов.

Теперь реализуем интерфейс и создадим сервер. Для простоты оформим все это в одном классе `SimpleRMIImpl`, приведенном в листинге 1.8. Реализация интерфейса должна расширять класс `UnicastRemoteObject`, занимающийся черновой работой по установлению и поддержке сетевой связи с удаленным объектом. Конструктор класса должен выбрасывать исключение `RemoteException`, потому что его выбрасывает конструктор суперкласса, который надо обязательно выполнять.

Листинг 1.8. Реализация интерфейса и сервера RMI

```
import java.rmi.*;
import java.rmi.server.*;

public class SimpleRMIImpl extends UnicastRemoteObject
                    implements SimpleRMI{
```

```
   private String result;
   public SimpleRMIImpl(String name, String result)
                              throws RemoteException{
     super();

     try{
        Naming.rebind(name, this);
     }catch(Exception e){
        System.err.println("From rebind: " + e);
     }
     this.result = result;
   }

   public String getResult() throws RemoteException{
      return result;
   }

   public static void main(String[] args){
      try{
        SimpleRMIImpl simp =
           new SimpleRMIImpl("myRMI", "Привет от RMI");

      }catch(Exception e){
        System.err.println("Can't create server: " +
           e.getMessage());
       }
   }
}
```

Конструктор этого класса сначала вызывает конструктор суперкласса, а затем регистрирует создаваемый объект в RMI-реестре под именем, заданным первым параметром `name`. Для этого используется статический метод `rebind(name, this)` класса `Naming`. *RMI-реестр* — это небольшой сервер имен, который будет прослушивать порт 1099 (по умолчанию), принимать запросы от клиентов и направлять их серверному объекту. Метод `main()` создает распределенный объект, регистрирует его в RMI-реестре под именем `"myRMI"` и передает ему строку "Привет от RMI".

Затем пишем клиентскую программу. Она приведена в листинге 1.9.

Листинг 1.9. Клиент системы RMI

```
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class SimpleRMIClient{
   public static void main(String[] args){
     try{
        SimpleRMI simp = (SimpleRMI)Naming.lookup
             ("rmi://" + args[0] + "/myRMI");

        System.out.println("Ответ от сервера: " +
            simp.getResult());

     }catch(Exception e){
        System.err.println("Can't find the server: " +
            e.getMessage());
     }
   }
}
```

В клиентской программе сначала устанавливается связь с сервером, затем у сервера запрашивается объект `simp`, представляющий заглушку на машине клиента. Все это выполняется одним методом `lookup()`. С объектом `simp` можно обращаться так, как будто это обычная реализация интерфейса `SimpleRMI`, выполненная на машине клиента.

Откомпилируем на серверной машине исходные серверные файлы:

```
javac SimpleRMI.java
javac SimpleRMIImpl.java
```

Потом запустим на серверной машине компилятор RMI:

```
rmic SimpleRMIImpl
```

Компилятор создает два файла: `SimpleRMIImpl_Stub.class`, содержащий класс заглушки, и необязательный файл `SimpleRMIImpl_Skel.class`, необходимый для совместимости со старыми версиями RMI.

Затем запускаем на серверной машине RMI-реестр. В UNIX запускаем его в фоновом режиме:

```
rmiregistry&
```

Для MS Windows используем команду `start`:

```
start rmiregistry
```

Команду `start` можно заменить командой `javaw`.

Запускаем нашу серверную программу, создающую объект:

```
start java SimpleRMIImpl
```

Объект автоматически регистрируется в реестре под именем `"myRMI"`.

С серверной стороны все готово. Объект создан, сервер имен ждет запросы.

На машине клиента компилируем клиентскую программу:

```
javac SimpleRMIClient.java
```

и запускаем ее, указывая адрес серверной машины:

```
java SimpleRMIClient your.server.com
```

Все сделано, на консоли появляется строка, полученная от сервера.

Приведенный пример иллюстрирует только основные возможности RMI, подробнее мы рассмотрим их *в главе 15*.

Использование средств CORBA

В новой версии Java 2 (JDK 1.2 и далее), выпущенной в 1998 году, фирма Sun значительно расширила набор классов, входящий в SDK Standard Edition, переработав и дополнив старые классы, добавив новые классы и пакеты классов.

В стандартную поставку были включены пакеты классов, обеспечивающие связь Java с технологией CORBA (Common Object Request Broker Architecture). Они позволяют объектам Java взаимодействовать с объектами CORBA, созданными на других языках программирования. Стало возможным использовать для пересылки объектов Java транспортный протокол IIOP (Internet Inter-ORB Protocol), применяемый в CORBA. Для включения технологии Java в технологию CORBA был разработан свой брокер объектных запросов Java ORB, удовлетворяющий спецификациям OMG (Object Management Group, консорциум, разрабатывающий стандарты CORBA). Конструкции RMI тоже могут работать отныне поверх IIOP, образуя так называемый механизм RMI-IIOP. Это позволяет объектам, созданным на других платформах, обращаться к RMI-серверам, и создавать CORBA-серверы средствами RMI, используя компилятор `rmic`.

Кроме того, в J2SDK SE включен компилятор `idlj`, преобразующий описания, сделанные на языке IDL (Interface Definition Language), в код Java, и другие средства создания объектов CORBA. Эти средства получили общее название *Java IDL*. Их могут применять программисты CORBA, желающие создавать распределенные приложения с использованием Java.

Реализуем предыдущий пример средствами Java IDL. Сначала опишем метод `getResult()` в интерфейсе, написанном на языке IDL, как показано в листинге 1.10. Язык IDL применяется в технологии CORBA в качестве общего языка, "эсперанто" для описания объектов CORBA. Для каждого конкретного языка должен быть написан компилятор из IDL в этот язык.

Листинг 1.10. Интерфейс распределенного объекта CORBA

```
module SimpleIDL{
  interface Simple{
    wstring getResult();
  };
};
```

Модуль (module) языка IDL — это аналог пакета Java, интерфейсы и методы языка IDL прямо соответствуют интерфейсам и методам языка Java. Тип `wstring` эквивалентен типу `String` языка Java.

Сохраним приведенное описание в файле Simple.idl и преобразуем его в файлы Java с помощью компилятора `idlj`. Для этого наберем в командной строке:

```
idlj -fall Simple.idl
```

Компилятор создает пакет `SimpleIDL`, названный по имени модуля IDL, приведенного чуть выше, в нем есть сразу шесть интерфейсов и классов, которые понадобятся как клиентской, так и серверной программе. Эти интерфейсы и классы занесены в следующие файлы каталога SimpleIDL:

- SimpleOperations.java — интерфейс, описывающий метод `getResult()`;
- Simple.java — интерфейс, определяющий основной тип `Simple` и расширяющий интерфейс `SimpleOperations`;
- _SimpleStub.java — класс-заглушка, реализующий интерфейс `Simple`;
- SimpleHelper.java — абстрактный класс, создающий потоки ввода/вывода для преобразования и передачи информации;
- SimpleHolder.java — класс, определяющий методы чтения из потока ввода и записи в поток вывода;
- SimplePOA.java — абстрактный класс, описывающий задачу и связывающий методы, определенные пользователем, с компонентами CORBA.

Эти интерфейсы и классы не описывают задачу полностью, например, из описания IDL не видно, что метод `getResult()` передает значение поля `result`, и компилятор `idlj` не может сгенерировать правильный код. Поэтому метод `getResult()` не реализован в классе `SimplePOA` и класс `SimplePOA` оставлен абстрактным. Его надо расширить, определив методы пользователя.

Напишем расширение класса `SimplePOA`, реализующее наш распределенный объект. Оно приведено в листинге 1.11.

Листинг 1.11. Реализация интерфейса CORBA

```
import SimpleIDL.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import java.util.Properties;

class SimpleImpl extends SimplePOA{

  private String result;

  public SimpleImpl(String result){
    this.result = result;
  }

  public String getResult(){
    return result;
  }
}
```

Создадим серверную часть программы (листинг 1.12).

Листинг 1.12. Сервер CORBA

```
public class SimpleServer{

  public static void main(String args[]){
    try{
      ORB orb = ORB.init(args, null);

      POA rootpoa = (POA)orb.resolve_initial_references("RootPOA");

      rootpoa.the_POAManager().activate();

      SimpleImpl simp = new SimpleImpl("Привет от CORBA");
```

```
      org.omg.CORBA.Object ref =
          rootpoa.servant_to_reference(simp);

      Simple href = SimpleHelper.narrow(ref);

      org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");

      NamingContextExt ncRef =
          NamingContextExtHelper.narrow(objRef);

      NameComponent path[] = ncRef.to_name("myCORBA");

      ncRef.rebind(path, href);

      orb.run();

    }catch (Exception e){
        System.err.println("ERROR: " + e);
    }
    System.out.println("SimpleServer exiting ...");
  }
}
```

Серверный код выглядит очень сложно и эта сложность не только кажущаяся. Технология CORBA сложна и запутанна, потому что она должна учитывать особенности множества платформ. К счастью для разработчика, большая часть кода стандартна и повторяется из программы в программу.

Создадим клиентскую программу. Она приведена в листинге 1.13.

Листинг 1.13. Клиент CORBA

```
import SimpleIDL.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class SimpleClient{
  static Simple simpleImpl;
```

```
  public static void main(String args[]){
     try{
        ORB orb = ORB.init(args, null);

        org.omg.CORBA.Object objRef =
           orb.resolve_initial_references("NameService");

        NamingContextExt ncRef =
            NamingContextExtHelper.narrow(objRef);

        simpleImpl =
            SimpleHelper.narrow(ncRef.resolve_str("myCORBA"));

        System.out.println("Ответ от сервера: " +
              simpleImpl.getResult());

     }catch(Exception e){
        System.out.println("ERROR : " + e);
     }
   }
}
```

Клиентская программа тоже выглядит сложной, потому что в начале программы требуется создать несколько объектов, необходимых для работы механизмов CORBA. Впрочем, код их создания стандартен и может повторяться без изменений.

Компилируем файлы, находящиеся в каталоге SimpleIDL:

```
javac SimpleIDL/*.java
```

Запускаем брокер объектных запросов `orbd`. В UNIX отправляем его на выполнение в фоновом режиме:

```
orbd -ORBInitialPort 1050&
```

В MS Windows пользуемся командой `start`:

```
start orbd -ORBInitialPort 1050
```

Брокер слушает порт 1050 и ждет запросы от клиентов.

Запускаем написанный нами сервер CORBA.

В UNIX:

```
java HelloServer -ORBInitialPort 1050&
```

В MS Windows:

```
start java SimpleServer -ORBInitialPort 1050
```

Наконец, запускаем клиента:

```
java SimpleClient -ORBInitialPort 1050\
        -ORBInitialHost some.server.com
```

и на экране появляется ответ сервера.

Как видите, создание распределенных объектов и связь с ними по технологии CORBA осуществляются гораздо сложнее, чем те же действия, выполненные по технологии RMI. Это плата за универсальность технологии CORBA и широкие возможности, которые она предоставляет.

Я привел самый простой пример, какой только можно написать в технологии CORBA. Подробнее связь Java и CORBA будет рассмотрена *в главе 19*.

Итак, сегодня, используя технологию Java, можно организовать сетевое соединение средствами низкого уровня через сокеты потока TCP, сокеты дейтаграмм UDP или посредством адресации URL. При этом очень многое остается на долю программиста и даже для создания простого сетевого приложения приходится писать огромное количество строк кода на языке Java. Механизм более высокого уровня RMI предоставляет возможность применять методы распределенных объектов. Взаимодействие с объектами CORBA средствами Java IDL позволяет использовать всю мощь технологии CORBA.

Использование технологии WWW

В наши дни все большее развитие получает технология World Wide Web. С помощью механизма *CGI* Web-сервер может вызвать на выполнение некоторую программу, передать на ее стандартный ввод параметры, полученные от клиента, и отправить обратно клиенту результаты выполнения программы, которые та направила в свой стандартный вывод.

Сервлеты

Такую программу можно написать и на языке Java (см. листинг 1.2), но для ее выполнения надо запускать виртуальную машину Java (JVM). Если вызовы программы будут повторяться, то создание JVM при каждом вызове программы приведет к большому расходу ресурсов. Гораздо удобнее не использовать CGI, а присоединить к Web-серверу дополнительный модуль, запускающий JVM при первом вызове первой программы и выполняющий все программы под ее управлением. Кроме запуска программ, такой модуль обычно создает несколько объектов, взаимодействующих с Web-сервером. Эти объекты принимают запросы, полученные Web-сервером, преобразуют

их для передачи прикладным программам, принимают ответы от этих программ, образуя, таким образом, операционную среду для выполнения программ, написанных на Java.

Web-сервер с таким дополнением называется *Java Web-сервером*, а программы, написанные на языке Java и выполняющиеся под управлением Java Web-сервера, называются *сервлетами* (servlets). Название этих программ неслучайно перекликается со словом "апплеты" — подобно тому, как апплеты расширяют возможности браузера, включая в него анимацию и интерактивность, сервлеты расширяют возможности Web-сервера, превращая его из простого переносчика данных в мощного обработчика информации.

Для облегчения разработки сервлетов фирма Sun создала большой набор интерфейсов и классов, получивший название Servlet API (Application Programming Interface). Эти классы скрывают детали реализации и обеспечивают взаимодействие с Java Web-сервером, перекладывая на него все функции сетевого соединения. Образуется так называемый *Web-контейнер*, в котором выполняются сервлеты. На жаргоне его называют "сервлетным движком" (servlets engine). Для встраивания в Web-контейнер сервлеты должны удовлетворять требованиям некоторых интерфейсов, входящих в Servlet API, и реализованных Java Web-сервером.

На рынке появилось уже много Web-контейнеров для различных Web-серверов. Это свободно распространяемый "движок" Tomcat, разрабатываемый сообществом Apache Software Foundation в рамках проекта Jacarta. Его можно получить по адресу **http://jacarta.apache.org/**. Этот Web-контейнер используется самостоятельно или в составе Web-сервера Apache. Кроме того, он является составной частью некоторых серверов приложений.

Большой популярностью пользуется Web-контейнер Resin фирмы Caucho (**http://www.caucho.com/products/resin/**).

Активно применяется Web-контейнер JRun фирмы Allaire (**http://www.allaire.com/products/jrun/**).

Java 2 Enterprise Edition

Пакеты интерфейсов и классов, составивших Servlet API, входят как составная часть в обширный набор утилит и пакетов классов, обеспечивающих реализацию технологии Java на сервере. Этот набор назван Java 2 SDK Enterprise Edition, сокращенно J2EE SDK. Его можно свободно получить по адресу **http://java.sun.com/j2ee/**. Он не заменяет набор пакетов классов и утилит Java 2 SDK Standard Edition (сокращенно J2SE), а дополняет его и устанавливается на машину после установки J2SE. Каждый выпуск J2EE SDK оптимизирован под некоторый выпуск J2SE SDK, например, J2EE SDK 1.3.1 оптимизирован под J2SE SDK 1.3.1_02.

Кроме Servlet API, в состав J2EE SDK входят еще пакеты *Java Mail*, обеспечивающие работу электронной почты, пакет *JMS* (Java Message Service) средств обмена сообщениями, пакеты управления транзакциями, пакеты для работы с документами XML, пакеты создания компонентов EJB (Enterprise Java Beans). *Части IV* и *V* книги полностью посвящены работе с пакетами J2EE SDK.

Сервер J2EE

Кроме пакетов классов, J2EE SDK содержит сервер, предоставляющий все перечисленные услуги. Он называется *J2EE-сервером* и запускается из командной строки вызовом на выполнение командного файла j2ee. В J2EE-сервер версии 1.3.1 встроен Web-контейнер Tomcat 4.0, в который внесены некоторые изменения по сравнению с отдельным выпуском контейнера. По умолчанию он прослушивает порт 8000, а для протокола HTTPS (HTTP over SSL) назначен порт 7000. Кроме того, EJB-сервер включен в службу имен COS Naming технологии CORBA, сообщения которой он прослушивает, используя порт 1050. EJB-контейнер прослушивает порт 9191. Для правильной работы J2EE-сервера эти порты не должны прослушиваться другими программами.

Такой многофункциональный сервер, который может и отправлять клиенту запрошенные файлы, и выполнять сервлеты, и служить почтовым агентом, и участвовать в обмене сообщениями, и быть брокером объектных запросов CORBA, называется *сервером приложений* (application server). На рынке приложений есть уже много серверов приложений: WebLogic фирмы BEA Systems (**http://www.weblogic.com/**), WebSphere фирмы IBM (**http://www.ibm.com/**), свободно распространяемый сервер JBoss (**http://www.jboss.org/**), EAServer фирмы Sybase (**http://www.sybase.com/**), Orbix E2A фирмы IONA (**http://www.iona.com/**), OAS (Oracle Application Server) фирмы Oracle Corporation (**http://www.oracle.com/**), BAS (Borland Application Server) фирмы Borland Software Corporation (**http://www.borland.com/**) и др. Они различаются ценой, производительностью и дополнительными сервисными возможностями.

Сравнением серверов приложений занимается много энтузиастов, результаты сравнения можно посмотреть на сайтах:

- **http://www.app-serv.com/contend.html**;
- **http://www.techmetrix.com/trendmarkers/techmetrixasd.php**;
- **http://www.flashline.com/Components/appservermatrix.jsp**.

Инсталляция J2EE SDK

Все, что входит в J2EE SDK 1.3, в том числе и документация, поставляется одним файлом, представляющим собой самораспаковывающийся архив величиной около 17 Мбайт. В процессе распаковки вы указываете каталог, в

который будет записано все содержимое архива, например, D:\j2sdkee1.3.1. После распаковки для использования классов, входящих в Servlet API, надо к переменной `CLASSPATH` добавить путь к ним, например, в MS Windows надо набрать в командной строке:

```
set CLASSPATH=%CLASSPATH%;D:\j2sdkee1.3.1\lib\j2ee.jar
```

Перед первым запуском J2EE-сервера переменной `JAVA_HOME`, находящейся в файле D:\j2sdkee1.3.1\bin\userconfig.bat, должно быть присвоено полное имя каталога, в который помещен J2SE, например:

```
set JAVA_HOME=D:\jdk1.4.0
```

а переменной `J2EE_HOME` — присвоить имя каталога, в который распакован J2EE SDK, например:

```
set J2EE_HOME=D:\j2sdkee1.3.1
```

Простейший сервлет

Вернемся к сервлетам. Основные интерфейсы и классы Servlet API собраны в пакете javax.servlet. Основа этого пакета — интерфейс `Servlet` — описывает метод `init()`, который инициализирует сервлет, метод `destroy()`, завершающий работу сервлета, метод `service()`, в котором сосредоточена вся работа сервлета. Интерфейс `Servlet` частично реализован абстрактным классом `GenericServlet` и его расширением — абстрактным классом `HttpServlet`, учитывающим особенности протокола HTTP. В этих классах не реализован только метод `service()`. Поэтому, чтобы создать свой сервлет, достаточно расширить класс `GenericServlet` или `HttpServlet`, реализовав в расширении метод `service()`.

В листинге 1.14 показано, как просто выглядит сервлет, решающий ту же задачу, что и CGI-программа, приведенная в листинге 1.2. Он принимает строку текста и отправляет ее обратно с некоторыми изменениями.

Листинг 1.14. Простейший сервлет

```
import java.io.*;
import javax.servlet.*;

public class SimpleServlet extends GenericServlet{
   public void service(ServletRequest req, ServletResponse res)
                  throws IOException, ServletException{

      // Указываем кодировку полученного сообщения.
      req.setCharacterEncoding("KOI8-R");
```

```
        // Получаем символьный входной поток.
        BufferedReader br = req.getReader();

        // Читаем первую строку сообщения.
        String mess = br.readLine();

        br.close();
        // . . . . . . . .
        // Здесь обработка полученной информации...
        // . . . . . . . .
        // Задаем MIME-тип и кодировку ответа.
        res.setContentType("text/plain;charset=KOI8-R");

        // Получаем символьный выходной поток.
        PrintWriter pw = res.getWriter();

        // Заносим в него строку.
        pw.println("Ответ сервлета: " + mess);

        // Очищаем буфер выходного потока.
        pw.flush();

        pw.close();
    }
}
```

Как видите, в сервлете почти ничего нет. Всю основную работу — сетевое соединение, прием и отправку сообщения, открытие входного и выходного потоков — берет на себя Web-контейнер. Он предоставляет сервлету два объекта. Первый объект `req` реализует интерфейс `ServletRequest` или его расширение `HttpServletRequest`. В этом объекте содержится запрос, направленный сервлету, и параметры запроса. Сервлет может получить тело запроса из символьного входного потока класса `BufferedReader`, который открывается методом `getReader()`, или из байтового потока класса `ServletInputStream`, открываемого методом `getInputStream()`. Для получения параметров запроса объект `req` предоставляет множество методов `getXxx()`.

Второй объект `res` пуст, в него надо занести результаты работы сервлета. Для этого интерфейс `ServletResponse` и его расширение `HttpServletResponse` предоставляют выходные потоки и методы `setXxx()` для задания параметров ответа.

В сервлете нет метода `main()`. Как видно из листинга 1.14, необязателен даже метод `init()`. Инициализацию сервлета берет на себя Web-контейнер, но ему надо сообщить о появлении нового сервлета. Разные серверы приложений делают это по-разному. Для одного сервера приложений достаточно просто откомпилировать сервлет и поместить файл, содержащий его байт-коды, в определенный каталог. Очень часто этот каталог называется servlets. Сервер приложений читает при запуске этот каталог и регистрирует у себя сервлеты, расположенные в нем. Для других серверов приложений сервлет надо поместить в Web-контейнер. Эта довольно сложная процедура называется *установкой* (deployment) сервлета в контейнер.

Установка сервлета в контейнер

При установке сервлета в контейнер сначала создается файл с расширением war (Web ARchive), содержащий описание сервлета, HTML-файлов, файлов с изображениями, короче говоря, всех файлов, связанных с сервлетом. В Web-приложении может быть несколько сервлетов, соответственно получается несколько WAR-файлов. Потом все такие описания собираются в файл приложения с расширением ear (Enterprise ARchive), в который могут входить описания и других компонентов, образующих приложение. Затем записывается *дескриптор установки* — файл описания приложения на языке XML (eXtended Markup Language). Наконец, создаются необходимые каталоги для хранения этих описаний и файлов.

Каждый сервер приложений, а также многие средства разработки имеют в своем составе утилиты или make-файл для облегчения процесса установки сервлетов. В J2EE SDK это утилита deploytool. Она запускается из командной строки, но затем открывает графическое окно и проводит разработчика через все стадии установки Web-приложения. Перед ее использованием следует запустить J2EE-сервер, набрав в командной строке `j2ee` или `j2ee -verbose`, если вы хотите получать сообщения о процессе запуска сервера.

На одной из стадий установки надо задать имя каталога Web-приложения (WAR Context Root). Для примера, приведенного в листинге 1.14, пусть это имя каталога будет SimpleWAR. Утилита deploytool создаст каталог SimpleWAR и поместит в него и в его подкаталоги все файлы, относящиеся к устанавливаемому Web-приложению, в том числе и копию сервлета.

Виртуальные каталоги

Web-серверы, серверы приложений, и J2EE-сервер в том числе, делают доступными для внешнего пользователя только определенные каталоги, меняя их имена и иерархию. Внешний пользователь видит *виртуальную структуру каталогов* сервера. Указав в адресе URL корневой каталог, например,

http://www.bhv.ru/, пользователь попадает, на самом деле, в каталог D:\Inetpub\wwwroot\, или D:\j2sdkee1.3\public_html\ машины, находящейся по указанному URL, или в какой-то другой каталог, это зависит от Web-сервера. По общему соглашению сервлеты обычно показываются в виртуальном каталоге с именем servlet, хотя на сервере они могут быть помещены совсем в другой каталог. Поэтому для вызова сервлета SimpleServlet, записанного в листинге 1.14 и установленного так, как описано в предыдущем абзаце, надо в листинге 1.1 указать адрес URL следующим образом:

```
http://www.bhv.ru:8000/SimpleWAR/servlet/SimpleServlet
```

Впрочем, сейчас это правило почти не соблюдается, при установке сервлета в Web-контейнер можно отобразить (to map) реальный каталог сервлета в любой виртуальный каталог.

При отладке сервлета его можно изменять и компилировать в его собственном каталоге, не обращаясь к копии, лежащей в одном из подкаталогов каталога SimpleWar. После компиляции во многих серверах приложений нужно переустановить сервлет. В утилите deploytool для этого надо в меню **Tools** выбрать пункт **Update and Redeploy**. Копию сервлета менять не надо, утилита все сделает автоматически.

Подробно работа с сервлетами будет рассмотрена *в главе 20*.

Работа сервлетов по протоколу HTTP

В технологии WWW связь между машинами чаще всего устанавливается с помощью протокола HTTP. Например, типичный запрос от HTML-формы

```
<html><body>
<form method="GET" action="SomeWAR/servlet/SomeServlet" name="f1">
   <input type="text" name="usr">
   <input type="text" name="pw">
   <input type="submit">
</form></body></html>
```

посылается на Web-сервер в виде строки

```
"GET /SomeWAR/servlet/SomeServlet?usr=John+Smith&pw=a%1C!xD
   HTTP/1.1\r\n"
```

За этой строкой идут строки заголовка запроса. В строке, написанной выше, после адреса URL за вопросительным знаком идут имена параметров, определенные в HTML-форме, и их значения, причем в значениях параметров пробелы заменены плюсом, непечатные и специальные символы записаны в 16-ричной форме в виде %FF, а параметры разделены амперсандом. Правила кодирования описаны в спецификации CGI. Каждая CGI-программа

должна выполнить синтаксический разбор полученной строки, сделав обратные замены и выделив полезную информацию.

В Servlet API для удобства работы по протоколу HTTP разработан пакет классов и интерфейсов javax.servlet.http. Для создания объекта, содержащего запрос, сделанный по протоколу HTTP, в этом пакете предусмотрен интерфейс `HttpServletRequest`. Web-контейнер реализует этот интерфейс, производя синтаксический разбор поступившего HTTP-запроса и предоставляя результаты разбора методами `getXxx()`.

После синтаксического разбора запроса Web-контейнер создает объект типа `HttpServletRequest`. В следующем примере этот объект назван `req`. Значения параметров запроса сервлет может получить из объекта `req` в виде строки, используя имена параметров, переданные из HTML-формы, следующим образом:

```
String user = req.getParameter("usr");
String password = req.getParameter("pw");
```

Чтобы использовать особенности методов передачи информации `GET`, `POST`, `HEAD`, `OPTIONS`, `PUT`, `TRACE` протокола HTTP, в классе `HttpServlet` есть методы `doGet()`, `doPost()`, `doHead()`, `doOptions()`, `doPut()` и `doTrace()`. Метод `service()` играет только роль диспетчера. Он определяет метод передачи данных протокола HTTP и обращается к конкретному методу `doXxx()`.

Результаты своей работы сервлет записывает в объект типа `HttpServletResponse`, назовем его `resp`, предварительно созданный Web-контейнером. В объекте `resp` ответ сервлета преобразуется по правилам протокола HTTP, а затем передается Web-серверу для отправки клиенту. При этом документ HTML сервлет должен создавать самостоятельно, примерно так:

```
// . . . . . . .
int n = 5;
Pupil pupil = new Pupil();

resp.setContentType("text/html;charset=windows-1251");

PrintWriter pw = resp.getWriter();

pw.println("<!doctype html public \"-//W3C//DTD"+
           "HTML 4.0 Transitional//EN\">");

pw.println("<html><head><title> Результат </title>");

pw.println("<META http-equiv=Content-Type" +
           "content=\"text/html; charset=windows-1251\">");
```

```
pw.println("</head><body>");

pw.println("<h2> Результат решения задачи " + n + "</h2>");

pw.println("<p>Ваше решение: " +
            pupil.task[n].getSolution() + "</p>");

pw.println("<p>Ваша оценка: " +
            pupil.task[n].getResult() + "</p>");

pw.println("</body></html>");

pw.flush();

pw.close();
```

Такую рутинную работу приходится выполнять всякий раз при написании сервлета. Посмотрите любой большой производственный сервлет — он весь испещрен обращениями к методам `println()`.

Язык JSP

Для того чтобы избавить разработчика от рутины при создании сервлетов, в рамках J2EE SDK разработан язык *JSP* (JavaServer Pages) — совокупность тегов специального вида и интерпретатор этих тегов, создающий по ним окончательный сервлет. Правила построения и использования языка JSP описаны в его спецификации, доступной по адресу **http://java.sun.com/products/jsp/**, а классы и интерфейсы интерпретатора языка JSP собраны в пакеты javax.servlet.jsp и javax.servlet.jsp.tagext, входящие в состав J2EE SDK.

Теги языка JSP включаются в обычный документ HTML подобно HTML-тегам, но в них включен (без пробелов!) знак процента следующим образом:

```
<% тело тега %>
```

Набор тегов определен спецификацией JSP. Например:

```
<%-- Комментарий, не посылается клиенту %>
<%@ page Задает параметры HTML-страницы %>
<%@ include file= Включает файл, имя файла задается в виде URL %>
<%! Операторы определения объектов Java %>
<%= Выражение Java, оно вычисляется,
    результат записывается в HTML-страницу %>
<% Фрагмент кода Java, так называемый скриптлет %>
```

При написании тегов JSP следует строго соблюдать правила их синтаксиса, особенно следить за отсутствием пробелов в нужных местах, и за выбором регистра букв — язык JSP, как и язык Java, различает прописные и строчные буквы.

Язык JSP допускает определение и своих собственных, *заказных*, тегов (custom tags). Вместе с заданием имени заказного тега надо определить классы Java, реализующие его выполнение.

Вот как можно написать предыдущий фрагмент сервлета с использованием JSP:

```
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN">

<%@ page contentType="text/html;charset=windows-1251" %>

<html>
<head><title> Результат </title>
<META http-equiv=Content-Type
           content="text/html; charset=windows-1251">
</head>
<body>
<%!
  int n = 5;
  Pupil pupil = new Pupil();
%>
<h2> Результат решения задачи <%= n %> </h2>

<p>Ваше решение:
<%= pupil.task[n].getSolution()%> </p>

<p>Ваша оценка:
<%= pupil.task[n].getResult() %></p>

</body>
</html>
```

Это описание сохраняется в файле с расширением jsp и устанавливается в Web-контейнер, подобно сервлету. Никакой предварительной компиляции не требуется, она выполняется автоматически при вызове JSP-файла. Для этого в HTML-форме или адресе URL указывается имя JSP-файла.

Подробно язык JSP будет рассмотрен *в главе 20*.

Компоненты EJB

Любую задачу на сервере можно, в принципе, решить сервлетами и документами JSP, но всякая нетривиальная задача потребует написать большое количество строк кода на языке Java. Сервлеты будут громоздкими и необозримыми, а главное, будут дублировать часть кода, связанную с сетевыми соединениями, установлением сеанса связи с пользователем и обеспечением надежности сеанса, идентификацией пользователя и правами доступа, отслеживанием транзакций и сохранением их промежуточных результатов, и т. д., и т. д.

Пытаясь облегчить жизнь разработчика серверных приложений, фирма Sun Microsystems пошла привычным путем. Она разработала спецификацию компонентов EJB (Enterprise Java Beans), пакет классов и интерфейсов javax.ejb, включенный в J2EE SDK, и реализовала эти интерфейсы в EJB-контейнере, входящем в сервер приложений. *EJB-контейнер* выполняет компоненты EJB подобно тому, как Web-контейнер выполняет сервлеты.

Компоненты EJB могут быть трех типов, определяемых интерфейсами `EntityBean`, `SessionBean` и `MessageDrivenBean` пакета javax.ejb.

- Первый тип — `EntityBean` — удобен для извлечения информации из баз данных и ее преобразования в объекты Java. С помощью методов этих объектов информация преобразуется к тому виду, который нужен клиенту. Компоненты этого типа могут взаимодействовать сразу с несколькими клиентами и связываться с несколькими базами данных. Они необходимы для создания постоянных (persistent) объектов, хранящихся в базе данных.
- Второй тип EJB-компонентов — `SessionBean` — более удобен для организации сеанса связи с пользователем и сложных вычислений по запросам клиента. Поэтому компоненты этого типа могут связываться только с одним клиентом. Они могут сохранять свое состояние, передавая его от запроса к запросу (stateful session beans), при создании сеанса связи с клиентом, или терять это состояние (stateless session beans).
- Третий тип — `MessageDrivenBean` — предназначен для асинхронного обмена сообщениями по технологии *JMS* (Java Message Service).

Для того чтобы создать свой компонент EJB, разработчик EJB-компонентов должен, разумеется, реализовать один из этих интерфейсов в своем классе, но этого недостаточно для установки компонента в EJB-контейнер и его использования. Необходимо написать еще два интерфейса, расширяющих интерфейсы `EJBObject` и `EJBHome`.

Первый интерфейс типа `EJBObject`, называемый *remote-интерфейсом* (remote interface), описывает методы, предоставляемые EJB-компонентом клиенту. Он подобен интерфейсу типа `Remote` механизма RMI (см. листинг 1.7), и IDL-описанию, принятому в технологии CORBA (см. листинг 1.10).

Второй интерфейс типа `EJBHome`, называемый *home-интерфейсом* (home interface), служит для создания объекта EJB, реализующего remote-интерфейс, методом `create()`, его поиска и удаления методом `remove()`. Дело в том, что EJB-контейнер создает и уничтожает объект EJB только по запросу от клиента. Клиентская программа не работает непосредственно с EJB-компонентом. Она имеет дело только с объектами, реализующими home- и remote-интерфейсы. Эти объекты взаимодействуют с EJB-контейнером, который и обращается к компоненту EJB для решения поставленной клиентом задачи.

Подробному построению EJB-компонентов посвящены *главы 21—23*, а пока поясню их создание на простом примере реализации с помощью EJB-контейнера задачи обращения к удаленному методу `getResult()`, которую мы решали методами RMI и CORBA (см. листинги 1.7—1.13).

Сначала напишем remote-интерфейс, назовем его `Simple`. В нем описываются методы, предоставляемые клиенту компонентом EJB. В нашем примере предоставляется только один метод, как показано в листинге 1.15.

Листинг 1.15. Простейший remote-интерфейс

```
import java.rmi.*;
import javax.ejb.*;

public interface Simple extends EJBObject{
   public String getNewStud() throws RemoteException;
   public Vector getCourses() throws RemoteException;

}
```

Потом описываем home-интерфейс. Его имя складывается из имени remote-интерфейса и слова Home. В нем сообщается только то, что метод `create()` должен создавать объект типа `Simple`: Интерфейс приведен в листинге 1.16.

Листинг 1.16. Простейший home-интерфейс

```
import java.rmi.*;
import javax.ejb.*;

public interface SimpleHome extends EJBHome{
   Simple create() throws CreateException, RemoteException;
}
```

Теперь создаем сам EJB-компонент. Он показан в листинге 1.17. Имя компонента обычно складывается из имени remote-интерфейса и слова EJB. В нашем простейшем примере он будет иметь тип `SessionBean` без сохранения состояния. Нам не понадобится ни один из четырех методов интерфейса `SessionBean`, мы применим их пустую реализацию. Это означает, что будет выполняться та реализация, которая сделана в EJB-контейнере. Кроме того, с той же целью мы используем пустую реализацию метода `ejbCreate()`. Этот метод применяется контейнером для создания объекта EJB.

Листинг 1.17. Простейший компонент EJB

```
import java.rmi.*;
import javax.ejb.*;

public class SimpleEJB implements SessionBean{

   private String result = "Привет от EJB";

   public String getResult(){
      return result;
   }

   // EJB-контейнер будет использовать свою реализацию.
   public void ejbCreate(){}

   // Реализация методов интерфейса.
   public void setSessionContext(SessionContext ctx){}
   public void ejbRemove(){}
   public void ejbActivate(){}
   public void ejbPassivate(){}
}
```

Наконец, в листинге 1.18, приведем клиентскую программу, которая будет использовать метод `getResult()` EJB-компонента `SimpleEJB`. Большую часть кода составляет поиск объекта EJB. Для этого используется система именования и поиска объектов *JNDI* (Java Naming and Directory Interface), входящая в технологию Java.

Листинг 1.18. Простейший клиент EJB

```
import javax.naming.*;
import javax.rmi.*;

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

    try{
      // Создаем начальный контекст.
      Context c = new InitialContext();

      // Ищем в нем ссылку на объект с именем "simple".
      Object ref = c.lookup("java:comp/env/ejb/simple");

      // Получаем объект по ссылке и приводим к нужному виду.
      SimpleHome simphome =
         (SimpleHome)PortableRemoteObject.narrow(ref, SimpleHome.class);

      // Создаем объект EJB.
      Simple simp = simphome.create();

      // Используем его методы.
      System.out.println("Результат: " + simp.getResult());

      c.close();

    }catch(Exception e){
      System.err.println("From client: " + e);
    }
  }
}
```

Исходные тексты EJB-приложения готовы, теперь их надо откомпилировать и установить в EJB-контейнер. Но сначала несколько слов о системе именования и поиска объектов JNDI.

Система именования JNDI

Система именования JNDI дает каждому зарегистрированному в ней объекту имя в виде строки типа `String` или объекта, реализующего интерфейс `Name`. В системе JNDI этот процесс называется *связыванием* (bind) имени и объекта. Это похоже на то, как в файловой системе определяются имена или дескрипторы файлов. Каждое имя объекта действует в некотором *контексте* (context) — объекте, реализующем интерфейс `Context`, — подобно тому, как имя файла действует в своем каталоге. Чтобы отличить друг от друга два файла с одним и тем же именем, надо уточнить их имена с помощью имени каталога. Точно так же в системе JNDI имя объекта уточняется именем контекста. Сходство усиливается тем, что контекст может храниться в другом контексте как один из его объектов. Образуется иерархия вложенных контекстов, аналогичная иерархии каталогов в файловой системе.

На этом сходство заканчивается. В системе JNDI нет корневого контекста. Перед работой с JNDI надо создать *начальный контекст* (initial context) — объект класса `InitialContext`. В него методами `bind(String, Object)` или `bind(Name, Object)` заносятся объекты под именами, заданными первым параметром этих методов. В контексте можно создать *вложенный контекст* (subcontext) методами `createSubcontext(String)` или `createSubcontext(Name)`. Список всех имен можно получить методами `list(String)` или `list(Name)`, а отыскать объект по имени — методами `lookup(String)` или `lookup(Name)`.

Для того чтобы можно было отыскать объект, находящийся в другом контексте, в системе JNDI создаются *ссылки* (reference) — объекты класса `Reference`. Они похожи на символические ссылки в файловой системе UNIX.

Подробному рассмотрению средств JNDI посвящена *глава 17*.

Установка компонента EJB в контейнер

Вернемся к нашему примеру. После компиляции файлов их надо установить в EJB-контейнер. В разных серверах приложений это осуществляется по-разному. В утилите deploytool после задания имени файла приложения, например, Simple.ear, надо создать объект `Enterprise Bean` и поместить в него файлы Simple.class, SimpleHome.class и SimpleEJB.class. При создании объекта надо задать его свойства: `Session`, `Stateless`, класс объекта `SimpleEJB` и интерфейсы, как показано на рис. 1.1.

После этого надо создать объект `Application Client` и поместить в него файл Client.class. Затем для этого объекта следует указать контекст, открыв вкладку **EJB Refs** и, щелкнув кнопкой мыши на кнопке **Add**, записать в электронной таблице строку

```
ejb/simple, Session, Remote, SimpleHome, Simple
```

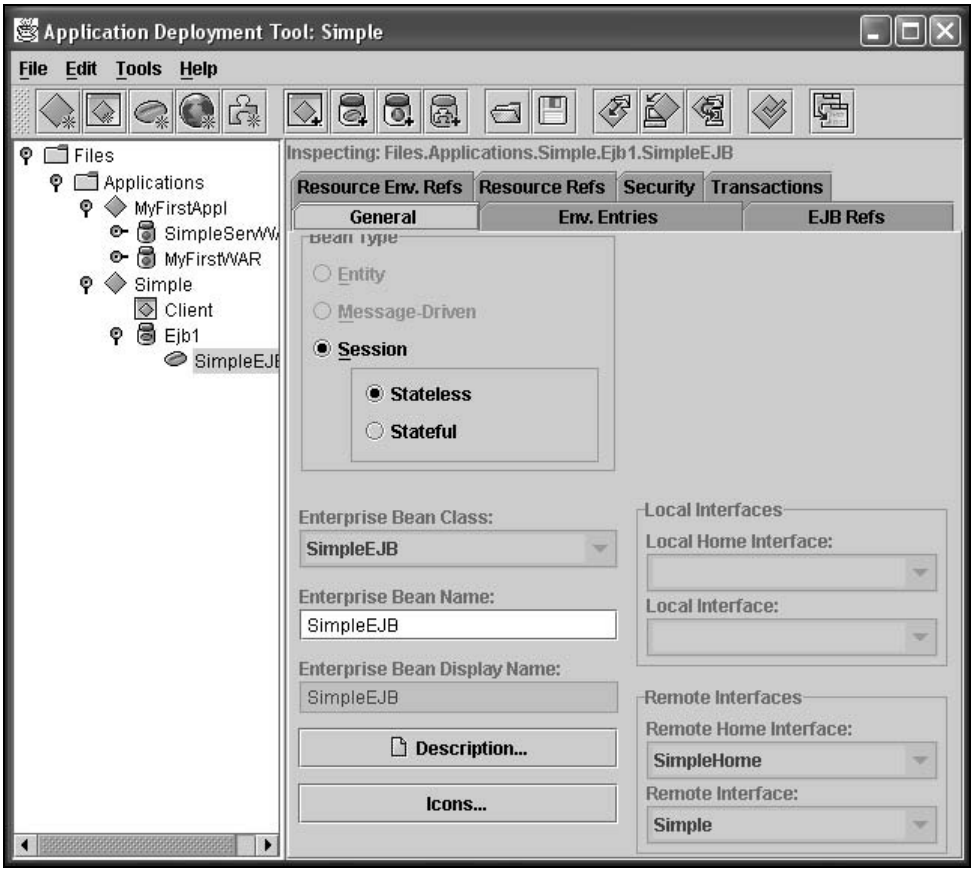


Рис. 1.1. Окно **Application Deployment Tool: Simple**

Выделив в дереве объектов слева объект `Simple`, надо открыть вкладку **JNDI Names** и записать в электронной таблице в колонке **JNDI Name** строку `Simp`. Ту же строку надо повторить ниже в таблице **References**. Строка в этой таблице должна выглядеть так:

```
EJB Ref,    Client,    ejb/simple,    Simp
```

В результате этих манипуляций в стандартном пространстве имен `java:comp` в контексте `env/ejb`, в котором J2EE-сервер хранит имена компонентов EJB, создается JNDI-имя `simple` нашего EJB-компонента. В листинге 1.18 мы обращаемся к объекту EJB по составному имени `java:comp/env/ejb/simple`. Конечно, для нашего примера все эти сложности излишни, но для большой системы с множеством распределенных объектов они необходимы.

Наконец, с помощью утилиты deploytool надо создать файл SimpleClient.jar в формате JAR, содержащий все файлы распределенного EJB-приложения, установив флажок **Return Client Jar**. Этот исполняемый файл можно запускать на виртуальной машине Java, отличной от той, на которой работает J2EE-сервер. После сетевого соединения J2EE-сервер загрузит из этого JAR-файла все необходимое для работы приложения. При запуске файла SimpleClient.jar надо задать массу параметров, подключающих библиотеки классов. Для облегчения этой задачи в каталоге %J2EE_HOME%\bin есть командный файл runclient.bat — утилита, которая содержит все параметры запуска клиентской программы. С помощью этой утилиты наш пример можно выполнить, написав следующую командную строку:

```
runclient -client Simple.ear -name Client -textauth
```

После запуска в командной строке будут запрошены имя и пароль, надо ввести имя `guest` и пароль `guest123`, после чего, наконец, появится результат выполнения метода `getResult()`.

Как видите, установка компонентов EJB в контейнер — задача непростая. Мы обсудим ее подробно в *главе 26*.

Распределенное приложение Java

Итак, мы рассмотрели основные средства создания распределенного приложения, предлагаемые технологией Java. Технология Java включает в себя еще средства обмена сообщениями JMS (Java Message Service), которое мы рассмотрим в *главе 16*, средства электронной почты JavaMail, изложенные в *главе 18*, средства управления транзакциями JTA (Java Transaction API), описанные в *главе 24*. Все большую роль сейчас начинает играть язык XML (eXtended Markup Language). В состав J2EE включено несколько пакетов для связи объектов Java с документами XML. Они составили комплекс средств JAXP (Java API for XML Parsing), который мы рассмотрим в *главе 25*.

Наконец, построение современной информационной системы немыслимо без связи с базами данных. Технология Java для связи с базами данных предлагает средства JDBC (это не аббревиатура, хотя часто расшифровывается как Java DataBase Connection). Им полностью посвящена *часть III* книги.

Эти средства позволяют создать распределенное приложение, используя только технологию Java. На стороне клиента можно запустить приложение Java, построенное средствами графических библиотек AWT или Swing, или загрузить апплет в обычный браузер. Из приложения или апплета с помощью JDBC можно обратиться с запросами к базе данных, расположенной на сервере, используя сетевые средства Java. Мы построим подобную систему в *части III* книги.

Такая двухслойная схема клиент-сервер проста в реализации, но неудобна в эксплуатации. Пересылка больших выборок из базы сильно загрузит сеть, а

обработка ее клиентской программой сделает клиентское приложение или апплет слишком громоздкими и потребует для клиента мощную дорогостоящую машину. Кроме того, всякое изменение алгоритма обработки данных приведет к необходимости замены клиентской программы на всех компьютерах.

Эти недостатки устраняет перенос обработки полученных из базы данных на сервер. На серверной машине запускается приложение, принимающее запросы от клиента и обращающееся для их выполнения к базе данных средствами JDBC. Это же приложение обрабатывает данные перед отправкой клиенту. Построить приложение можно средствами RMI или сервлетами плюс JSP при наличии сервера приложений.

Такое приложение образует *промежуточный слой* (middleware) трехслойного распределенного приложения. Введение промежуточного слоя позволяет разгрузить клиентскую машину. Клиентская часть приложения теперь только обслуживает интерфейс пользователя, принимает его указания и показывает отчеты. В большинстве случаев достаточно использовать обычный браузер, отправляя запросы из HTML-форм и принимая документы, написанные на языке HTML или на языке XML.

Теперь вся сложность распределенного приложения перемещается в промежуточный слой. Он может работать на сервере или на отдельной машине. Его можно разработать как простое приложение Java, организовав связь с другими частями приложения через сокеты или механизм RMI. Но это потребует написания кода большого объема, реализующего рутинные операции. Гораздо удобнее запустить сервер приложений со встроенным Web-контейнером и реализовать промежуточный слой сервлетами и страницами JSP. Такой способ позволит возложить на сервер приложений всю работу по сетевому соединению. Облегчить обработку информации в промежуточном слое помогут EJB-компоненты. Они возьмут на себя создание сеанса связи с клиентом, выборку информации из базы данных, управление транзакциями.

Таким образом, промежуточный слой распределенного приложения делится еще на два слоя: *Web-слой*, реализованный сервлетами и JSP и управляемый Web-контейнером, и *EJB-слой*, реализованный компонентами EJB и управляемый EJB-контейнером. Web-слой располагается ближе к клиенту. Он принимает запросы от клиента, анализирует их и передает в EJB-слой. EJB-слой находится ближе к серверу. Компоненты EJB с помощью JDBC извлекают из базы данных запрошенную информацию, преобразуют ее и передают обратно в Web-слой. Тот формирует документ HTML или XML и отправляет клиенту.

Итак, развитое распределенное приложение, построенное средствами Java, состоит из четырех слоев:

- клиент, реализованный как апплет или приложение Java;
- Web-слой, реализованный сервлетами и JSP;

- EJB-слой, реализованный компонентами EJB;
- система управления базами данных на сервере.

Один или несколько слоев могут отсутствовать. Это придает гибкость проектированию распределенного приложения. Кроме того, разработчики систем управления базами данных сейчас внедряют в свои серверы Web- и EJB-контейнеры, придавая им свойства серверов приложений. Для таких СУБД достаточно написать клиентское приложение, связывающееся с СУБД по сети. Происходит возврат к схеме клиент-сервер на новом уровне.

Таким путем идет корпорация Oracle. Она встроила виртуальную машину Java в ядро своей СУБД Oracle Server и реализовала драйвер JDBC. Утилита администратора Oracle Enterprise Manager представляет собой распределенное приложение Java. Администратор может управлять базой Oracle и следить за ее работой с любого браузера, связавшись с базой по URL.

Глава 2

Пример: система дистанционного обучения

В этой главе мы начнем разрабатывать большой пример распределенного приложения, использующего технологию Java. Такой пример должен удовлетворять четырем попарно исключающим друг друга краеугольным требованиям. Он должен быть актуальным, разъяснять насущную тему, чтобы заинтересовать читателя. При этом он не должен быть банальным, переходящим из книги в книгу, чтобы читатель нашел в примере что-то новое и полезное для себя. Его следует сделать большим и сложным, чтобы отразить все аспекты технологии. Но он не должен быть громоздким и трудным для понимания, потому что листинги длиной больше двух страниц никем не читаются и вообще отпугивают читателя. Самый распространенный пример распределенного приложения — это электронный, другими словами, online-магазин. Торговля по Internet, так или иначе, отражается в каждой книге по распределенным технологиям, например, [3, 6, 7, 12], примеры электронных магазинов есть на многих сайтах в Internet, более того, есть средства разработки электронных магазинов, в том числе и для Java: JShop Professional, Power WebShop Builder, Online Shop Creator и др. В фирменный учебник J2EE Tutorial (**http://java.sun.com/j2ee/doc/tutorial/**) включен пример банка "Duke's Bank". В рамках учебного пособия Blueprints (**http://java.sun.com/j2ee/blueprints/**) фирма Sun Microsystems разработала электронный зоомагазин Pet Shop, который следует считать образцом распределенного приложения, написанного на Java. Его можно получить по тому же адресу **http://java.sun.com/j2ee/blueprints/**. Итак, этот хороший и актуальный пример полностью разобран во многих изданиях, и мы электронную коммерцию отвергнем.

Выберем в качестве примера вторую по распространенности распределенную систему — систему дистанционного обучения (СДО). Обучение по Internet все шире распространяется по всему миру. Многие учебные заведения и даже корпоративные сайты предлагают различные электронные обучающие курсы online. Есть и специализированные обучающие сайты. Ведущие вузы России организуют свои университеты дистанционного обучения.

Для того чтобы не превращать книгу в инструкцию по созданию обучающих курсов и сборник длинных и скучных листингов, мы не будем детально выписывать все проверки исходных данных, обработку исключительных ситуаций, определять большое число типов данных, что заставляет то и дело выполнять приведение типов. Зададим только схему распределенного приложения, которую можно будет наполнять конкретным содержанием.

Для еще большего упрощения листингов нашей СДО будем считать, что ученик в каждый момент времени изучает только один курс и переходит последовательно от одной темы этого курса к другой. Каждый учитель ведет только один курс.

Подумаем, как реализовать нашу СДО. В этом нам поможет объектно-ориентированный анализ и проектирование.

Объектно-ориентированное проектирование

В последние 15—20 лет усилиями многих специалистов, из которых нужно выделить Грейди Буча (Grady Booch), Айвара Джекобсона (Ivar Jacobson), Джеймса Рамбо (James Rumbaugh), Бертрана Майера (Bertrand Meyer), Питера Коада (Peter Coad), Эдварда Йордана (Edward Yourdon) Мартина Фоулера (Martin Fowler), сложилось объектно-ориентированное направление разработки информационных систем [2—4, 8, 9, 14]. Это направление разбивает процесс разработки на *объектно-ориентированный анализ OOA* (Object-Oriented Analysis), *объектно-ориентированное проектирование OOD* (Object-Oriented Design) и *объектно-ориентированное программирование OOP* (Object-Oriented Programming). Этапы анализа, проектирования и программирования выполняются несколько раз. После первого прохода уже создается рабочий прототип системы, выполняющий простейшие функции. Следующие проходы развивают систему, добавляют новые функции, устраняют ошибки, сделанные ранее. Обычно после второго прохода система признается рабочей и передается для эксплуатации.

Достоинство объектно-ориентированного подхода к разработке информационных систем заключается в том, что полученная система легко сопровождается и модернизируется.

Возникло несколько школ, предлагающих свою методику объектно-ориентированной разработки. Эти школы создают и развивают языки, описывающие такие методики. Уже созданы компиляторы с этих языков в объектно-ориентированные языки программирования: C++, Java, Smalltalk, Eiffel, Object Pascal и др.

Наибольшее развитие получила объединенная методика, как говорят, *нотация* (notation), первых трех из перечисленных выше авторов: Г. Буча,

А. Джекобсона и Дж. Рамбо. Их часто называют "Три друга". Они разработали *унифицированный язык моделирования UML* (Unified Modeling Language) [3, 4, 9, 14], получивший широкое распространение. Разрабатываемый проект описывается на языке UML в виде диаграмм нескольких строго определенных типов и связей между диаграммами. На этапе программирования диаграммы компилируются в классы, а связи — в расширения классов и обращения к методам классов.

Для реализации своей нотации авторы организовали фирму Rational Software Corporation (**http://www.rational.com/**), выпускающую несколько программных продуктов. Основной продукт фирмы — визуальное средство разработки Rational Rose. Это средство позволяет графически построить диаграммы, провести связи между ними и задать свойства диаграмм и связей. Затем Rational Rose может откомпилировать проект в один из нескольких объектно-ориентированных языков программирования, в том числе и в язык Java.

Корпорация Rational Software предложила и методику разработки информационных систем с использованием UML, получившую название RUP (Rational Unified Process). С ней можно ознакомиться на сайте корпорации.

Мы не можем в этой книге заниматься объектно-ориентированным проектированием, но будем следовать методике RUP в разработке нашей системы дистанционного обучения. Читатель может познакомиться с OOD, обратившись к книгам [2—4, 8—9, 14].

Design Patterns (паттерны проектирования)

При воплощении в жизнь объектно-ориентированного проекта часто встает вопрос о выборе того или иного варианта его реализации. Разные разработчики выбирают разные варианты, возникают споры, дискуссии, в результате которых вырабатывается наилучшее решение. Это решение затем применяется в других проектах, обкатывается, обобщается. Такое обобщенное и обкатанное решение нетривиальной задачи образует то, что названо *design pattern*. Русский термин еще не устоялся, говорят о *паттернах проектирования* [5], *образцах проектирования* [4], *шаблонах проектирования* [9], *схемах разработки* [2].

На русский язык, наконец-то, переведена книга [5], широко известная под названием "GoF" ("Gang of Four" — так назвали четырех авторов книги). Авторы этой книги собрали, классифицировали и дали имена наиболее употребительным паттернам проектирования. Каждый разработчик может найти среди них решение большинства встающих перед ним проблем. В этой книге я буду ссылаться на паттерны из книги [5], называя их просто по именам.

Приведу еще несколько ссылок. Джеймс Купер [19] изложил паттерны проектирования из книги "GoF" на языке Java. Брюс Эккель [20] написал постоянно обновляемую книгу "Thinking in Patterns with Java", которую можно посмотреть на его сайте **http://www.bruceeckel.com/**. Вот еще несколько сайтов с разделами, посвященными паттернам:

- **http://st-www.cs.uiuc.edu/users/patterns/**;
- **http://www.patterndepot.com/**;
- **http://www.theserverside.com/patterns/**.

Наконец, паттерны, предложенные для разработок J2EE, собраны на сайте фирмы Sun по адресу **http://developer.java.sun.com/developer/technicalArticles/J2EE/patterns/**. Некоторые из этих паттернов изложены в учебном пособии Blueprints.

Пользователи СДО

Всех пользователей, которые будут работать с нашей системой дистанционного обучения (СДО), можно разделить на три группы: ученик, учитель и администратор СДО. Администратор не участвует в непосредственной работе СДО, он внешний наблюдатель, но мы должны создать ему рабочее место и поэтому будем держать его в поле зрения. Представители остальных групп, называемые в теории объектно-ориентированного анализа и проектирования *actors* (актеры, исполнители, деятели), выполняют определенные действия в СДО. Последовательность действий, совершаемая для получения определенного результата, называется *сценарием*. Набор всех сценариев, приводящих к одному и тому же результату, в теории OOD называется *use case*. На русский язык этот термин переводят по-разному: *прецедент* [4, 9], *вариант использования* [14]. Рассмотрим действия представителей каждой группы.

Действия ученика

Перед тем как стать учеником, пользователь должен зарегистрироваться. Ниже перечислены необходимые для этого действия.

Регистрация

1. Ученик заходит на сайт СДО и выбирает на начальной странице предложение зарегистрироваться или получает такое предложение после выбора учебного курса.
2. Ученик заполняет поля формы регистрации.
3. После отправки заполненной формы ученик получает подтверждение регистрации, свой регистрационный номер и предложение выбрать учебный курс.
4. Ученик переходит к действиям "учебный процесс" или завершает работу.

Учебный процесс

1. Ученик заходит на сайт СДО и просматривает список учебных курсов.
2. Он выбирает курс. На экране появляется окно входа, в поля которого ученик вводит имя и свой регистрационный номер. Ученик может вызвать окно входа перед выбором курса. В этом же окне есть предложение зарегистрироваться новому ученику.
3. На экране появляется первая тема выбранного курса, если ученик только начинает изучение данного курса, или следующая тема, если ученик уже изучил несколько тем.
4. Ученик работает с материалом темы.
5. После изучения темы ученику предлагается ответить на несколько вопросов, после чего на экран выводится оценка полученных знаний и рекомендация перейти к следующему разделу или повторить изучение данного раздела.
6. Ученик выбирает следующий раздел или повторяет данный раздел либо завершает сеанс обучения.

Оплата учебы

На первом этапе проектирования мы не будем определять эти действия.

Действия учителя

1. Учитель заходит на сайт СДО на страницу **teacher**.
2. Он вводит свое имя в поле появившегося на экране окна входа.
3. Появляется новое окно с названием учебного курса, проводимого учителем, и указанием возможных действий: создать новый курс, изменить существующий курс, удалить курс. Учитель выбирает одно из этих действий.
4. Если выбрано первое действие, то на экране появляется пустая форма для ввода материала нового курса. Если выбрано второе действие, то появляется содержимое изменяемого курса. Учитель вносит новый курс или изменяет существующий курс.
5. Учитель завершает работу.

Действия администратора

1. Администратор запускает графическую утилиту администратора.
2. Он вводит имя и пароль в поля появившегося окна. Открывается основное окно утилиты администратора.
3. Администратор запрашивает и получает отчеты о работе СДО.

4. Администратор просматривает базу учебных материалов, восстанавливая при необходимости испорченные данные из резервной копии.
5. Администратор следит за работой сервера СДО, при необходимости меняя его настройки.
6. Администратор при необходимости меняет забытые пароли и восстанавливает испорченные имена учеников и учителей.
7. Администратор завершает работу.

Графические интерфейсы

Для осуществления всех этих действий СДО должна предоставить каждому пользователю графический интерфейс. Его составляют одна или несколько страниц, последовательно появляющихся на экране. Каждая страница содержит управляющие элементы, помогающие выполнить определенные действия. У каждой группы пользователей должен быть свой набор страниц, отвечающий варианту использования (use case) этой группы. Кроме того, СДО должна иметь начальную страницу.

Начальная страница

Начальная страница нашей системы дистанционного обучения должна содержать общие сведения об СДО, список и краткое описание учебных курсов, порядок регистрации ученика, правила работы с обучающей системой.

Страницы ученика

Для выполнения всех своих действий ученику необходимо предоставить следующие страницы графического интерфейса.

Страница регистрации

Эта страница должна содержать поля фамилии, имени и отчества, адреса или адреса электронной почты (e-mail). Ограничимся пока этими сведениями, на следующих этапах можно будет внести дополнительные сведения.

Страница подтверждения регистрации

На этой странице повторяются введенные учеником сведения, сообщается его регистрационный номер и предлагается перейти к выбору учебной темы или сообщается об ошибках, сделанных учеником при регистрации.

Страница учебной темы

Данная страница выводит на экран ученика текущую тему выбранного учебного курса. Вид ее сильно зависит от изучаемого материала и не может быть здесь описан.

Страница вопросов

На этой странице появляются вопросы и поля для ввода ответов. Эту страницу можно оформить самыми разными способами, которые мы выберем позднее.

Страница оценки

Эта страница содержит оценку ответа ученика и рекомендацию для дальнейшего изучения. Страницу можно оформить как простой текст.

Страницы учителя

Для выполнения учителем своих обязанностей ему надо предоставить следующие страницы.

Начальная страница

Начальная страница учителя включает в себя название его курса с датой последнего обновления и элементы управления, позволяющие вызвать курс для редактирования или форму для ввода нового курса.

Страница нового курса

Эта страница содержит форму для ввода материала нового курса. Вид этой формы сильно зависит от содержания курса и не будет здесь описываться.

Страница изменения курса

На данную страницу выводится текущая тема курса и элементы управления, позволяющие выбрать ту или иную тему для редактирования.

Утилита администратора

Утилита администратора должна выглядеть как стандартное приложение Java, обеспечивающее выполнение всех перечисленных выше функций администратора. Читатель, разумеется, знаком с подобными утилитами и хорошо представляет, как они выглядят.

Функции и модули СДО

Анализ действий каждого из пользователей нашей системы дистанционного обучения выявляет следующие процессы, происходящие в СДО:

- регистрация нового ученика;
- предоставление ученику следующей темы для изучения;

- задание ученику контрольных вопросов;
- анализ ответов ученика и их оценка;
- внесение учителем нового учебного курса;
- изменение учителем существующего курса;
- предоставление отчетов администратору.

Каждый процесс будет реализован методами одного или нескольких классов, составляющих отдельный модуль распределенного приложения. В нашей системе дистанционного обучения выделяются такие модули.

- *Модуль регистрации ученика*, собирающий вводимые им сведения, проверяющий их и заносящий эти сведения в базу данных.
- *Модуль учебных курсов*, извлекающий из базы и предоставляющий ученику или учителю список курсов и их описаний, а также текущую тему.
- *Модуль заданий*, задающий вопросы ученику, сравнивающий ответы ученика с эталонными ответами и выводящий оценку.
- *Модуль генерации отчетов*, предоставляющий отчеты администратору.
- *Управляющий модуль*, согласовывающий работу остальных модулей.

Эти модули могут быть реализованы в разных слоях распределенного приложения: клиентском, серверном или в промежуточном слое.

Структура СДО

После выявления участников системы дистанционного обучения и процессов, происходящих в ней, переходим к определению объектов СДО и связей между ними.

Объекты СДО

Каждое существительное, участвующее в описании действий лиц, вовлеченных в систему дистанционного обучения, определяет какое-то понятие, более или менее важное для построения СДО. Некоторые понятия означают объекты системы, другие понятия обозначают процессы, происходящие в системе. Анализ этих существительных приводит к выделению таких понятий.

- Student — ученик, очевидный кандидат на роль объекта.
- Teacher — учитель и его учебный курс, объект.
- Administrator — администратор. Его нельзя считать объектом СДО, он не участвует в учебном процессе и не связан с другими объектами системы.
- StudentAccount — информация об ученике и его учебном процессе, его учетная карточка, объект.

- Registration — сбор информации о новом ученике, процесс.
- Course — учебный курс, выбранный учеником или изменяемый учителем, состоит из названия, краткого описания и нескольких тем, объект.
- Topic — тема, предложенная ученику или просматриваемая учителем. В простейшем случае это обычный текст или документ HTML или XML. В более сложном случае это гипертекст, состоящий из нескольких документов, снабженных рисунками и чертежами, звуковым сопровождением, видеофильмами. Мы пока ограничимся простейшим случаем, но создадим объект, который в дальнейшем можно будет расширить.
- Task — вопросы, предложенные ученику после изучения темы, объект.
- Answer — ответы ученика и их оценка, объект. Его удобно объединить с объектом Task.
- Mark — подсчет оценки знаний, процесс.
- Report — отчет, предоставляемый администратору, внешний процесс, не влияющий на состояние системы.

Из этих понятий выделяются объекты, их атрибуты и процессы. Объекты легко описать интерфейсами и классами Java, а процессы — методами интерфейсов и классов. Классы Java пока останутся абстрактными, поскольку реализация их конструкторов и методов зависит от того, в каком слое распределенного приложения располагаются объекты этих классов.

Связи между объектами

После выделения объектов рассмотрим связи между ними. Эти связи можно проследить, анализируя описание действий участников СДО. В описании действий связи между объектами задаются глаголами. Не все глаголы задают связи, и не все связи существенны для описания СДО, но начинать их исследование удобно с анализа глаголов в описании действий.

В теории OOD связи между объектами называются *отношениями* (relationship). Те отношения, которые описывают взаимодействие объектов, называются *ассоциациями* (association). Все отношения между объектами системы дистанционного обучения являются ассоциациями. При анализе информационной системы можно отследить большее или меньшее число ассоциаций в зависимости от детализации отношений между объектами. Мы отметим следующие ассоциации.

- Student — StudentAccount. По сведениям, представленным учеником при регистрации, создается учетная карточка.
- Student — Course. Ученик выбирает курс.
- StudentAccount — Course. Выбранный курс заносится в учетную карточку ученика.
- Course — Topic. Ученик выбирает тему учебного курса.

- Topic — Student. Ученику посылается учебная тема.
- Topic — StudentAccount. Это отмечается в учетной карточке ученика.
- Topic — Task. Выбираются вопросы по изучаемой теме.
- Task — Student. Ученику посылаются вопросы.
- Student — Task. Ученик дает ответы и получает оценку знаний.
- Task — StudentAccount. Ответы заносятся в учетную карточку ученика.
- Teacher — Course. Учитель вводит новый курс или изменяет свой курс.
- Teacher — Topic. Учитель получает учебную тему для редактирования.
- Teacher — Task. Учитель получает вопросы для редактирования.

На этапе программирования эти отношения реализуются методами классов, главным образом, `get`- и `set`-методами доступа, называемыми на жаргоне "геттерами" и "сеттерами". Например, последнее из перечисленных отношений можно реализовать `set`-методом

```
void Teacher.setTask(Task task);
```

Некоторые отношения реализуются внутри методов путем создания ассоциированных объектов и обращения к их методам. Не все отношения одинаково важны для правильного функционирования системы, некоторые из них не будут учтены при программировании, зато могут возникнуть другие отношения.

Атрибуты объектов

Анализ страниц, которые СДО предоставляет всем пользователям системы, позволяет определить атрибуты каждого объекта СДО. Лучший способ их описать — просто показать классы Java. Сделаем предварительный набросок каждого класса, не пытаясь детально проработать их структуру, тем более что классы могут в дальнейшем трансформироваться в сервлеты, JSP или компоненты EJB. В следующих частях книги мы будем определять методы этих классов, наполняя их функциональным содержанием, как говорят, *бизнес-логикой* (business logic).

Класс *Student*

Класс `Student` содержит сведения о состоянии дел ученика. Объект этого класса создается впервые после регистрации нового ученика, затем возникает при всяком входе ученика в СДО.

```
import java.io.*;


public class Student implements Serializable{
```

```
private int id; // Регистрационный номер ученика.
private Course course; // Изучаемый курс.
private Topic topic; // Изучаемая тема курса.
private Task task; // Вопросы, предложенные ученику.
private int mark; // Текущая оценка.

public Student(){}

public Student(int id, Course course, Topic topic,
               Task task, int mark){
   this.id = id; this.course = course; this.topic = topic;
   this.task = task; this.mark = mark;
}

public int getId(){ return id; }
public void setId(int id){ this.id = id; }

public Course getCourse(){ return course; }
public void setCourse(Course course){
   this.course = course;
}

private Topic getTopic(){ return topic; }
private void setTopic(Topic topic){
   this.topic = topic;
}

private Task getTask(){ return task; }
private void setTask(Task task){
   this.task = task;
}

private int getMark(){ return mark; }
private void setMark(int mark){
   this.mark = mark;
}

public void addMark(int newMark){
   mark += newMark;
}
```

```
   public boolean isNewcomer(){
      return id == 0;
   }

   public boolean isStudent(){
      return id > 0;
   }

   public String toString(){
      return "Student: " + id + "  " +
              course + "  " + task + "  " + mark;
   }
}
```

Класс *StudentAccount*

Класс `StudentAccount` содержит сведения о личности и процессе обучения ученика. Это учетная карточка ученика.

```
import java.util.*;
import java.io.*

public class StudentAccount implements Serializable{

   private Student stud; // Текущее состояние дел ученика.
   private String name;  // Имя ученика.
   private String address; // Адрес ученика.
   private Vector courses; // Список изученных курсов.
   private Vector marks; // Список полученных оценок.

   public StudentAccount(){}

   public StudentAccount(Student stud, String name,
                  String address, Vector courses,
                  Vector marks){
      this.stud = stud; this.name = name;
      this.address = address; this.courses = courses;
      this.marks = marks;
   }
```

```
   public Student getStudent(){ return stud; }
   public void setStudent(Student stud){ this.stud = stud; }

   public String getName(){ return name; }
   public void setName(String name){ this.name = name; }

   public String getAddress(){ return address; }
   public void setAddress(String address){
      this.address = address;
   }

   public Vector getCourses(){ return courses; }
   public void setCourses(Vector courses){
      this.courses = courses;
   }

   public Vector getMarks(){ return marks; }
   public void setMarks(Vector marks){ this.marks = marks; }

   public void addCourse(Course newCourse){
      courses.add(newCourse);
   }

   public void addMark(int newMark){
      marks.add(new Integer(newMark));
   }

   public String toString(){
      return "StudentAccount: " + stud + "  " +
              name + "  " + address;
   }
}
```

Класс *Teacher*

Класс `Teacher` содержит сведения об учителе и курсе, который он ведет. Мы не разделяем эти сведения по разным классам, поскольку они очень тесно связаны.

```
import java.util.*;
import java.io.*;
```

```
public class Teacher implements Serializable{

   private String name;   // Имя учителя.
   private Course course; // Курс, который ведет учитель.
   private Topic topic;   // Тема, изменяемая учителем.
   private Task task;     // Вопросы, задаваемые учителем.
   private Date date;     // Дата последнего изменения курса.

   public Teacher(){}

   public Teacher(String name, Course course,
                  Topic topic, Task task, Date date){
      this.name = name; this.course = course;
      this.topic = topic; this.task = task;
      this.date = date;
   }

   public String getName(){ return name; }
   public void setName(String name){ this.name = name; }

   public Course getCourse(){ return course; }
   public void setCourse(Course course){ this.course = course; }

   private Topic getTopic(){ return topic; }
   private void setTopic(Topic topic){
         this.topic = topic;
   }

   private Task getTask(){ return task; }
   private void setTask(Task task){
         this.task = task;
   }

   public Date getDate(){ return date; }
   public void setDate(Date date){ this.date = date; }

   public String toString(){
      return "Teacher: " + name + "  " + course + "  " +
```

```
                topic + "  " + task + "  " + date;
    }
}
```

Класс *Course*

Класс `Course` содержит название учебного курса, его краткое описание и список тем, входящих в курс.

```
import java.util.*;
import java.io.*;

public class Course implements Serializable{

   private String name; // Название курса.
   private String description; // Описание курса.
   private Vector topic; // Темы курса.

   public Course(){}

   public Course(String name, String descr, Vector topic){
      this.name = name; description = descr;
      this.topic = topic;
   }

   public String getName(){ return name; }
   public void setName(String name){ this.name = name; }

   public String getDescr(){ return description; }
   public void setDescr(String descr){ description = descr; }

   public Vector getTopics(){ return topic; }
   public void setTopics(Vector topic){ this.topic = topic; }

   public String toString(){
      return "Course: " + name;
   }
}
```

Класс *Topic*

Класс `Topic` пока содержит только один текстовый документ для каждой темы курса.

```
import java.io.*;

public class Topic implements Serializable{

   private Course course;  // Курс, к которому относится тема.
   private String content; // Учебный материал — пока только строка.

   public Topic(){}

   public Topic(Course course, String content){
      this.course = course; this.content = content;
   }

   public Course getCourse(){ return course; }
   public void setCourse(Course course){
      this.course = course;
   }

   public String getContent(){ return content; }
   public void setContent(String content){
      this.content = content;
   }

   public String toString(){
      return "Topic: " + course + "  " + content;
   }
}
```

Класс *Task*

Класс `Task` содержит список вопросов по изучаемой теме и варианты ответов на них. Он содержит также эталонные ответы на вопросы, сравнивает их с ответами ученика и выводит оценку. Для простоты считаем, что ответами будут просто целые числа, например, номер правильного, по мнению ученика, варианта. Если ученик не дал ответа, то заносится ноль. За каждый правильный ответ дается один балл.

```
import java.io.*;

public class Task implements Serializable{

   private Topic topic; // Ссылка на учебную тему.
   private String[] question; // Набор вопросов.
   String[][] answer; // Набор ответов на каждый вопрос.
   int[] master; // Номера правильных ответов.

   public Task(){}

   public Task(Topic topic, String[] question,
               String[][] answer, int[] master){
      this.topic = topic; this.question = question;
      this.answer = answer; this.master = master;
   }

   public Topic getTopic(){ return topic; }
   public void setTopic(Topic topic){ this.topic = topic; }

   public String[] getQuestion(){ return question; }
   public void setQuestion(String[] question){
     this.question = question;
   }

   public String[][] getAnswer(){ return answer; }
   public void setAnswer(String[][] answer){
      this.answer = answer;
   }

   public int[] getMaster(){ return master; }
   public void setMaster(int[] master){ this.master = master; }

   public int makeMark(int[] answer){
      int mark = 0;
      for (int k=0; k < master.length; k++)
          if (master[k] == answer[k]) mark++;
      return mark;
   }
```

```
   public String toString(){
      return "Task: " + topic;
   }
}
```

База данных SDO

После того как определены атрибуты всех объектов, следует обдумать, какие объекты и их атрибуты будут храниться в базе данных. После этого надо определить структуру базы и распределить хранимые объекты и атрибуты по таблицам базы данных. Назовем нашу базу данных SDO.

Нам надо сохранять от сеанса к сеансу следующие сведения об ученике: регистрационный номер, имя, адрес, список уже изученных им курсов и оценки знаний, полученные за них, изучаемый в настоящее время курс и тему курса, текущую оценку. Сведения об учителе содержат его имя и курс, который он ведет, а также дату последнего изменения курса. Кроме того, в базе SDO должен храниться учебный материал, контрольные вопросы и эталонные ответы.

После некоторых размышлений и приведений к одной из многочисленных нормальных форм занесем всю информацию в шесть таблиц базы данных SDO. Кроме того, определим последовательность `reg_seq`, выдающую регистрационные номера от 1 до 99999 с шагом 1. Она хранит последний выданный регистрационный номер и следующий номер в полях `CURRVAL` и `NEXTVAL`.

```
CREATE SEQUENCE reg_seq MAXVALUE 99999
```

Таблица `students` содержит сведения об учениках, изучаемые ими в данный момент курсы и темы, текущую оценку знаний по изучаемому курсу.

```
CREATE TABLE students(
   id            NUMBER(5) PRIMARY KEY,
   name          VARCHAR2(64) NOT NULL,
   address       VARCHAR2(128),
   course_name   VARCHAR2(64),
   topic_N       NUMBER(2),
   mark          NUMBER(2)
)
```

Таблица `studied_courses` содержит для каждого ученика список изученных им курсов и полученных оценок.

```
CREATE TABLE studied_courses(
   id            NUMBER(5) REFERENCES students(id),
```

```
   course_name   VARCHAR REFERENCES courses(course_name),
   mark          NUMBER(2)
)
```

Таблица `courses` содержит имя и краткое описание каждого курса.

```
CREATE TABLE courses(
   course_name        VARCHAR2(100) PRIMARY KEY,
   description        VARCHAR2(1000)
)
```

Таблица `topics` для каждого курса содержит номер темы и учебный материал темы.

```
CREATE TABLE topics(
   course_name    VARCHAR2(100) REFERENCES courses(course_name),
   topic_N        NUMBER(2) NOT NULL,
   topic_content  CLOB,
)
```

Таблица `tasks` для каждой темы содержит контрольные вопросы и не более пяти вариантов ответов, а также номер верного ответа.

```
CREATE TABLE tasks(
   course_name    VARCHAR2(100) REFERENCES courses(course_name),
   topic_N        NUMBER(2) NOT NULL,
   question       VARCHAR2(500),
   answer1        VARCHAR2(300),
   answer2        VARCHAR2(300),
   answer3        VARCHAR2(300),
   answer4        VARCHAR2(300),
   answer5        VARCHAR2(300),
   valid_answer   NUMBER(1)
)
```

Таблица `teachers` содержит имена учителей, названия курсов, которые они ведут, и дату последнего изменения курса.

```
CREATE TABLE teacher(
   name            VARCHAR2(100),
   course_name     VARCHAR2(100) REFERENCES courses(course_name),
   mod_date        DATE
)
```

Заключение

Итак, мы прошли стадии анализа и проектирования нашей системы дистанционного обучения. Подведем итоги проделанной работы.

Мы определили страницы интерфейса ученика и учителя, содержание утилиты администратора СДО. В *части II* книги мы рассмотрим основное средство Java для создания графического интерфейса пользователя — библиотеку графических компонентов Swing, — и реализуем с ее помощью все выявленные здесь интерфейсы.

Нам удалось проследить процессы, происходящие в СДО, и выделить программные модули, которые будут управлять этими процессами. В дальнейшем мы разместим эти модули по слоям нашего многослойного распределенного приложения.

Мы описали объекты СДО и их атрибуты в виде классов Java. Пока это только предварительный набросок. В дальнейшем эти классы будут преобразованы в апплеты, сервлеты, JSP, компоненты EJB и будут наполнены функциональным содержанием.

Отмечены те атрибуты, которые должны постоянно храниться в базе данных, названной нами SDO, и описаны таблицы базы данных SDO.

В *части III* книги мы построим двухслойное приложение: клиентский апплет или приложение Java будет непосредственно связываться с базой данных посредством JDBC. Там же проанализируем все достоинства и недостатки такого двухслойного приложения.

В *части IV* выделим промежуточный слой. Реализуем его с помощью сервлетов и JSP, перенеся в них всю обработку запросов клиента. Наша СДО будет разработана как трехслойное распределенное приложение.

В *части V* книги мы воспользуемся компонентами EJB для упрощения создания промежуточного слоя и разделим таким образом промежуточный слой на Web-слой и EJB-слой. Получим окончательное четырехслойное распределенное приложение, полностью построенное по технологии Java.

Часть II

Клиентская часть распределенного приложения Java

Глава 3

Оформление клиентского приложения компонентами Swing

Стандартная поставка Java 2 SDK, Standard Edition (J2SDK 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.
- *Swing* — новая библиотека "легких" (lightweight) графических компонентов, дополняющая и во многом заменяющая библиотеку AWT. Занимает шестнадцать пакетов с префиксом javax.swing.
- *Java 2D* — часть библиотеки AWT, обеспечивающая воспроизведение графики, выбор цвета, вывод изображений и фигурного текста, а также преобразование их перед выводом.
- *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 включены в 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 2 Runtime Environment (J2RE), в частности, в браузере Netscape Communicator 6.*x*. Для браузеров, в которые не встроен пакет J2RE, фирма SUN Microsystems выпускает загружаемый модуль Java Plug-in, автоматически загружающийся с сайта **http://java.sun.com/products/plugin/** при запуске апплета, использующего классы библиотеки Swing. Модуль Java

Plug-in входит в состав J2RE и автоматически подгружается в браузер, работающий там, где установлен пакет J2RE. Нужно лишь, чтобы браузер распознавал теги <OBJECT> или <EMBED>, в котором указан класс апплета.

В число демонстрационных программ J2SE SDK входит апплет SwingSet2, расположенный в каталоге $JAVA_HOME/demo/jfc/SwingSet2/. Он показывает большинство возможностей Swing. Тут же можно посмотреть исходные тексты соответствующих классов апплета. 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. С их помощью очень легко создать стандартное окно приложения. В листинге 3.1 показан шаблон для приложения, использующего библиотеку Swing.

Листинг 3.1. Шаблон приложения, использующего Swing

```
import java.awt.*;           // Базовые классы AWT.
import java.awt.event.*;     // Интерфейсы и классы основных событий.
import javax.swing.*;        // Основные классы Swing.
import javax.swing.event.*; // Интерфейсы и классы событий Swing.

public class SwingApplicationTemplate extends JFrame
                                      implements XxxListener{

   public SwingApplicationTemplate(String title){

         // Создаем основное окно.
      super(title);

         // Получаем контейнер верхнего уровня.
      Container c = getContentPane();
```

```
        // Задаем менеджер размещения
        // компонентов. По умолчанию BorderLayout.
    c.setLayout(new XxxLayout());

        // Помещаем компонент в контейнер.
    c.add(xxxx);

        // Присоединяем обработчик событий.
    xxxx.addXxxListener(this);

        // Прочие установки...

        // Задаем начальную ширину и высоту окна.
    setSize(width, height);

        // Завершаем работу приложения при закрытии окна.
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Выводим окно на экран.
    setVisible(true);
  }
  public void someEventHandler(XxxEvent e){
    // Обработка события...
  }
  public static void main(String[] args){
    new SwingApplicationTemplate("Заголовок основного окна");
  }
}
```

Как может выглядеть шаблон апплета, использующего библиотеку Swing, показано в листинге 3.2.

Листинг 3.2. 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.
}
```

Для обработки событий, специфичных для Swing, кроме пакета java.awt.event, содержащего средства обработки стандартных событий AWT, понадобится пакет javax.swing.event.

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, четыре пакета javax.swing.plaf.* задают внешний вид и поведение приложения (*Look and Feel*, L&F) в различных графических средах. Можно сделать внешний вид и поведение приложения независимым от графической оболочки операционной системы. Тогда оно в любой графической оболочке будет выглядеть одинаково и одинаково реагировать на действия мыши и клавиатуры. Есть возможность 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) по желанию пользователя. Эта возможность реализуется 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 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 фирмы Sun Microsystems есть подробнейшее руководство "Java Look and Feel Design Guidelines" по созданию графического интерфейса пользователя в стиле Java L&F, хранящееся по адресу **http://java.sun.com/products/jlf2/book/**.

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, с которым захочется работать. 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", расположенном 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. Начнем с вершины иерархии компонентов — класса `JComponent`.

Класс *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 возможности.

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, передающий им 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)`.

Неотъемлемой частью каждого компонента является 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 можно 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)`, а также собственную локаль [15] 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 созданы по схеме MVC. Рассмотрим подробнее эту схему.

Схема MVC в компонентах Swing

Конструктивная схема *Модель—Вид—Контроллер* MVC (Model—View—Controller) предполагает, что компонент логически состоит из трех частей.

Первую часть "Модель" (Model) составляет один или несколько классов, в которых хранится вся информация, обрабатываемая и формируемая компонентом, и текущее состояние объектов, созданных этим компонентом. Эти классы обладают методами `setXxx()` ввода и изменения информации.

Вторая часть — один или несколько классов, составляющих "Вид" (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 `getXxx()` и `isXxx()` *Модели*.

Третья часть — классы, образующие "Контроллер" (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 `setXxx()` *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 `setXxx()` *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 `fireXxx()`, после чего *Виды* забирают измененную информацию, обращаясь к методам `getXxx()` и `isXxx()` *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 интерфейсов, некоторые интерфейсы реализованы несколькими классами. Эти интерфейсы и классы, реализующие их, приведены в табл. 3.1.

Таблица 3.1. *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
	JTreeEmptySelectionModel

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 `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 обращается 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 по горизонтали.

В конструкторах с двумя элементами `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 установленной 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 применять 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 метод 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`: 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)` или задать всплывающую подсказку методом `setToolTipText(String)`. Есть возможность даже задать реакцию на внешние воздействия, но для этого лучше применять кнопки.

Кнопки

Библиотека Swing предлагает целую иерархию кнопок, показанную на рис. 3.1. В нее включены и пункты меню `JMenuItem`, и флажки `JCheckBox`, и переключатели `JRadioButton`.

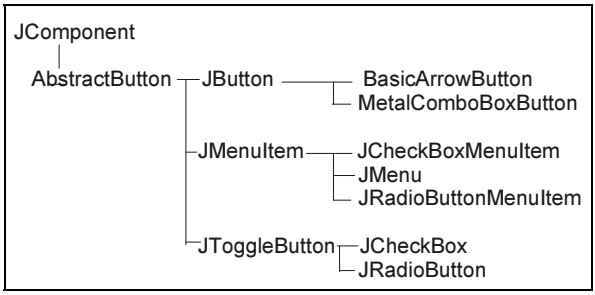


Рис. 3.1. Иерархия классов кнопок

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 изображение, если оно было на кнопке, методом `setIcon(Icon)`.

Как и в компоненте класса `JLabel`, можно сменить изображение на кнопке, временно недоступной, т. е. на кнопке, к которой применен метод `setEnabled(false)`, методом `setDisabledIcon(Icon)`. 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 можно 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 все эти 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 на ней находится курсор мыши. Контроллер отмечает это состояние методом `setRollover(boolean)` модели, а вид определяет его методом `isRollover()`.
- В состояние "наготове" (armed) кнопка переходит при нажатии над 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 содержать текст и/или изображение, так же, как и `JLabel`. 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, 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.

Флажок *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, соответствующее стилю 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 можно методом `setSelected(boolean)`.

Экземпляры класса `JToggleButton` создаются восьмью 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 3.3 дан простейший пример флажка с двумя состояниями.

Листинг 3.3. Простейший флажок с двумя состояниями

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

class DummyToggleButton extends JFrame implements ChangeListener{

   private JToggleButton tb;

   public DummyToggleButton(){
      tb = new JToggleButton("<html><u>Д</u>а?<p>Нет?");
      tb.setMnemonic(KeyEvent.VK_L);
      tb.setToolTipText("Сделайте выбор");
      tb.addChangeListener(this);

      getContentPane().add(tb);

      setSize(300,300);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        setVisible(true);
    }

    public void stateChanged(ChangeEvent ae){
        if (tb.isSelected()){
            tb.setForeground(Color.white);
            tb.setText("Да");
        }else{
            tb.setForeground(Color.black);
            tb.setText("Нет");
        }
    }

    public static void main(String[] args){
        new DummyToggleButton();
    }
}
```

В этом примере можно было бы использовать событие классов `ItemEvent` или `ActionEvent`.

Класс `JToggleButton` применяется обычно как суперкласс для создания кнопок выбора нестандартного вида. Для получения стандартных кнопок выбора используются его подклассы `JCheckBox` и `JRadioButton`.

Флажок *JCheckBox*

Компонент `JCheckBox` — это стандартный флажок выбора с меткой или изображением слева или справа от надписи, в котором показывается состояние `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`.

Переключатель *JRadioButton*

Стандартный переключатель 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`.

Переключатели имеет смысл использовать только в составе группы, обеспечивая выбор одного из нескольких значений. Для получения группы сначала создается пустая группа переключателей как экземпляр класса `ButtonGroup`, затем она заполняется переключателями методом `add(AbstractButton)`.

Группа переключателей никак не выделяется на экране, это только логическое объединение элементов. Чтобы выделить переключатели, входящие в группу, их обычно размещают на отдельной панели, окружая эту панель рамкой. В листинге 3.4 показано обычное размещение группы переключателей, которые в листинге устанавливают цвет фона окна приложения. На рис. 3.2 показан вид этой группы переключателей.

Листинг 3.4. Группа переключателей

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

class RadioButtonTest extends JFrame
                 implements ActionListener{
   Container c;

   public RadioButtonTest(){
      c = getContentPane();
      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 void actionPerformed(ActionEvent ae){
      String selected = ae.getActionCommand();

      if (selected.equals("red"))
         c.setBackground(Color.red);
      else if (selected.equals("green"))
         c.setBackground(Color.green);
      else
         c.setBackground(Color.blue);
   }

   public static void main(String[] args){
      new RadioButtonTest();
   }
}
```

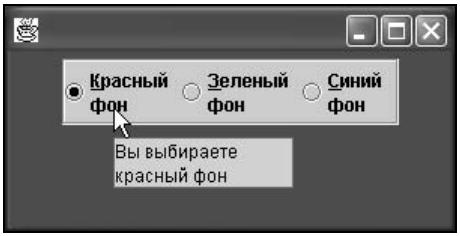


Рис. 3.2. Группа переключателей и всплывающая подсказка

Группу переключателей 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`.

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 после описания класса `JList`.

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 пусто. Это удобно в случае редактируемого списка, в окно которого можно вводить новый элемент или редактировать выбранный элемент. 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 списке 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. Реализации методов этих интерфейсов только выбрасывают исключения, например:

```
public void contentsChanged(ListDataEvent e){
    throw new RuntimeException("Obsolete method. " +
            "ItemEvent is fired with setSelectedItem()");
}
```

Есть еще несколько нестыковок. В классе `JComboBox` элемент называется `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 после описания класса `JList`.

Список выбора *JList*

Вместо группы переключателей можно создать список класса `JList`. В таком списке можно выбрать 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 заданным 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 линейкой 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 произвольное число любых элементов, держа клавишу <Ctrl> нажатой. После применения метода

```
setSelectionMode(ListSelectionModel.SINGLE_SELECTION)
```

в списке можно будет выбрать только один элемент. Третья возможность — выбрать один диапазон подряд идущих элементов — достигается использованием в качестве параметра этого метода константы SINGLE_INTERVAL_SELECTION. Указанные три возможности выбора элементов списка — результат реализации интерфейса ListSelectionModel классом DefaultListSelectionModel. Эта реализация модели выбора применяется в классе JList по умолчанию. Если она почему-либо не устраивает разработчика, то можно реализовать интерфейс ListSelectionModel с помощью своего класса 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, используя идею 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 очень большому расходу оператив-

ной 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 берется в тонкую рамку). 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 выше фрагмента и применив унаследованные от класса `JLabel` методы `setText(((JLabel)value).getText())` и `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 этом классе следует переопределить, кроме методов `getXxx()/setXxx()`, 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 классы `JLabel` или `JPanel`.

Счетчик *JSpinner*

Перебирать элементы списка часто бывает удобнее с помощью счетчика — небольшого редактируемого окошка 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 окно можно ввести любое значение в указанном диапазоне, 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 значения 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 эти методы возвращают `null`.

При всяком изменении текущего значения, а также окончании ввода в окно 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` задает 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 этого класса `SpinnerListModel(Object[])` 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 включить любые объекты, в окне будет показана только строка, полученная методом `toString()` текущего объекта. Это про-

исходит потому, что редактор по умолчанию, заложенный в класс `Jspinner`, — это экземпляр класса `JFormattedTextField`. Редактор определяется вложенным в `JSpinner` классом `DefaultEditor` и его подклассами `DateEditor`, `ListEditor` и `NumberEditor`.

Линейка прокрутки *JScrollBar*

Линейки прокрутки используются многими компонентами и контейнерами Swing. Во многих случаях достаточно поместить компонент на панель типа `JScrollPane`, как это сделано в листинге 3.3, чтобы обеспечить прокрутку содержимого компонента.

Линейки прокрутки определяются четырьмя целочисленными свойствами, хранящимися в модели данных. Это наименьшее значение линейки `minimum`, наибольшее значение `maximum`, текущее значение `value` и шаг изменения `extent`. От последнего числа зависит размер видимой области компонента, связанного с линейкой прокрутки и длина ползунка в линейке прокрутки. Действия с этими числами описаны в интерфейсе `BoundedRangeModel`. При всяком изменении этих свойств происходит событие `ChangeEvent`. Класс `JScrollBar` использует в качестве модели данных реализацию `DefaultBoundedRangeModel` этого интерфейса.

Линейки 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 линейки 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)`.

Линейка 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 линейка прокрутки.

Компонент `JSlider` тоже основан на модели данных класса `DefaultBoundedRangeModel`, используя наименьшее, наибольшее и текущее значения, а также шаг 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 остальных 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 устанавливается методом `setPaintTicks(true)`. К штрихам можно добавить числовые значения методом `setPaintLabels(true)`. Между штрихами можно разместить более мелкие штрихи методом `setMinorTickSpacing(int)`. Если применить метод `setSnapToTicks(true)`, то указатель ползунка будет останавливаться только напротив штрихов.

Основную линейку ползунка можно убрать методом `setPaintTrack(false)`, оставив только шкалу.

Числовые значения на шкале ставятся напротив каждого штриха. Методом `createStandardLabels(int incr, int start)` можно изменить это правило, задав другой шаг для расстановки чисел `incr` на шкале, другую начальную точку отсчета `start` и установив шаг `incr` на шкале методом `setLabelTable(Dictionary)`. Все это удобно делать одновременно, 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. 3.3.

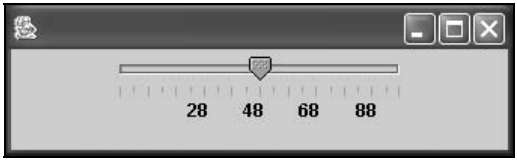


Рис. 3.3. Ползунок `JSlider`

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. 3.3 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 указателя в ползунке происходит событие `ChangeEvent`. В процессе обработки этого события можно получить местоположение указателя методом `getValue()`.

Индикатор *JProgressBar*

Индикатор, часто называемый "градусником", показывает степень выполнения какого-то процесса, чаще всего в процентах. Процесс должен характеризоваться каким-нибудь целым числом. В конструкторе индикатора

```
JProgressBar(int orientation, int min, int max)
```

задается наименьшее (`min`) и наибольшее (`max`) значение этого числа. В других конструкторах опущены некоторые из этих 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 на 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 время окончания 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 индикатора в стиле 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`, описанный в *главе 6*.

Дерево объектов *JTree*

Дерево `JTree` располагает объекты в иерархической структуре. Она создается только на экране, но не в оперативной памяти. На уровне 0 находится один *корневой* (root) объект, на уровнях 1, 2 находятся его потомки (child) — *узловые* (node) объекты, имеющие своих потомков и одного предка (parent). На самом нижнем уровне расположены *листья* (leaves). 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()` — число 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 так далее…
```

Для простоты каждый узел в этом дереве содержит строку класса `String`. Полученное дерево показано на рис. 3.4.

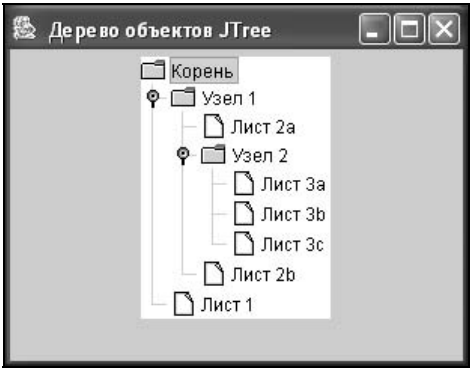


Рис. 3.4. Дерево класса `JTree`

Возле каждого узла выводится содержимое его объекта, преобразованное методом `toString()`.

Второй способ — создать 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 необозначенными, на экран выводится содержимое узлов в виде строки.

Второй недостаток можно устранить использованием конструктора `JTree(Hashtable)`. 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" (параметры метода `put()`), хранящейся в хэш-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 хэш-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 хранится в модели выбора, описанной интерфейсом `TreeSelectionModel` и реализованной классом `DefaultTreeSelectionModel`. Этот класс хранит массив типа `TreePath[]` пу-

тей 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. Событие этого класса позволяет получить полезные данные о дереве, в частности, метод `getTreePath()` возвращает путь к предку тех узлов, которые были изменены, добавлены или удалены, в виде объекта класса `TreePath`.

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`.

Этот интерфейс реализован классом `DefaultTreeCellRenderer`, добавляющим массу свойств и методов и расширяющим класс `JLabel`. Наличие свойств класса `JLabel` позволяет вывести на экран различные иконки для узлов и листьев, открытых и закрытых ветвей дерева.

В состав J2SE SDK входит пример построения дерева шрифтов, расположенный в каталоге $JAVA_HOME/demo/jfc/SampleTree/. В нем показано, как можно изменить стандартную визуализацию дерева.

Построение меню

Все компоненты, составляющие меню: 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 строке 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` или `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 "кнопке". Как видно из рис. 3.1, меню `JMenu` отно-

сится 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 то, 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`, флажки класса `JCheckBoxMenuItem` и переключатели 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 с 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()` эти методы 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 примера из предыдущего раздела:

```
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 меню, выборе пунктов меню и закрытии меню.

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:

```
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.

Меню *JCheckBoxMenuItem*

Вставить в меню флажки 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 флажка. 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("");
   }
});
```

Меню *JRadioButtonMenuItem*

Для размещения в меню группы переключателей удобно воспользоваться классом JRadioButtonMenuItem. Он не добавляет функциональности своему

суперклассу `JMenuItem` и используется точно так же, как в обычной группе переключателей.

Добавим группу переключателей 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 графической подсистеме MS Windows и стиле 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 компоненту, над которым находится курсор мыши. Таким образом, контекстное меню не привязывается к строке меню или другому меню, а связано с одним или несколькими компонентами. Оно добавляется к компонентам при обработке событий, связанных с мышью 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 возникновении всякого события, связанного с мышью. Когда метод `isPopupTrigger()` покажет, что надо вызывать всплывающее меню, вернув `true`, следует обратиться к методу `show(Component, int, int)`, в котором 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, связанные с мышью, т. е. к нему присоединен методом `addMouseListener()` хотя бы один слушатель [1, 11, 15, 18] или в нем есть обращение к методу

```
enableEvents(AWTEvent.MOUSE_EVENT_MASK).
```

Действия с 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 экране 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` означает, 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()` возвращает 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 позволяет указать для выбора цвета не только экземпляр класса `JColorChooser`, но и любое расширение 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 не нужно создавать отдельное диалоговое окно, а требуется только панель выбора цветов, то можно воспользоваться конструктором `JColorChooser(Color)`, в котором задается начальный цвет, или конструктором `JColorChooser()`, устанавливающим вначале белый цвет.
- В этом случае определять момент окончания выбора цвета придется самостоятельно. Класс `JColorChooser` отслеживает только событие `PropertyChangeEvent` изменения свойств JavaBean, которое можно обработать, присоединив обработчик унаследованным от класса `JComponent` методом

  `addPropertyChangeListener(PropertyChangeListener)`.

- Еще один 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 в классе `DefaultColorSelectionModel`. Экземпляр этого класса можно получить методом `getSelectionModel()`. Данная модель отслеживает событие `ChangeEvent`.

  При необходимости изменения модели данных можно сделать другую реализацию интерфейса `ColorSelectionModel` и установить ее при создании объекта конструктором `JColorChooser(ColorSelectionModel)`.

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 показывает скрытые (hidden) файлы. Чтобы задать их отображение, надо обратиться к методу `setFileHidingEnabled(false)`.

По умолчанию в окне можно выбрать 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` (не перепутайте с интерфейсом `FileFilter` из пакета java.io) и установить полученный фильтр в окно выбора файла методом `addChoosableFileFilter(FileFilter)`. Этот метод можно применить несколько раз с разными параметрами, установив несколько фильтров в одно окно.

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 показана в поле **Files of Type** (Тип файлов) окна выбора.

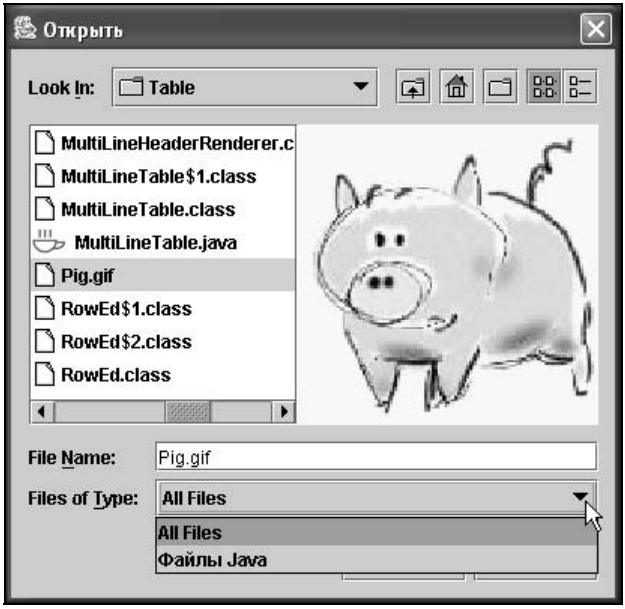


Рис. 3.5. Окно открытия файла

Вот пример фильтра, отбирающего только файлы с расширением java. Вид окна открытия файла с этим фильтром показан на рис. 3.5.

```
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";
    }
}
```

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 "Open" или "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 приведенного примера, методы `showXxxDialog()` возвращают целое число — одну из трех констант, соответствующих выбору файла и щелчку по кнопке **Open** или **Save** — константа `APPROVE_OPTION`, щелчку по

кнопке **Cancel** — константа `CANCEL_OPTION` или появлению ошибки — константа `ERROR_OPTION`.

После того как пользователь выбрал 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 программе, вызвавшей окно выбора.

Дополнительный компонент

Возможности окна выбора файлов можно расширить, добавив в него произвольный компонент методом `setAccessory(JComponent)`. Чаще всего добавляется небольшое окно предварительного просмотра выбранного 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. 3.5.

Замена изображений

В окне выбора файла перед каталогами и файлами выводятся стандартные иконки. 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 дымящуюся чашечку 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 стандартная иконка перед исходными файлами Java с расширением java будет заменена изображением из файла javacup.gif, как показано на рис. 3.5.

Более содержательный пример создания окна выбора файлов есть в составе J2SE SDK 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, например, в книге [15]. Остается лишь русифицировать стандартные надписи, что видно на рис. 3.6.

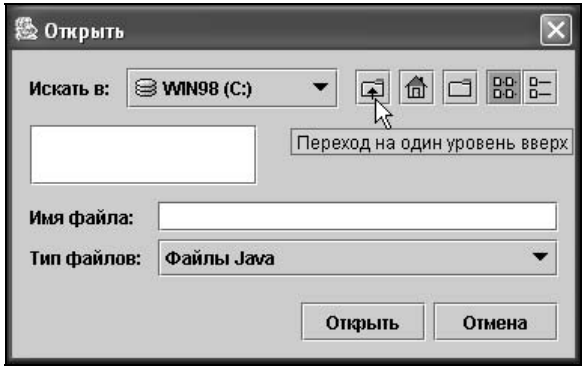


Рис. 3.6. Русифицированное окно открытия файла

Для этого надо написать несколько файлов ресурсов (properties), которые, если они есть, просматриваются классами-рисовальщиками компонентов перед выводом на экран. Подробнее об этом написано в *главе 9*, а пока надо сказать лишь, что эти файлы уже написаны Сергеем Астаховым и лежат на его домашней странице в файле **http://people.comita.spb.ru/users/sergeya/java/swing_ru.jar**. 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. 3.6.

Сергей Астахов держит на своей странице постоянно обновляемый файл ruschars.html, в котором собрано все, касающееся русификации Java. По всем вопросам, возникающим при работе с кириллицей, безусловно, нужно обращаться к этому ресурсу.

Глава 4

Создание форм с помощью текстовых компонентов

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 нередактируемым. В редактируемом окне часто приходится задавать смену 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 компоненте, как последовательность символов Unicode. Количество символов в тексте можно узнать методом `getLength()`. Последовательность символов нумеруется, начиная с 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: текущий этот символ 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 ин-

декса 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, надо сказать, 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 туда записать. Это 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 названия 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 этого интерфейса — класс SimpleAttributeSet. С помощью этого класса можно определить любые пары "имя—значение", но общепринятые атрибуты удобнее задавать с помощью констант и статических методов класса StyleConstants и четырех вложенных в него подклассов: 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 слушатель [1, 11, 15, 18] 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);
```

Этот метод 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 изменения, фильтруя, таким образом, 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 переопределения таким образом методов и расширения класса `DocumentFilter` полученный фильтр надо установить в документ методом

```
void setDocumentFilter(DocumentFilter filter);
```

класса `AbstractDocument`. После этого всякое обращение к методам `insertString()` и `remove()` документа будет пропускаться через методы созданного фильтра.

Пример применения фильтра, пропускающего только цифры, приведен в листинге 4.1, *см. разд. "Поле ввода JTextField" данной главы.*

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 можно 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 можно задать 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, можно 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 описывает 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 элемента в документ и удаление элемента из него.

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, состоящую из отдельных элементов — строк. Корневой элемент этой 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 двухмерное 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 двухмерную фигуру класса `Shape`. Параметры метода задают начальную `startPos` и конечную `endPos` позиции текста, направления `b0` и `b1` от этих позиций к преобразуемому 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.

Класс `View` напоминает класс `Component`. 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. 4.1.

Как видно из рис. 4.1, 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`.

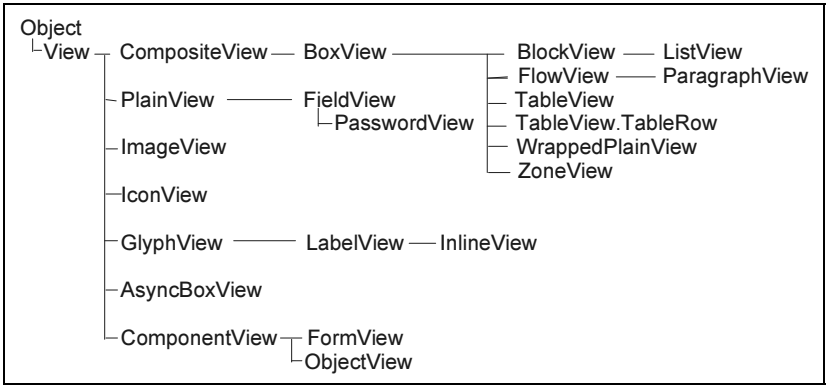


Рис. 4.1. Иерархия классов-видов

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 виде. Определить двухмерную 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 задается 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, связанные с мышью 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 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 переопределения методов и расширения, таким образом, класса `NavigationFilter`, 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, описывающих 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 др. 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` обслуживают целых 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 (Document Type Definition) документа и таблицы стилей CSS (Cascading Style Sheets), разработать свой интерпретатор (parser) документа HTML.

Как видите, схема MVC получила в компоненте `JTextComponent` развитое детальное воплощение. Разработчик при желании может совершенно изменить стандартное воплощение 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 ниже.

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`, — это установка раскладки клавиатуры для ввода текста.

Для обеспечения такой возможности при нажатии или отпускании клавиши создается объект класса `KeyStroke`. Этот объект содержит код клавиши и состояние клавиш-модификаторов <Shift>, <Alt>, <Ctrl> и <Meta>.

Класс `KeyStroke` предоставляет фабричные [1, 11, 15, 18] методы для создания своих объектов одним из следующих статических методов.

- `KeyStroke getKeyStroke(char keyChar)` — в нем задается символ `keyChar`, отвечающий нажатию клавиши.

- `KeyStroke getKeyStroke(int keyCode, int modifiers)` — здесь 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)` — здесь 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)` — здесь 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 6*.

Такой механизм задания реакции на действия с клавиатурой заменяет обычную обработку событий клавиатуры, заключающуюся в добавлении 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[]` получается 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 будет найдена в данном объекте, то ее поиск продолжится 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)`.

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)` — смысл аргументов `text` и `columns` такой же, как и у предыдущих конструкторов;
- `JTextField(Document doc, String text, int columns)` — задается модель данных `doc`.

Модель данных можно заменить методом `setDocument(Document)`. Можно заменить 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()`, или 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, для создания поля только для чтения следует воспользоваться унаследованным методом `setEditable(false)`.

В поле можно установить новый курсор методом `setCaret(Caret)` или просто изменить цвет курсора методом `setCaretColor(Color)`. Позицию курсора можно отследить методом `getCaretPosition()`, а задать программно — методом `setCaretPosition(int)`. Перемещать курсор программно, выделив, таким образом, участок текста, следует 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()`.

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 4.1 приведен пример текстового поля для ввода только цифр.

Листинг 4.1. Поле ввода цифр

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.text.*;

public class NumberText extends JFrame implements ActionListener{
   JTextField tf = new JTextField(5);
   JLabel l = new JLabel("Вводите цифры:");

   NumberText(){
      super("text");
      Container c = getContentPane();
      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);
      // Присоединяем обработчик событий.
      tf.addActionListener(this);
      l.setLabelFor(tf);

      c.add(l);
      c.add(tf);

      setSize(400, 400);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
      setVisible(true);
   }

   public void actionPerformed(ActionEvent e){
      l.setText(tf.getText());
   }

   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`, следовательно, к нему относится все сказанное выше. 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 — 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, например, для ввода дат (`Date`) и чисел (`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 позволяет использовать редактор объектов и для проверки и фильтрации текста, допуская ввод в поле только определенных символов.

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` используется 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 для ввода действительных чисел с двумя знаками после десятичной точки.

Второй класс `InternationalFormatter` прямо использует для своих преобразований класс `Format`, 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 `getXxxFormatter()`, 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 надо 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. Этот режим 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)`.

В области ввода допускается 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 редакторы `DefaultEditorKit`, `HTMLEditorKit` или `RTFEditorKit` для редактирования. Для того чтобы учесть MIME-тип текста, применяется конструктор

```
JEditorPane(String type, String text);
```

Этот конструктор вызывает метод `setContentType(type)`, задающий MIME-тип текста, а затем вызывает 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 текущий документ, обрабатываемый редактором!

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 в редактор начальный текст, хранящийся по адресу, заданному в форме URL. Адрес задается объектом класса `URL` или строкой:

```
JEditorPane(URL url);
JEditorPane(String url);
```

Информацию, хранящуюся по адресу URL, можно занести в редактор в любое время методом `setPage(URL)` или `setPage(String url)`. Учтите, что при изменении MIME-типа текста изменится тип документа и редактора. Если занесена страница HTML, то будет установлена 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)`.

Существует возможность задать множество атрибутов для отдельных символов, выделенных в тексте, или вставляемых в текст, методом `setCharacterAttributes(AttributeSet attr, boolean replace)`. Можно задать атрибуты сразу целому элементу, который выделен, или тому, 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.

Глава 5

Помещение результата запроса в таблицу

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 тоже в виде таблицы, снабженной заголовком, комментариями, оформленной цветом и различными шрифтами. В графической библиотеке Swing для этого есть 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-представителей. В своей работе методы класса `JTable` используют 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 имеется 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`.

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)` создает пустую редактируемую таблицу с заданным числом строк (`rows`) и столбцов (`columns`) с моделью данных, принятой по умолчанию, которая предполагает хранить в таблице объекты типа `Object` в виде вектора типа `Vector`, состоящего из элементов типа `Vector`. В нее можно вводить данные прямо с клавиатуры.
- Конструктор `JTable(Object[][] data, Object[] colNames)` создает таблицу, заполненную объектами `data`. Второй параметр `colNames` содержит имена столбцов таблицы. Все строки массива `data` должны содержать столько же элементов, сколько содержится в массиве `colNames`. Пример создания таблицы этим конструктором приведен в листинге 5.1.
- Конструктор `JTable(Vector data, Vector colNames)` делает то же самое, но параметры заданы векторами. Пример его использования приведен в листинге 5.3, *см. разд. "Модель ячеек таблицы" данной главы.*

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, 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 5.1 показывает простой пример определения таблицы с помощью массива объектов, в данном случае, строк.

Листинг 5.1. Простое определение таблицы

```
import java.awt.*;
import javax.swing.*;

public class SimpTable extends JFrame{
```

```
    SimpTable(){
      super("My Table");
      Container c = getContentPane();
      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(JFrame.EXIT_ON_CLOSE);
      setVisible(true);
    }

    public static void main(String[] args){
      new SimpTable();
    }
}
```

Таблицу можно заполнить методом `setValueAt(Object data, int row, int column)`. Этот метод 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 методом `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 описана в *главе 3*.

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 с нуля. Общий суперкласс всех ячеек столбца с индексом `ind` можно получить методом `getColumnClass(int ind)`. Текущее число строк в таблице можно определить 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 извлечь из модели данных в виде объекта класса `Object` методом `getValueAt(int rowInd, int colInd)`, а занести в модель, если ячейка редактируемая, — методом `setValueAt(Object data, int rowInd, int colInd)`.

Проверить, разрешено ли редактировать ячейку, 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 5.2 приведен пример расширения класса `AbstractTableModel` — модель ячеек таблицы, запрещающая редактировать первый столбец таблицы.

Листинг 5.2. 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();
     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. 5.1, объект класса `Boolean` показан в таблице компонентом `JCheckBox`. Это результат действия метода `getColumnClass()`. На рис. 5.2 тот же объект показан в стандартном виде. Но объект класса `Color` по-прежнему показан строкой — результатом действия метода `toString()`.

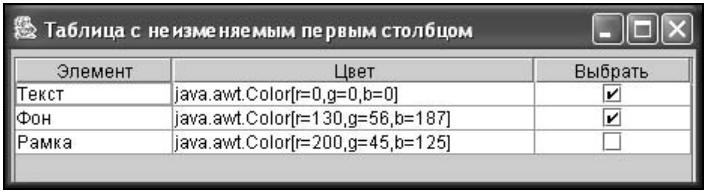


Рис. 5.1. Таблица со сложными объектами

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)` и `DefaultTableModel(Object[][] data, Object[] colNames)` сразу же заполняют модель данными.

Еще два конструктора `DefaultTableModel(Vector colNames, int rowCount)` и `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 `getXxx()` 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 5.3 показан пример такого расширения — модель заполняется сведениями о файлах, полученными из каталога, имя которого указывается в командной строке.

Листинг 5.3. 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(JFrame.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 этого события может быть присоединен к модели методом `addTableModelListener(TableModelEvent)`.

Сочетание типа `UPDATE` с константой `ALL_COLUMNS` и константой `MAX_VALUE`, полученной в качестве значения метода `getLastRow()` означает, 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 5.2.

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 `getXxx()/setXxx()`.

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, хранится ширина промежутков между колонками и общая ширина таблицы. Модель хранит объект класса `ListSelectionModel`, определяющий правила выделения столбцов и хранящий 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, как определено в модели ячеек таблицы класса `DefaultTableModel`.

Строка заголовков — это объект класса `JTableHeader` из пакета javax.swing.table. Этот 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 5.4, 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.

Листинг 5.4. Отдельный вывод заголовка таблицы

```
import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;

public class HeadTable extends JFrame{

   HeadTable(){
     super("My Table");
     Container c = getContentPane();

     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(JFrame.EXIT_ON_CLOSE);
      setVisible(true);
    }

    public static void main(String[] args){
      new HeadTable();
    }
}
```

На рис. 5.2 показан вывод программы, записанной в листинге 5.4. Как видно из рисунка, объект класса `Boolean` показан строкой — результатом действия метода `toString()`.

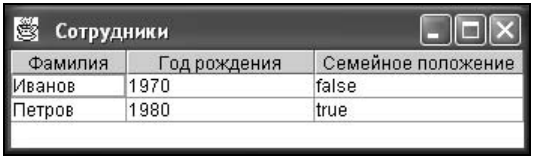


Рис. 5.2. Отдельный вывод заголовка таблицы

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 3* 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 получить 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 представитель 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 3*.

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 5.2 объекты класса `Color` выводятся текстовой строкой (см. рис. 5.1). Чтобы в какой-либо 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`, надо обратиться к методу `setDefaultRenderer(Class, TableCellRenderer)` класса `JTable`. В листинге 5.2 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 5.5 показывает, как можно реализовать его для вывода многострочных ячеек, воспользовавшись текстовым редактором `JTextArea`, описанным в *главе 4*.

Листинг 5.5. 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 5.5, 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-представитель. 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 метода. 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 события методом `addCellEditorListener(CellEditorListener)`.

Интерфейс `TableCellEditor` реализован классом `DefaultCellEditor`. Конструкторы этого класса используют 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, но этот редактор обрабатывает только текст, получающийся применением метода `toString()` такого объекта.

Компонент `JCheckBox` используется для показа в виде кнопки выбора логического содержимого ячейки типа `boolean` (см. рис. 5.1). Этот редактор позволяет изменять истинность содержимого ячейки.

Компонент `JComboBox` применяется для ввода в ячейку одного из нескольких значений, содержащихся в раскрывающемся списке.

В листинге 5.6 показан пример программы, позволяющей устанавливать разные редакторы для редактирования разных ячеек таблицы. В примере ячейка в третьей строке и во втором столбце редактируется компонентом `JComboBox`. Результат показан на рис. 5.3.

Листинг 5.6. 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;
   }
}
```

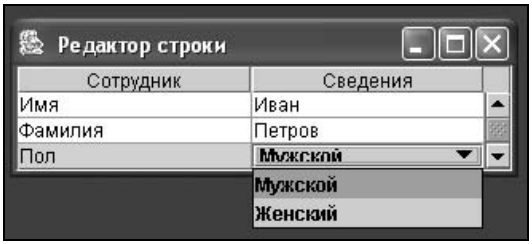


Рис. 5.3. Компонент `JComboBox` в ячейке таблицы

Примеры создания таблиц и связи их с базами данных имеются в документации J2SE SDK в каталоге $JAVA_HOME/demo/jfc/TableExample/.

Глава 6

Контейнеры

В процессе создания графического интерфейса пользователя каждый компонент по правилам языка должен быть помещен в контейнер. Контейнеры тоже являются компонентами и они, в свою очередь, могут располагаться в контейнере. Так образуется иерархия вложенных контейнеров. Контейнер верхнего уровня, содержащий все остальные контейнеры, должен взаимодействовать с оконным менеджером графической оболочки операционной системы компьютера, на котором работает приложение. В библиотеке Swing четыре таких контейнера. Кроме класса `JFrame`, которым мы все время пользуемся, это окно `JWindow`, диалоговое окно `JDialog` и класс апплета `JApplet`. Они по необходимости выполнены как "тяжелые" (heavyweight) компоненты в отличие от остальных "легких" (lightweight) компонентов Swing.

Поскольку класс `JComponent` — прямой наследник класса `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. Эта глава дает обзор таких контейнеров.

Панель *JPanel*

Класс `JPanel` реализует простую невидимую панель для размещения компонентов.

Конструктор по умолчанию `JPanel()` применяет к панели менеджер размещения компонентов `FlowLayout`. Напомним, что этот менеджер размещает компоненты рядами, укладывая их слева направо и сверху вниз как кирпичи в порядке обращения к методам `add()`, устанавливая при этом такой размер каждого компонента, какой 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 для него 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` использует двойную буферизацию (double buffering) для перерисовки своего содержимого. Описание метода двойной буферизации есть во многих книгах, например, [15]. 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 нем компонентов методами, унаследованными от класса `Container`.

- `add(Component comp)` — добавить компонент `comp` в конец списка компонентов, лежащих на панели.
- `add(Component comp, Object constraints)` — добавить компонент `comp` в конец списка компонентов, лежащих на панели и передать менеджеру размещения параметры `constraints`, суть которых зависит от типа менеджера.
- `add(Component comp, int ind)` — вставить компонент `comp` в указанную позицию `ind` списка компонентов панели.
- `add(Component comp, Object constraints, int ind)` — содержит все 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 на панель прокрутки сразу же при ее создании конструктором `JScrollPane(Component)` или позднее методом `setViewportView(Component)`. Линейки 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 в классе `JScrollPane`.

Панель прокрутки имеет сложное строение. Кроме своего содержимого и двух линеек 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)`, который используется в листинге 6.1. Структурные элементы панели показаны на рис. 6.1, созданном программой листинга 6.1. 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, можно применить другой менеджер размещения методом `setLayout(LayoutManager)`, но новый менеджер может быть только расширением менеджера размещения `ScrollPaneLayout`.

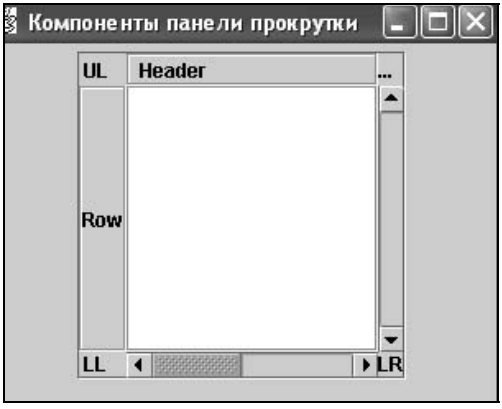


Рис. 6.1. Компоненты панели прокрутки

Листинг 6.1. Компоненты панели прокрутки

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ScrollComps extends JFrame{

   ScrollComps(){
     super(" Компоненты панели прокрутки");
     c = getContentPane();
     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();
    }
}
```

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 контейнер `JViewport` может содержать в себе только один компонент, размещением 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);
```

Двойная панель *JSplitPane*

На панели класса `JSplitPane` располагаются два компонента, разделенные тонкой полосой, которую можно перемещать с помощью мыши, меняя, таким образом, взаимные размеры компонентов. Компоненты могут располагаться один за другим, по горизонтали, что определяется константой `HORIZONTAL_SPLIT` класса `JSplitPane`, или один под другим, 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 сдвига или после окончательной установки разделительной черты. Это определяется вторым параметром конструктора `JSplitPane(int orientation, boolean continuous)`.

В этих 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. 6.2.

Четвертый конструктор `JSplitPane(int orientation, Component left, Component right)`, кроме расположения `orientation`, сразу задает компо-

нент, который будет располагаться слева (сверху) `left`, и тот, что будет располагаться справа (снизу) `right`.

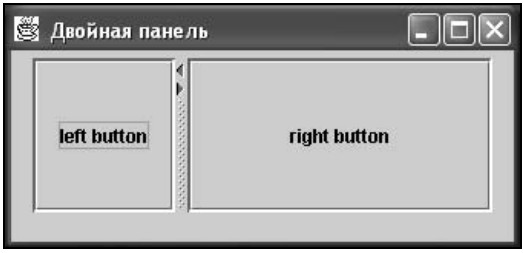


Рис. 6.2. Двойная панель

Очень часто компоненты, размещаемые на 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 определяется 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 долях от 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 устанавливается методом `setDividerSize(int)`, параметр которого задается в пикселах. По умолчанию в Java L&F толщина равна 8 пикселам.

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)` (см. рис. 6.2). При щелчке кнопкой мыши на одной из этих стрелок один компонент занимает всю панель, а другой исчезает полностью.

Компоненты устанавливаются на панель или заменяются другими компонентами методами `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.

Пример использования панели класса `JSplitPane` приведен в *главе 10*, в листинге 10.2.

Панель со вкладками *JTabbedPane*

Класс `JTabbedPane` создает сразу несколько панелей. На экране видна только одна из них, остальные панели показаны как *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 `TOP`, `BOTTOM`, `LEFT`, `RIGHT` класса `JTabbedPane`. 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 в несколько рядов или прокручиваться, для чего в строке вкладок появляются кнопки прокрутки, как показано на рис. 6.3. Способ расположения вкладок в один ряд обозначается константой `WRAP_TAB_LAYOUT`, в несколько рядов — константой `SCROLL_TAB_LAYOUT` класса `JTabbedPane`. Второй параметр третьего конструктора `JTabbedPane (int pos, int tab)` задает один из этих двух способов.

Как видите, конструкторы класса не создают панели и не помещают на них компоненты. Эти действия выполняются 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)` — пишет на вкладке строку, хранящуюся в первом параметре.
- `void addTab(String, Component)` — пишет на вкладке строку, содержащуюся 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 `setXxx()` позволяют установить отдельные элементы панелей и вкладок. Кроме того, можно установить цвет фона методом `setBackgroundAt(Color)` как показано на рис. 6.3 и в листинге 6.2. Это удобно для того, чтобы пометить вкладки разными цветами.

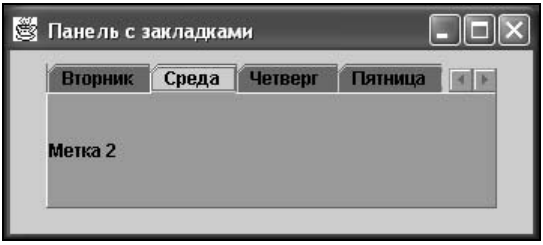


Рис. 6.3. Панель со вкладками

Листинг 6.2. Панель с разноцветными вкладками

```
import java.awt.*;
import javax.swing.*;

public class Tabbed extends JFrame{

   Tabbed(){
     super(" Панель со вкладками");
     Container c = getContentPane();
     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(JFrame.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 ли 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)`.

Размещением 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 других 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 пытается выровнять ширину компонентов по самому широкому компоненту. Если это 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 благодаря тому, 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);
```

В листинге 6.3 приведен пример размещения текстовой области и двух кнопок на панели класса `Box`. Для того чтобы сдвинуть кнопки вправо, применена "подушка".

Листинг 6.3. 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();
      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(JFrame.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 то, что компоненты можно обвести рамкой нужной 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 размещения компонентов, но при одновременном использовании большого числа компонентов приходится вкладывать экземпляры класса `Box` друг в друга. К тому же, частое применение компонентов-разделителей приводит к неоправданной сложности компоновки.

Вообще говоря, диапазон возможностей размещения компонентов в контейнерах графических библиотек AWT и Swing очень велик.

На одном конце этого диапазона абсолютное размещение, при котором прямо указываются координаты (`x, y`), ширина `width` и высота `height` компонента в координатной системе контейнера. Такое размещение 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);
```

но такой способ сложен 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 к размеру контейнера.

В состав J2SE SDK, 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, максимальный и предпочтительный размеры. Эти размеры могут использоваться как размеры промежутков между компо-

нентами. 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`.

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 значением. 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 значения 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 меняет положение левого верхнего угла компонента в заданных объектами `x` и `y` пределах. Поэтому координаты 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 — `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 решили расположить несколько компонентов `comp[0]`, `comp[1]`, `comp[2]` в одну строку с фиксированными промежутками между ними величиной в 6 пикселов. Кроме того, хотели бы оставить отступы от границ контейнера размером в 10 пикселов. Листинг 6.4 содержит программу, выполняющую такое размещение, а на рис. 6.4 показан результат размещения.

Листинг 6.4. 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();
```

```
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();
    }
}
```

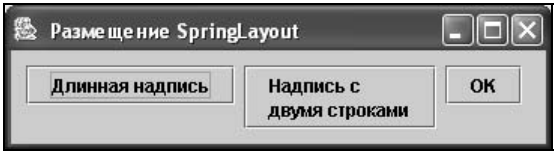


Рис. 6.4. Размещение `SpringLayout`

Панель инструментальных кнопок *JToolBar*

Класс `JToolBar` создает панели инструментальных кнопок. Обычно такая панель занимает строку, расположенную 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 J2SE SDK рекомендует управлять контейнером, в который помещают панель инструментов, с помощью менеджера размещения `BorderLayout` и ничего не располагать в граничных областях этого контейнера. В этом случае панель чаще всего помещают на "Север". Слева (сверху) панели есть полоса с "насечкой". Она видна на рис. 6.5. Наведя курсор мыши на эту полосу, панель можно перемещать по контейнеру. При переносе панели на "Запад", "Восток" или на "Юг", она занимает эту область, располагая свои компоненты по вертикали или по горизонтали. При перемещении панели в "Центр" или за пределы окна контейнера, она 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.

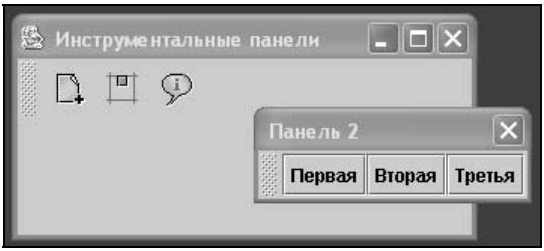


Рис. 6.5. Инструментальные панели

Панель инструментов можно сделать неперемещаемой методом `setFloatable(false)`. Полоска с "насечкой" исчезает, панель нельзя двигать по экрану.

Обычно инструментальные кнопки на панели обведены тонкой рамкой как на панели **Панель 2** (рис. 6.5), но после применения метода `setRollover(true)` рамка будет появляться только тогда, когда курсор мыши окажется над кнопкой.

Для размещения компонентов класс `JToolBar` использует свой внутренний менеджер размещения `DefaultToolBarLayout`, основанный на менеджере `BoxLayout`, следовательно, панель инструментов может использовать свойства этого менеджера размещения. В листинге 6.5 дается пример создания панели инструментов.

Листинг 6.5. Панели инструментальных кнопок

```
import java.awt.*;
import javax.swing.*;

public class MyTool extends JFrame{

   MyTool(){
     super(" Инструментальные панели");
     Container c = getContentPane();

     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(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

  public static void main(String[] args){
    new MyTool();
  }
}
```

В составе J2SE SDK, в каталоге $JAVA_HOME/demo/jfc/Notebook/ есть пример текстового редактора с панелью инструментальных кнопок класса `JToolBar`. Его можно запустить, перейдя в этот каталог и набрав в командной строке

```
java -jar Notebook.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` предназначен 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 объектов, определяемых в интерфейсе, — 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.
- `LONG_DESCRIPTION` — описание действия для всплывающей справки.
- `MNEMONIC_KEY` — код командной клавиши типа `int`.
- `NAME` — имя действия, записываемое в пункт меню.
- `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 следующим образом:

```
ImageIcon ic = (ImageIcon)act.getValue(Action.SMALL_ICON);
```

Доступность объекта, реализующего интерфейс `Action` (enabled), 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 входит `JMenu`, `JPopupMenu` и `JToolBar`, умеют применять 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 дискетки или из всплывающего контекстного меню. В любом случае сохранение файла выполняет один объект типа `Action`, присоединенный к этим графическим элементам их методом `add(Action)`.

Примеры такого применения интерфейса `Action` приведены в *главе 10* (см. листинг 10.2).

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, перекрывают визуально компоненты, лежащие в нижних слоях. Слои на 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`: его второй параметр определяет 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 других слоев. Этот 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)`. В него 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 отпускания кнопки мыши и завершения перетаскивания 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)` и `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, а компонент с номером (−1) лежит под всеми другими компонентами этого слоя.

Компоненты можно перемещать в своем слое следующими 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`.

Для того чтобы поместить компонент в другой слой, надо сначала указать ему новый слой одним из следующих методов.

- `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()`. Она — контейнер класса `Container`, в котором обычно размещаются большинство компонентов и строка меню. Управляет их размещением 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));
```

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 располагаться необязательная строка меню класса `JMenuBar`.

Итак, корневая панель `JRootPane` не содержит панель содержимого непосредственно. В нее включен 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, генерируемых мышью.

Дело в том, что обычно действия мыши обрабатываются компонентом, над которым расположен курсор мыши, точнее, обработчиком событий `MouseEvent`, присоединенным к этому компоненту. Такой обработчик можно присоединить к прозрачной панели и обрабатывать события, генерируемые мышью, 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 9*.

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.

Окно `JWindow` может быть связано с уже существующим, "родительским" (parent) окном. Для этого его надо создать конструкторами `JWindow(Frame)`,

`JWindow(Window)` или `JWindow(Window, GraphicsConfiguration)`. Окно располагается поверх родительского окна. Ему можно передать фокус ввода и, например, ввести текст в поле ввода, находящееся в окне, а затем передать введенный текст родительскому окну.

Если окно создано конструкторами `JWindow()` или `JWindow(GraphicsConfiguration)`, то оно не связано с родительским окном. Ему нельзя передать фокус ввода, следовательно, в нем нельзя ничего сделать. Такое окно может скрываться под другими окнами.

Окно `JWindow` непосредственно содержит всего один компонент — корневую панель класса `JRootPane`. На эту корневую панель и кладутся все компоненты, как описано в предыдущем разделе. 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: 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)`. Этот метод защищен (protected), он предназначен для использования в расширениях 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 и является "тяжелым" компонентом. Он создает модальные (modal) или немодальные диало-

говые окна. Из модального окна нельзя удалить фокус ввода, не выполнив в нем все требуемые действия.

Каждое диалоговое окно обязательно связано с родительским окном класса `Dialog` или `Frame`. Даже конструктор по умолчанию `JDialog()` создает скрытое родительское окно.

Конструкторы `JDialog(Frame)`, `JDialog(Frame, String)`, `JDialog(Dialog, String)`, `JDialog(Dialog)` создают немодальные диалоговые окна с заголовком или без него.

Конструкторы `JDialog(Frame, boolean)`, `JDialog(Frame, String, boolean)`, `JDialog(Dialog, String, boolean)`, `JDialog(Dialog, boolean)` формируют модальные диалоговые окна, если последний параметр равен `true`, с заголовком или без него.

Модальность окна и его заголовок можно изменить унаследованными методами `setModal(boolean)` и `setTitle(String)`.

Диалоговое окно, как и окно `JWindow`, непосредственно содержит только один компонент — корневую панель `JRootPane` и точно так же предоставляет непосредственный доступ к панелям корневой панели методами `getContentPane()`, `getLayeredPane()`, `getGlassPane()` и к самой корневой панели методом `getRootPane()`.

Все компоненты следует помещать на панели, расположенные на корневой панели. Размещение компонентов непосредственно в диалоговом окне выполняется по тем же правилам, 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, заданная в конструкторе. В строке заголовка есть кнопка **Закрыть**, реакцию на которую, а заодно и способ реагирования на нажатие комбинации клавиш <Alt>+<F4>, можно установить методом

```
setDefaultCloseOperation(int);
```

Поведение окна при закрытии 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 унаследованный метод `setResizable(boolean)` позволяет отказаться от этой возможности.

По умолчанию диалоговое окно появляется с рамкой и со строкой заголовка, оформленными по правилам графической оболочки операционной системы. Это оформление можно убрать методом `setUndecorated(true)`, но вместе с этим будет потеряна и стандартная реакция на действия с элементами оформления, инициируемые мышью. После применения этого метода можно установить оформление текущего стиля Look and Feel методом `setWindowDecorationStyle(int)` класса `JRootPane`. Подробно о стиле Look and Feel написано в *главе 9*.

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 ниже.

В листинге 10.2 *главы 10* приведен пример модального диалогового окна входа в утилиту администратора SDO.

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 него. Чтобы вывести такое окно 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, посвященных `Jwindow` и `JDialog`, за исключением модальности и родительского окна.

Реакция на щелчок кнопкой мыши по кнопке закрытия окна тоже определяется методом `setDefaultCloseOperation(int)`, но параметр может принимать еще одно, четвертое значение: `EXIT_ON_CLOSE` — завершить работу приложения методом `System.exit(0)`. Это значение не следует применять в апплетах.

Напомним, 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 `JFrame`, выполненное по правилам оконного менеджера графической оболочки операционной системы, методом `setUndecorated(true)`, установив затем оформление текущего стиля Look and Feel методом

```
getRootPane().setWindowDecorationStyle(JRootPane.FRAME);
```

Подробнее об этом написано в *главе 9*.

Внутреннее окно *JInternalFrame*

Класс `JInternalFrame` создает окно, очень похожее на окно класса `JFrame`, но существующее только внутри другого окна, обычно 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 стиль 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 пиктограмму и разворачиваться на все объемлющее окно, закрываться. По умолчанию все эти четыре свойства отсутствуют. У внутреннего окна, созданного конструктором по умолчанию `JInternalFrame()`, нет кнопок: **Свернуть**, **Развернуть**, **Закрыть**. При установке курсора мыши на границу окна он не меняет свой вид и не позволяет изменять размеры окна, заголовок также 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 `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. 6.6 показано два внутренних окна, созданных разными конструкторами. Первое окно сдвинуто влево вниз, оно частично обрезано границами внешнего окна. В листинге 6.6 приведена программа, создающая эти окна.

Листинг 6.6. Внутренние окна

```
import java.awt.*;
import javax.swing.*;
```

```
public class IntFrame extends JFrame{

   IntFrame(){
      super(" Окно с внутренними окнами");
      c = getContentPane();
      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(JFrame.EXIT_ON_CLOSE);
      setVisible(true);
   }

   public static void main(String[] args){
     new IntFrame();
   }
}
```

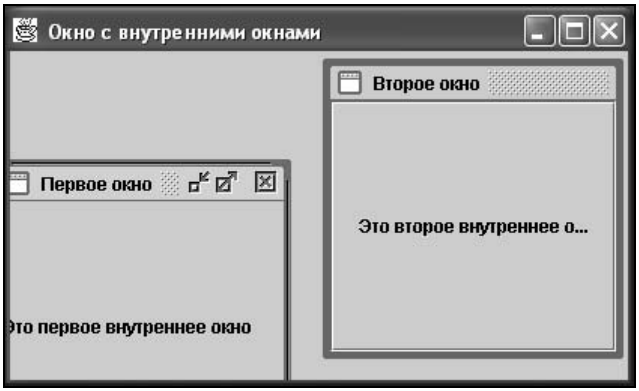


Рис. 6.6. Внутренние окна

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 значение параметра, равное `EXIT_ON_CLOSE`.

Рабочий стол *JDesktopPane*

Внутренние окна можно поместить в любой контейнер, но удобнее всего расположить их на 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 он не управляется никаким менеджером размещения. Поэтому позицию и размеры компонента следует задавать методом `setBounds()`, как написано в листинге 6.7. На рис. 6.7 показан результат работы программы листинга 6.7.

Листинг 6.7. 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();
  }
}
```

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);
```

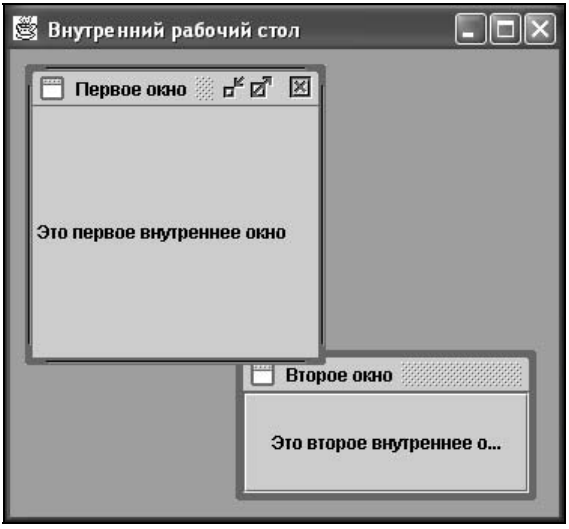


Рис. 6.7. Внутренний рабочий стол

Стандартные диалоговые окна *JOptionPane*

Класс `JDialog` позволяет создавать любые диалоговые окна, но в подавляющем большинстве случаев их назначение сводится к выводу сообщений (message), подтверждению или отмене каких-то действий (confirm/cancel) и вводу коротких сведений (input).

Класс `JOptionPane` предоставляет более двадцати статических методов для создания модальных диалоговых окон. Это три метода `showMessageDialog()` с разными аргументами для окон сообщений, четыре метода `showConfirmDialog()` для окон подтверждения, шесть методов `showInputDialog()` для окон ввода, и один метод для диалогового окна общего вида `showOptionDialog()`. Эти методы создают диалоговые окна класса `JDialog`. Еще более десяти методов вида `showInternalXxxDialog()` формируют внутренние диалоговые окна класса `JInternalFrame`.

Существует нескольких типов окон сообщений, снабжаемых определенными ярлыками и обозначаемых следующими константами.

- `INFORMATION_MESSAGE` — просто сообщение, для стиля Java L&F это синий круг с восклицательным знаком внутри (см. рис. 6.15) (по умолчанию).
- `WARNING_MESSAGE` — предупреждение, желтый треугольник с восклицательным знаком внутри (см. рис. 6.10).

- `ERROR_MESSAGE` — сообщение об ошибке, красный шестиугольник с "кирпичом" внутри (см. рис. 6.8).
- `QUESTION_MESSAGE` — вопрос, зеленый квадрат с вопросительным знаком внутри (см. рис. 6.9).
- `PLAIN_MESSAGE` — какое-то сообщение без определенного ярлыка.

Сообщение любого типа можно снабдить любым изображением типа `Icon` (см. рис. 6.11). Это делается самым развитым в этой группе методов статическим методом

```
static void showMessageDialog(Component parent, Object message,
    String title, int type, Icon icon);
```

Для работы этого метода необходимо задать родительское окно `parent`, строку заголовка `title`, сообщение `message`, его тип `type` — одна из констант, перечисленных выше, и ярлык `icon`.

Как видите, сообщение `message` описывается как объект самого общего типа `Object`. Сообщением может быть компонент класса `Component` или 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. 6.8 показано окно сообщения, созданное методом

```
JOptionPane.showMessageDialog(this,
   " В поле \"Код\" могут быть только цифры.",
   " Ошибка", JOptionPane.ERROR_MESSAGE);
```

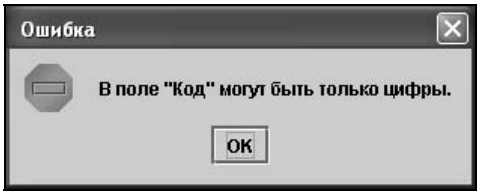


Рис. 6.8. Окно сообщения типа "Ошибка"

Окно подтверждения или отмены действий содержит две или три кнопки: **Да**, **Нет**, **Отмена** (Yes, No, Cancel). Количество кнопок задается константами `YES_NO_OPTION` или `YES_NO_CANCEL_OPTION` соответственно 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. 6.9 показано окно подтверждения, используемое, например, так:

```
int n = JOptionPane.showConfirmDialog(this,
    "Сохранить этот или предыдущий вариант?",
    " 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:
}
```

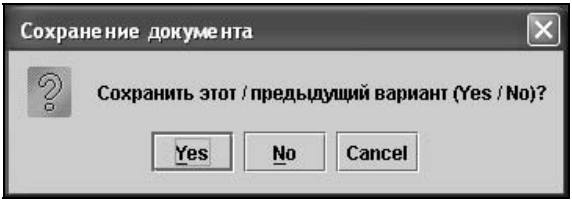


Рис. 6.9. Окно подтверждения типа "Вопрос"

Диалоговое окно ввода содержит поле для ввода краткого ответа, возвращаемого методом, или список выбора, а также кнопки **OK** и **Cancel**. На рис. 6.10 показано окно, созданное методом

```
String s = (String)JOptionPane.showInputDialog(this,
   " Запишите ответ: ", " Ответ", JOptionPane.WARNING_MESSAGE);
```

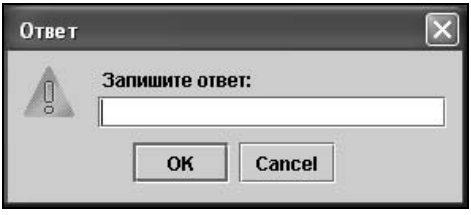


Рис. 6.10. Окно ввода типа "Предупреждение"

На рис. 6.11 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. 6.11. 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. 6.9, 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. 6.12.

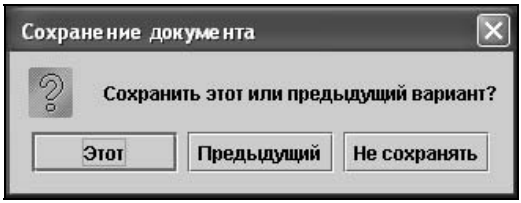


Рис. 6.12. Окно с надписями на русском языке

Как видно из рисунков, каждое стандартное диалоговое окно содержит такие элементы: предопределенный выбранным стилем 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 нужно поместить диалоговую панель `JOptionPane`. Для этого есть семь конструкторов. Конструктор по умолчанию `JOptionPane()` создает диалоговую панель с кнопкой **OK** и стандартной строкой сообщения. Она показана на рис. 6.13. Этот рисунок создан методами

```
JOptionPane op = new JOptionPane();
JDialog d = op.createDialog(this, " Простейшее диалоговое окно");
d.setVisible(true);
```

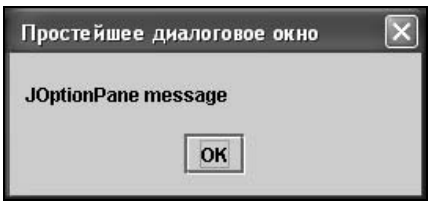


Рис. 6.13. Простейшее диалоговое окно

Конструктор с наибольшим числом параметров выглядит так:

```
JOptionPane(Object message, int messType, int optType,
   Icon icon, Object[] options, Object init);
```

у остальных конструкторов для отсутствующих параметров приняты значения по умолчанию. На рис. 6.14 показано 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);
```

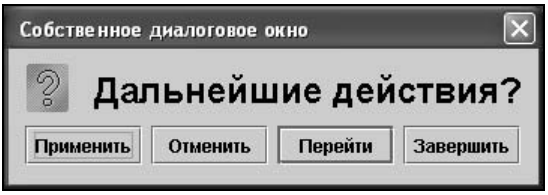


Рис. 6.14. Собственное диалоговое окно

Окно с индикатором *ProgressMonitor*

Еще один вид диалоговых окон позволяет следить за ходом какого-нибудь процесса, такое окно 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 3*.

В листинге 6.8 приведен пример использования окна индикатора. Для изменения значения индикатора запущен простейший подпроцесс. Рис. 6.15 показывает результат работы программы листинга 6.8.

Листинг 6.8. Создание окна индикатора

```
import java.awt.*;
import javax.swing.*;

public class Progress extends JFrame{
```

```
  Progress(){
    super(" Окно индикатора");

    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();
    }
}
```

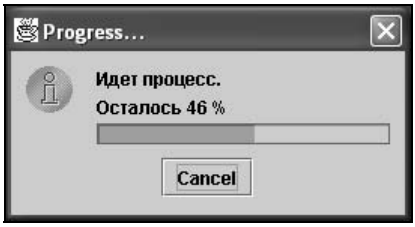


Рис. 6.15. Окно индикатора

В заключение хотелось бы отметить, что следить за загрузкой большого входного потока класса `InputStream` удобно с помощью специально разработанного фильтра входного потока — класса `ProgressMonitorInputStream`, расширяющего класс `FilterInputStream`.

Апплет *JApplet*

Апплеты, созданные средствами библиотеки Swing, расширяют класс `JApplet`, являющийся непосредственным расширением класса `Applet`. Как и всякий апплет, они наследуют четыре метода: `init()`, `start()`, `stop()`, `destroy()`. Подобно другим окнам верхнего уровня библиотеки Swing они содержат всего один компонент — корневую панель класса `JRootPane`. Поэтому к апплетам относится все сказанное выше о размещении компонентов в окне верхнего уровня `JFrame`. В частности, панелью содержимого по умолчанию управляет менеджер размещения `BorderLayout`.

Если в браузере, выполняющем апплет, есть 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 Java 2 JRE и автоматически устанавливается при инсталляции JRE. Если же модуля Java Plug-in на компьютере нет, то браузер предлагает загрузить его из Internet при первой попытке выполнения апплета. Загрузить Java Plug-in можно с сайтов **http://java.sun.com/** или **http://www.microsoft.com/java/** либо 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 SDK входит утилита $JAVA_HOME/lib/htmlconverter.jar. Она переписывает HTML-файл, содержащий тег `<APPLET>`, добавляя теги `<OBJECT>` и `<EMBED>`. Написанный выше 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 2 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 J2SE SDK, в каталоге $JAVA_HOME/docs/guide/plugin/, есть руководство разработчика, подробно объясняющее правила работы с Java Plug-in.

Глава 7

Обработка событий в Swing

Для обработки событий (events), происходящих при нажатии и отпускании клавиш на клавиатуре, работе с мышью, перемещении и изменении размеров окон и других действиях пользователя, графическая библиотека Swing использует ту же модель, называемую *моделью делегирования*, что и библиотека AWT. Напомним ее суть.

Событие может произойти в любом компоненте Swing или его модели данных. Компонент или контейнер, в котором произошло событие, называется *источником* (source) события.

Все события классифицированы и образуют весьма разветвленное дерево классов с корнем — классом `EventObject` пакета java.util. Его прямой наследник — класс `AWTEvent` — дает начало дереву классов-событий AWT, собранных в пакет java.awt.event. Они описаны во многих книгах, например, в [11, 15, 18]. Дополнительные события, используемые в библиотеке Swing, собраны в пакет javax.swing.event. По соглашению имена классов-событий имеют вид `XxxEvent`, вначале краткое имя события, потом слово `Event`.

При возникновении события исполняющая система Java распознает причину события и создает объект класса, соответствующего этой причине. В этом объекте содержится ссылка на компонент, в котором произошло событие, время события и различные сведения, зависящие от причины события. Например, для события, связанного с мышью, содержатся координаты ее курсора в системе координат компонента, отмечается состояние кнопок мыши. Для событий, поступивших от клавиатуры — код клавиши, уточнение: нажата клавиша или отпущена и состояние клавиш-модификаторов.

Созданный объект, содержащий сведения о событии, передается методам-обработчикам событий. Метод-обработчик, получив сведения о произошедшем событии, должен предпринять какие-то действия, как-то отреагировать на событие. Создание методов обработки событий — это задача разработчика, но сигнатуры этих методов строго описаны в нескольких интерфейсах, называемых *слушателями* (listeners) события. Имена этих интерфейсов, по соглашению, имеют вид `XxxListener`, повторяя имя события и заканчиваясь словом `Listener`. В пакетах java.awt.event и javax.swing.event собрано около сорока таких интерфейсов.

Разработчик, желающий обработать событие, должен реализовать соответствующий интерфейс, определив описанные в нем методы. Некоторые интерфейсы, например интерфейс `ActionListener`, имеют всего один метод. Разработчик, пишущий обработку события класса `ActionEvent`, должен определить только один метод `actionPerformed(ActionEvent)`, например,

```
class MilliSeconds implements ActionListener{
   private JLabel label;
   private long msecs, old;
   public MilliSeconds(JLabel label, long msecs){
      this.label = label; this.msecs = old = msecs;
   }
   public void actionPerformed(ActionEvent ae){
      if (ae.getActionCommand().equals("Миллисекунды")){
         msecs = ae.getWhen(); // Время события в миллисекундах.
         label.setText("" + (msecs — old));
         old = msecs;
      }
   }
}
```

Этот обработчик события работает следующим образом. Как только происходит событие класса `ActionEvent`, исполняющей системой Java автоматически выполняется метод `actionPerformed()` класса `MilliSeconds`. Если источник события отмечен словом "Миллисекунды", то в поле `label` появляется число миллисекунд, прошедших со времени предыдущего события.

Большинство интерфейсов описывает несколько методов, например, интерфейсы `InternalFrameListener`, `MouseInputListener`, `WindowListener` описывают по семь методов каждый. Их реализация превращается в скучное занятие по определению всех методов, особенно если разработчика интересует только один из них. Для облегчения этой задачи введены *классы-адаптеры*, реализующие интерфейсы, содержащие несколько методов, например, `InternalFrameAdapter`, `MouseInputAdapter`, `WindowAdapter`. По соглашению их имена повторяют имя интерфейса, но заканчиваются словом `Adapter`, а не `Listener`. Классы-адаптеры устроены предельно просто: они осуществляют пустую реализацию каждого метода своего интерфейса. Расширяя класс-адаптер, достаточно определить только нужные методы.

После того как класс-обработчик написан, его надо присоединить к компоненту — источнику события методом `addXxxListener(XxxListener)`, в имени которого повторяется имя события. Для приведенного выше примера можно присоединить обработчик `MilliSeconds` к кнопке:

```
JButton b = new JButton("Миллисекунды");
b.setMnemonic(KeyEvent.VK_L);
```

```
MilliSeconds ms = new MilliSeconds(label, System.currentTimeMillis());
b.addActionListener(ms);
```

После этого при каждом щелчке кнопкой мыши на кнопке `b` или нажатии комбинации командных клавиш <Alt>+<L> в то время, когда фокус ввода находится на кнопке `b`, автоматически будет выполнен метод `actionPerformed()` класса `MilliSeconds`. В результате надпись `label` будет содержать число миллисекунд, прошедших со времени предыдущего щелчка кнопкой мыши или нажатия упомянутых клавиш.

Так реализуется идея делегирования. Компонент не обрабатывает сам возникшее в нем событие, а делегирует обработку объекту класса-обработчика. Это очень удобно. Всегда можно присоединить новый обработчик события, предварительно удалив старый обработчик методом `removeXxxListener(XxxListener)`. Один обработчик можно присоединить к нескольким компонентам. С другой стороны, к одному компоненту можно присоединить несколько обработчиков разных событий. Правда, при этом нет гарантии, что обработка будет происходить в том же порядке, в котором присоединялись обработчики.

Если же обработчик реализуется вместе с компонентом, или нужно расширить класс-адаптер, а компонент уже расширяет другой класс, то удобно сделать класс-обработчик вложенным классом, использующим поля класса-компонента. В листинге 7.1 предыдущий пример реализован вложенным классом, который использует общее поле `label`.

Листинг 7.1. Обработка события вложенным классом

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ActInner extends JFrame{

   private JButton b = new JButton("Миллисекунды");
   private JLabel label = new JLabel("Отсчет", JLabel.RIGHT);
   private long msecs = System.currentTimeMillis();

   ActInner(){
     super(" Отсчет времени");
     Container c = getContentPane();
     c.setLayout(new FlowLayout());
     c.add(label);
```

```
      b.setMnemonic(KeyEvent.VK_L);
      c.add(b);

      MilliSeconds ms = new MilliSeconds(msecs);
      b.addActionListener(ms);

      setSize(400, 400);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setVisible(true);
    }

    public static void main(String[] args){
       new ActInner();
    }

    class MilliSeconds implements ActionListener{

       private long msecs, old;

       public MilliSeconds(long msecs){
          this.msecs = old = msecs;
       }

       public void actionPerformed(ActionEvent ae){
           if (ae.getActionCommand().equals("Миллисекунды")){
              msecs = ae.getWhen();
              label.setText("" + (msecs - old));
              old = msecs;
           }
        }
    }
}
```

Еще один способ — сделать сам класс обработчиком событий, происходящих в его компонентах. Для этого класс должен реализовать один или несколько интерфейсов-слушателей. Примеры такой реализации приведены в *главах 3*, *4* (см. листинги 3.1, 3.3, 3.4, 4.1).

Среди всех событий выделяются события *низкого уровня* (low-level events):

- `ComponentEvent` — происходит при изменении размеров и положения компонента или его перекрытии другим компонентом.

- `ContainerEvent` — возникает при добавлении компонента в контейнер или удалении компонента из него.
- `FocusEvent` — происходит при передаче фокуса ввода.
- `WindowEvent` — активизируется при открытии, закрытии, сворачивании и разворачивании окна, передаче ему фокуса ввода.
- `InputEvent` — возникает при вводе данных, например, мышью или с клавиатуры.
- `KeyEvent` — происходит при нажатии или отпускании клавиши.
- `MouseEvent` — генерируется действиями мыши.

Все остальные события называются *семантическими* (semantic). У них нет определенной причины возникновения. Например, событие класса `ActionEvent` возникает не только при щелчке кнопкой мыши на кнопке `JButton`, но и при нажатии клавиши <Enter> в поле ввода `JTextField`. Кроме того, семантические события могут вызываться событиями низкого уровня.

События низкого уровня происходят во всех компонентах классов `Component` и их наследниках. Их всегда можно отследить подходящим методом:

- `addComponentListener(ComponentListener)`;
- `addFocusListener(FocusListener)`;
- `addInputListener(InputListener)`;
- `addKeyListener(KeyListener)`;
- `addMouseListener(MouseListener)`;
- `addMouseMotionListener(MouseMotionListener)`;
- `addMouseWheelListener(MouseWheelListener)`.

В классе `Container` есть еще один метод

`addContainerListener(ContainerListener)`.

Семантические события происходят не во всех компонентах. Какие именно семантические события происходят в том или ином компоненте, можно посмотреть в документации к этому компоненту. Например, в классе `JComponent` есть три метода:

- `addAncestorListener(AncestorListener)`;
- `addPropertyChangeListener(PropertyChangeListener)`;
- `addVetoableChangeListener(VetoableChangeListener)`.

Таким образом, все компоненты Swing могут следить не только за событиями низкого уровня, но и за событиями `AncestorEvent` и `PropertyChangeEvent`. Рассмотрим их подробнее.

Событие изменения предка *AncestorEvent*

Событие класса `AncestorEvent` возникает при перемещении контейнера, в котором содержится компонент, по экрану, при удалении его с экрана или появлении на экране. Объект этого класса хранит не только ссылку на контейнер, в котором лежит компонент, но и ссылку на родительский контейнер, содержащий этот контейнер. Методы класса `AncestorEvent` предоставляют эти ссылки:

- `public Container getAncestor()` — контейнер, содержащий компонент;
- `public Container getAncestorParent()` — контейнер, содержащий контейнер;
- `public JComponent getComponent()` — компонент.

Статические поля класса позволяют узнать причину события.

- `static int ANCESTOR_ADDED` — контейнер появился на экране.
- `static int ANCESTOR_MOVED` — контейнер переместился по экрану.
- `static int ANCESTOR_REMOVED` — контейнер исчез с экрана.

Объект класса `AncestorEvent`, созданный действиями с контейнером, передается методам интерфейса `AncestorListener`. Интерфейс `AncestorListener` описывает три метода:

- `public void ancestorAdded(AncestorEvent event);`
- `public void ancestorRemoved(AncestorEvent event);`
- `public void ancestorMoved(AncestorEvent event);`

Эти методы должны обработать соответствующие причины события: появление, перемещение и удаление контейнера с экрана.

Событие изменения свойств *PropertyChangeEvent*

Все компоненты Swing являются в то же время и компонентами JavaBeans. Некоторые элементы JavaBeans выделяются как *связанные свойства* (bound properties). При изменении связанного свойства компонента JavaBean происходит событие класса `PropertyChangeEvent`. Кроме унаследованного метода `getSource()`, возвращающего ссылку на JavaBean, в котором произошло событие, объект класса `PropertyChangeEvent` содержит еще следующие методы:

- `String getPropertyName()` — имя изменившегося свойства;
- `Object getOldValue()` — старое значение свойства;
- `Object getNewValue()` — новое значение свойства.

Согласно спецификации JavaBeans приложение, использующее компонент JavaBean, должно иметь возможность отреагировать на событие и до фактического изменения свойств компонента, и после него. Поэтому процесс прослушивания события `PropertyChangeEvent` описывают сразу два интерфейса.

Первый интерфейс-слушатель, `PropertyChangeListener`, описывает всего один метод:

```
public void propertyChange(PropertyChangeEvent evt);
```

Он будет выполнен при каждом изменении любого связанного свойства компонента, который присоединил к себе объект — обработчик события методом

```
addPropertyChangeListener(PropertyChangeListener);
```

Такой способ удобен для задания реакции на изменение полей компонента, независимо от того, по какой причине это изменение произошло.

Метод, изменяющий свойство, должен возбуждать событие по следующей схеме:

```
private SwingPropertyChangeSupport pcs =
          new SwingPropertyChangeSupport(this);
private int someProp;
public void setSomeProp(int someProp){
   int old = this.someProp;
   this.someProp = someProp;
   pcs.firePropertyChange("someProp",
          new Integer(old), new Integer(someProp));
}
```

Здесь использован класс-утилита `SwingPropertyChangeSupport`, обладающий методами возбуждения события `firePropertyChange()`.

Среди свойств компонента JavaBean выделяются *ограниченные свойства* (constrained properties) — свойства, для которых можно задать границы их значений. Например, дни месяца можно ограничить значениями от 1 до 31. Ограниченность свойства обеспечивается проверкой его значения перед занесением в компонент JavaBean, а возможность проверки описана в интерфейсе `VetoableChangeListener`.

Второй интерфейс, `VetoableChangeListener` — слушатель события `PropertyChangeEvent` — тоже описывает только один метод:

```
public void vetoableChange(PropertyChangeEvent evt);
```

Важное различие между двумя интерфейсами, описывающими обработку события `PropertyChangeEvent`, заключается в том, что метод `vetoableChange()` должен выполняться до фактического изменения свойств

компонента, а метод `propertyChange()` — уже после изменения свойств. Алгоритм работы метода, изменяющего ограниченное свойство таков:

```
private VetoableChangeSupport vcs = new VetoableChangeSupport(this);
private int someProp;
public void setSomeProp(int someProp){
   vcs.fireVetoableChange("someProp",
          new Integer(this.someProp), new Integer(someProp));
   this.someProp = someProp;
}
```

Первоначально событие `PropertyChangeEvent` было введено для того, чтобы устанавливать свойства компонента JavaBean при встраивании его в приложение с помощью инструментального визуального средства, но это событие оказалось полезным и при создании графического интерфейса пользователя. Поскольку все компоненты AWT и Swing являются в то же время и компонентами JavaBeans, всегда можно отследить изменения их связанных и ограниченных свойств и как-то отреагировать на них. Это полезно в тех случаях, когда причина изменения свойства не важна для обработки события.

Управление колесиком мыши

Недавно появившееся колесико, дополнившее среднюю кнопку мыши, оказалось очень удобным и быстро завоевало популярность. Его использование встроено в J2SE SDK, начиная с версии JDK 1.4, и применяется, например, в контейнере `JScrollPane`. Если же разработчику нужно создать свой компонент, в котором прокрутка выполняется 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 структурной единицы — статическое поле `WHEEL_UNIT_SCROLL`. Тип прокрутки, а также смысл понятий "блок" и "единица" определяются графической оболочкой операционной системы. Тип прокрутки 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, на которой выполняется программа.

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);
```

которому передается объект-событие, происшедшее в результате вращения колесика мыши. В листинге 7.1 приведен тренировочный пример работы с колесиком мыши.

Листинг 7.2. 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();
    }
}
```

Рис. 7.1 показывает конкретные значения, полученные этой программой в операционной системе Windows XP.

Рис. 7.1. Значения, полученные от колесика мыши

Событие изменения состояния *ChangeEvent*

Еще одно событие общего характера `ChangeEvent` происходит при изменении состояния компонента: щелчке кнопкой мыши на кнопке 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` описываемое событие — основное.

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:

```
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");
   }
}
```

Остальные события Swing более специализированы. Они были описаны в *главах 3—5* вместе с генерирующими их компонентами.

Глава 8

Оформление рамок

Каждый стандартный графический компонент — это прямоугольная область на экране со сторонами, параллельными сторонам экрана. Стороны прямоугольника выделяются каким-то образом. Например, стороны кнопки `JButton` нарисованы так, что создают впечатление ее выпуклости. При нажатии кнопки мыши на ней графическое оформление сторон кнопки `JButton` меняется, создавая впечатление ее "вдавленности".

У некоторых компонентов, например, `JLabel`, вообще не оформлены границы.

В любом случае библиотека Swing позволяет изменить оформление границ любого компонента, в том числе и контейнера с помощью рамок разного вида.

Самые общие характеристики всех рамок описаны интерфейсом `Border` из пакета javax.swing.border. Вычерчивание рамки выполняется 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, либо чуть больше или чуть меньше их.

Рамка может быть прозрачной или непрозрачной. Это отмечается логическим методом `isBorderOpaque()`.

Третий и 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 есть 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 8.1 один из компонентов `JLabel` создан методами

```
JLabel l = new JLabel("Надпись с толстой синей рамкой");
l.setBorder(BorderFactory.createLineBorder(Color.blue, 3));
```

Рассмотрим подробнее некоторые типы рамок. Простые типы рамок, созданные программой листинга 8.1, показаны на рис. 8.1.

Листинг 8.1. Простые рамки

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class SimpBorders extends JFrame{

   SimpBorders(){
     super(" Простые рамки");
     Container c = getContentPane();
     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(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

  public static void main(String[] args){
    new SimpBorders();
  }
}
```

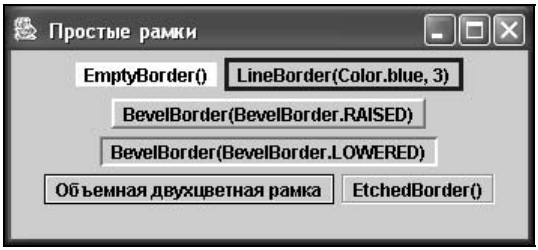


Рис. 8.1. Простые рамки

Пустая рамка *EmptyBorder*

Класс `EmptyBorder` представляет самую простую рамку. Это просто пустое пространство, окружающее компонент, как показано на рис. 8.1 сверху слева. Конструкторы класса `EmptyBorder(Insets)` и `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 для всех сторон рамки. Она показана на рис. 8.1 сверху справа. Рамка заданного цвета толщиной в один пиксел создается конструктором `LineBorder(Color)`. Конструктор `LineBorder(Color, int)` задает 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. 8.1 во второй строке. Если же поменять местами темные и светлые линии, то компонент выглядит вдавленным в поверхность контейнера. Это тип `LOWERED`, на рис. 8.1 он показан в третьей строке. Именно так создается кнопка `JButton`.

Конструктор `BevelBorder(int type)` создает 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()` и `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`.

Объемная рамка с закругленными краями *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, `RAISED`, или вдавливаясь, `LOWERED`, в его поверхность. Такая рамка показана на рис. 8.1 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 задаются 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. 8.2 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 класса `MatteBorder` 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 8.2 содержит программу, создавшую рис. 8.2.

Листинг 8.2. 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();
     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(JFrame.EXIT_ON_CLOSE);
     setVisible(true);
   }

   public static void main(String[] args){
     new MatBorders();
   }
}
```

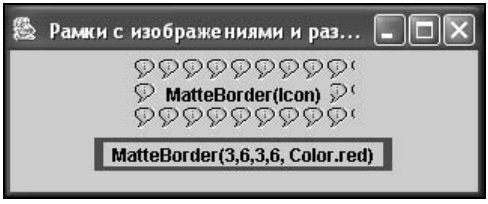


Рис. 8.2. Рамка с изображениями и рамка с линиями разной толщины

Рамка с надписью *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. 8.3 сверху.

Надпись можно вставить в рамку любого типа. Для этого используется конструктор `TitledBorder(Border, String)` или статический метод

```
createTitledBorder(Border, String);
```

На рис. 8.3 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 — допустимые значения параметра `pos` в конструкторах и методах, описанных ниже.

По умолчанию надпись располагается слева, `LEFT`, но ее можно расположить по центру, `CENTER`, или справа, `RIGHT`. Эти константы — допустимые значения параметра `just` в конструкторах и методах, описанных ниже.

Полный набор констант, определяющих место надписи, перечислен в листинге 8.5.

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. 8.3.

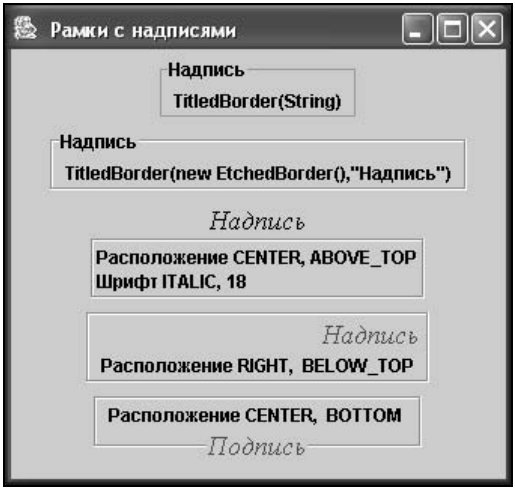


Рис. 8.3. Рамки с надписями

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 задать 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);
```

Листинг 8.3 содержит программу, создавшую рис. 8.3. В ней приведены все описанные возможности создания рамок с надписями.

Листинг 8.3. 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();
     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(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

  public static void main(String[] args){
    new TitBorders();
  }
}
```

Сдвоенная рамка *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`.

Число возможных комбинаций двух рамок очень велико. На рис. 8.4 показаны некоторые из них, а в листинге 8.4 приведена программа, создающая это окно.

Листинг 8.4. Сдвоенные рамки

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class CompBorders extends JFrame{

   CompBorders(){
     super(" Сдвоенные рамки");
     Container c = getContentPane();
     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(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

  public static void main(String[] args){
    new CompBorders();
  }
}
```

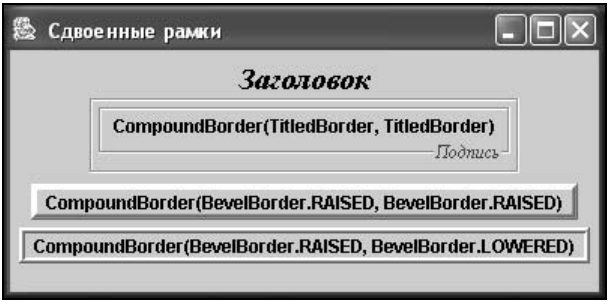


Рис. 8.4. Сдвоенные рамки

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 непрозрачна, то надо переопределить метод `isBorderOpaque()` так, чтобы он возвращал `true`.

Свою рамку можно создать, расширив какой-либо класс рамок. В листинге 8.5 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`. Эти действия описаны в листинге 8.5, а результат показан на рис. 8.5.

Листинг 8.5. 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();

      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(JFrame.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. 8.5. Рамка с компонентом `JLabel`

Глава 9

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 определяет его форма, тип и цвет рамки, цвет фона, цвет, тип и размер шрифта, форма курсора мыши. *Поведение* компонента (feel) определяет та или иная реакция на действия мыши, набор командных клавиш, способ перемещения окон и т. д. Набор таких характеристик всех компонентов приложения определяет его вид и поведение и называется стилем Look and Feel, сокращенно L&F. Всего набирается несколько сотен качеств, 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 эти вид и поведение называются "CDE/Motif".
- `WindowsLookAndFeel` — вид и поведение Win32, характерные для платформы MS Windows.

- `MacLookAndFeel` — вид и поведение, свойственные 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 для соответствующей платформы.

Платформа Java определяет и свой собственный, независимый от платформы, стиль Java Look and Feel, короче называемый Java L&F, еще короче — JLF. В документации сохраняется его старое название "Metal", данное ему за схожесть этого стиля с гравировкой по металлу. Java L&F в русской технической литературе называется "приборным" стилем из-за сходства с алюминиевыми панелями научных приборов. Приложение, работающее в стиле JLF, будет выглядеть одинаково на всех платформах. Именно этот стиль использован во всех примерах, приведенных в предыдущих главах.

Информация о виде и поведении, характерных для стиля Java L&F, собрана в четвертом наследнике класса `BasicLookAndFeel` — классе `MetalLookAndFeel`.

Сведения, хранящиеся 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 стиля L&F: "родной" для данной платформы, CDE/Motif и Java L&F. Они показаны на рис. 9.1.

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 может оформлять окна, то все создаваемые окна классов `JFrame` и `JDialog` будут оформлены текущим L&F (см. рис. 9.4).

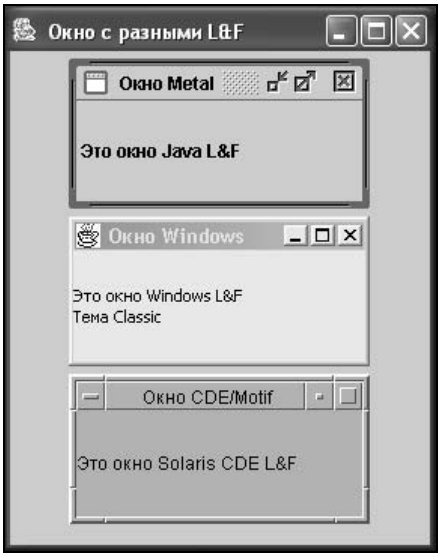


Рис. 9.1. Стандартные L&F

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 иметь внешний вид, определенный текущим оконным менеджером, могут следовать правилам другого оконного менеджера или быть оформлены в своей манере. Вид и поведение графического приложения определяются перед его загрузкой, но их можно изменить уже во время работы приложения.

Это качество графических элементов библиотеки Swing получило название Pluggable Look and Feel (Изменяемый вид и поведение), сокращенно PL&F, PLAF или plaf.

Получение сведений о свойствах L&F

Для получения значений свойств текущего стиля L&F и установки нового L&F вместо классов `XxxLookAndFeel` удобнее использовать класс `UIManager` из пакета javax.swing. Он содержит множество статических методов, позво-

ляющих получить сведения об элементах L&F, изменить некоторые элементы или вообще сменить стиль 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 хранящийся 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 нет файла swing.properties, то устанавливается стиль 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 использовании 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 такого метода есть парный метод `getXxx(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 стиля L&F без изменения. Посмотреть установки L&F можно статическим методом `geLookAndFeelDefaults()`, возвращающим экземпляр класса `UIDefaults`.

Для того чтобы легче изменять вид и поведение приложения, класс `UIManager` хранит несколько L&F с произвольными 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 стилей 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 стиля L&F

Очень легко задать какой-либо из стандартных стилей PL&F, воспользовавшись одним из статических методов `setLookAndFeel(LookAndFeel)` или `setLookAndFeel(String)` класса `UIManager`. Аргумент второго из этих методов — строка, содержащая полное имя класса `XxxLookAndFeel` со всеми подпакетами. Например, строка "javax.swing.plaf.metal.MetalLookAndFeel" задает имя класса, определяющего стиль 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, стиль Java L&F устанавливается по умолчанию, как

можно видеть в многочисленных примерах, приведенных в предыдущих главах. Еще один статический метод `getSystemLookAndFeelClassName()` класса `UIManager` возвращает полное имя класса, определяющего стандартный стиль 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 стиль 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 9.1 приведена программа, создавшая рис. 9.1. В ней каждое внутреннее окно строится по правилам одного из стандартных стилей L&F.

Листинг 9.1. Внутренние окна со стандартными стилями L&F

```
import java.awt.*;
import javax.swing.*;

public class DiffLAF extends JFrame{

   DiffLAF(){
     super(" Окно с разными L&F");
     Container c = getContentPane();
     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(JFrame.EXIT_ON_CLOSE);
      setVisible(true);
    }

    public static void main(String[] args){
      new DiffLAF();
    }
}
```

Дополнительные стили L&F

Кроме текущего стиля L&F библиотека Swing может одновременно использовать дополнительные (auxiliary) стили L&F. Они являются расширениями класса `MultiLookAndFeel` из пакета javax.swing.plaf.multi. Полные имена их классов `XxxLookAndFeel` 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 стиля L&F, а затем объекты дополнительных стилей L&F в порядке их записи в строке `swing.auxiliarylaf` в файле swing.properties.

Дополнительные стили 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 стилем L&F. Поэтому не рекомендуется использовать в качестве дополнительных графических стилей L&F-классы, унаследованные от `BasicLookAndFeel`, а наследовать их прямо от `MultiLookAndFeel`.

Смена всего стиля L&F

Разработчик GUI может сменить весь стиль L&F своего приложения или его отдельные свойства. Для смены всего стиля L&F сначала устанавливается новый L&F:

```
UIManager.setLookAndFeel(new MyCoolLookAndFeel());
```

Затем надо привести вид и поведение приложения в соответствие с новым стилем L&F.

При каждом изменении какого-либо свойства, входящего в L&F, или всего стиля 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 стиля L&F. Это удобно сделать статическим методом `updateComponentTreeUI(Component)` класса `SwingUtilities`. В аргументе этого метода достаточно указать контейнер верхнего уровня, метод передаст сообщение всем вложенным контейнерам и компонентам.

Итак, изменение стиля 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 стиль L&F для этого компонента.

В листинге 9.2 приведен пример программы, дающей возможность пользователю менять стиль L&F во время работы приложения, выбрав нужный пункт меню. На рис. 9.2 показан результат работы этой программы.

Листинг 9.2. 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");
     final Container c = getContentPane();
     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){}
        }
    }
}
```

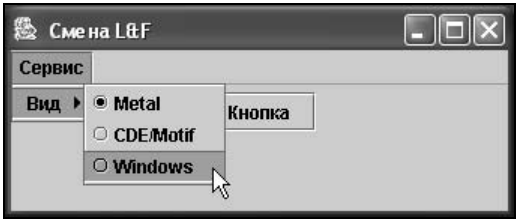


Рис. 9.2. Смена стиля L&F

В составе J2SE SDK, в каталоге $JAVA_HOME/demo/jfc/SimpleExample/ приведен пример смены стиля L&F во время работы приложения. Для его запуска достаточно перейти в этот каталог и набрать в командной строке

```
java -jar SimpleExample.jar
```

Замена отдельных свойств стиля L&F

Во время работы приложения можно заменить не весь текущий стиль 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 надо присоединить обработчик событий, получив ссылку на него методом `getDefaults()`. Потом надо оповестить все заинтересованные компоненты о сделанных изменениях методом `updateComponentTreeUI(Component)`.

В листинге 9.3 приведена программа, в которой проделаны эти действия. Она позволяет заменить размер шрифта в текстовых полях другим размером, выбранным из меню.

Листинг 9.3. 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");
     final Container c = getContentPane();
     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(JFrame.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. 9.3 показан результат работы программы листинга 9.3. 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.

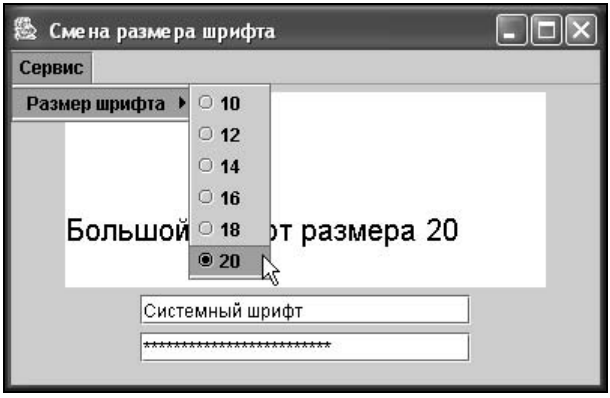


Рис. 9.3. Поля ввода с измененным размером шрифта

Темы стиля Java L&F

Вид и поведение Java L&F основаны на трех основных (primary) цветах и трех дополнительных (secondary) фоновых цветах, из которых образуется колорит оформления приложения. Кроме того, выбран цвет для ввода текста и цвет фона текстовых полей. Выбор каких-то 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 — это 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 для оформления рамок неактивных компонентов, цвет secondary3 — цвет фона неактивных компонентов.

Кроме этих цветов тема `DefaultMetalModel` определяет черный цвет для текста, вводимого в текстовые поля, и белый цвет — для фона текстовых полей.

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`.

Методы, задающие 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)`. Таким образом, метод, возвращающий первый основной цвет, может выглядеть так:

```
public ColorUIResource getPrimary1(){
   return new ColorUIResource(36, 124, 225);
}
```

Методы, определяющие шрифты, должны возвращать объект класса `FontUIResource`, для создания которого есть два конструктора: `FontUIResource(Font)`, `FontUIResource(String name, int style, int size)`. Метод, задающий 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 стиля 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 листин-

ге 9.4 приведен пример метода, устанавливающего новые рамки текстовых полей.

После того как написан класс темы, расширяющий `MetalTheme` или `DefaultMetalTheme`, надо установить новую тему статическим методом `setCurrentTheme(MetalTheme)` класса `MetalLookAndFeel`, как показано в листинге 9.4. В нем установлены только серые цвета, что удобно для печати иллюстраций в книге. Результат приведен на рис. 9.4.

Листинг 9.4. 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();
     getContentPane().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(JFrame.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);
   }
}
```

В стандартной поставке J2SE SDK, в каталоге $JAVA_HOME/demo/jfc/MetalWorks/ хранятся 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
```

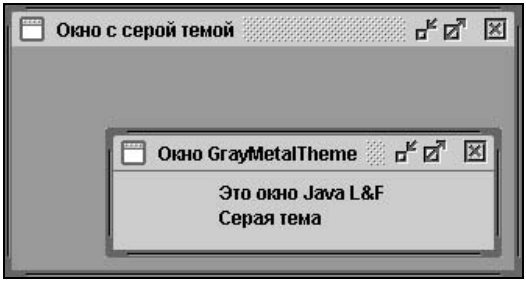


Рис. 9.4. Окна, оформленные в "серой" теме

Там же есть исходные тексты приложения, из которых видно, как создаются и используются темы Java L&F.

Глава 10

Прочие свойства Swing

В этой, заключительной, главе *части II* книги, собраны разнообразные сведения о библиотеке Swing, не вошедшие в предыдущие главы.

Индивидуальные 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. 6.5). 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()` использует это свойство. Приведем 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 отличать один экземпляр от другого:

```
Integer numb = (Integer)comp.getClientProperty("Number");
```

Подобная схема применена в листинге 10.1.

Прокрутка содержимого компонента

Содержимое компонентов Swing, находящихся на панели `JScrollPane`, легко прокручивать с помощью линеек прокрутки или колесика мыши. Но возможность прокрутки заложена не только в компоненты, реализующие интерфейс `Scrollable`. Эта возможность есть у каждого компонента Swing.

Возможность прокрутки содержимого компонента активизируется методом `setAutoscrolls(true)`. Если после выполнения этого метода вывести курсор мыши за пределы компонента при нажатой кнопке мыши (drag), то возникает событие, инициируемое мышью. Обычная обработка этого события заключается в прокрутке содержимого компонента. Для того чтобы сделать это, надо присоединить к компоненту обработчик события мыши и обратиться 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 одноименному методу объемлющего контейнера. Некоторые контейнеры, например, `JViewport`, переопределяют этот метод таким образом, чтобы в окне появился участок компонента, входящий в 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 графических 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. Обычный порядок следования 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 линеек 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 этот интерфейс, и `true`, если передача события на этом заканчивается. Событие доводится до этого метода *раньше*, чем до слушателя `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 клави-

ши. В нем определены следующие статические константы, соответствующие указанным комбинациям клавиш.

- `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.

Эти 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 считать следующим при применении навигационных клавиш, делается абстрактным классом `FocusTraversalPolicy`. Этот класс содержит методы, возвращающие ссылки:

- `getComponentAfter(Container, Component)` — на следующий компонент и его контейнер;
- `getComponentBefore(Container, Component)` — на предыдущий компонент и его контейнер;
- `getFirstComponent(Container)` — на первый компонент в контейнере;
- `getLastComponent(Container)` — на последний компонент в контейнере;
- `getDefaultComponent(Container)` — на компонент, которому передается фокус при активизации контейнера;
- `getInitialComponent(Window)` — на компонент, которому передается фокус при активизации окна.

Два конкретных порядка следования компонентов определены двумя подклассами класса `FocusTraversalPolicy`.

Первый подкласс `ContainerOrderFocusTraversalPolicy` описывает порядок, 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 отвечающие хотя бы одному из перечисленных четырех условий.

У класса `ContainerOrderFocusTraversalPolicy` есть расширение `DefaultFocusTraversalPolicy`, которое переопределяет метод `accept(Component)` таким образом, что четвертое условие `isFocusable()` учитывается, только 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 использует порядок расположения компонентов, определенный некоторой реализацией интерфейса `Comparator`, сделанной разработчиком. Описание этого интерфейса и правила его реализации можно посмотреть в книге [15]. Конкретная реализация интерфейса задается в конструкторе класса `SortingFocusTraversalPolicy(Comparator)`.

Класс `SortingFocusTraversalPolicy`, в свою очередь, расширен классом `LayoutFocusTraversalPolicy`. Он определяет порядок размещения компонентов, основываясь на их размерах и положении в контейнере. Именно этот класс используется по умолчанию при создании стиля L&F, описанного классом `BasicLookAndFeel` и его расширениями.

После того как определен порядок следования компонентов, методом `setDefaultFocusTraversalPolicy(FocusTraversalPolicy)`, он 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 аналогичные переходы устанавливаются методами `transferFocus()`, `transferFocusBackward()` класса `Component`.

В листинге 10.1 приведен простой пример передачи фокуса от текстового поля к кнопке и обратно. На рис. 10.1 показан протокол этой программы.

Листинг 10.1. 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(JFrame.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();
    }
}
```

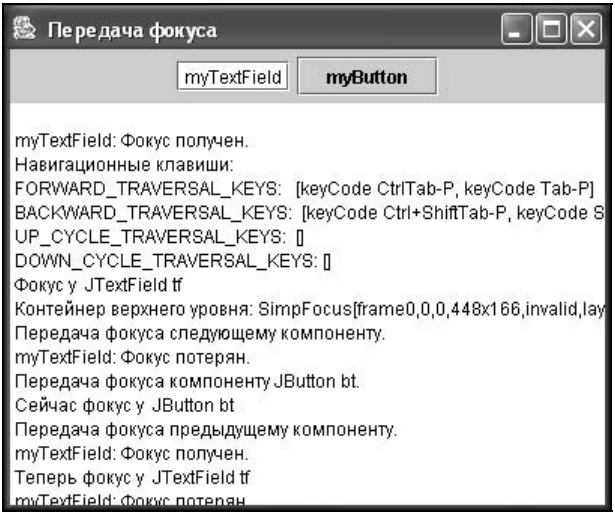


Рис. 10.1. Протокол передачи фокуса

Перенос данных методом Drag and Drop

В компоненты Swing `JColorChooser`, `JFileChooser`, `JList`, `JTable`, `JTree`, `JTextComponent` и в их потомков встроена возможность переноса данных с помощью мыши (Drag and Drop) или командных клавиш (для команд **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 требованиям интерфейса `Transferable`, описывающего 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. Он 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 графических платформ копирование происходит при нажатой клавише <Ctrl>, но это зависит от используемого стиля 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 10.2 показывает простой пример переноса выделенного текста из поля ввода класса `JTextField` в надпись класса `JLabel` перетаскиванием мышью.

Листинг 10.2. 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(JFrame.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 инициируется подпроцесс, в котором через заданное время

millisec будет 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). Есть возможность отдельно задать начальную задержку методом setInitialDelay(int).

Если обратиться к методу setLogTimers(true), то при каждой генерации события ActionEvent в стандартный вывод System.out будет выводиться протокол, а именно, сообщение "Timer ringing:" со ссылкой на объект Timer.

В листинге 10.3 приведен простой пример использования класса Timer. Сообщение о событии выводится в область ввода. Нажатие на кнопку прекращает работу "будильника".

Листинг 10.3. Применение методов 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(JFrame.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 10.3 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.

Интерфейс пользователя СДО

Наконец, после этого длинного обзора возможностей графической библиотеки Swing, растянувшегося на восемь глав, мы можем построить графический интерфейс системы дистанционного обучения (СДО). В *главе 2* мы описали графические страницы, которые должны появляться на экране при различных действиях ученика, учителя и администратора СДО.

Поскольку наша система дистанционного обучения рассчитана на использование в Internet, есть три варианта реализации графического интерфейса средствами Java.

- Страницы HTML, хранящиеся на сервере СДО или динамически генерируемые сервлетами или JSP.
- Апплеты Java с графическим интерфейсом Swing, загружаемые с сервера и запускаемые с помощью браузера.
- Приложения Java с графическим интерфейсом Swing, загружаемые с сервера и запускаемые на клиенте после инсталляции.

Самое простое решение — оформить интерфейс страницами HTML. Их можно просматривать в любом браузере. HTML-формы позволяют пересылать на сервер небольшие порции информации. Там, где не требуется соблюдать секретность, нет сложной обработки больших объемов информации, нет анимации и сложной графики страницы HTML — наилучшее решение. В системе дистанционного обучения страницами HTML можно выразить большинство элементов графического интерфейса: начальную страницу сайта СДО, страницу регистрации ученика, страницу ее подтверждения, страницу оценки ответов ученика, начальную страницу учителя.

Апплеты и приложения Java удобны в том случае, когда обработку информации надо проводить непосредственно на клиенте, не отвлекаясь на пересылку данных по сети. Библиотека Swing позволяет создать графический интерфейс, отвечающий самым высоким требованиям. При этом апплеты сильно ограничены в своих действиях требованиями безопасности [1, 11, 15, 18]. Кроме того, не во всяком браузере реализована библиотека Swing, хотя положение меняется: в новейшие версии браузеров вводится библиотека Swing, и всегда есть возможность загрузить Java Plug-in. В системе дистанционного обучения апплетами можно выразить материал учебной темы, страницу ответов ученика, страницу ввода учебного материала учителем.

Утилиту администратора, который постоянно работает с СДО, удобнее оформить как приложение Java, которое администратор может инсталлировать на сервере или в любом удобном для него месте. В листинге 10.4 приведен исходный код графического интерфейса утилиты администратора. На рис. 10.2 показано основное окно утилиты.

Листинг 10.4. Графический интерфейс утилиты администратора СДО

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;
```

```
class AdminSDO extends JFrame{

  JPopupMenu popup = new JPopupMenu("Контекстное меню");

  public AdminSDO(){
     super(" Консоль администратора СДО");
     enableEvents(AWTEvent.MOUSE_EVENT_MASK);
     Container c = getContentPane();

// -------- Меню и панель инструментальных кнопок -----
    JToolBar tb = new JToolBar();
    c.add(tb, BorderLayout.NORTH);

    JMenuBar mb = new JMenuBar();
    setJMenuBar(mb);

// -------------- Меню "Файл" -------------------------
   JMenu menu = new JMenu("<html><u>Ф</u>айл");
    mb.add(menu);
    menu.setMnemonic(KeyEvent.VK_A);

    Action act = new AbstractAction("Открыть",
                      new ImageIcon("open16.gif")){
      {
        putValue(Action.SHORT_DESCRIPTION, "Открыть");
        putValue(Action.ACCELERATOR_KEY,
          KeyStroke.getKeyStroke(KeyEvent.VK_O, Event.CTRL_MASK));
      }
      public void actionPerformed(ActionEvent e){
          JFileChooser fch = new JFileChooser("Открытие документа");
          fch.showOpenDialog(null);
      }
    };
    tb.add(act);
    menu.add(act);
    popup.add(act);

    act = new AbstractAction("Сохранить",
                 new ImageIcon("Save16.gif")){
```

```
      {
        putValue(Action.SHORT_DESCRIPTION, "Сохранить");
        putValue(Action.ACCELERATOR_KEY,
          KeyStroke.getKeyStroke(KeyEvent.VK_S, Event.CTRL_MASK));
      }

      public void actionPerformed(ActionEvent e){
         // Запись в файл
         JOptionPane.showInputDialog(this, "Заглушка");
      }
    };
   tb.add(act);
   menu.add(act);
   popup.add(act);

   act = new AbstractAction("Сохранить как...",
                 new ImageIcon("SaveAs16.gif")){
      {
       putValue(Action.SHORT_DESCRIPTION, "Сохранить как");
      }

      public void actionPerformed(ActionEvent e){
         JFileChooser fch = new JFileChooser("Сохранение документа");
         fch.showSaveDialog(null);
      }
    };
   tb.add(act);
   menu.add(act);
   popup.add(act);

   tb.addSeparator();
   menu.addSeparator();
   popup.addSeparator();

   JMenuItem  exit = new JMenuItem("Выход");
   exit.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e){
          System.exit(0);
      }
```

```
    });
    menu.add(exit);

// ---------------- Меню "Правка" ---------------------
    mb.add(menu = new JMenu("<html><u>П</u>равка"));
    menu.setMnemonic(KeyEvent.VK_G);

    act = new AbstractAction("Вырезать",
                  new ImageIcon("Cut16.gif")){
      {
       putValue(Action.SHORT_DESCRIPTION, "Вырезать");
       putValue(Action.ACCELERATOR_KEY,
         KeyStroke.getKeyStroke(KeyEvent.VK_X, Event.CTRL_MASK));
      }

      public void actionPerformed(ActionEvent e){
           // Вырезать
          JOptionPane.showInputDialog(this, "Заглушка");
      }
    };
    tb.add(act);
    menu.add(act);
    popup.add(act);

    act = new AbstractAction("Копировать",
                   new ImageIcon("Copy16.gif")){
      {
       putValue(Action.SHORT_DESCRIPTION, "Скопировать");
       putValue(Action.ACCELERATOR_KEY,
         KeyStroke.getKeyStroke(KeyEvent.VK_C, Event.CTRL_MASK));
      }

      public void actionPerformed(ActionEvent e){
             // Скопировать
          JOptionPane.showInputDialog(this, "Заглушка");
      }
   };
```

```
    tb.add(act);
    menu.add(act);
    popup.add(act);

    act = new AbstractAction("Вставить",
                  new ImageIcon("Paste16.gif")){
      {
       putValue(Action.SHORT_DESCRIPTION, "Вставить");
       putValue(Action.ACCELERATOR_KEY,
          KeyStroke.getKeyStroke(KeyEvent.VK_V, Event.CTRL_MASK));
      }

      public void actionPerformed(ActionEvent e){
            // Вставить
          JOptionPane.showInputDialog(this, "Заглушка");
      }
    };
    tb.add(act);
    menu.add(act);
    popup.add(act);

    tb.addSeparator();

// -------------------- Меню "База" ---------------------
    mb.add(menu = new JMenu("<html>Б<u>а</u>за"));
    menu.setMnemonic(KeyEvent.VK_F);
    JMenuItem conf = menu.add("Конфигурация...");
    conf.addActionListener(new ActionListener(){
       public void actionPerformed(ActionEvent e){
           JOptionPane.showInputDialog(this, "Заглушка");
       }
    });

// -------------------- Меню "Отчет" -------------------
    mb.add(menu = new JMenu("<html><u>О</u>тчет"));
    menu.setMnemonic(KeyEvent.VK_J);
       JMenuItem stud   = menu.add("Ученики");
       JMenuItem stCourses  = menu.add("Предметы");
       JMenuItem courses = menu.add("Курсы");
```

```
        JMenuItem topics   = menu.add("Темы");
        JMenuItem tasks  = menu.add("Задачи");
        JMenuItem teacher = menu.add("Учителя");

// -------------------- Меню "Справка" -----------------
    mb.add(menu = new JMenu("<html><u>C</u>правка"));
    menu.setMnemonic(KeyEvent.VK_C);

    act = new AbstractAction("Справка",
                   new ImageIcon("Help16.gif")){
      {
       putValue(Action.SHORT_DESCRIPTION, "Справка");
      }

      public void actionPerformed(ActionEvent e){
            // Вызов справки
          JOptionPane.showMessageDialog(null, "Заглушка");
      }
    };
    tb.add(act);
    menu.add(act);
    menu.addSeparator();
    JMenuItem about  = menu.add("О программе");
    about.addActionListener(new ActionListener(){
       public void actionPerformed(ActionEvent e){
           JOptionPane.showMessageDialog(null, "Заглушка");
       }
    });

// ------------- Конец описания меню ------------------

// ------------- Двойная панель -----------------------
    JSplitPane sp = new JSplitPane();
    sp.setOneTouchExpandable(true);
    c.add(sp);

// --------------- Текстовая область ---------------
//   final JTextArea ta = new JTextArea();
//   sp.setRightComponent(new JScrollPane(ta));
```

```
// -------------- Дерево -------------------------

    DefaultMutableTreeNode root =
       new DefaultMutableTreeNode("База СДО");

    JTree tr = new JTree(root);
    sp.setLeftComponent(new JScrollPane(tr));

    DefaultMutableTreeNode students =
       new DefaultMutableTreeNode("Студенты");
    root.add(students);

    DefaultMutableTreeNode studIds =
       new DefaultMutableTreeNode("Сведения");
    students.add(studIds);

    DefaultMutableTreeNode studCourses =
       new DefaultMutableTreeNode("Курсы");
    students.add(studCourses);

    DefaultMutableTreeNode teach =
        new DefaultMutableTreeNode("Учителя");
    root.add(teach);

    DefaultMutableTreeNode teachCourses =
       new DefaultMutableTreeNode("Курсы");
    teach.add(teachCourses);

    DefaultMutableTreeNode tsks =
       new DefaultMutableTreeNode("Задачи");
    root.add(tsks);

    tr.addTreeSelectionListener(new TreeSelectionListener(){
       public void valueChanged(TreeSelectionEvent e){
          TreePath[] path = e.getPaths();
          for (int i = 0; i < path.length; i++){
//              ta.append(path[i].getLastPathComponent() + "\n");
          }
       }
    });
```

```
// --------------- Таблица -------------------------
    String[][] data = {
      {"Иванов", "Кулинария", "25 июля 2001"},
      {"Петров", "Астрология", "10 декабря 2001"},
      {"Сидоров", "Уфология", "23 октября 2001"},
      {"Токарев", "Философия", "15 ноября 2001"}};
    String[] head = {"Фамилия", "Название", "Дата"};

   JTable tbl = new JTable(data, head);
   sp.setRightComponent(new JScrollPane(tbl));

// ---------- Конец оформления -------------------

    LoginWin lw = new LoginWin(this);
    lw.setVisible(true);

    setSize(500, 400);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

  public void processMouseEvent(MouseEvent e){
    if (e.isPopupTrigger())
        popup.show(e.getComponent(), e.getX(), e.getY());
    else super.processMouseEvent(e);
  }

  public static void main(String[] args){
    new AdminSDO();
  }
}

class LoginWin extends JDialog{
    LoginWin(JFrame fr){
      super(fr, "Окно входа", true);
      Container c = getContentPane();
      c.add(new JLabel(
        " Вход в утилиту администратора СДО "), BorderLayout.NORTH);
```

```
        JPanel p1 = new JPanel(false);
        p1.setLayout(new BoxLayout(p1, BoxLayout.X_AXIS));
        c.add(p1);

        JPanel  p2 = new JPanel(false);
        JButton b1 = new JButton("Войти");
        JButton b2 = new JButton("Отменить");
        p2.add(b1); p2.add(b2);
        c.add(p2, BorderLayout.SOUTH);

        JPanel p3 = new JPanel(false);
        p3.setLayout(new GridLayout(0, 1));
        p1.add(p3);

        p3.add(new JLabel("Имя: ", JLabel.RIGHT));
        p3.add(new JLabel("Пароль: ", JLabel.RIGHT));

        JPanel p4 = new JPanel(false);
        p4.setLayout(new GridLayout(0, 1));
        p1.add(p4);
        JTextField name = new JTextField(10);
        p4.add(name);
        JPasswordField pass = new JPasswordField(10);
        p4.add(pass);

        pack();
    }
}
```

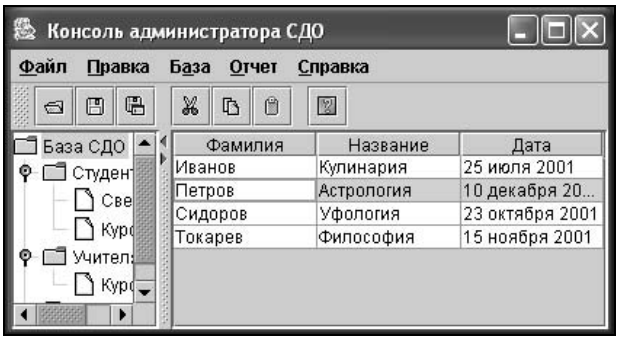


Рис. 10.2. Главное окно утилиты администратора

Часть III

Связь приложения Java с базами данных

Глава 11

Соединение приложения с базой данных

Громадное количество накопленной человечеством информации хранится в разнообразных базах данных, разбросанных по всему свету. Каждое приложение, работающее с постоянно хранимой информацией, должно уметь сохранять и извлекать информацию из одной или нескольких баз данных.

У каждой системы управления базами данных (СУБД) свои способы связи с внешним миром. Большинство современных СУБД основано на реляционной модели. Их язык общения — это язык SQL (Structured Query Language). Реляционная СУБД должна предоставить библиотеку функций API (Application Programming Interface), вызывающих или реализующих операторы SQL, для того чтобы приложение, написанное на языке программирования высокого уровня, могло обращаться к ней. Такие функции пишутся на том же самом языке высокого уровня. В составе СУБД есть либо препроцессор, преобразующий вызовы этих функций в операторы SQL, либо компилятор, переводящий их непосредственно в машинный язык.

Естественно возникает желание стандартизировать набор функций вызова операторов SQL, чтобы приложение могло обращаться к разным СУБД, не переписывая эти функции. Такой стандарт, названный *CLI* (Call Level Interface), был предложен совместно комитетом по стандартизации X/Open (сейчас этот комитет называется Open Group) и комитетом SAG (SQL Access Group), позднее вошедшим в состав X/Open.

Фирма Microsoft Corporation, основываясь на стандарте CLI, разработала обобщенный набор прототипов функций для связи и работы с СУБД, названный *интерфейсом ODBC* (Open Database Connectivity). Эти прототипы, описанные на языке C, не используют особенности той или иной СУБД, а только стандарт SQL. Следовательно, они могут выполняться в любой базе данных, для которой есть реализация этих прототипов. Такая реализация, названная *драйвером ODBC*, должна быть написана для каждой СУБД, предоставляющей интерфейс ODBC.

Интерфейс ODBC оказался удобным и быстро стал популярным. Драйверы ODBC написаны для всех известных СУБД. Одни из них входят в стандартную поставку СУБД, другие поставляются вместе с операционной системой,

третьи создаются независимыми фирмами. Созданы тысячи приложений СУБД с использованием ODBC. Технология Java не могла не использовать такой популярный интерфейс. Но он сильно связан с языком C, а это не только противоречит объектно-ориентированному стилю Java, но и ограничивает переносимость и безопасность приложения.

По этим причинам для связи с СУБД в технологии Java создан аналогичный ODBC *интерфейс JDBC* (это товарный знак, а не аббревиатура, хотя часто расшифровывается как "Java Database Connectivity"), тоже основанный на стандарте CLI. Интерфейсы и классы JDBC составили два пакета java.sql и javax.sql. Эти пакеты входят и в состав Java 2 SDK, Standard Edition, и в состав Java 2 SDK, Enterprise Edition.

Для использования интерфейса JDBC, как и для ODBC, следует написать *драйвер JDBC*, реализующий интерфейсы и методы JDBC для конкретной СУБД. Таких драйверов, написанных разными производителями для разных СУБД, насчитывается уже несколько десятков. Они перечислены на Web-странице фирмы Sun Microsystems **http://industry.java.sun.com/products/jdbc/drivers/**. Технология Java делит драйверы JDBC на четыре типа.

1. Мост JDBC-ODBC. Этот драйвер преобразует обращения к методам JDBC в вызовы функций ODBC. Кроме моста JDBC-ODBC, на клиентскую машину приходится загружать и соответствующий драйвер ODBC, что неудобно во многих случаях. В драйверах JDBC первого типа вместо ODBC может использоваться какой-нибудь другой сторонний способ доступа к базе данных.
2. Драйвер, преобразующий обращения к методам JDBC в клиентский интерфейс конкретных СУБД. Использует программное обеспечение клиентской машины. Как и в предыдущем случае, на клиентскую машину приходится загружать дополнительное программное обеспечение, а именно, клиента СУБД. Например, для СУБД Oracle на клиентской машине следует инсталлировать Net8 и SQL Plus.
3. Драйвер, преобразующий обращения к методам JDBC в команды сетевого протокола, независимого от СУБД, например, HTTP. Такой драйвер целиком написан на языке Java. Сервер СУБД принимает команды сетевого протокола своими сетевыми средствами и выполняет их.
4. Драйвер Pure Java, преобразующий обращения к методам JDBC в команды сетевого протокола конкретной СУБД. При этом клиентская программа обращается непосредственно к серверу СУБД через его сетевые средства. Для Oracle такое "родное" сетевое средство — это Net8.

Драйверы типа 1 и 2, опирающиеся на промежуточные программы, не принадлежат полностью технологии Java и требуют загрузки дополнительного программного обеспечения. Поэтому они неудобны в эксплуатации. Драйверы типа 3 и 4 гораздо удобнее. Они полностью соответствуют технологии

Java и могут загружаться вместе с клиентскими приложениями Java, даже вместе с апплетами.

В стандартную поставку J2SE SDK входит мост JDBC-ODBC, реализованный классом `JdbcOdbcDriver` из пакета sun.jdbc.odbc. Другие драйверы JDBC поставляются вместе с СУБД или предлагаются независимыми фирмами. Например, в Oracle 8.1.7 входят следующие драйверы.

- Драйвер второго типа JDBC OCI 8.1.7, для которого на клиентской машине необходимо установить OCI (Oracle Call Interface) версии 8.1.7. Этот драйвер не может быть загружен по сети, его надо специально устанавливать на клиентской машине. По этой причине он не может использоваться апплетами. Драйвер дает наибольшую производительность и удобен для использования серверами приложений в сервлетах и EJB.
- Драйвер четвертого типа JDBC Thin 8.1.7. Он называется "тонким" (thin), потому что очень компактен и может загружаться вместе с апплетом. Связывается с сервером базы данных через создаваемые им сокеты потокового типа.
- Драйвер четвертого типа JDBC Thin Server-side 8.1.7, используемый в промежуточном слое (middleware) многослойного приложения;
- Драйвер четвертого типа JDBC Kprb 8.1.7, используемый хранимыми процедурами Oracle. Он загружается на сервере Oracle, прямо связан с исполняющей системой SQL Engine, и используется сервером для выполнения хранимых процедур Java, EJB и других программ Java.

Для использования этих драйверов надо к значению переменной `PATH` добавить путь к библиотеке классов JDBC, обычно это $ORACLE_HOME/jdbc/lib, а к значению переменной `CLASSPATH` добавить пути к сжатым файлам классов: $ORACLE_HOME/jdbc/lib/classes12.zip и $ORACLE_HOME/jdbc/lib/nls_charset12.zip.

Загрузка драйверов JDBC

Каждый драйвер JDBC должен реализовать методы интерфейса `Driver` из пакета java.sql. Этот интерфейс описывает несколько информационных методов, примеры использования которых можно посмотреть в листинге 11.1, и метод соединения с базой данных

```
public Connection connect(String url, Properties info);
```

Метод возвращает `null`, если соединение с базой по заданному адресу `url` не получилось, и объект, реализующий интерфейс `Connection`, если соединение установлено. Параметр `info` содержит сведения, нужные серверу базы данных для соединения, например, имя и пароль пользователя. Этот метод не используется напрямую, к нему обращается класс `DriverManager`.

Приложение может загрузить несколько драйверов JDBC. Для управления ими используются статические методы класса `DriverManager`. Этот класс во время инициализации читает системную переменную `jdbc.drivers`, если она определена, и загружает драйверы, перечисленные в ней. Драйверы в системной переменной должны разделяться двоеточием. Например:

```
java -Djdbc.drivers =
   sun.jdbc.odbc.JdbcOdbcDriver:oracle.jdbc.driver.OracleDriver
     MyJDBCAppl
```

Можно задать загрузку драйверов явно, используя статический метод `registerDriver(Driver)`, или конструктор класса драйвера, либо метод `Class.forName(String)`. Последние два способа возможны потому, что драйвер JDBC по требованиям спецификации JDBC должен уметь загружать себя и регистрироваться в классе `DriverManager`. Как правило, для этого он обращается к методу `registerDriver(Driver)` и какое-либо специальное обращение к этому методу становится излишним. Два способа загрузки драйверов даны в листинге 11.1.

Листинг 11.1 Загрузка драйверов JDBC и их свойства

```
import java.sql.*;
import java.util.*;

class DriverApp{
  public static void main(String args[]){
    try{
      DriverManager.registerDriver(
        new oracle.jdbc.driver.OracleDriver());

      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

      Enumeration drs = DriverManager.getDrivers();

      System.out.println("Загруженные драйверы:");

      while(drs.hasMoreElements()){

        Driver dr=(Driver)drs.nextElement();

        System.out.println("Драйвер: " +
           dr.getClass().getName());
```

```
        System.out.println("Версия: " +
           dr.getMajorVersion());
        System.out.println("Модификация: " +
           dr.getMinorVersion());
        System.out.println("Тестирован: " +
           dr.jdbcCompliant());

        DriverPropertyInfo props[] =
           dr.getPropertyInfo("", null);

        if(props != null){
          System.out.println("Свойства: ");
          for(int i = 0; i < props.length; i++){
            System.out.println("Имя: " + props[i].name);
            System.out.println("Описание: " +
               props[i].description);
            System.out.println("Значение: " + props[i].value);

            if(props[i].choices != null){
              System.out.println("Значения: ");
              for(int j = 0; j < props[i].choices.length; j++)
                    System.out.println(" " + props[i].choices[j]);
            }
            System.out.println("Требуется: "+props[i].required);
          }
        }
      }
    }catch(Exception e){
        System.err.println(e);
        System.exit(0);
    }
  }
}
```

В листинге 11.1 делается попытка загрузить драйверы Oracle JDBC, реализованные классом `OracleDriver` из пакета oracle.jdbc.driver, входящего в поставку Oracle Server, и мост ODBC-JDBC, реализованный классом `JdbcOdbcDriver`. Загруженные драйверы регистрируются классом `DriverManager`. Затем просматривается список зарегистрированных драйверов, полученный методом `getDrivers()`, и выводятся их свойства.

При соединении с базой данных `DriverManager` автоматически подбирает подходящий драйвер из всех зарегистрированных у него драйверов JDBC, используя метод `acceptsURL(String)`, описанный в интерфейсе `Driver`.

Соединение с базой данных

Соединение с базой данных выполняется одним из следующих статических методов класса `DriverManager`:

- `public static Connection getConnection(String url);`
- `public static Connection getConnection(String url, String name, String password);`
- `public static Connection getConnection(String url, Properties info);`

Адрес базы, задаваемый параметром `url`, имеет специфический вид:

```
jdbc:<subprotocol>:<subname>
```

Здесь `<subprotocol>` — условное имя драйвера или способа соединения. Например, мост JDBC-ODBC обозначается именем `odbc`, драйверы СУБД Oracle носят имя `oracle`, драйверы Sybase обозначаются словом `sybase`, драйверы IBM DB2 — `db2`. Каждый производитель драйверов JDBC должен зарегистрировать условное имя по адресу **jdbc@eng.sun.com**.

От `<subprotocol>` зависит вид третьей части адреса — `<subname>`. Он определяется разработчиками драйвера. В этой части уточняется имя драйвера, задается адрес базы, чаще всего в виде //host:port/database, и передаются некоторые параметры соединения. Если `<subprotocol>` — это `odbc`, т. е. мост JDBC-ODBC, то `<subname>` определяется по правилам ODBC. Вот несколько примеров:

```
String url = "jdbc:odbc:SDO";
String url = "jdbc:odbc:SDO;UID=sdoadmin;PWD=ftE2dF";
String url = "jdbc:oracle:thin:@localhost:1521:SDO";
String url = "jdbc:oracle:oci8:scott/tiger@homexp";
String url = "jdbc:oracle:kprb:";
String url =
   "jdbc:sybase://db.utah.com:8192/aircom;user=guest;password=guest";
String url = "jdbc:merant:db2://somehost.com:50000/somebase";
String url = "jdbc:db2://somehost.com:50000/somebase";
```

Теперь соединение можно установить так:

```
Connection con = DriverManager.getConnection(url);
```

или так:

```
Connection con = DriverManager.getConnection(url, "scott", "tiger");
```

Можно воспользоваться и третьим методом:

```
Properties info = new Properties();
info.put("name", "scott");
info.put("password", "tiger");
Connection con = DriverManager.getConnection(url, info);
```

Использование JNDI для соединения

Вместо того чтобы прямо задавать адрес базы данных при установлении соединения, можно зарегистрировать адрес в распределенной системе именования JNDI (Java Naming and Directory Interface), а затем применить интерфейс `DataSource` из пакета javax.sql. Краткое описание системы JNDI приведено в *главе 1*, подробное описание — в *главе 17*.

Включение адреса базы в распределенную систему имен JNDI дает больше возможностей, чем прямое соединение. Оно позволяет создавать пул соединений и организовывать распределенные транзакции. *Источником данных* может быть не только база данных, но и электронная таблица, и файл, информация в котором хранится в виде таблицы, и что-то иное. Надо лишь, чтобы источник данных реализовал интерфейс `DataSource`.

Разумеется, для применения этого способа нужно, чтобы и драйвер JDBC реализовал интерфейс `DataSource`.

Интерфейс *DataSource*

Интерфейс `DataSource`, находящийся в пакете javax.sql, описывает два метода соединения:

```
public Connection getConnection();
public Connection getConnection(String user, String password);
```

Первый метод задает соединение по умолчанию, второй — соединение пользователя `user`, имеющего пароль `password`.

Метод

```
public void setLoginTimeout(int seconds);
```

задает максимальный интервал `seconds` между попытками соединения в секундах. Нулевое значение аргумента означает системную установку интервала.

Метод

```
public void setLogWriter(PrintWriter pw);
```

задает вывод сообщений всех методов, пользующихся этим объектом типа `DataSource`, в заданный символьный выходной поток `pw`. По умолчанию поток пуст — `null`.

Еще два информационных метода:

```
public int getLoginTimeout();
public PrintWriter getLogWriter();
```

возвращают установленный интервал между попытками соединения и заданный выходной поток для сообщений.

Каждая реализация интерфейса `DataSource` пишется для конкретного источника данных и добавляет к этим методам свойства, специфичные для источника данных, например, IP-адрес или DNS-имя источника данных, имя и пароль пользователя. Эти сведения оформляются как свойства JavaBean со своими методами `setXxx()`, `getXxx()`.

Например, реализация `OracleDataSource` фирмы Oracle Corporation содержит методы `setDatabaseName(String)`, `setDataSourceName(String)`, `setDescription(String)`, `setPassword(String)`, `setDriverType(String)`, `setNetworkProtocol(String)`, `setPortNumber(int)`, `setServerName(String)`, `setTNSEntryName(String)` и соответствующие методы `getXxx()`.

Работа с источником данных

Работа с источником данных начинается с создания экземпляра источника данных:

```
OracleDataSource ds = new OracleDataSource();
```

Здесь использована реализация интерфейса `DataSource` из стандартной поставки СУБД Oracle 8.1.7.

Потом задаются свойства источника данных, определяющие его местоположение, имя и другие сведения:

```
ds.setServerName("parus");
ds.setPortNumber(1521);
ds.setDatabaseName("SDO");
ds.setDescription("Система дистанционного обучения");
```

Регистрируем источник данных в распределенной системе именования JNDI в начальном контексте `jdbc` под именем `SDO`:

```
InitialContext ct = new InitialContext();
ct.bind("jdbc/SDO", ds);
ct.close();
```

Все это обычно делается на сервере или в промежуточном слое распределенного приложения. Теперь источник данных зарегистрирован в системе

именования JNDI под именем "jdbc/SDO". Любое приложение может обратиться к распределенной системе именования и связаться с источником данных по его имени.

Для этого, после регистрации, на любом компьютере, включенном в систему JNDI, находим источник данных:

```
InitialContext ct = new InitialContext();
DataSource ds = (DataSource)ct.lookup("jdbc/SDO");
```

и устанавливаем соединение с источником данных:

```
Connection con = ds.getConnection();
```

или

```
Connection con = ds.getConnection("sdoadmin", "sdoadmin");
```

В любой момент времени можно заменить источник данных, зарегистрированный под именем "jdbc/SDO", ничего не меняя в приложениях, использующих это имя. Пусть, например, мы решили заменить базу данных SDO другой базой NSDO, находящейся на том же сервере. Для этого достаточно перерегистрировать источник данных под тем же именем:

```
OracleDataSource ds = new OracleDataSource();


ds.setServerName("parus");
ds.setPortNumber(1521);
ds.setDatabaseName("NSDO");
ds.setDescription("Новая система дистанционного обучения");


InitialContext ct = new InitialContext();


ct.rebind("jdbc/SDO", ds);
ct.close();
```

Клиентское приложение продолжает связываться с базой данных по имени "jdbc/SDO", не подозревая, что база изменилась.

Когда надобность в источнике данных отпадет, его можно удалить из системы JNDI:

```
InitialContext ct = new InitialContext();


ct.unbind("jdbc/SDO", ds);
ct.close();
```

Пул соединений

Обычная реализация интерфейса `DataSource` устанавливает только одно соединение с источником данных, начинающее клиентскую сессию, с помощью объекта, реализующего интерфейс `Connection`. Когда этот объект выполняет свой метод `close()` или приложение завершает работу, сессия прекращается, а вместе с тем разрывается соединение с источником данных. Следующая сессия снова должна устанавливать физическое соединение, что приводит к неоправданным накладным расходам.

Накладные расходы можно уменьшить, организовав *пул физических соединений*, используемый несколькими логическими соединениями. Логическое соединение при своем создании забирает одно физическое соединение из пула, а по окончании работы возвращает его в пул. Физическое соединение сохраняется в пуле до своего закрытия или закрытия всех логических соединений.

Методы работы с пулом физических соединений описаны интерфейсом `ConnectionPoolDataSource`. Они устанавливают максимальное время ожидания соединения в секундах `setLoginTimeout(int sec)`, выходной поток для записи протокола в журнальный файл `setLogWriter(PrintWriter)`. Два основных метода этого интерфейса

```
public PooledConnection getPooledConnection();
public PooledConnection getPooledConnection(String name,
    String password);
```

пытаются установить физическое соединение с источником данных, пригодное для помещения в пул соединений, и, в случае удачи, возвращают объект, описанный интерфейсом `PooledConnection`. Этот объект помещается в пул и обеспечивает физическое соединение с источником данных. Он предоставляет приложениям возможность логического соединения с пулом своим методом

```
public Connection getConnection();
```

Как правило, интерфейс `ConnectionPoolDataSource` реализуется драйвером JDBC, используемым сервером приложений. Когда клиентское приложение устанавливает соединение методом `getConnection()` класса, реализующего `DataSource`, сервер приложений обращается к одноименному методу реализации интерфейса `ConnectionPoolDataSource` и предоставляет приложению соединение из пула физических соединений. Если пул пуст, то сервер приложений обращается к методу `getPooledConnection()` и драйвер JDBC устанавливает новое физическое соединение типа `PooledConnection`. Оно тут же выполняет свой метод `getConnection()` и возвращает клиенту только что созданное соединение.

Все это происходит внутри драйвера JDBC. Для сервера приложений все делается почти так же, как описано в предыдущем пункте.

```
OracleConnectionPoolDataSource pds =
            new OracleConnectionPoolDataSource();
pds.setServerName("parus");
pds.setPortNumber(1521);
pds.setDatabaseName("SDO");
pds.setDescription("Система дистанционного обучения");
```

Здесь использована реализация `OracleConnectionPoolDataSource` интерфейса `ConnectionPoolDataSource`, выполненная фирмой Oracle Corporation.

Регистрируем созданный источник данных `pds` в начальном контексте `jdbc` под именем `SDO`:

```
InitialContext ct = new InitialContext();
ct.bind("jdbc/SDO", pds);
```

Создаем источник данных для логических соединений:

```
OracleDataSource ds = new OracleDataSource();
ds.setDataSourceName("jdbc/SDO");
ct.bind("jdbc/SDO", ds);
```

Все эти действия может выполнять и сервер приложений, и сам драйвер JDBC, реализующий пул соединений. И в том и в другом случае клиент совершенно не замечает наличия или отсутствия пула соединений. Клиентская программа создает объект типа `DataSource` и связывает его с зарегистрированным источником данных точно так же, как описано в предыдущем пункте.

```
OracleDataSource ds = new OracleDataSource();
InitialContext ct = new InitialContext();
ct.bind("jdbc/SDO", ds);
```

Затем клиентская программа просто пользуется методом `getConnection()` объекта `ds`, реализующего интерфейс `DataSource`.

```
Connection con = ds.getConnection();
```

или

```
Connection con = ds.getConnection("sdoadmin", "sdoadmin");
```

После создания объекта типа `ConnectionPoolDataSource` можно определить его свойства методами `setMaxStatements(int)`, `setInitialPoolSize(int)`, `setMinPoolSize(int)`, `setMaxPoolSize(int)` и другими методами. Такие методы создаются разработчиками драйверов JDBC.

Когда приложение разрывает соединение методом `close()`, происходит событие класса `ConnectionEvent`. Сервер приложений или драйвер JDBC про-

слушивают это событие, реализуя интерфейс `ConnectionEventListener`. В методе `connectionClosed(ConnectionEvent)` этого интерфейса они возвращают физическое соединение `PooledConnection` в пул соединений.

Физическое соединение существует все время работы драйвера. Только когда будет выполнен метод `close()` физического соединения `PooledConnection`, оно перестает существовать.

Соединение для распределенной транзакции

Драйвер JDBC можно разработать так, чтобы соединения, устанавливаемые им, могли участвовать в распределенных транзакциях. Для этого драйвер должен реализовать интерфейс `XADataSource`. Этот интерфейс организован подобно интерфейсу `ConnectionPoolDataSource`. Он тоже описывает два метода соединения

```
public XAConnection getXAConnection();
public XAConnection getXAConnection(String name, String password);
```

Методы пытаются установить физическое соединение с источником данных и, в случае удачи, возвращают объект, реализующий интерфейс `XAConnection`. Этот интерфейс расширяет интерфейс `PooledConnection`, значит, создает пул физических соединений и обладает методом `getConnection()` для логических соединений.

Интерфейс `XAConnection` описывает один метод

```
public XAResource getXAResource();
```

связывающий соединение с ресурсами распределенной транзакции. Он создает объект, реализующий интерфейс `XAResource` из пакета javax.transaction.xa. Этот объект затем используется диспетчером транзакций (transaction manager) для включения соединения в распределенную транзакцию. Подробнее он будет описан в *главе 24*.

Драйвер JDBC реализует все эти интерфейсы, и сам управляет соединением. С точки зрения сервера приложений соединение устанавливается точно так же, как описано в предыдущих пунктах.

```
OracleXADataSource xads = new OracleXADataSource();
xads.setServerName("parus");
xads.setPortNumber(1521);
xads.setDatabaseName("SDO");
xads.setDescription("Система дистанционного обучения");
InitialContext ct = new InitialContext();
ct.bind("jdbc/SDO", xads);
```

Здесь использована реализация `OracleXADataSource` интерфейса `XADataSource`, выполненная фирмой Oracle Corporation.

```
OracleDataSource ds = new OracleDataSource();
ds.setDataSourceName("jdbc/SDO");
InitialContext ct = new InitialContext();
ct.bind("jdbc/SDO", ds);
```

Для клиентской программы ничего не меняется:

```
OracleDataSource ds = new OracleDataSource();
InitialContext ct = new InitialContext();
ct.bind("jdbc/SDO", ds);
Connection con = ds.getConnection();
```

или

```
Connection con = ds.getConnection("scott", "tiger");
```

Соединение *Connection*

Одно приложение может соединиться с несколькими источниками данных или установить несколько соединений с одним и тем же источником данных. С точки зрения драйвера JDBC каждое *соединение* — это пользовательская сессия. Соединение содержит идентификатор сессии, набор операторов SQL, выполняемых в течение сессии, выборки, сделанные этими операторами.

Управление транзакцией

По умолчанию после выполнения каждого оператора SQL транзакция завершается (commit). Соединение, не участвующее в распределенной транзакции, может изменить это правило методом `setAutoCommit(false)`. Этот метод, кроме отмены автоматического завершения транзакции, сразу же завершает текущую транзакцию. Дальнейшие транзакции следует завершать явно методом `commit()` или выполнять их откат (rollback) методом `rollback()`.

Откат можно сделать на определенную *точку сохранения* (savepoint) методом `rollback(Savepoint)`. Точка сохранения — это объект, реализующий интерфейс `Savepoint`. Она устанавливается методом `setSavepoint()` или методом `setSavepoint(String)`. Аргумент второго метода задает имя для точки сохранения. Оба метода, кроме того, создают объект типа `Savepoint`. Например:

```
Connection con = ds.getConnection();
Statement st = con.createStatement();
int n = st.executeUpdate("INSERT INTO tbl1 VALUES(5, 'Иванов')");
```

```
Savepoint sp1 = con.setSavepoint("Точка 1");
int m = st.executeUpdate("INSERT INTO tbl1 VALUES(6, 'Петров')");
Savepoint sp2 = con.setSavepoint();
//   . . . . . .
con.rollback(sp1);
```

После этого первая вставка `INSERT` сохраняется, а вторая отменяется.

После отката точки сохранения отменяются. Отменить точку сохранения можно и явно, методом `releaseSavepoint(Savepoint)`.

Если соединение участвует в распределенной транзакции, то управлять транзакцией методами `commit()` и `rollback()` нельзя. При попытке их применения выбрасывается исключение класса `SQLException`.

Соединение может запросить транзакцию, предназначенную только для чтения, методом `setReadOnly(true)`. Это указание передается серверу базы данных, который может оптимизировать выполнение такой транзакции и увеличить производительность соединения.

Уровни изоляции

Соединение способно запросить установку уровня изоляции методом `setIsolationLevel(int)`, параметр которого принимает одно из значений стандарта SQL:

- `TRANSACTION_READ_UNCOMMITTED`;
- `TRANSACTION_READ_COMMITTED`;
- `TRANSACTION_REPEATABLE_READ`;
- `TRANSACTION_SERIALIZABLE`.

Эти константы определены в интерфейсе `Connection`. Сервер базы данных может учесть это пожелание, если он обеспечивает нужный уровень изоляции.

Прочие методы

Драйвер JDBC всегда преобразует оператор SQL, записанный в виде, принятом для этого драйвера, в форму, используемую в базе данных, с которой установлено соединение. Метод

```
public String nativeSQL(String sql);
```

позволяет посмотреть преобразованный оператор `sql`. Это только информационный метод, его не надо применять специально.

Завершить соединение можно и нужно методом `close()`.

Основу же интерфейса `Connection` образуют многочисленные методы, создающие объекты типов `XxxStatement`, о которых мы поговорим в следующей главе.

Еще один интересный метод `getMetaData()` возвращает множество сведений об источнике данных в виде объекта, реализующего интерфейс `DatabaseMetaData`. Рассмотрим его подробнее.

Свойства базы данных

Итак, следующая строка

```
DatabaseMetaData meta = con.getMetaData();
```

создает объект `meta`, содержащий обширные сведения о базе данных, с которой установлено соединение.

Интерфейс `DatabaseMetaData` описывает почти две сотни информационных методов, выдающих адрес, название и версию базы данных, типы данных, хранящихся в базе, набор допустимых операторов SQL, предельный объем таблиц, набор функций, имеющихся в базе, и т. п.

Большинство этих методов проверяют то или иное свойство базы и возвращают логические значения `true` или `false`. Например, метод `supportsSavepoints()` позволяет узнать, можно ли расставлять точки сохранения в транзакции. Метод `isReadOnly()` позволяет проверить, позволяет ли данное соединение изменить данные, хранящиеся в базе.

Другие методы определяют числовые характеристики базы, например, метод `getMaxRowSize()` возвращает максимально допустимую длину строки в таблицах базы. Метод `getMaxColumnNameLength()` выдает наибольшую длину имени столбца. Ноль означает, что ограничения нет или оно неизвестно.

Некоторые методы возвращают строку символов, примеры таких методов приведены в листинге 11.2. Есть и методы, возвращающие целую выборку типа `ResultSet`.

Драйвер JDBC считает, что все таблицы базы находятся в *каталогах* (catalog). Метод `getCatalogs()` дает список каталогов базы в виде объекта типа `ResultSet`. Каталоги разбиты на *схемы* (schema), их список можно получить методом `getSchemas()` как это сделано в листинге 11.2. В схемах хранятся таблицы. Чтобы обратиться к таблице, надо указать ее каталог и схему. Например, метод:

```
public ResultSet getPrimaryKeys(String catalog,
   String schema, String table);
```

возвращает перечень первичных ключей таблицы `table`, находящейся в схеме `schema` каталога `catalog`.

Для того чтобы указать текущий каталог или схему, надо задать пустую строку "". Если в базе нет понятия каталога или схемы, следует указывать значение null, как это сделано в листинге 11.2. Определить, использует ли база понятие каталога, можно несколькими логическими методами supportsCatalogsInXxx(). Несколько логических методов supportsSchemasInXxx() возвращают подобные сведения о схеме.

Драйвер JDBC различает несколько типов таблиц, задаваемых следующими константами: TABLE, VIEW, SYSTEM TABLE, GLOBAL TEMPORARY, LOCAL TEMPORARY, ALIAS, SYNONYM. Узнать, какие из этих типов определены в базе данных, можно методом getTableTypes(), как показано в листинге 11.2. Например, в СУБД Oracle 8.1.7 определены типы TABLE, VIEW и SYNONYM.

С помощью методов интерфейса DatabaseMetaData можно настроить приложение так, чтобы оно полностью учитывало свойства присоединенной базы данных.

Листинг 11.2. Получение сведений о базе данных

```
import java.sql.*;
import oracle.jdbc.driver.*;

public class MetaDataApp{
  public static void main(String args[]){
    try{
       OracleDriver od = new OracleDriver();

       Connection con = DriverManager.getConnection(
          "jdbc:oracle:thin:@homexp:1521:SDO",
          "sdoadmin", "sdoadmin");

       DatabaseMetaData meta=con.getMetaData();

       System.out.println("Сервер: " +
          meta.getDatabaseProductName());

       System.out.println("Версия: " +
          meta.getDatabaseProductVersion());

       System.out.println("Пользователь: " +
          meta.getUserName());
```

```
        System.out.println("Драйвер JDBC: " +
           meta.getDriverName());

        System.out.println("Версия драйвера: " +
           meta.getDriverVersion());

        System.out.println("Процедура: " +
           meta.getProcedureTerm());

        System.out.println("Адрес: " +
           meta.getURL());

        ResultSet rs = meta.getSchemas();
        while (rs.next())
          System.out.println("Схема: "+ rs.getObject(1));

        rs = meta.getTableTypes();
        while (rs.next())
           System.out.println("Тип таблицы: "+ rs.getObject(1));

        con.close();

     }catch(Exception e){
        System.err.println(e);
        System.exit(0);
    }
   }
}
```

Глава 12

Оформление запроса

В предыдущей главе был описан процесс связи приложения с источником данных с помощью драйвера JDBC. Мы остановились на стадии получения соединения `con`, аккумулирующего сведения об установленной связи. Соединение устанавливается методами `getConnection()` класса `DriverManager`, например:

```
Connection con =
   DriverManager.getConnection("jdbc:oracle:thin:@homexp:1521:SDO");
```

или методами `getConnection()` источника данных `DataSource`, например:

```
Connection con = ds.getConnection();
```

Следующий шаг — получение объекта, реализующего интерфейс `Statement`.

Интерфейс *Statement*

Интерфейс `Statement` определяет несколько констант и описывает методы `executeXxx()`, формирующие запрос к серверу СУБД, а также информационные методы `getXxx()`, возвращающие характеристики созданного объекта и методы `setXxx()`, изменяющие эти характеристики.

Из последней группы методов наиболее полезны следующие методы:

- `public void setQueryTimeout(int secs)` — устанавливает время ожидания выполнения запроса `secs` в секундах. Ноль означает неограниченное время ожидания;
- `public void setMaxRows(int max)` — ограничивает количество строк в выборке числом `max`. Ноль означает неограниченное число строк;
- `public void setMaxfieldSize(int max)` — ограничивает длину столбца выборки символьных и двоичных значений числом `max` в байтах. Ноль означает, что ограничения нет;
- `public void setFetchDirection(int dir)` — определяет направление просмотра выборки. Аргумент метода `dir` — одна из констант `FETCH_FORWARD` (просмотр в прямом направлении) (по умолчанию), `FETCH_REVERSE` (просмотр в обратном направлении) или `FETCH_UNKNOWN` (направление просмотра не задано).

Реализация интерфейса `Statement` должна обеспечить весь цикл выполнения оператора SQL: формирование запроса, передачу его серверу СУБД, получение сведений о выполнении запроса и получение результата запроса, чаще всего в виде объекта, реализующего интерфейс `ResultSet`.

Все эти действия выполняются в рамках определенного соединения, поэтому объект, реализующий интерфейс `Statement`, создается методами, описанными интерфейсом `Connection`.

Получение экземпляра *Statement*

Экземпляр объекта типа `Statement` можно получить после соединения тремя методами. Основной метод имеет три аргумента:

```
public Statement createStatement(int type, int concur, int hold);
```

Первый аргумент метода — `type` — задает тип результата запроса — выборки. Это одна из констант, определенных в интерфейсе `ResultSet`:

- `TYPE_FORWARD_ONLY` — курсор выборки перемещается только вперед;
- `TYPE_SCROLL_INSENSITIVE` — курсор перемещается в обоих направлениях, во время работы выборка не изменяется;
- `TYPE_SCROLL_SENSITIVE` — курсор перемещается в обоих направлениях, выборка изменяется при изменении в базе выбранных строк и столбцов.

Узнать, реализует ли драйвер JDBC тот или иной тип выборки, можно логическим методом `supportsResultSetType(int)` интерфейса `DatabaseMetaData`, в который подставляется одна из перечисленных констант.

Второй аргумент — `concur` — задает режим изменения выборки. Это одна из двух констант:

- `CONCUR_READ_ONLY` — выборку нельзя изменять;
- `CONCUR_UPDATABLE` — выборку можно изменять.

Узнать, используется ли тот или иной режим драйвером JDBC, можно логическим методом `supportsResultSetConcurrency(int)` интерфейса `DatabaseMetaData`, аргумент которого — одна из этих двух констант.

Третий аргумент — `hold` — определяет состояние выборки после завершения транзакции:

- `HOLDS_CURSORS_OVER_COMMIT` — выборка остается открытой после завершения транзакции;
- `CLOSE_CURSORS_AT_COMMIT` — выборка закрывается после завершения транзакции.

Опять-таки, узнать о наличии того или иного режима в драйвере JDBC можно логическим методом `getResultSetHoldability(int)` интерфейса `DatabaseMetaData`.

Второй метод

```
public Statement createStatement(int type, int concur);
```

закрывает выборку после завершения транзакции.

Третий метод

```
Statement st = con.createStatement();
```

определяет простейший тип выборки: курсор перемещается только вперед, при совместной работе нельзя изменять выборку, после завершения транзакции выборка закрывается.

Формирование и выполнение запроса

После того как объект типа `Statement` сформирован и получен, можно обратиться с запросом к серверу СУБД. Простейший метод для этого

```
public ResultSet executeQuery(String sql);
```

Аргумент метода `sql` — это оператор SQL, чаще всего `SELECT`, записанный в виде строки символов. Например:

```
ResultSet rs = st.executeQuery("SELECT * FROM tbl1");
```

или

```
String sel = "SELECT имя, должность FROM сотрудники " +
   "WHERE год = 1948";
ResultSet rs = st.executeQuery(sel);
```

Метод преобразует аргумент к виду, принятому в присоединенной базе данных, посылает запрос серверу базы данных, дожидается ответа и создает объект, реализующий интерфейс `ResultSet`, если ответ получен. Если выполнить запрос не удалось, возвращается `null`. Если аргумент содержит ошибки, то выбрасывается исключение класса `SQLException`.

В листинге 12.1 приведен пример типичного соединения с базой данных и получения выборки, которая записывается в таблицу класса `JTable`. На рис. 12.1 показан результат выполнения этой программы.

Листинг 12.1. Получение выборки из базы данных

```
import java.sql.*;
import java.awt.*;
import java.awt.event.*;
```

```
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;

public class JdbcSelGui extends JFrame{
   Connection con;
   JTextField addr = new JTextField("jdbc:oracle:thin:@homexp:1521:SDO");
   JTextField name = new JTextField("sdoadmin");
   JPasswordField pass = new JPasswordField("sdoadmin");
   JTextField tf = new JTextField("SELECT name, address FROM students");
   JLabel sf = new JLabel("Строка состояния");
   JButton b = new JButton("Выполнить");
   JTable table;
   DBTableModel model;

   public JdbcSelGui(){

      super(" Выборка");
      Container c = getContentPane();

      JPanel p1 = new JPanel(false);
      p1.setLayout(new BoxLayout(p1, BoxLayout.X_AXIS));
      c.add(p1, BorderLayout.NORTH);

      JPanel p3 = new JPanel(false);
      p3.setLayout(new GridLayout(0, 1));
      p1.add(p3);

      p3.add(new JLabel("Адрес ", JLabel.RIGHT));
      p3.add(new JLabel("Имя ", JLabel.RIGHT));
      p3.add(new JLabel("Пароль ", JLabel.RIGHT));
      p3.add(new JLabel("Запрос ", JLabel.RIGHT));
      p3.add(new JPanel());

      JPanel p4 = new JPanel(false);
      p4.setLayout(new GridLayout(0, 1));
      p1.add(p4);
```

```
 p4.add(addr);
 p4.add(name);
 p4.add(pass);
 p4.add(tf);

 String[] defCols = {"Имя", "Адрес"};
 Object[] defData = {"", ""};

 model = new DBTableModel(defCols, defData, 0);
 table = new JTable(model);

 table.setRowHeight(20);
 table.setToolTipText("Содержимое выборки");

 sf.setToolTipText("Строка состояния");
 sf.setBackground(Color.lightGray);
 sf.setPreferredSize(new Dimension(300, 29));

 tf.setToolTipText(" Введите сюда запрос");
 tf.setPreferredSize(new Dimension(100, 29));

 b.setText("Выполнить");
 b.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent evt){
       connect(addr.getText(), name.getText(),
                 pass.getPassword());
       selectRecords(tf.getText());
    }
 });

JScrollPane sp = new JScrollPane(table);

JPanel p = new JPanel();
p.add(b);
p4.add(p);

c.add(sp);
c.add(sf, BorderLayout.SOUTH);
```

```
    setSize(400, 300);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

 public void connect(String addr, String name, char[] pass){
   try{
      sf.setText("Идет соединение...");

      DriverManager.registerDriver(
          new oracle.jdbc.driver.OracleDriver());

      String s = new String(pass);
      con = DriverManager.getConnection(addr, name, s);
      con.setAutoCommit(false);

      sf.setText("Успешное соединение");

  }catch(Exception e){
      sf.setText("Ошибка соединения... " + e);
  }
 }

 public void selectRecords(String query){
    try{
       Statement st = con.createStatement();
       ResultSet rs = st.executeQuery(query);

       model.clearTable();
       model.insertFromDB(rs);

       rs.close();
       st.close();

    }catch(SQLException e){
       sf.setText("Ошибка выборки... " + e);
    }
  }
```

```
   public static void main(String[] args){
      new JdbcSelGui();
   }
}

public class DBTableModel extends AbstractTableModel{

  int cols;
  Vector data;
  String[] colNames;

  public DBTableModel(String[] colNames, Object[] values, int rows){

    cols = colNames.length;

    this.colNames = new String[cols];
    for (int i = 0; i < cols; i++)
       this.colNames[i] = colNames[i];

    data = new Vector();
    for (int i=0; i < rows; i++){
       Vector v = new Vector();
       for (int j=0; j < cols; j++)
          v.addElement(values[j]);
       data.addElement(v);
    }
  }

  public Object getValueAt(int row, int col){
    Vector v = (Vector)data.elementAt(row);
    return v.elementAt(col);
  }

  public int getRowCount(){
    return data.size();
  }

  public int getColumnCount(){
    return cols;
  }
```

```
public void setValueAt(Object obj, int row, int col){
  Vector v = (Vector)data.elementAt(row);
  v.setElementAt(obj, col);
}

public void insertRow(Vector row){
  data.addElement(row);
  super.fireTableDataChanged();
}

public void deleteRow(int row){
  data.removeElementAt(row);
  super.fireTableDataChanged();
}

public Vector getRow(int row){
  return (Vector)data.elementAt(row);
}

public void updateRow(Vector updatedRow, int row){
  data.setElementAt(updatedRow, row);
  super.fireTableDataChanged();
}

public void insertFromDB(ResultSet rs){

  try{
    ResultSetMetaData rsmd = rs.getMetaData();
    cols = rsmd.getColumnCount();

    colNames = new String[cols];
    for (int i = 0; i < cols; i++)
       colNames[i] = rsmd.getColumnName(i+1);

    data = new Vector();

    while (rs.next()){
       Vector col = new Vector();
       for (int i = 1; i <= cols; i++)
          col.addElement(rs.getObject(i));
```

```
        data.addElement(col);
      }

      rs.close();

    }catch (SQLException e){
        e.printStackTrace();
    }

    fireTableChanged(null);
  }

  public String getColumnName(int col){
    return colNames[col];
  }

  public Class getColumnClass(int col){
    return getValueAt(0, col).getClass();
  }

  public void clearTable(){
    cols = 0;
    data = new Vector();
    colNames = null;
    super.fireTableDataChanged();
  }
}
```

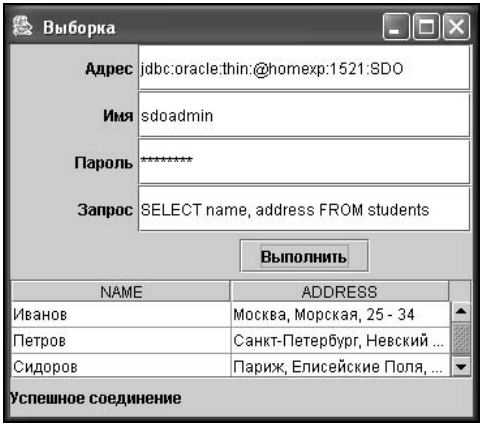


Рис. 12.1. Выполнение запроса

Изменение таблиц базы

Операторы SQL, изменяющие таблицы: INSERT, UPDATE и DELETE, не возвращают выборку. Они возвращают количество обработанных строк таблицы. Чтобы учесть это, для выполнения операторов, изменяющих таблицы базы, применяется метод

```
public int executeUpdate(String sql);
```

возвращающий число обработанных строк. Например:

```
int n = st.executeUpdate("INSERT INTO сотрудники " +
    "VALUES(189342, 'Петров', 1968, 'Программист')");
int n = st.executeUpdate("UPDATE сотрудники " +
    "SET должность = 'Менеджер' WHERE таб_N = 123456");
int n = st.executeUpdate("DELETE FROM сотрудники " +
    "WHERE таб_N = 456321");
```

Этот же метод можно применить для выполнения операторов DDL, не возвращающих значения. В таком случае метод возвращает нуль. В листинге 12.2 приводится пример создания и заполнения таблицы средствами JDBC.

Листинг 12.2. Создание и заполнение таблицы

```
import java.sql.*;

public class Example1{
   public static void main(String args[]){
      Connection con;
      Statement st;
      ResultSet rs;

      try{
        Class.forName("oracle.jdbc.driver.OracleDriver");

        con = DriverManager.getConnection(
        "jdbc:oracle:thin:sdoadmin/sdoadmin@homexp:1521:SDO");

        st = con.createStatement();

        try{
           st.execute("DROP TABLE test_table");
        }catch(SQLException e){
```

```
          System.err.println("Caught drop table: " + e);
        }
        st.execute(
           "CREATE TABLE test_table (col1 CHAR(10), col2 INT)");
        System.out.println("Table created.");

        st.execute("INSERT INTO test_table VALUES ('value0', 0)");
        st.execute("INSERT INTO test_table VALUES ('value1', 1)");
        st.execute("INSERT INTO test_table VALUES ('value2', 2)");
        st.execute("INSERT INTO test_table VALUES (null,  null)");
        st.execute("INSERT INTO test_table VALUES ('value3', 3)");
        System.out.println("Rows inserted");

        con.commit();

        rs = st.executeQuery("SELECT col1, col2 FROM test_table");
        System.out.println("Query executed");

    }catch SQLException ex){
       System.err.println("SQLException: " + ex);
    }
  }
}
```

Общий случай

В некоторых случаях, например, при выполнении хранимой процедуры, заранее неизвестно, одну выборку возвратит запрос, несколько выборок или вообще не возвратит результат. В других случаях оператор SQL неизвестен заранее, например, при интерактивной работе. Для таких случаев разработан логический метод

```
public boolean execute(String sql);
```

Он выполняет запрос и возвращает `true`, если запрос ответил хотя бы одной выборкой типа `ResultSet`, и `false`, если запрос вернул число или не вернул ничего.

Если метод `execute()` возвратил `true`, значит, в объекте типа `Statement` содержится выборка. Получить ее можно методом

```
public ResultSet getResultSet();
```

Если метод `execute()` вернул `false`, то получить число можно методом

```
public int getUpdateCount();
```

который возвращает число обработанных строк, или −1, если в результате выполнения оператора SQL получился объект типа `ResultSet` или нет результата.

Эти методы можно применить несколько раз, последовательно получая выборки одну за другой до тех пор, пока один из методов не возвратит `null` или −1. Всякий раз для перехода к следующей выборке следует применять логический метод

```
public boolean getMoreResults(int current);
```

аргумент которого принимает одно из значений — констант интерфейса `Statement`:

- `CLOSE_CURRENT_RESULT` — закрыть текущую выборку;
- `KEEP_CURRENT_RESULT` — не закрывать текущую выборку;
- `CLOSE_ALL_RESULTS` — закрыть все выборки.

Этот метод выполняет действие, предписанное аргументом, и перемещается к следующей выборке. Метод возвращает `true`, если следующая выборка имеет тип `ResultSet`, и `false`, если тип `int` или вообще нет результата.

Если же перед перемещением надо всегда закрывать текущую выборку, то следует применить метод без аргумента

```
public boolean getMoreResults();
```

Листинг 12.3 содержит пример программы, обрабатывающей интерактивные запросы.

Листинг 12.3. Получение результатов запроса общего вида

```
import java.sql.*;
import oracle.jdbc.driver.*;

class StApp{
  public static void main(String[] args){
    if (args.length != 1){
      System.out.println("Usage: java StApp statement");
      System.exit(0);
    }
    try{
       OracleDriver od = new OracleDriver();
```

```
      Connection con = DriverManager.getConnection
           ("jdbc:oracle:thin:@homexp:1521:SDO",
            "sdoadmin", "sdoadmin");

      Statement st = con.createStatement();

      String sql=args[0];
      System.out.println(sql);

      boolean hasResults = st.execute(sql);

      if (hasResults)
        do{
           ResultSet rs = st.getResultSet();
           if (rs != null) displayResults(rs);
        }while (st.getMoreResults());

      st.close();
      con.close();

   }catch(Exception e){
      System.out.println(e);
      System.exit(0);
   }
}

static void displayResults(ResultSet rs)
                            throws SQLException{

   ResultSetMetaData rsmd = rs.getMetaData();

   int numCols=rsmd.getColumnCount();

   for (int i = 1; i < numCols; i++)
        System.out.print(rsmd.getColumnName(i) + "\t");
   System.out.println(rsmd.getColumnName(numCols));

   while (rs.next()){
      for (int i = 1; i < numCols; i++)
```

```
            System.out.print(rs.getString(i) + "\t");
        System.out.println(rs.getString(numCols).trim());
    }
  }
}
```

Предварительно откомпилированные запросы

Во многих случаях, особенно при вставке `INSERT`, возникает необходимость выполнить один и тот же запрос несколько раз, может быть, с разными параметрами. Большинство серверов баз данных в таких случаях компилируют запрос только один раз, подставляя в дальнейшем параметры в уже откомпилированную последовательность действий, выполняющую запрос. Драйверы JDBC используют эту возможность или сами сохраняют откомпилированный запрос. Действия, необходимые для этого, описаны интерфейсом `PreparedStatement`, расширяющем интерфейс `Statement`.

Объект, реализующий интерфейс `PreparedStatement`, создается для установленного соединения одним из трех методов, описанных в интерфейсе `Connection`. Основной метод

```
public PreparedStatement prepareStatement(String sql,
    int type, int concur, int hold);
```

создает объект, содержащий уже откомпилированный запрос `sql`. Значения остальных аргументов метода полностью совпадают со значениями, описанными в *разд. "Получение экземпляра Statement"*.

Два других метода

```
public PreparedStatement prepareStatement(String sql,
    int type, int concur);
public PreparedStatement prepareStatement(String sql);
```

задают пропущенные аргументы по умолчанию, как и при получении объекта типа `Statement`.

Параметры откомпилированного запроса

Интерфейс `ParameterMetaData` описывает константы и методы для получения информации об откомпилированном запросе. Получить экземпляр класса, реализующего этот интерфейс, можно методом

```
public ParameterMetaData getParameterMetaData();
```

описанным в интерфейсе `PreparedStatement`.

Метод `getParameterCount()` возвращает число параметров откомпилированного запроса.

Метод `getParameterType(int param)` возвращает тип параметра с номером `param` в виде константы класса `Types`, а метод `getParameterTypeName(int param)` возвращает имя типа в виде строки `String`. Параметры нумеруются, начиная с 1.

Для числовых параметров методы `getPrecision(int param)` и `getScale(int param)` возвращают разрядность и точность параметра, а логический метод `isSigned(int param)` позволяет определить, может ли числовой параметр быть отрицательным.

Метод `getParameterMode(int param)` возвращает модификатор типа аргумента `param` (модификатор на языке SQL записывается как IN, OUT, INOUT) в виде одной из констант `parameterModeIn`, `parameterModeOut`, `parameterModeInOut`. Если тип параметра неизвестен, то возвращается константа `parameterModeUnknown`.

Логический метод `isNullable(int param)` позволяет узнать, может ли значение параметра оставаться неопределенным.

Выполнение откомпилированного запроса

После создания откомпилированного запроса он выполняется одним из методов

```
public ResultSet executeQuery();
public int executeUpdate();
public boolean execute();
```

так же, как и в предыдущих случаях.

Перед выполнением запроса в него можно занести значения переменных параметров. Переменные параметры отмечаются при создании объекта типа `PreparedStatement` вопросительными знаками, например:

```
PreparedStatement pst = con.prepareStatement("INSERT INTO tbl1 " +
   VALUES(?, ?, ?)");
```

а определяются методами `setXxx()`, например:

```
pst.setInt(1, 25);
pst.setString(2, "Иванов");
pst.setDate(3, new Date());
int n = pst.executeUpdate();
```

У методов `setXxx()` два аргумента. Первый аргумент — это порядковый номер вопросительного знака в запросе — первый, второй, третий. Второй аргумент — значение переменного параметра запроса, обозначаемого этим во-

просительным знаком. В интерфейсе `PreparedStatement` описано около двадцати методов `setXxx()` для различных типов вставляемых значений. Кроме обычных типов Java, точно соответствующих стандартным типам SQL, можно вставить любой Java-объект `obj` методом

```
public void setObject(int index, Object obj);
```

Драйвер JDBC приводит объект к одному из типов, хранимых в базе, руководствуясь спецификацией JDBC. Например, если написать

```
Integer n = new Integer(555);
pst.setObject(1, n);
```

то значение `n` будет преобразовано к типу `INTEGER`, описанному в классе `Types`.

Можно задать SQL-тип явно методом

```
public void setObject(int index, Object obj, int sqlType);
```

последний аргумент которого, `sqlType`, задает тип SQL одной из констант класса `Types`. Если этот тип `NUMERIC` или `DECIMAL`, то последним аргументом `scale` метода

```
public void setObject(int index, Object obj, int sqlType, int scale);
```

можно задать количество цифр в дробной части вещественного числа.

Основную трудность вызывает передача больших объемов информации. Хотя методы `setBytes()` и `setString()` позволяют передать неограниченное число байтов и символов, многие СУБД принимают данные порциями. Чтобы учесть это, в интерфейсе `PreparedStatement` описаны методы

```
public void setAsciiStream(int index, InputStream x, int length);
public void setBinaryStream(int index, InputStream x, int length);
public void setCharacterStream(int index, Reader x, int length);
```

Последний аргумент этих методов, `length`, задает длину данных. Например:

```
File f = new File("ivanov.gif");
InputStream fis = new FileInputStream(f);
PreparedStatement pst = con.prepareStatement(
   "UPDATE сотрудники SET фото = ? WHERE таб_N = ?");
pst.setBinaryStream (1, fis, f.length());
pst.setInt(2, 184563);
pst.executeUpdate();
```

Еще один способ передачи в базу большого объема информации — воспользоваться SQL-типом `BLOB` или `CLOB`. Для реализации этих типов в пакете java.sql определены интерфейсы `Blob` и `Clob`. В них описаны методы, создающие входной и выходной потоки для передачи объекта. Методы

`setBlob(int, Blob)` и `setClob(int Clob)` используют эти интерфейсы. Мы поговорим о работе с типами `BLOB` и `CLOB` в *главе 14*.

Установленное одним из методов `setXxx()` значение повторяется во всех последующих запросах, если оно в дальнейшем не изменено повторным применением метода `setXxx()` или все переменные параметры не очищены методом `clearParameters()`.

Хранимые процедуры

Третий интерфейс — `CallableStatement` — описывает выполнение хранимых на сервере СУБД процедур. Он расширяет интерфейс `PreparedStatement` и содержит, кроме унаследованных методов `setXxx(int, Xxx)`, задающих входные параметры процедуры типа `IN`, соответствующие методы `getXxx(int)` и `getXxx(String)` для каждого типа данных, позволяющие получить выходные параметры процедуры типа `OUT`. Кроме того, интерфейс описывает методы `setXxx(String, Xxx)`, первый параметр которых задает не целочисленный порядковый номер параметра процедуры, а его имя в виде строки `String`. Это удобно в тех случаях, когда надо задавать только некоторые, идущие не по порядку параметры.

Пусть, например, на языке Java написана хранимая процедура

```
public class StudChange{
   public void chAddress(int id, String addr){
       String sql = "UPDATE students SET address = ? " +
                    "WHERE id = ?";
      try{
        Connection con = new OracleDriver().defaultConnection();
        PreparedStatement pst = con.prepareStatement(sql);
        pst.setString(1, addr);
        pst.setInt(2, id);
        pst.executeUpdate();
        pst.close();
        con.close();
      }catch (SQLException e){
        System.err.println(e.getMessage());
     }
  }
}
```

Заносим ее в базу данных:

```
loadjava -u sdoadmin/sdoadmin@homexp:1521:SDO
   -r -t StudChange.java
```

Регистрируем в словаре данных:

```
CREATE OR REPLACE PROCEDURE
ch_addr(stud_id NUMBER DEFAULT 1, stud_addr VARCHAR2)
AS LANGUAGE JAVA
NAME 'StudChange.chAddress(int, String)';
```

Создаем объект и вызываем его, указывая только второй параметр:

```
CallableStatement cst = con.prepareCall("{CALL ch_addr(?)}");
cst.setstring("stud_addr", "Морская, 2 — 15");
ResultSet rs = cst.executeQuery();
```

Регистрация выходных параметров

Каждый выходной параметр типа `OUT` или `INOUT`, значение которого надо получить, следует зарегистрировать методом

```
public void registerOutParameter(int index, int type);
```

первый аргумент которого `index` — порядковый номер параметра процедуры, второй аргумент `type` — SQL-тип параметра, определенный одной из констант класса `Types`. Если этот тип `NUMERIC` или `DECIMAL`, то для задания количества цифр в дробной части числа `scale` применяется метод

```
public void registerOutParameter(int index, int type, int scale);
```

Два аналогичных метода регистрируют выходной параметр по имени, а не по порядковому номеру:

```
public void registerOutParameter(String name, int type);
public void registerOutParameter(String name, int type, int scale);
```

Еще один метод регистрирует параметр ссылочного типа или типа, определенного пользователем `typeName`:

```
public void registerOutParameter(String name, int type, String typeName);
```

После регистрации можно получать значения выходных параметров методами `getXxx()`.

Выполнение хранимой процедуры

Объект типа `CallableStatement` создается одним из трех методов, описанных в интерфейсе `Connection` точно так же, как и объект типа `PreparedStatement`:

```
public CallableStatement prepareCall(String sql,
    int type, int concur, int hold);
```

```
public CallableStatement prepareCall(String sql,
    int type, int concur);
public CallableStatement prepareCall(String sql);
```

аргумент `sql` в этих методах, описывающий вызов процедуры или функции, записывается в фигурных скобках, например, один входной параметр типа `IN`:

```
CallableStatement cst = con.prepareCall("{CALL get_info(?)}");
cst.setInt(1, 555);
ResultSet rs = cst.executeQuery();
```

Выходной параметр типа `OUT`:

```
CallableStatement cst = con.prepareCall("{? = CALL get_ID(?)}");
cst.registerOutParameter(1, Types.INTEGER);
cst.setString(2, "Петров");
cst.execute();
int id = cst.getInt(1);
```

Параметр типа `INOUT`:

```
CallableStatement cst = con.prepareCall("{CALL proc(?)}");
cst.registerOutParameter(1, Types.INTEGER);
cst.setInt(555);
cst.execute();
int n = cst.getInt(1);
```

Escape-синтаксис

Все системы управления базами данных содержат набор скалярных функций, позволяющих получить сведения о пользователях и системе, дату, время, преобразующих хранимые значения из одного формата в другой и выполняющих другие стандартные действия. Синтаксис этих функций сильно отличается у разных баз. Драйвер JDBC позволяет обратиться к таким функциям единообразно с помощью так называемого *ESC-синтаксиса* (escape syntax). Вызов функции записывается в фигурных скобках с указанием `fn` после открывающей скобки, например:

```
{fn SIN(x)}
{fn UCASE("dfdf")}
{fn SUBSTRING("dfddf", 5.10)}
{fn NOW()}
```

Драйвер преобразует эту ESC-запись в запись, специфичную для присоединенной базы данных, и передает вызов серверу СУБД. Результат выполнения функции замещает содержимое, заключенное в фигурные скобки.

Получить набор функций, хранящихся в базе, можно методами `getNumericFunctions()`, `getStringFunctions()`, `getSystemFunctions()`, `getTimeDateFunctions()`, описанными в интерфейсе `DatabaseMetaData`. Они возвращают строку `String`, в которой через запятую перечислены имена функций.

Escape-синтаксис даты и времени

Аналогично определяется ESC-синтаксис для даты, времени и того и другого вместе:

```
{d 'yyyy-mm-dd'}
{t 'hh:mm:ss'}
{ts 'yyyy-mm-dd hh:mm:ss.f'}
```

Драйвер JDBC преобразует этот формат даты и времени в "родной" формат присоединенной базы данных.

Escape-синтаксис внешних соединений

Большие различия существуют в записи *внешних соединений таблиц* (outer joins) разными базами данных. Некоторые СУБД различают левые (`LEFT`) и правые (`RIGHT`) внешние соединения, другие только полные (`FULL`), третьи вообще не позволяют создавать внешние соединения. Сведения об этом можно получить логическими методами `supportsOuterJoins()`, `supportsFullOuterJoins()`, описанными в интерфейсе `DatabaseMataData`. Драйверы JDBC в этом случае тоже применяют Escape-синтаксис, который формально записывается так:

```
{oj <outer-join>}
```

где `<outer-join>` имеет вид

```
table {LEFT|RIGHT|FULL} OUTER JOIN {table | <outer-join>}
    ON search-condition
```

Escape-синтаксис экранирования

Спецсимволы, образующие шаблоны в операторе `LIKE`, такие как знак процента `%`, означающий любое количество любых символов, или знак подчеркивания `_`, означающий одиночный символ, иногда нужно понимать как обычные символы. В таком случае они чаще всего экранируются обратной наклонной чертой: `\%`, `\_`. Не все СУБД понимают обратную наклонную черту как символ экранирования, поэтому для нее или для другого символа, назначаемого для экранирования, следует сделать объявление

```
{escape '\'}
```

Сгенерированные ключи

Некоторые СУБД при каждой вставке записи автоматически генерируют для нее уникальный ключ. Узнать об этом можно логическим методом `supportsGetGeneratedKeys()`, описанным в интерфейсе `DatabaseMetaData`. Если же ключи генерируются, то получить их можно так:

```
DatabaseMetaData meta=con.getMetaData();
if (meta.supportsGetGeneratedKeys()){
   ResultSet rs = st.getGeneratedKeys();
   while (rs.next())
     System.out.println("name: " + rs.getObject(1));
}
```

Здесь `st` — это объект, выполнивший вставку `INSERT` одним из следующих методов:

```
public int executeUpdate(String sql, int auto);
public int executeUpdate(String sql, String[] colNames);
public int executeUpdate(String sql, int[] colIndexes);
```

Аргумент `auto` первого метода только указывает, нужно ли будет возвращать сгенерированные ключи. Его значением может быть одна из двух констант `RETURN_GENERATED_KEYS` (возвращать сгенерированные ключи) или `NO_GENERATED_KEYS` (не возвращать сгенерированные ключи) интерфейса `Statement`. Второй и третий методы указывают колонки вставленной строки, которые следует вернуть вместе с ключами.

Такие же варианты есть и у метода `execute()`.

Глава 13

Обработка результатов запроса

В предыдущей главе мы выяснили, что в двух случаях запрос возвращает в качестве результата экземпляр класса, реализующего интерфейс `ResultSet`. В первом случае такой экземпляр создается методом `executeQuery(String)` Во втором случае он создается методом `execute(String)`, но создание экземпляра происходит не всегда. Объект типа `ResultSet` возвращает метод `getResultSet()`, но сначала надо выполнить метод `getMoreResults()` и убедиться, что он вернул `true`.

Познакомимся подробнее с интерфейсом `ResultSet`.

Интерфейс *ResultSet*

Выборка — объект типа `ResultSet` — устроена как таблица. Она состоит из строк и столбцов, пронумерованных, начиная с 1.

Вместе с объектом типа `ResultSet` создается курсор, позволяющий просматривать полученные результаты запроса. Сначала курсор находится перед первой строкой результата. Логический метод `next()` перемещает его на первую строку. Каждое следующее применение метода `next()` перемещает курсор на следующую строку. В конце концов, курсор перемещается за последнюю строку, а метод возвращает `false`.

Описанное поведение курсора принято по умолчанию, так ведет себя курсор выборки типа `TYPE_FORWARD_ONLY`. К ней можно применить только метод `next()`. Обычное применение этого метода — последовательный просмотр выборки:

```
ResultSet rs = st.executeQuery("SELECT col1, col2 FROM tbl”);


while (rs.next()){
      // Получаем значение первого столбца текущей строки.
   int n = rs.getInt(1);
      // Получаем значение второго столбца текущей строки.
   String s = rs.getString(2);
      // Обрабатываем значения n и s...
```

```
}
      // Закрываем выборку.
rs.close();
```

Расширенный просмотр выборки

Если тип выборки не `TYPE_FORWARD_ONLY`, а `TYPE_SCROLL_SENSITIVE` или `TYPE_SCROLL_INSENSITIVE`, о которых говорилось в *главе 12*, то у курсора появляются новые возможные направления перемещения.

Можно переместить курсор сразу в конечную позицию за последней строкой результата методом `afterLast()`.

Появляется возможность поставить курсор на последнюю строку результата логическим методом `last()`, можно вернуть курсор в начальную позицию перед первой строкой результата методом `beforeFirst()` или поместить его на первую строку логическим методом `first()`.

Передвинуть курсор на одну строку назад можно логическим методом `previous()`.

Курсор помещается на определенную строку с номером `row` логическим методом `absolute(int row)`. Номер первой строки 1, `absolute(1)` эквивалентно `first()`. Метод `absolute(0)` устанавливает курсор перед первой строкой. Отрицательный номер означает отсчет от последней строки: `absolute(-1)` перемещает курсор на последнюю строку, `absolute(-2)` — на предпоследнюю строку.

Логический метод `relative(int pos)` перемещает курсор относительно текущей строки: `relative(0)` оставляет курсор на месте, `relative(1)` эквивалентно `next()`, `relative(-1)` эквивалентно `previous()` и т. д.

Эти методы возвращают `false`, когда курсор выходит за пределы выборки.

Вот пример перемещения курсора в обратном направлении:

```
Statement st = con.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY);

ResultSet rs = st.executeQuery("SELECT col1, col2 FROM tbl");

rs.afterLast();

while (rs.previous()){
   int n = rs.getInt(1);
```

```
   String s = rs.getString(2);
}
rs.close();
```

Еще пример:

```
rs.absolute(10); // Теперь текущая строка — десятая.
rs.relative(-2); // Текущей строкой становится восьмая строка.
rs.next();       // После этого текущей будет девятая строка.
rs.previous();   // Текущая — снова восьмая строка.
```

Дополнительные свойства выборки

Выборка типа `TYPE_SCROLL_SENSITIVE` обладает еще одним полезным свойством. Она поддерживает связь с базой вплоть до своего закрытия и автоматически обновляется при обновлении в базе выбираемых ею строк и столбцов. Чтобы быть уверенным, что текущая строка выборки совпадает со строкой базы данных, можно применить метод `refreshRow()`.

Конечно, дополнительные свойства перемещения курсора и обновления выборки требуют дополнительных ресурсов и уменьшают производительность. Поэтому применять их следует только при необходимости.

Есть два способа повышения производительности работы с выборкой. Первый способ — настройка размера буфера для перемещения курсора — обеспечивается методом `setFetchSize(int rows)`. Некоторые драйверы JDBC не размещают всю выборку сразу в оперативной памяти клиента, а подгружают ее строки по мере надобности. Метод `setFetchSize(int rows)` запрашивает у драйвера `rows` строк.

Второй способ — указать драйверу направление перемещения курсора методом `setFetchDirection(int)`. Аргумент метода — одна из констант `FETCH_FORWARD`, `FETCH_REVERSE` или `FETCH_UNKNOWN`.

Эти два метода — только пожелания (hints) драйверу, которые он может учесть, а может и не принимать во внимание.

Такие же два метода описаны и в интерфейсе `Statement`. Они позволяют сформулировать пожелания драйверу сразу для всех выборок данной инструкции типа `Statement`.

Основу интерфейса `ResultSet` составляет множество методов `getXxx()` и `updateXxx()`, позволяющих получить сведения из выборки и изменить ее. Перейдем к изложению этих методов, но сначала посмотрим, как можно получить общие свойства выборки.

Интерфейс *ResultSetMetaData*

Кроме выборки и курсора, объект типа `ResultSet` содержит еще общие сведения о выборке. Они описаны интерфейсом `ResultSetMetaData`. Объект типа `ResultSetMetaData` создается методом

```
public ResultSetMetaData getMetaData();
```

описанным в интерфейсе `ResultSet`. Методы интерфейса `ResultSetMetaData` позволяют получить следующие сведения.

Число выбранных столбцов возвращает метод `getColumnCount()`. Имя столбца с номером `n` возвращает метод `getColumnName(int n)`. В тех случаях, когда имя столбца неизвестно, может помочь метод `getColumnLabel(int n)`.

Имя класса Java, которому принадлежат данные `n`-го столбца выборки, можно получить методом `getColumnClassName(int n)`, а числовую константу класса `Types`, определяющую тип SQL столбца, — методом `getColumnType(int n)`. Тип данных столбца, определенный в базе данных, возвращает метод `getColumnTypeName(int n)`.

Метод `getColumnDisplaySize(int n)` возвращает наибольшую ширину `n`-го столбца в символах. Если в `n`-м столбце выборки находятся числа, то метод `getPrecision(int n)` дает наибольшее количество цифр, а метод `getScale(int n)` — точность представления числа.

Вот как можно сформировать строку заголовков столбцов для распечатки:

```
ResultSetMetaData rsmd = rs.getMetaData();
int n = rsmd.getColumnCount();
String cols = "";

for (int i = ; i <= n; i++){
   String c = rsmd.getColumnName(i);
   int w = rsmd.getColumnDisplaySize(i) - c.length();
   int m = (w > 0) ? w / 2 : 0;
   for (int k = 0; k < m; k++) cols += " ";
   cols += c + " ";
   for (int k = 0; k < m; k++) cols += " ";
}
```

Методы `getCatalogName(int n)`, `getSchemaName(int n)`, `getTableName(int n)` позволяют узнать имя каталога, схемы и таблицы, из которой выбран `n`-й столбец.

Логические методы `isReadOnly(int n)`, `isWritable(int n)`, `isNullable(int n)`, `isCaseSensitive(int n)` дают возможность проверки соответствующих свойств `n`-го столбца.

Обработка выборки

После установки курсора на определенную строку выборки, эта строка становится текущей строкой. Столбцы в строке пронумерованы, начиная с 1, и в большинстве случаев снабжены именами, взятыми из базы данных. Для получения значения каждого столбца есть два метода: метод вида `getXxx(int n)`, возвращающий значение `n`-го столбца, и метод вида `getXxx(String name)`, возвращающий значение столбца с именем `name`. Такие методы описаны для каждого типа Java, всего насчитывается около тридцати пар методов `getXxx()`.

Среди этих методов выделяются методы `getObject(int)` и `getObject(String)`, возвращающие объект самого общего вида. Их можно применять в тех случаях, когда тип столбца неизвестен. Кроме того, это единственные методы для вывода JDBC-типов `STRUCT` и `JAVA OBJECT`. Вот как можно распечатать всю выборку этими методами:

```
ResultSet rs = st.executeQuery("SELECT * FROM tbl");

ResultSetMetaData rsmd = rs.getMetaData();
int cols = rsmd.getColumnCount();

for (int i = 1; i < cols; i++)
  System.out.print(rsmd.getColumnName(i) + "\t");

System.out.println(rsmd.getColumnName(cols));

while (rs.next()){
   for (int i = 1; i < cols; i++)
      System.out.print(rs.getObject(i) + "\t");
   System.out.println(rs.getObject(cols));
}
rs.close();
```

У методов `getString(int)`, `getString(String)` тоже широкая область применения. Они годятся для получения значений почти всех SQL-типов. Исключение составляют JDBC-типы `STRUCT`, `JAVA OBJECT` и типы `BLOB`, `CLOB`, `ARRAY`, `REF`. Для получения значений этих типов есть специальные методы `getBlob()`, `getClob()`, `getArray()`, `getRef()`. О них мы поговорим в следующей главе.

Но все-таки для каждого SQL-типа лучше использовать специально предназначенный для него метод `getXxx()`. Таблицы соответствия типов SQL и

методов getXxx() приведены в спецификации JDBC, находящейся в документации J2SE SDK в каталоге $JAVA_HOME/docs/guide/jdbc/spec/.

Особое место занимает получение больших значений SQL типа LONGVARCHAR, LONGVARBINARY. Их можно получить методами getString() и getBytes(), но многие драйверы JDBC подгружают эти значения по частям, а в этом случае упомянутые методы не годятся. Вместо них лучше использовать методы getAsciiStream(), getBinaryStream(), открывающие входной байтовый поток типа InputStream, и метод getCharacterStream(), открывающий символьный входной поток типа Reader.

Обновляемые результаты

Значения базы данных можно изменить не только непосредственно методом executeUpdate(String) с SQL-операторами INSERT, UPDATE, DELETE в качестве аргумента. Это можно сделать и с помощью выборки типа CONCUR_UPDATABLE. Такое изменение происходит в два этапа.

На первом этапе методами updateXxx(int, Xxx), updateXxx(String, Xxx) изменяются значения столбцов текущей строки выборки. При этом изменяется только выборка, но не соответствующие столбцы базы данных. Изменения, сделанные на этом этапе, можно отменить методом cancelRowUpdates().

На втором этапе измененная строка выборки заносится в базу данных методом updateRow().

Оба этапа выполняются примерно так:

```
Statement st = con.createStatement(ResultSet.TYPE_FORWARD_ONLY,
   ResultSet.CONCUR_UPDATABLE);

ResultSet rs = st.executeQuery("SELECT address FROM students " +
   "WHERE id = 152");

rs.next();

rs.updateString("address", "Елисейские поля, 4");
rs.updateRow();
```

Так же как и для получения значений столбцов выборки, для каждого типа Java есть свой метод updateXxx(). Руководствуясь таблицами, приведенными в пункте "Data Type Conversion Tables" спецификации JDBC, можно подобрать наилучший метод для изменения столбцов выборки и таблиц базы данных.

Вот пример такого подбора для изменения `k`-го столбца текущей строки выборки:

```
String value = "Какое-то значение в виде строки символов";
ResultSetMetaData rsmd = rs.getMetaData();
int type = rsmd.getColumnType(k);
switch (type){
   case Types.CHAR:
   case Types.VARCHAR:
   case Types.VARCHAR:
      rs.updateString(k, value);
      break;
   case Types.BIT:
   case Types.BOOLEAN:
      rs.updateByte(k, Byte.valueOf(value).byteValue());
      break;
   case Types.TYNYINT:
   case Types.SMALLINT:
   case Types.INTEGER:
      rs.updateInt(k, Integer.parseInt(value));
      break;
   case Types.BIGINT:
      rs.updateLong(k, Long.valueOf(value).longValue());
      break;
   case Types.FLOAT:
      rs.updateFloat(k, Float.valueOf(value).floatValue());
      break;
   case Types.DOUBLE:
      rs.updateDouble(k, Double.valueOf(value).doubleValue());
      break;
   case Types.DATE:
      rs.updateDate(k, java.sql.Date.valueOf(value));
      break;
   case Types.TIME:
      rs.updateTime(k, java.sql.Time.valueOf(value));
      break;
   case Types.TIMESTAMP:
      rs.updateTimestamp(k, java.sql.Timestamp.valueOf(value));
      break;
```

```
   default:
      rs.updateObject(k, value);
      break;
   }
rs.updateRow();
```

Удаление строки

Текущую строку выборки и базы данных можно удалить методом `deleteRow()`, например:

```
Statement st = con.createStatement(ResultSet.TYPE_FORWARD_ONLY,
   ResultSet.CONCUR_UPDATABLE);

ResultSet rs = st.executeQuery("SELECT * FROM students " +
   "WHERE id = 32");

rs.next();

rs.deleteRow();
```

Вставка строки

Вставка строки в выборку и в таблицы базы данных выполняется в три этапа.

На первом этапе создается вставляемая строка, и курсор перемещается на нее. Это делается одним методом `moveToInsertRow()`.

На втором этапе строка заполняется значениями столбцов с помощью методов `updateXxx()`, соответствующих типу столбца. Это необходимо сделать для каждого столбца с ограничением целостности `NOT NULL`.

На третьем этапе сформированная строка вставляется в выборку и в таблицы базы данных методом `insertRow()`.

Затем можно вернуть курсор на прежнее место методом `moveToCurrentRow()`.

Например:

```
ResultSet rs = st.executeQuery("SELECT * FROM courses");
rs.moveToInsertRow();
rs.updateString(1, "Домоводство");
rs.insertRow();
rs.moveToCurrentRow();
```

Выборка типа *RowSet*

Интерфейс `RowSet`, расширяющий интерфейс `ResultSet`, описывает выборку как компонент JavaBean. Выборка, реализующая этот интерфейс, обладает всеми описанными ранее качествами выборки и дополнительно обладает свойствами (properties) компонента JavaBean со своими методами `getXxx()` и `setXxx()` для каждого свойства. К таким свойствам относятся адрес URL и имя источника данных, имя и пароль пользователя, уровень изоляции транзакции, выполняемый запрос, время ожидания ответа и другие характеристики запроса.

Вместе с интерфейсом `RowSet` надо реализовать вспомогательные интерфейсы `RowSetInternal`, `RowSetMetaData`, `RowSetReader`, `RowSetWriter`. Как и всякий компонент JavaBean, классы, реализующие интерфейсы `RowSetXxx`, и прочие классы компонента упаковываются в JAR-архив и в таком виде распространяются по сети. Архив можно использовать не только как библиотеку классов приложения, но и в графических средствах разработки (IDE) для включения его в создаваемое приложение как один из компонентов.

Компоненты `RowSet` могут создаваться любыми разработчиками. Каждый разработчик может придать им те или иные дополнительные свойства. Все компоненты делятся на две большие группы: компоненты, поддерживающие постоянное соединение с базой данных (connected rowsets), и компоненты без постоянного соединения (disconnected rowsets). Компоненты второй группы получают выборку, помещают ее в оперативную память клиента и закрывают соединение, возобновляя его, только если нужно обновить выборку.

Подразделение Java Software фирмы Sun Microsystems разрабатывает три реализации интерфейса `RowSet`. Их общие свойства собраны в абстрактный класс `BaseRowSet`, который расширен тремя классами.

- Класс `JDBCRowSet` — это компонент с постоянным соединением, работающий как часть драйвера JDBC. Он просто оформляет реализацию интерфейса `ResultSet` как компонент JavaBean.
- Класс `CachedRowSet` — это компонент без постоянного соединения. Кроме того, он кэширует выборку в оперативной памяти. Из-за этого он не может работать с очень большой выборкой, но зато расширяет возможности драйвера JDBC, что хорошо подходит для "тонкого" клиента.
- Класс `WebRowSet` — расширение класса `CachedRowSet` — это компонент без постоянного соединения, общающийся по протоколу HTTP с сервлетом, который связывается с базой данных. Он представляет выборку в виде документа XML и удобен для применения в Web-приложениях. Заголовок DTD (Document Type Definition — определение типа документа) документа описан в файле RowSet.dtd.

Ко времени написания книги эти реализации интерфейса `RowSet` не входили в стандартную поставку Sun Java 2 SDK, но их можно было загрузить с адреса разработчиков Java **http://developer.java.sun.com/developer/earlyAccess/crs/**.

Получив компонент в виде файла rowset.jar, его надо, не распаковывая, поместить в каталог, к которому в переменной `CLASSPATH` указан путь, или просто в каталог $JAVA_HOME/jre/lib/ext/. Классы, упакованные в архив rowset.jar, составляют пакет sun.jdbc.rowset.

Соединение с источником данных

Работа с реализациями интерфейса `RowSet` ведется обычным образом. Конструктором по умолчанию создается экземпляр класса, определяются его свойства, и выполняется запрос, как показано в листинге 13.1, в методе `go()`.

Листинг 13.1 приводит полный цикл работы с выборкой класса `CachedRowSet`. К этому примеру мы будем обращаться несколько раз по мере изучения методов этого класса.

Листинг 13.1. Соединение с помощью `RowSet`

```
import java.sql.*;
import javax.sql.*;
import sun.jdbc.rowset.*;

public class ConRowSet{

   public static void main(String[] args){
      ConRowSet crs = new ConRowSet();
      try{
         crs.setup();
         crs.go(false);
         crs.go(true);
      }catch(SQLException e){
         System.err.println(e);
      }
   }

   private void go(boolean populated) throws SQLException {
      CachedRowSet crs = new CachedRowSet();

      crs.setUrl("jdbc:odbc:SDO");
      crs.setUsername("sdoadmin");
```

```
    crs.setPassword("sdoadmin");
    crs.setCommand("SELECT col1, col2 FROM test_table ORDER BY 1");

    crs.execute();

    if (!populated){
      populateRowset(crs);
      System.err.println("Выборка заполнена.");

      crs.acceptChanges();
      System.err.println("Таблица изменена.");
    }else scrollCursor(crs);
  }

  private void setup() throws SQLException{

    Connection conn = getConn();
    Statement stmt = conn.createStatement();

    try{
      stmt.execute("DROP TABLE test_table");
    }catch(SQLException ex){
      System.err.println("Ошибка удаления таблицы.");
    }

    stmt.execute("CREATE TABLE test_table (col1 INT, col2 FLOAT)");
    System.out.println("Таблица создана");

    stmt.close();
    conn.commit();
    conn.close();
  }

  private Connection getConn(){

    try{
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    }catch(ClassNotFoundException e){
      System.err.println(e);
    }
```

```
    try{
       return DriverManager.getConnection
         ("jdbc:odbc:SDO;UID=sdoadmin;PWD=sdoadmin");
    }catch(SQLException e){
       System.err.println(e);
    }
    return null;
  }

  private void scrollCursor(CachedRowSet crs)
                              throws SQLException{
    System.out.println("Просмотр выборки.");

    String v1;  float v2;

    while (crs.next()){

      v1 = crs.getString(1);
      if (!crs.wasNull())
           System.out.println("v1 = " + v1);
      else System.out.println("v1 is null");

      v2 = crs.getFloat("col2");
      if (!crs.wasNull())
           System.out.println("v2 = " + v2);
      else System.out.println("v2 is null");
    }

    if (crs.isAfterLast()){
       System.out.println("Курсор за последней строкой.");
       System.out.println("Текущая строка: " + crs.getRow());
    }

    System.out.println("Просмотр в обратном направлении.");

    while (crs.previous()){

      v1 = crs.getString("col1");
      if (!crs.wasNull())
```

```
            System.out.println("v1 = " + v1);
        else System.out.println("v1 is null");

        v2 = crs.getFloat(2);
        if (!crs.wasNull())
            System.out.println("v2 = " + v2);
        else System.out.println("v2 is null");
      }

      if (crs.isBeforeFirst())
         System.out.println("Курсор перед первой строкой.");

      crs.first();
      if (crs.isFirst())
         System.out.println("Курсор на первой строке.");

       System.out.println("Текущая строка: " + crs.getRow());

       crs.last();
       if (crs.isLast())
          System.out.println("Курсор на последней строке.");

       System.out.println("Текущая строка: " + crs.getRow());

       if (crs.isAfterLast())
          System.out.println("Курсор за последней строкой.");
    }

    private void populateRowset(CachedRowSet crs)
                                throws SQLException{

       for (int i=0; i < 10 ; i++){
          crs.moveToInsertRow();
          crs.updateInt(1, i);
          crs.updateFloat(2, ((((float)i) * 100) + i));
          crs.insertRow();
          crs.moveToCurrentRow();
       }
   }
}
```

Соединение может быть установлено не только с базой данных, но и с любым источником данных, включенным в распределенную систему именования JNDI. Например, в листинге 13.1 можно установить соединение следующим образом:

```
WebRowSet wrs = new WebRowSet();

wrs.setDataSourceName("jdbc/SDO");

wrs.setUsername("sdoadmin");
wrs.setPassword("sdoadmin");
wrs.setTransactionIsolation(
    Connection.TRANSACTION_READ_COMMITTED);
wrs.setCommand("SELECT * FROM students");

wrs.execute();
```

Заполнение выборки

Метод `execute()` устанавливает соединение с источником данных, создает объект типа `Statement` и с его помощью получает объект типа `ResultSet`, с которым и работает компонент `RowSet`.

Если соединение уже установлено, то класс `CachedRowSet` позволяет выполнить запрос методом `execute(Connection)`. Листинг 13.2 показывает пример такого соединения.

Листинг 13.2. Выборка `RowSet` из установленного ранее соединения

```
import java.sql.*;
import javax.sql.*;
import sun.jdbc.rowset.*;

public class CnRowSet{

   public static void main(String[] args){
     try{
       Class.forName("oracle.jdbc.driver.OracleDriver");

       Connection con = DriverManager.getConnection
         ("jdbc:oracle:thin:sdoadmin/sdoadmin@homexp:1521:SDO");
```

```
        CachedRowSet crs = new CachedRowSet();

        crs.setCommand("SELECT * FROM students");

        crs.execute(con);
        System.out.println("RowSet populated.");

      }catch(Exception e){
        System.err.println("ConRowSet: " + e);
      }
    }
}
```

Если используется реализация интерфейса `RowSet` без постоянного соединения, то после выполнения `execute()` соединение закрывается.

Класс `CachedRowSet` предлагает еще один способ заполнения компонента данными: после получения объекта типа `ResultSet` передать записанные в нем данные в объект класса `CachedRowSet` методом `populate(ResultSet)`.

```
ResultSet rs = st.executeQuery("SELECT * FROM students");
CachedRowSet crs = new CachedRowSet();
crs.populate(rs);
rs.close();
st.close();
con.close();
crs.absolute(3);
crs.previous();
```

Это удобно, когда драйвер JDBC не позволяет сделать выборку `rs` с расширенными возможностями просмотра и обновления.

Обработка результатов

После выполнения выборки обработка результатов идет точно так же, как обработка выборки типа `ResultSet`. Это показано в методе `scrollCursor(CachedRowSet)` листинга 13.1.

По умолчанию у выборки класса `CachedRowSet` тип `TYPE_SCROLL_INSENSITIVE`, который позволяет просматривать выборку в любом направлении. При желании можно изменить тип выборки методом `setType(int)`, например:

```
crs.setType(ResultSet.TYPE_SCROLL_SENSITIVE);
```

По умолчанию выборка не обновляема, ее тип `CONCUR_READ_ONLY`. При необходимости динамического обновления выборки надо изменить ее тип методом `setConcurrency(int)`, например:

```
crs.setConcurrency(ResultSet.CONCUR_UPDATABLE);
```

Если выборка без постоянного соединения, как выборка класса `CachedRowSet`, то методы `updateRow()`, `insertRow()`, `deleteRow()` меняют только выборку, но не связанный с ней источник данных. Такое обновление выполняется в листинге 13.1 методом `populateRowset()`. Для того чтобы внести изменения в источник данных, надо выполнить метод `acceptChanges()`, что и сделано в листинге 13.1 после применения метода `populateRowset()`.

Метод `acceptChanges()` восстанавливает соединение и пытается записать измененные значения в источник данных. Если это не удается сделать, то выбрасывается исключение класса `SQLException`, и курсор устанавливается на первую из измененных в выборке строк. В этом случае можно восстановить старые значения выборки методом `restoreOriginal()`.

Если выборка получена из соединения, установленного заранее, то можно воспользоваться методом `acceptChanges(Connection)`.

Все эти действия приведены в листинге 13.3. В нем выполняется выборка класса `WebRowSet`, расширяющего класс `CachedRowSet`, и используется тестовая таблица `test_table`, созданная программой листинга 13.1.

Листинг 13.3. Действия с выборкой класса `WebRowSet`

```
import java.sql.*;
import javax.sql.*;
import sun.jdbc.rowset.*;
import java.io.*;

public class WbRowSet{

   public WbRowSet(){
     try{
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

        WebRowSet wrs = new WebRowSet();

        wrs.setUrl("jdbc:odbc:SDO");
        wrs.setUsername("sdoadmin");
        wrs.setPassword("sdoadmin");
```

```
      wrs.setCommand("SELECT col1, col2 FROM test_table");
      wrs.setType(ResultSet.TYPE_SCROLL_SENSITIVE);
      wrs.setConcurrency(ResultSet.CONCUR_UPDATABLE);
      wrs.setTransactionIsolation(
        java.sql.Connection.TRANSACTION_READ_UNCOMMITTED);

      wrs.execute();
      System.out.println("Выборка создана.");

         // Просматриваем и изменяем выборку и базу.
      scrollCursor(wrs);

         // Создаем файл для записи выборки.
      FileWriter fw = new FileWriter("WbRowSet.xml");

         // Записываем выборку в файл в формате XML.
      wrs.writeXml(fw);

         // Открываем файл для чтения.
      FileReader fr = new FileReader("WbRowSet.xml");

         // Создаем новую выборку из XML-файла.
      WebRowSet wrs2 = new WebRowSet();
      wrs2.readXml(fr);

         // Что-нибудь делаем с новой выборкой...
         // и записываем ее в XML-файл.
      FileWriter fw2 = new FileWriter("WbRowSet_2.xml");
      wrs2.writeXml(fw2);

   }catch(Exception e){
      System.out.println(e);
   }
}

private void scrollCursor(WebRowSet wrs) throws SQLException{
   System.out.println("Просмотр и изменение выборки.");

   String v1;  int v2;
```

```
wrs.beforeFirst();
   // Просмотр выборки.
while (wrs.next()){

   v1 = wrs.getString(1);
   if (!wrs.wasNull())
        System.out.println("v1 = " + v1);
   else System.out.println("v1 is null");

   v2 = wrs.getInt("col2");
   if (!wrs.wasNull()){
      System.out.println("v2 = " + v2);

      if (v2 == 2){
        System.out.println("Меняем строку.");
        wrs.updateInt("col2", 9999);
        wrs.updateString("col1", "Big Number");
        wrs.updateRow();

        System.out.println("Строка выборки изменена.");

             // Просматриваем измененную строку.
        v1 = wrs.getString(1);
        if (!wrs.wasNull())
             System.out.println("v1 = " + v1);
        else System.out.println("v1 is null");

        v2 = wrs.getInt("col2");
        if (!wrs.wasNull())
             System.out.println("v2 = " + v2);
        else System.out.println("v2 is null");

           // Вставляем строку в выборку.
        wrs.moveToInsertRow();
        wrs.updateInt("col2", 5);
        wrs.updateString("col1", "go");
        wrs.insertRow();
        wrs.moveToCurrentRow();
```

```
        wrs.next();
            // Изменяем еще одну строку.
        wrs.updateString("col1", "five");
        wrs.updateRow();
        wrs.previous();
      }
   }else System.out.println("v2 is null");
}

if (wrs.isAfterLast()){
   System.out.println("Курсор за последней строкой.");
   System.out.println("Текущая строка: " + wrs.getRow());
}

  // Вносим изменения в источник данных.
wrs.acceptChanges();

System.out.println("Курсор перемещается назад.");

while (wrs.previous()){

   v1 = wrs.getString("col1");
   if (!wrs.wasNull())
         System.out.println("v1 = " + v1);
   else System.out.println("v1 is null");

   v2 = wrs.getInt(2);
   if (!wrs.wasNull())
        System.out.println("v2 = " + v2);
   else System.out.println("v2 is null");
}

if (wrs.isBeforeFirst())
  System.out.println("Курсор перед первой строкой.");

 wrs.first();
 if (wrs.isFirst())
   System.out.println("Курсор на первой строке.");
```

```
      System.out.println("Текущая строка: " + wrs.getRow());

      wrs.last();
      if (wrs.isLast())
        System.out.println("Курсор на последней строке.");

      System.out.println("Текущая строка: " + wrs.getRow());
    }
   public static void main(String args[]){
      new WbRowSet();
   }
}
```

Как видно из листинга 13.3, класс `WebRowSet` обладает дополнительным свойством. Он может записывать выборку в выходной поток в виде документа XML и читать документ XML из входного потока. Для этого в классе `WebRowSet` есть методы `writeXml(Writer)`, `readXml(Reader)` и статический метод `writeXml(ResultSet, Writer)`.

Последний метод удобен для записи выборки типа `ResultSet` в виде документа XML. Он создает объект класса `WebRowSet` из выборки, указанной первым аргументом метода, и записывает его в выходной поток, указанный вторым аргументом, например:

```
ResultSet rs = st.executeQuery("SELECT * FROM tbl");
FileWriter fr = new FileWriter("newrowset.xml");
WebRowSet.writeXml(rs, fr);
```

Обработка событий

Как всякий компонент JavaBean, выборка типа `RowSet` реагирует на события, а именно, на событие класса `RowSetEvent`. Такое событие происходит в трех случаях: при изменении строки выборки, при изменении всей выборки или при перемещении курсора выборки. Соответственно, в интерфейсе `RowSetListener` описаны три метода, приведенные в листинге 13.4.

Листинг 13.4. Прослушивание событий в `RowSet`

```
import java.sql.*;
import javax.sql.*;
import sun.jdbc.rowset.*;

public class EvRowSet implements RowSetListener{
```

```
   EvRowSet(){
     try{
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

        CachedRowSet jrs = new CachedRowSet();

        jrs.addRowSetListener(this);
        jrs.setUrl("jdbc:odbc:SDO");
        jrs.setUsername("sdoadmin");
        jrs.setPassword("sdoadmin");
        jrs.setCommand("SELECT * FROM students ORDER BY id");

        jrs.execute();

        jrs.last();
        if (jrs.isLast())
          System.out.println("Последним зарегистрировался " +
                jrs.getString(2));

        jrs.close();
     }catch(Exception e){
        System.err.println(e);
     }
   }

   public void rowSetChanged(RowSetEvent event){
      System.out.println("Выборка изменилась.");
   }

   public void rowChanged(RowSetEvent event){
      System.out.println("Строка выборки изменилась.");
   }

   public void cursorMoved(RowSetEvent event){
      System.out.println("Курсор выборки переместился.");
   }

   public static void main(String[] args){
     new EvRowSet();
  }
}
```

Глава 14

Прочие свойства интерфейса JDBC

В интерфейсе JDBC есть еще несколько общих свойств, предназначенных для увеличения производительности или для более полного охвата возможностей языка SQL. В этой главе мы познакомимся с этими свойствами. В конце главы покажем двухслойную схему реализации системы дистанционного обучения, о которой мы говорили в *главе 2*.

Пакетное изменение источника данных

Методы `executeUpdate()` посылают каждый запрос на выполнение SQL-операторов `INSERT`, `UPDATE`, `DELETE` по отдельности, даже если запросы идут подряд в рамках одной транзакции. Производительность выполнения этих запросов можно увеличить, если посылать их вместе в одном пакете (batch). Проверить, допускает ли соединение с источником данных пакетное выполнение запросов, можно логическим методом `supportsBatchUpdates()`, описанным в интерфейсе `DatabaseMetaData`.

Пакетное выполнение операторов изменения базы данных происходит в два этапа. Сначала методами `addBatch(String sql)` операторы SQL собираются в пакет. Затем пакет посылается серверу СУБД методом `executeBatch()`. Все это выглядит примерно так:

```
Statement st = con.createStatement();

con.setAutoCommit(false);

st.addBatch("INSERT INTO tbl1 VALUES(15, 'Иванов')");
st.addBatch("INSERT INTO tbl1 VALUES(20, 'Петров')");
st.addBatch("INSERT INTO tbl2 VALUES('Иванов', 1975, 732456)");
st.addBatch("INSERT INTO tbl2 VALUES('Петров', 1967, 437694)");

int[] chCount = st.executeBatch();

con.commit();
```

Перед созданием пакета следует отключить автоматическое завершение транзакции после выполнения каждого запроса, которое установлено по умолчанию. Для отмены автоматического завершения транзакции применяется метод `setAutoCommit(false)`. После этого завершение или откат транзакции придется выполнять явно, методами `commit()` или `rollback()`.

Метод `executeBatch()` возвращает целочисленный массив, каждый элемент которого содержит число обработанных соответствующим запросом строк или константу `SUCCESS_NO_INFO`. Константа записывается в массив, если запрос выполнен, но число обработанных строк неизвестно. Если выполнить пакет операторов не удалось или какой-то запрос из пакета возвращает выборку типа `ResultSet`, то выбрасывается исключение класса `BatchUpdateException`.

Драйвер JDBC может прекратить выполнение пакета после возникновения исключительной ситуации или выполнить оставшиеся в пакете запросы. В любом случае при обработке исключения можно методом `getUpdateCount()` класса `BatchUpdateException` получить массив результатов запросов пакета. В первом случае число элементов массива будет равно числу выполненных запросов, во втором случае — числу запросов, содержащихся в пакете.

Если отпала необходимость в выполнении пакета запросов, то его можно расформировать методом `clearBatch()`.

В пакет можно собрать и значения предварительно откомпилированного запроса типа `PreparedStatement`. Перед занесением в пакет следует заполнить все неопределенные параметры такого запроса, например:

```
con.setAutoCommit(false);

PreparedStatement pst = con.prepareStatement(
    "UPDATE tbl SET name = ? WHERE id = ?");

pst.setString(1, "Фролов");
pst.setInt(2, 50);
pst.addBatch();

pst.setString(1, "Хрулев");
pst.setInt(2, 60);
pst.addBatch();

// и так далее...

int[] chCount = pst.executeBatch();
con.commit();
```

Для занесения запроса в пакет здесь применяется метод `addBatch()` без аргументов.

Пакетное выполнение хранимых процедур типа `CallableStatement` возможно, если процедуры не содержат параметров типа `OUT` или `INOUT`. В остальном занесение запросов в пакет и его выполнение не отличается от пакетного выполнения предварительно откомпилированного запроса:

```
con.setAutoCommit(false);

CallableStatement cst =
        con.prepareCall("{call addMark(?, ?)}");
cst.setInt(1, 37);
cst.setInt(2, 4);
cst.addBatch();

cst.setInt(1, 19);
cst.setInt(2, 5);
cst.addBatch();

int[] chCounts = cst.executeBatch();
con.commit();
```

Типы данных *BLOB* и *CLOB*

Основной принцип реляционных СУБД — хранение в столбцах таблиц простых, неделимых, как говорят, атомарных значений, — чисел, дат, коротких строк. Но часто приходится хранить и большие объемы неделимой информации, содержащей звук, изображение, статьи и целые книги. Для описания таких данных в язык SQL введен *тип* `BLOB` (Binary Large Object), содержащий длинную последовательность бинарных данных, и *тип* `CLOB` (Character Large Object), содержащий большое количество символов.

Хранение и извлечение больших объектов типа `BLOB` и `CLOB` имеет особенности. Для их учета драйверу JDBC нужно реализовать интерфейсы `Blob` и `Clob`.

Интерфейсы *Blob* и *Clob*

Интерфейсы `Blob` и `Clob` описывают методы работы с большими объектами.

Оба интерфейса позволяют получить длину большого объекта методом `length()`, возвращающим тип `long`. Длина объекта типа `Blob` измеряется в байтах, а длина объекта типа `Clob` — в символах.

Оба интерфейса позволяют найти позицию заданного подобъекта `patt` в виде целого числа типа `long` методами `position(Blob patt, long pos)`, `position(byte[] patt, long pos)` для объекта типа `Blob` и методами `position(Clob patt, long pos)`, `position(String patt, long pos)` для объекта типа `Clob`. Второй аргумент этих методов `pos` задает начальную позицию поиска подобъекта. Его наименьшее значение равно 1.

Большие объекты обычно заносятся в базу данных и извлекаются из нее по частям. Чтобы учесть это обстоятельство, оба интерфейса описывают методы создания входного потока для представления большого объекта в виде потока и выходного потока для записи объекта в базу данных.

Интерфейс `Blob` описывает метод `getBinaryStream()` для получения входного потока типа `InputStream` и метод `setBinaryStream(long)`, возвращающий выходной поток типа `OutputStream` для записи части объекта типа `Blob`, начиная с позиции, указанной аргументом.

Интерфейс `Clob`, кроме байтовых потоков, возвращаемых методами

```
public InputStream getAsciiStream();
public OutputStream setAsciiStream(long);
```

предписывает создавать еще символьные потоки, возвращаемые методами

```
public Reader getCharacterStream();
public Writer setCharacterStream(long);
```

Оба интерфейса позволяют выделять подобъект заданной длины `length`, начиная с позиции `pos`. Интерфейс `Blob` делает это методом `getBytes(long pos, int length)`, возвращающим массив типа `byte[]`, а интерфейс `Clob` — методом `getSubString(long pos, int length)`, возвращающим строку типа `String`.

Кроме того, интерфейс `Blob` позволяет изменить часть объекта, записав в него массив байтов `b` или его часть методами

```
public int setBytes(long pos, byte[] b);
public int setBytes(long pos, byte[] b, int offset, int length);
```

а интерфейс `Clob` — строку `s` или ее часть методами

```
public int setString(long pos, String s);
public int setString(long pos, String s, int offset, int length);
```

В этих методах аргумент `pos` — позиция, начиная с которой вставляется массив или строка, `offset` — начальная позиция вставляемой части массива `b` или строки `s`, `length` — длина вставляемой части. Методы возвращают число записанных данных.

Драйверы JDBC могут изменять непосредственно объект или создавать его измененную копию. Во втором случае для изменения содержимого базы дан-

ных потребуется дополнительное применение метода `executeUpdate()`. Узнать поведение драйвера можно логическим методом `locatorsUpdateCopy()`, описанным в интерфейсе `DatabaseMetaData`. Метод возвращает `true`, если драйвер изменяет копию объекта, и `false`, если драйвер изменяет сам объект. Чтобы учесть оба случая, надо написать примерно такой код:

```
byte[] b = {0, 1, 2, 3, 4};

Blob bl = rs.getBlob(1);

int n = bl.setBytes(1, b);

if (dbmd.locatorsUpdateCopy()){
   PreparedStatement pst = con.prepareStatement(
       "UPDATE tbl SET data = ?");
   pst.setBlob(1, bl);
   pst.executeUpdate();
}
```

Получение большого объекта

Большие объекты извлекаются из источника данных обычным образом. Сначала они заносятся в экземпляр `ResultSet`, например, методом `executeQuery()`, а затем извлекаются из него методами `getBlob()`, `getClob()`, например

```
ResultSet rs = st.executeQuery("SELECT photo FROM emp " +
   "WHERE name = 'Иванов'");
rs.next();
Blob bl = rs.getBlob(1);
```

Созданный объект `bl` — только указатель на элемент столбца `photo`, хранящегося в объекте `rs`. Для работы с большим объектом его можно перевести в массив байтов:

```
byte [] data = bl.getBytes(1, (int)bl.length());
```

или записать в поток:

```
InputStream is = bl.getBinaryStream();
```

который затем можно направить в файл:

```
FileOutputStream fos = new FileOutputStream("ivanov.gif");
int b;
while ((b = is.read()) != -1) fos.write(b);
fos.close();
```

или создать изображение для вывода на экране:

```
ImageIcon im = new ImageIcon(
   bl.getBytes(1, (int)bl.length()));
JLabel photo = new JLabel(im);
add(photo);
```

Аналогично работают и с большим символьным объектом:

```
ResultSet rs = st.executeQuery("SELECT book FROM lib " +
    "WHERE book = 'Золушка'");
rs.next();
Clob cl = rs.getClob(1);
String text = cl.getSubString(1, (int)cl.length());
JTextArea ta = new JTextArea(text);
add(ta);
```

Если использование метода `getSubString()` затруднено из-за передачи информации по сети, то для большого символьного объекта `cl` можно создать байтовый или символьный входной поток:

```
InputStream is = cl.getAsciiStream();
Reader r = cl.getCharacterStream();
```

и использовать его таким же образом:

```
InputStreamReader in =
   new InputStreamReader(cl.getAsciiStream());
JTextArea text = new JTextArea(readString(in));
add(text);
```

Здесь мы применили известный метод `readString(InputStreamReader)` из сборника "Java Receipts". Вот его исходный текст:

```
public static String readString(InputStreamReader in)
                                    throws IOException{
   StringBuffer buf = new StringBuffer();
   int n;
   char[] b = new char[1024];

   while((n = in.read(b)) > 0) buf.append(b, 0, n);

   return buf.toString();
}
```

Занесение большого объекта в источник данных

Большой объект можно записать в базу данных точно так же, как и данные других типов, например:

```
PreparedStatement pst = con.prepareStatement(
   "UPDATE emp SET photo = ?, text = ? WHERE name = ?");
pst.setBlob(1, bl);
pst.setClob(2, cl);
pst.setString(3, "Иванов");
pst.executeUpdate();
```

Кроме того, можно воспользоваться потоками с помощью одного из двух методов:

```
pst.setBinaryStream(1, bl.getBinaryStream(), (int)bl.length());
pst.setCharacterStream(2, cl.getCharacterStream(), (int)cl.length());
```

или методом `setObject()`:

```
pst.setObject(1, bl);
pst.setObject(2, cl);
```

или занести большой объект в базу данных прямо из файла:

```
File f1 = new File("myface.jpg") ;
FileInputStream fis1 = new FileInputStream(f1);
pst.setBinaryStream(1, fis1, (int)f1.length());
File f2 = new File("book.txt");
FileInputStream fis2 = new FileInputStream(f2);
pst.setAsciiStream(2, fis2, (int)f2.length());
```

При изменении источника данных через обновляемый объект типа `ResultSet` следует использовать методы

```
public void updateBlob(int, Blob);
public void updateBlob(String, Blob);
public void updateClob(int, Clob);
public void updateClob(String, Clob);
```

Составные типы данных

Новый стандарт SQL3 позволяет хранить в одной ячейке таблицы целый массив или структуру данных разных типов, подобно структуре в языке C или записи в языке Pascal. Для этого стандарт вводит SQL-типы `ARRAY` и

STRUCT. Драйверы JDBC обрабатывают эти типы методами, описанными в интерфейсах Array и Struct.

Объект, реализующий интерфейс Array, можно получить, как обычно, методами getArray(int), getArray(String), описанными в интерфейсе ResultSet, а для получения объекта типа Struct нет специальных методов. Его получают методами getObject(int), getObject(String) с приведением к типу Struct.

Рассмотрим пример. Сначала создадим тип массива phones и тип структуры address.

```
CREATE TYPE phones AS VARRAY(5) OF INTEGER;
CREATE TYPE address AS OBJECT(
   flat INTEGER,
   build INTEGER,
   street VARCHAR(100));
```

Объединим эти типы:

```
CREATE TYPE emp_type AS OBJECT(
   id INTEGER,
   addr ADDRESS,
   phon PHONES);
```

Создадим таблицу:

```
CREATE TABLE emp OF emp_type;
```

Заполним ее:

```
INSERT INTO emp VALUES(46,
   PHONES(234567, 453690, 568945, 231406, 678420),
   ADDRESS(45, 12, 'Пушкина'));
```

Теперь извлечем строку таблицы emp:

```
ResultSet rs = st.executeQuery("SELECT phon, addr " +
   "FROM emp WHERE id =46");
rs.next();
Array ar = rs.getArray(1);
Struct st = (Struct)rs.getObject(2);
int[] phone = (int[])ar.getArray();
Object[] attr = st.getAttributes();
```

Интерфейс *Array*

Основу интерфейса `Array` составляют четыре метода `getArray()` и четыре метода `getResultSet()`, которые возвращают содержимое объекта типа `Array` или его часть в виде экземпляра класса `Object` или в виде объекта типа `ResultSet`. Два самых развитых метода

```
public Object getArray(long ind, int count, Map map);
public ResultSet getResultSet(long ind, int count, Map map);
```

возвращают `count` элементов массива, начиная с элемента с индексом `ind`. Индекс первого элемента равен 1. Последний аргумент `map` задает отображение типов Java на типы SQL.

Другие методы `getArray()` и `getResultSet()` опускают некоторые аргументы, возвращая весь массив и пользуясь стандартным отображением типов.

Интерфейс *Struct*

Интерфейс `Struct` проще, чем интерфейс `Array`. Он описывает два метода

```
public Object[] getAttributes(Map map);
public Object[] getAttributes();
```

возвращающих массив ссылок на элементы структуры. При этом аргументом `map` может быть задано нестандартное отображение типов Java на типы SQL.

Ссылочный тип данных *REF*

Новый стандарт SQL3 определяет ссылочный тип данных `REF`. Этот тип хранит ссылку на данные. Данные этого типа извлекаются из объекта типа `ResultSet` методами `getRef(int)` и `getRef(String)`, возвращающими объект, реализующий интерфейс `Ref`.

Интерфейс `Ref` описывает методы `getObject()`, `getObject(Map)`, возвращающие объект, ссылка на который хранится в объекте типа `REF`, а также метод `setObject(Object)`, устанавливающий этот объект. Например:

```
ResultSet rs = st.executeQuery("SELECT point FROM tbl");
rs.next();
Ref r = rs.getRef(1);
PointRef pr = (PointRef)r.getObject();
```

Типы данных пользователя, UDT

Многие современные СУБД позволяют пользователю создавать собственные типы данных UDT (User Defined Type). Для этого в язык SQL введен оператор `CREATE TYPE`, конкретный вид которого зависит от СУБД. Объект такого типа занимает одну ячейку или одну строку в таблице базы данных. В *разд. "Составные типы данных"* мы уже создавали тип массива и структуры. Для этих типов есть соответствующие интерфейсы `Array` и `Struct`, описывающие методы работы с ними. Но пользователь может создать тип данных, не описанный ни одним интерфейсом Java. Для таких случаев предусмотрен интерфейс `SQLData`. Его следует реализовать при определении типа пользователя.

Интерфейс `SQLData` описывает три метода:

```
public String getSQLTypeName();
public void readSQL(SQLInput stream, String typeName);
public void writeSQL(SQLOutput stream);
```

с помощью которых драйвер JDBC будет определять тип объекта, читать и записывать его. Среди аргументов этих методов встречаются объекты еще двух видов `SQLInput` и `SQLOutput`.

Интерфейс `SQLInput` описывает множество методов `readXxx()` чтения разных типов JDBC, а интерфейс `SQLOutput` — множество таких же методов `writeXxx()` записи типов JDBC. Эти интерфейсы реализуются драйвером JDBC.

Немного выше мы определили тип `address` как тип структуры:

```
CREATE TYPE address AS OBJECT(
   flat INTEGER,
   build INTEGER,
   street VARCHAR(100));
```

Оформим его как UDT. Реализуем интерфейс `SQLData`:

```
public class Address implements SQLData{
   public int flat;
   public int build;
   public String street;
   private String sqlType;

   public Address(int flat, int build, String street){
      this.flat = flat;
      this.build = build;
      this.street = street;
   }
```

```
   public Address(){
      this(0,0, "");
   }

   public String getSQLTypeName(){
      return sqlType;
   }

   public void readSQL(SQLInput stream, String type)
                        throws SQLException{
      sqlType = type;
      flat = stream.readInt();
      build = stream.readInt();
      street = stream.readString();
   }

   public void writeSQL(SQLOutput stream)
                       throws SQLException{
      stream.writeInt(flat);
      stream.writeInt(build);
      stream.writeString(street);
   }

   public void sendMail(){
      // Какой-то метод пользователя.
   }
}
```

Теперь надо отобразить Java-класс `Address` на новый SQL-тип `address`. Для этого получим ссылку на стандартное отображение и дополним его новой строкой:

```
Map map = con.getTypeMap();
map.put("SDOADMIN.ADDRESS", Class.forName("Address"));
con.setTypeMap(map);
```

После этого можно создавать объект класса `Address` и заносить его в базу данных:

```
Address addr = new Address(34, 56, "Солнечная");
PreparedStatement pst = con.prepareStatement(
         "INSERT INTO emp (address) VALUES(?)");
```

```
pst.setObject(1, addr);
pst.executeUpdate();
```

Драйвер JDBC преобразует Java-объект `add` к SQL-типу `address`, руководствуясь дополненным стандартным отображением.

При извлечении данных SQL-типа `address` из базы данных их можно преобразовать в объект класса `Address` и пользоваться его методами:

```
ResultSet rs = st.executeQuery("SELECT address " +
   "FROM emp WHERE id =46");
rs.next();
Address addr = (Address)rs.getObject(1);
addr.sendMail();
```

Тип данных *DISTINCT*

Примером типа, определяемого пользователем, является подтип какого-нибудь стандартного типа ячеек таблиц базы данных. Вот пример из спецификации JDBC:

```
CREATE TYPE MONEY AS NUMERIC(10, 2) FINAL
```

Такие подтипы объявляются как новый SQL-тип `DISTINCT`. Драйверы JDBC обрабатывают типы `DISTINCT` как значения базового типа. В этом примере можно обрабатывать SQL-тип `MONEY` как Java-тип `BigDecimal` — стандартное отображение SQL-типа `NUMERIC`:

```
BigDecimal mn = rs.getBigDecimal(1);
```

Двухслойная схема СДО

Итак, средства интерфейса JDBC позволяют приложению Java устанавливать связь с базой данных просто по адресу URL. При этом совершенно неважно, где расположена база — на той же машине, где работает сервер базы данных, или на удаленной машине. Важно только качество связи и пропускная способность сети.

Используя JDBC, можно организовать работу распределенного приложения по двухслойной схеме клиент-сервер. Клиентская часть приложения принимает информацию, вводимую пользователем, и через JDBC заносит ее в базу данных. Потом по запросам пользователя информация через JDBC извлекается из базы данных, преобразуется к виду, нужному пользователю, и выводится на экран средствами графической библиотеки Swing. Серверную часть приложения представляет сервер СУБД.

Клиентскую часть можно реализовать либо апплетом, либо приложением Java. Приложение предоставляет все возможности для полноценной работы. Оно полностью владеет всеми ресурсами клиентской машины и позволяет связаться по сети с любой удаленной машиной. Но работа приложения требует, как минимум, наличия JRE [1, 11, 15, 18] на клиентской машине. Кроме того, приложение необходимо загрузить и инсталлировать на клиентской машине.

Апплет сильно ограничен своей "песочницей" (sandbox) [1, 11, 15, 18]. Он не может обращаться к файловой системе компьютера, на котором он работает или связаться с какой-нибудь удаленной машиной, кроме той, откуда он загружен. Однако он не требует никакого оборудования на клиентской машине, кроме браузера. Это очень часто оказывается решающим аргументом. Обычного пользователя нельзя заставлять заниматься несвойственной ему работой. Ему надо предоставить привычную рабочую среду.

В *главе 2* мы описали действия участников системы дистанционного обучения СДО: ученика, учителя и администратора. Для обеспечения действий ученика по соображениям, изложенным в предыдущем абзаце, мы выбираем апплеты. Действия учителя могут быть обеспечены и апплетами, и приложениями. Для администратора мы разрабатываем приложение — консоль администратора СДО.

Действия ученика состоят из регистрации в СДО и учебного процесса. Их можно обеспечить двумя апплетами. В листинге 14.1 приведен апплет регистрации ученика. Для уменьшения длины листинга в него включен только необходимый для функционирования апплета код, не описан графический интерфейс, проверки, обработка исключений.

Листинг 14.1. Апплет регистрации ученика СДО

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.util.*;
import java.sql.*;

public class RegPrep extends JApplet{

   private int id;
   private JTextField tf1, tf2, tf3, tf4;
   private JTextArea ta;
   private JButton b1, b2;
```

```
private String fullname = "", name = "",
               secname = "", surname = "", addr = "";

private String driver = "oracle.jdbc.driver.OracleDriver",
   url = "jdbc:oracle:thin:@homexp:1521:SDO",
   user = "sdoadmin", password = "sdoadmin";

private Vector courses = new Vector();

private Statement st;

public void init(){

  driver = getParameter("driver");
  url = getParameter("url");
  user = getParameter("user");
  password = getParameter("password");

  JPanel p = new JPanel();

  setContentPane(p);

  p.setBackground(Color.white);
  p.setLayout(new GridBagLayout());

  GridBagConstraints gbc = new GridBagConstraints();
  gbc.insets = new Insets(3,3,3,3);

  JLabel l = new JLabel("Для регистрации занесите " +
     "сведения о себе");
  l.setFont(new Font("Serif", Font.PLAIN, 18));
  JLabel ll = new JLabel("в следующие поля:");

  ll.setFont(new Font("Serif", Font.PLAIN, 20));
  gbc.anchor = GridBagConstraints.WEST;
  gbc.gridwidth = 2;
  p.add(l, gbc);
  gbc.gridx = 0;
  p.add(ll, gbc);
```

```
JLabel l1 = new JLabel("Фамилия:", JLabel.RIGHT);
l1.setLabelFor(tf1);
gbc.anchor = GridBagConstraints.EAST;
gbc.gridwidth = 1;
gbc.gridx = 0;
p.add(l1, gbc);

JLabel l2 = new JLabel("Имя:", JLabel.RIGHT);
l2.setLabelFor(tf2);
gbc.gridx = 0;
p.add(l2, gbc);

JLabel l3 = new JLabel("Отчество:", JLabel.RIGHT);
l3.setLabelFor(tf3);
gbc.gridx = 0;
p.add(l3, gbc);

JLabel l4 = new JLabel("E-mail:", JLabel.RIGHT);
l4.setLabelFor(tf4);
gbc.gridx = 0;
p.add(l4, gbc);

JLabel l5 = new JLabel("Замечания:", JLabel.LEFT);
l5.setLabelFor(ta);
gbc.gridx = 0;
p.add(l5, gbc);

tf1 = new JTextField(30);
tf1.setMinimumSize(new Dimension(250, 25));
tf1.setPreferredSize(new Dimension(250, 25));
gbc.anchor = GridBagConstraints.CENTER;
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.gridx = 1;
p.add(tf1, gbc);

tf2 = new JTextField(30);
tf2.setMinimumSize(new Dimension(250, 25));
tf2.setPreferredSize(new Dimension(250, 25));
gbc.gridx = 1;
p.add(tf2, gbc);
```

```
    tf3 = new JTextField(30);
    tf3.setMinimumSize(new Dimension(250, 25));
    tf3.setPreferredSize(new Dimension(250, 25));
    gbc.gridx = 1;
    p.add(tf3, gbc);

    tf4 = new JTextField(30);
    tf4.setMinimumSize(new Dimension(250, 25));
    tf4.setPreferredSize(new Dimension(250, 25));
    gbc.gridx = 1;
    p.add(tf4, gbc);

    ta = new JTextArea(5, 50);
    ta.setMinimumSize(new Dimension(250, 25));
    ta.setPreferredSize(new Dimension(250, 125));
    ta.setEditable(false);
    gbc.gridx = 1;
    p.add(ta, gbc);

    JButton b1 = new JButton("Зарегистрировать");
    gbc.gridx = 1;
    gbc.fill = GridBagConstraints.NONE;
    p.add(b1, gbc);
    b1.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent ae){
          register();
      }
    });

    try{
        Class.forName(driver);
    }catch(Exception e){
       ta.append("From init exc: " + e + "\n");
    }
}

public void register(){
    surname = tf1.getText();
    name = tf2.getText();
```

```
secname = tf3.getText();
addr = tf4.getText();
if (surname.length() * name.length() *
    secname.length() * addr.length() == 0){

   JOptionPane.showMessageDialog(this,
       "Заполните, пожалуйста, все поля.",
       "Замечание",
       JOptionPane.ERROR_MESSAGE);
   return;
 }

fullname = surname.trim() + " " + name.trim() +
    " " + secname.trim();

try{
   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();

  st = con.createStatement();
  ResultSet rs =  st.executeQuery(
     "SELECT id, name FROM students ORDER BY id DESC");

  rs.next();

  id = rs.getInt(1);
  fullname = rs.getString(2);

  rs.close();

  courseSelect();
```

```
      con.close();

    }catch(Exception e){
       ta.append("Проверьте, пожалуйста, Ваши сведения.");
    }
 }

 public void courseSelect() throws SQLException{

    JPanel c1 = new JPanel();
    c1.setLayout(new GridBagLayout());
    c1.setBackground(Color.white);
    c1.setFont(new Font("Serif", Font.PLAIN, 16));

    GridBagConstraints gbc = new GridBagConstraints();
    gbc.insets = new Insets(3,3,3,3);

    StringTokenizer sttok = new StringTokenizer(fullname);
    sttok.nextToken();
    name = sttok.nextToken();
    secname = sttok.nextToken();
    JLabel l6 = new JLabel("Добро пожаловать, " + name + " " +
        secname + "!");
    gbc.anchor = GridBagConstraints.WEST;
    gbc.gridx = 0; c1.add(l6, gbc);
    JLabel l66 = new JLabel("Вы зарегистрированы в СДО.");
    gbc.anchor = GridBagConstraints.WEST;
    gbc.gridx = 0; c1.add(l66, gbc);

    JLabel l7 = new JLabel("Ваш регистрационный номер  " + id);
    gbc.gridx = 0; c1.add(l7, gbc);

    JLabel l8 = new JLabel("Выберите учебный курс: ");
    gbc.gridx = 0; c1.add(l8, gbc);

    ResultSet rs =
       st.executeQuery("SELECT course_name FROM courses");

    while (rs.next()) courses.addElement(rs.getString(1));
```

```
        rs.close();

        JList list = new JList(courses);
        list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        list.addListSelectionListener(new ListSelectionListener(){
           public void valueChanged(ListSelectionEvent e){
              getCourse(e.getFirstIndex());
           }
         });

        JScrollPane sp = new JScrollPane(list);
        gbc.gridx = 0;
        c1.add(sp, gbc);

        setContentPane(c1);
    }
}
```

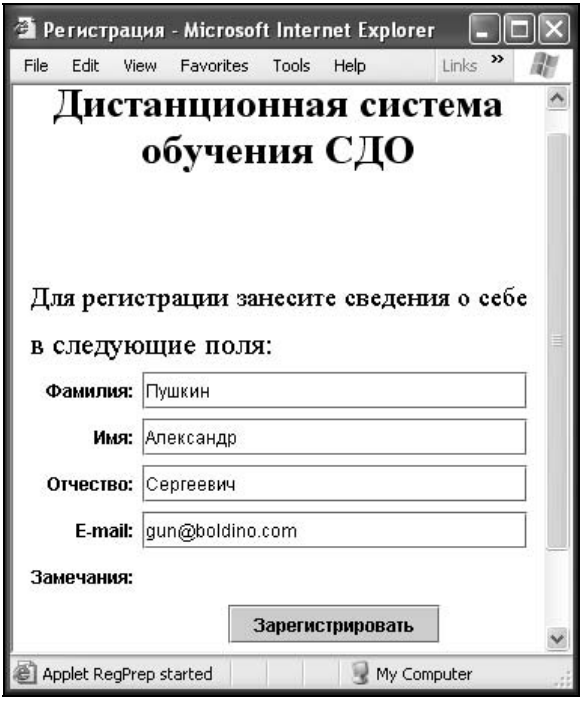


Рис. 14.1. Страница регистрации

При использовании апплета для загрузки драйвера JDBC четвертого типа Oracle Thin надо в каталог апплета поместить архивы classes12.zip и nls_charset12.zip, входящие в стандартную поставку Oracle Server. Можно упаковать их в JAR-архив вместе с апплетом и загружать в браузер весь архив. Подробности упаковки можно посмотреть в книге [15].

На рис. 14.1 показано окно регистрации ученика, выводимое методом `init()` апплета, а на рис. 14.2 — окно подтверждения регистрации и выбора курса, выводимое методом `courseSelect()`.

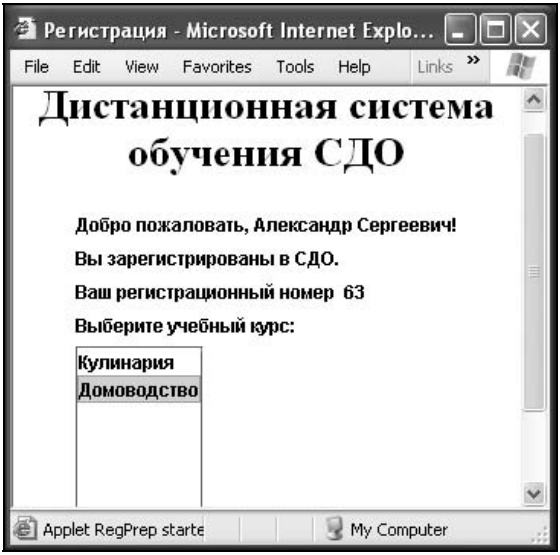


Рис. 14.2. Страница подтверждения регистрации

В листинге 14.2 приведен код HTML-файла, загружающего апплет регистрации ученика СДО.

Листинг 14.2. Файл HTML, загружающий апплет регистрации ученика СДО

```
<html>
  <head>
    <title>Регистрация</title>
  </head>
  <body>
    <h2 align="center">Дистанционная система обучения СДО</h2>
```

```
    <applet code="RegPrep.class" codebase="."
            width=100% height=100%>
      <param name=driver   value="oracle.jdbc.driver.OracleDriver">
      <param name=url      value="jdbc:oracle:thin:@homexp:1521:SDO">
      <param name=user     value="sdoadmin">
      <param name=password value="sdoadmin">
    </applet>
  </body>
</html>
```

Часть IV

Сетевая часть приложения Java

Глава 15

Обращение к удаленным методам

Как мы видели в предыдущей части книги, технология Java позволяет легко связаться с базой данных по сети, используя интерфейс JDBC. При этом вся обработка полученной из базы информации, как говорят, бизнес-логика, выполняется клиентом. Интерфейс JDBC позволяет, конечно, выполнять процедуры, хранимые в базе данных, в том числе и процедуры, написанные на языке Java. Но иногда возникает необходимость выполнять процедуры и функции, находящиеся на сервере, но не в составе базы данных.

Для выполнения удаленных процедур и функций на сервере и пересылки результатов клиенту давно разработана технология удаленного вызова процедур *RPC* (Remote Procedure Call). Цель этой технологии — спрятать внутри механизма RPC процесс связи с сервером и последовательность сетевого обмена, создав для клиента иллюзию вызова локальной процедуры. Для этого на клиентской машине создается *заглушка* (stub). Это процедура, имитирующая интерфейс вызываемой процедуры. Вместо тела вызываемой процедуры заглушка содержит команды связи с сервером и передачи ему всей информации, необходимой для выполнения процедуры. Процесс вызова удаленной процедуры происходит так.

1. Клиент вызывает заглушку и передает ей входные аргументы процедуры.
2. Заглушка собирает аргументы в сетевые пакеты, возможно, преобразуя их в свой формат. Этот процесс называется *сборкой* (marshalling). Затем заглушка устанавливает связь с сервером (binding) и пересылает ему собранные пакеты.
3. На сервере имеется *каркас* (skeleton) — программа, принимающая запрос от заглушки клиента. Каркас *разбирает* полученные пакеты, выполняет *обратное преобразование аргументов* (unmarshalling) и вызывает процедуру, передавая ей входные аргументы.
4. Каркас дожидается выполнения процедуры, получает ее выходные параметры, собирает их (marshalling) и посылает заглушке на машину клиента.
5. Заглушка разбирает полученные результаты (unmarshalling) и передает их вызывающей программе.

Итак, с точки зрения клиентской программы происходит обычный вызов локальной процедуры. С точки зрения сервера тоже обычным образом вызывается и выполняется процедура. Все сетевое взаимодействие сосредоточено в заглушке и каркасе, которые предоставляют стандартный интерфейс, как для платформы клиента, так и для платформы сервера. При этом обычно используются транспортные протоколы TCP или UDP, но это необязательно. Технология RPC может работать и по другим протоколам, даже специально разработанным для этой цели.

Технология RPC успешно применяется во многих распределенных системах, например, в сетевой файловой системе NFS (Network File System). Естественно, ее решили применить в технологии Java. Но весь механизм RPC рассчитан на вызов отдельных самостоятельных процедур, а методы Java выполняются только объектом, содержащим их. Необходимость создания объектов для выполнения методов Java заставила переработать механизм RPC. В результате возник механизм удаленного вызова методов RMI (Remote Method Invocation).

Структура системы RMI

Система удаленного выполнения методов RMI строится по архитектуре клиент-сервер. *Удаленный объект* (remote object), называемый также *распределенным объектом* (distributed object), находится на сервере, обращение к его методам осуществляет клиент. Клиент и сервер могут находиться на одной машине, важно то, что они выполняются разными виртуальными машинами Java.

Для того чтобы клиент мог найти удаленный объект, на сервере, создается *RMI-реестр* (RMI registry). Это небольшая база данных, хранящая список удаленных объектов, находящихся на этом сервере. Каждый удаленный объект зарегистрирован в RMI-реестре под каким-то уникальным именем. Для работы с базой данных создается серверный процесс, тоже входящий в понятие RMI-реестра. Этот процесс работает по протоколу TCP, он прослушивает по умолчанию порт 1099 и ждет запросы от клиента.

RMI-реестр — это тоже удаленный объект, имеющий общеизвестное местоположение.

Обращение к методам удаленного объекта происходит по следующей схеме.

1. Удаленный объект создается на сервере, регистрируется в RMI-реестре, находящемся на том же сервере, и ждет запрос от клиента на выполнение своих методов. Кроме того, на сервере создается и хранится *заглушка* (stub) объекта.
2. RMI-реестр прослушивает порт, по умолчанию 1099, и ждет запрос от клиента.

3. Клиент, желающий обратиться к методам удаленного объекта, запрашивает у реестра объект по имени, зарегистрированному в реестре.
4. Реестр отыскивает заглушку объекта (stub), и пересылает ее клиенту. В заглушке, в частности, содержится *местоположение* (codebase) удаленного объекта.
5. Заглушка запрашивает у сервера описание методов удаленного объекта, указав его местоположение.
6. Сервер составляет список сигнатур методов удаленного объекта и отправляет их заглушке.
7. Клиент обращается к методам заглушки так, как будто это методы самого объекта.
8. Если аргументы метода — локальные объекты, то заглушка сериализует их и пересылает на сервер. Если аргументы — удаленные объекты, то пересылается ссылка на них.
9. Сервер десериализует параметры, передает их методу удаленного объекта, метод выполняется, результат сериализуется и пересылается заглушке. Та десериализует результат и передает клиенту.

Таким образом, для клиентской программы работа с удаленным объектом идет обычным образом: создается экземпляр удаленного объекта (вместо чего пересылается заглушка), и выполняются его методы. Все остальное — связь с сервером, поиск удаленного объекта, пересылку аргументов метода серверу и результатов его выполнения обратно клиенту — берет на себя система RMI. Понятно, что механизм RMI не может быть простым. И действительно, интерфейсы, и классы, обеспечивающие работу RMI, составляют целых пять пакетов.

В пакете java.rmi находятся самые общие классы: интерфейс `Remote`, классы `MarshalledObject`, `Naming`, `RMISecurityManager` и классы исключений, возникающих при работе системы RMI.

Удаленный объект, вместе с другими объектами, обеспечивающими его работу и связь с ним, образует так называемый *RMI-сервер*. Интерфейсы и классы этих объектов собраны в пакет java.rmi.server.

RMI-сервер обычно существует все время работы системы RMI. Однако можно построить работу RMI так, что RMI-сервер будет активизироваться только на время обращения к его методам. Активизацию выполняют объекты классов, собранных в пакет java.rmi.activation.

Связь клиента с удаленным объектом происходит через RMI-реестр. Его интерфейсы и классы образуют пакет java.rmi.registry.

Распределенный характер работы RMI усложняет "сборку мусора". Для очистки памяти в RMI разработана специальная система *DGC* (Distributed Garbage Collection). Интерфейсы и классы этой системы составляют пакет java.rmi.dgc.

Создание удаленного объекта

Удаленный объект системы RMI — это экземпляр класса Java. Как правило, этот класс реализует интерфейс, описывающий те его методы, которые предоставляются клиенту. Такой интерфейс называется *удаленным интерфейсом* (remote interface). Он необходим потому, что заглушка, передаваемая клиенту, содержит реализацию только тех методов, которые описаны в удаленном интерфейсе.

Удаленный интерфейс

Удаленный интерфейс — это всякий открытый (public) интерфейс Java, расширяющий интерфейс `Remote`. Интерфейс `Remote` не содержит ни одного метода. Поэтому надо только написать заголовок

```
public interface MyRemote extends Remote{
```

и описывать константы и методы удаленного объекта. Такое "расширение" служит лишь пометкой, относящей создаваемый интерфейс к типу удаленных объектов.

Удаленный интерфейс обязательно должен быть открытым, `public`, чтобы его "видели" удаленные клиенты.

Третья особенность удаленного интерфейса заключается в том, что каждый метод удаленного интерфейса должен выбрасывать исключение класса `RemoteException` или его суперкласса. Непосредственным суперклассом класса `RemoteException` служит класс `IOException`, расширяющий класс `Exception`.

Четвертая особенность — аргументы методов или возвращаемые значения, являющиеся удаленными объектами, должны описываться типом своего удаленного интерфейса, а не реализующего его класса.

Итак, с учетом этих четырех особенностей удаленный интерфейс выглядит примерно так, как записано в листинге 15.1.

Листинг 15.1. Удаленный интерфейс

```
import java.rmi.*;

public interface MyRemote extends Remote{

   String getName() throws RemoteException;
   void setName(String name) throws RemoteException;
}
```

Удаленный объект

Почему же удаленный объект часто строится как расширение удаленного интерфейса? Дело в том, что если удаленный объект передается как аргумент или возвращаемое значение некоторого метода, то в этом методе следует указывать тип интерфейса удаленного объекта, а не реализующего его класса, поскольку клиенту передаются только интерфейсы, а не классы.

Как бы то ни было, удаленный объект обязательно должен реализовать хотя бы один удаленный интерфейс, чтобы быть помеченным в системе RMI.

Кроме того, методы `hashCode()`, `equals()`, `toString()`, обязательные для каждого объекта Java и определенные в классе `Object`, должны учитывать то обстоятельство, что все клиентские заглушки одного и того же удаленного объекта должны иметь один и тот же хэш-код и возвращать `true` при их сравнении. Эти особенности учтены в абстрактном классе `RemoteObject`, расширяющем класс `Object` и переопределяющем эти методы. При переопределении класс `RemoteObject` реализует методы `remoteHashCode()`, `remoteEquals(RemoteRef)`, `remoteToString()` интерфейса `RemoteRef`.

Интерфейс `RemoteRef` описывает также очень важный метод

```
public Object invoke(Remote obj, Method method,
    Object[] args, long hash);
```

который как раз и вызывает удаленный метод `method` удаленного объекта с заглушкой `obj`, передает методу аргументы `args` и возвращает результат его выполнения.

Класс `RemoteObject` не реализует интерфейс `RemoteRef` явно, а использует его в своей внутренней реализации как защищенное (protected) поле `ref`.

Класс `RemoteObject` служит суперклассом всех заглушек и удаленных объектов, используемых в системе RMI. Для создания удаленных объектов предназначено его расширение `RemoteServer`. Это тоже абстрактный класс, он не используется сам по себе, а служит суперклассом для создания классов удаленных объектов.

В пакете java.rmi.server есть класс `UnicastRemoteObject`, расширяющий класс `RemoteServer`. Вот этот класс и служит основой для создания удаленных объектов. Экземпляр этого класса существует все время работы той виртуальной машины Java, которая его создала. Он всегда готов отвечать на запросы клиента и совместно используется всеми заглушками, обращающимися к образованному им удаленному объекту. Для связи с заглушками, находящимися на машине клиента, объект класса `UnicastRemoteObject` создает сокеты потокового типа, работающие по протоколу TCP. Номер порта, который прослушивают эти сокеты, определяется произвольно во время выполнения, но его можно задать конструктором

```
UnicastRemoteObject(int port);
```

Кроме того, класс `UnicastRemoteObject` реализует важную операцию — *экспорт* (export) удаленного объекта. Эта операция регистрирует удаленный объект в системе RMI и делает его видимым и доступным для клиентов. Экспорт осуществляется автоматически конструктором класса `UnicastRemoteObject`, который для этого обращается к одному из методов `exportObject()`, определенных в том же классе.

Методы `exportObject()` определены как статические методы, их следует применять для явного экспорта удаленных объектов, если класс удаленного объекта не расширяет класс `UnicastRemoteObject`.

Таким образом, для создания удаленного объекта легче всего расширить класс `UnicastRemoteObject`. При этом в классе удаленного объекта обязательно должен быть конструктор, выбрасывающий исключение класса `RemoteException` или его суперкласса, поскольку его выбрасывает метод `exportObject()`. Непосредственным суперклассом класса `RemoteException` служит класс `IOException`, расширяющий класс `Exception`.

С учетом всего сказанного выше, реализация удаленного объекта выглядит примерно так, как написано в листинге 15.2, реализующем удаленный интерфейс листинга 15.1.

Листинг 15.2. Удаленный объект

```
import java.rmi.*;

public class MyRemoteImpl extends UnicastRemoteObject
                                    implements MyRemote{
   private String name;

   public String getName() throws RemoteException{
      return name;
   }

   public void setName(String name) throws RemoteException{
      this.name = name;
   }

   public MyRemoteImpl() throws RemoteException{
      super(); // Необязательно, мы просто подчеркиваем, что
               // должен выполняться конструктор суперкласса.
   }
```

```
   public MyRemoteImpl(String name) throws RemoteException{
      this.name = name;
   }
}
```

Создание RMI-сервера

После того как написан класс удаленного объекта, можно создавать его экземпляры — удаленные объекты. Программа, создавшая один или несколько удаленных объектов, называется в системе удаленных вызовов *RMI-сервером*. Эта программа должна сначала проверить, установлен ли менеджер безопасности класса `RMISecurityManager` или другого класса, расширяющего класс `SecurityManager`, и если нет, то установить его, а после создания удаленного объекта должна зарегистрировать его в RMI-реестре. Все это можно описать в методе `main()` класса удаленного объекта, превратив его в RMI-сервер, а можно написать отдельный класс, как, например, в листинге 15.3, где создается RMI-сервер для удаленного объекта листинга 15.2.

Листинг 15.3. RMI-сервер

```
import java.rmi.*;
import java.rmi.server.*;

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

      if (System.getSecurityManager() == null)
        System.setSecurityManager(
             new RMISecurityManager());

      try{
         MyRemoteImpl mri = new MyRemoteImpl();

         Naming.rebind("serverName", mri);

     }catch(Exception e){}
   }
}
```

Менеджер безопасности обязательно участвует в процессе загрузки классов Java и должен быть установлен не только в распределенной системе, но и во

всех программах, в которых загружаются классы. Браузер всегда устанавливает его при загрузке апплета, но в приложениях следует делать это самостоятельно.

Регистрация удаленного объекта `obj` в RMI-реестре производится одним из двух статических методов

```
public static void bind(String url, Remote obj);
public static void rebind(String url, Remote obj);
```

класса `Naming`. Разница между этими методами заключается в том, что метод `bind()` выбрасывает исключение класса `AlreadyBoundException`, если имя `url` уже связано с каким-то объектом, а метод `rebind()` просто заменяет объект.

В небольшом классе `Naming` собраны статические методы связи удаленных объектов и их клиентов с системой вызова удаленных объектов RMI. Кроме написанных выше методов в нем есть метод

```
public static void unbind(String url);
```

удаляющий имя `url` из RMI-реестра, метод

```
public static String[] list(String url);
```

выдающий список всех зарегистрированных RMI-серверов, и метод

```
public static remote lookup(String url);
```

используемый клиентом для поиска RMI-сервера.

Строка адреса `url` во всех этих методах строится как адрес URL, но со своими особенностями. Она имеет вид

```
"//hostname:port/serverName"
```

В ней не указывается схема (подразумевается `rmi:`). После двух слэшей записывается DNS-имя или IP-адрес серверной машины. Если часть `hostname` адреса опущена, то подразумевается `localhost` или 127.0.0.1. Часть адреса `port` — это номер порта, который прослушивает RMI-реестр, по умолчанию 1099. Последняя часть адреса `serverName` — это имя RMI-сервера, оно задается произвольно, и никак не связано с именем класса.

Создание RMI-клиента

Клиентское приложение, желающее обратиться к методам удаленного объекта, должно выполнить поиск удаленного объекта и связь с ним. Это делается по имени, зарегистрированному в RMI-реестре, всего одним статическим методом `lookup(String)` класса `Naming` примерно так:

```
try{
   MyRemote mr = (MyRemote)Naming.lookup(
```

```
        "//hostname:port/serverName");
}catch(Exception e){}
```

Аргумент метода `lookup(String)` — это адрес удаленного объекта в виде строки URL, заданной по правилам, описанным в предыдущем абзаце. Обратите внимание на то, что при создании объекта `mr` указывается его интерфейс `MyRemote`, а не класс реализации `MyRemoteImpl`.

Теперь можно обращаться к методам полученной заглушки `mr` как к методам обычного объекта. Простейшая клиентская программа может выглядеть так, как в листинге 15.4. Здесь RMI-клиент обращается к удаленному объекту листинга 15.2 через удаленный интерфейс листинга 15.1.

Листинг 15.4. Клиент системы RMI

```
import java.rmi.*;

public class MyClient{

   public static void main(String[] args){

     if (System.getSecurityManager() == null)
        System.setSecurityManager(
            new RMISecurityManager());

      try{
         MyRemote mr = (MyRemote)Naming.lookup(
              "//hostname/serverName");
         mr.setName("Иванов");
         System.out.println("Фамилия: " + mr.getName());
      }catch(Exception e){
         System.out.println("Ошибка RMI: " + e);
      }
   }
}
```

Компиляция и запуск системы RMI

Все исходные файлы для системы RMI созданы. Надо их откомпилировать, сначала те файлы, которые должны располагаться на сервере

```
cd myrmifiles
javac *.java
```

После этого на сервере в каталоге myrmifiles появятся файлы MyRemote.class и MyRemoteImpl.class.

Следующий шаг — создать на сервере класс заглушки. Для этого на сервере запускаем специальный RMI-компилятор — утилиту rmic, входящую в стандартную поставку J2SE SDK и находящуюся в каталоге $JAVA_HOME/bin/:

```
rmic MyRemoteImpl
```

Эта утилита создает класс заглушки, хранящийся в файле с именем MyRemoteImpl_Stub.class, и файл MyRemoteImpl_Skel.class. Второй файл с классом — диспетчером запросов (skeleton) не нужен в Java 2, он остается в системе RMI для совместимости со старыми версиями.

Если вы хотите посмотреть исходный код заглушки на языке Java, то запустите компилятор `rmic` с ключом `-keep`:

```
rmic -keep MyRemoteImpl
```

Компилятор оставит исходные файлы MyRemoteImpl_Stub.java и MyRemoteImpl_Skel.java в том же каталоге. В листинге 15.5 приведен файл MyRemoteImpl_Stub.java, созданный компилятором `rmic`.

Листинг 15.5. Заглушка

```
// Stub class generated by rmic, do not edit.
// Contents subject to change without notice.

public final class MyRemoteImpl_Stub
          extends java.rmi.server.RemoteStub
             implements MyRemote, java.rmi.Remote{
   private static final java.rmi.server.Operation[] operations = {
      new java.rmi.server.Operation("java.lang.String getName()"),
      new java.rmi.server.Operation("void setName(java.lang.String)")
   };

   private static final long interfaceHash = 5625545834854829872L;

   private static final long serialVersionUID = 2;

   private static boolean useNewInvoke;
   private static java.lang.reflect.Method $method_getName_0;
   private static java.lang.reflect.Method $method_setName_1;

   static {
     try {
```

```
    java.rmi.server.RemoteRef.class.getMethod("invoke",
       new java.lang.Class[] {
          java.rmi.Remote.class,
          java.lang.reflect.Method.class,
          java.lang.Object[].class,
          long.class
       });
     useNewInvoke = true;
     $method_getName_0 = MyRemote.class.getMethod("getName",
          new java.lang.Class[] {});
     $method_setName_1 = MyRemote.class.getMethod("setName",
          new java.lang.Class[] {java.lang.String.class});
  } catch (java.lang.NoSuchMethodException e) {
     useNewInvoke = false;
 }
}

// constructors
public MyRemoteImpl_Stub() {
  super();
}
public MyRemoteImpl_Stub(java.rmi.server.RemoteRef ref) {
  super(ref);
}

// methods from remote interfaces

// implementation of getName()
public java.lang.String getName()
          throws java.rmi.RemoteException {
 try {
       if (useNewInvoke) {
         Object $result = ref.invoke(this, $method_getName_0,
              null, 6317137956467216454L);
         return ((java.lang.String) $result);
       } else {
         java.rmi.server.RemoteCall call =
             ref.newCall((java.rmi.server.RemoteObject) this,
                    operations, 0, interfaceHash);
```

```
            ref.invoke(call);
            java.lang.String $result;
            try {
               java.io.ObjectInput in = call.getInputStream();
               $result = (java.lang.String) in.readObject();
            } catch (java.io.IOException e) {
                 throw new java.rmi.UnmarshalException(
                    "error unmarshalling return", e);
            } catch (java.lang.ClassNotFoundException e) {
                 throw new java.rmi.UnmarshalException(
                    "error unmarshalling return", e);
            } finally {
                 ref.done(call);
            }
            return $result;
         }
      } catch (java.lang.RuntimeException e) {
            throw e;
      } catch (java.rmi.RemoteException e) {
            throw e;
      } catch (java.lang.Exception e) {
            throw new java.rmi.UnexpectedException(
              "undeclared checked exception", e);
      }
   }

   // implementation of setName(String)
   public void setName(java.lang.String $param_String_1)
                           throws java.rmi.RemoteException {
     try {
        if (useNewInvoke) {
           ref.invoke(this, $method_setName_1,
               new java.lang.Object[] {$param_String_1},
               1344297395548290975L);
        } else {
           java.rmi.server.RemoteCall call =
              ref.newCall((java.rmi.server.RemoteObject) this,
                    operations, 1, interfaceHash);
```

```
            try {
               java.io.ObjectOutput out = call.getOutputStream();
               out.writeObject($param_String_1);
            } catch (java.io.IOException e) {
                throw new java.rmi.MarshalException(
                    "error marshalling arguments", e);
            }
            ref.invoke(call);
            ref.done(call);
         }
      } catch (java.lang.RuntimeException e) {
          throw e;
      } catch (java.rmi.RemoteException e) {
          throw e;
      } catch (java.lang.Exception e) {
          throw new java.rmi.UnexpectedException(
             "undeclared checked exception", e);
      }
   }
}
```

Теперь все файлы откомпилированы и готовы к выполнению. Далее на сервере надо запустить RMI-реестр утилитой rmiregistry. В UNIX реестр запускается в фоновом режиме

```
# rmiregistry&
```

В Win 32 утилита запускается командой

```
start rmiregistry
```

или

```
javaw rmiregistry
```

У утилиты rmiregistry есть один параметр — номер порта, который прослушивает RMI-реестр, например:

```
# rmiregistry 1234&
```

По умолчанию номер порта 1099.

На сервере остается запустить RMI-сервер

```
java -Djava.server.codebase=http://hostname/myrmifiles/
     -Djava.security.policy=$HOME/policy     MyRemoteServer
```

При запуске устанавливаются два системных свойства.

Системное свойство `java.server.codebase` содержит один или несколько адресов заглушек и классов, необходимых для их работы. Адреса записываются в форме URL. Они могут указывать каталоги или JAR-архивы и отделяются друг от друга пробелами, поэтому в других местах — после буквы `D`, до и после знака равенства, — пробелов быть не должно. Если адрес указывает каталог, то в конце его обязательно должен стоять слэш.

При поиске заглушки сервер сначала просматривает свои каталоги и JAR-архивы, руководствуясь своей переменной `CLASSPATH`. Если поиск не дал результатов, то сервер обращается к значению системного свойства `java.server.codebase`. При этом, возможно, серверу придется обратиться к удаленному объекту и стать на время клиентом системы RMI.

Если системное свойство `java.server.codebase` не определено (у него нет значения по умолчанию), то сервер не имеет возможности обращаться к удаленным объектам.

Второе системное свойство `java.security.policy` задает местоположение файла, содержащего права доступа к ресурсам сервера (policy file). Если это свойство не определено, то по умолчанию используется файл $JAVA_HOME/jre/lib/security/java.policy, который создается при инсталляции J2SE SDK и не содержит ограничений — всем предоставляются все права.

На клиентской машине компилируем и запускаем обычным образом программу клиента:

```
javac MyClient.java
java MyClient
```

После этого метод `lookup(String)` клиента открывает сокет потокового типа и устанавливает связь с RMI-реестром по протоколу TCP через порт номер 1099. RMI-реестр находит заглушку удаленного объекта и пересылает ее клиенту. Клиент через заглушку передает RMI-серверу сериализованный аргумент "Иванов" удаленного метода `setName(String)`. Удаленный объект выполняет метод. Затем клиент обращается к удаленному методу `getName()`. Сервер выполняет этот метод и передает сериализованный результат заглушке клиента. Заглушка десериализует результат и передает его клиентской программе.

Аргументы и результат удаленного метода

При вызове метода удаленного объекта клиент задает его аргументы. Поскольку заглушка сериализует аргументы метода перед передачей их на сервер, аргументы должны быть простого типа Java или реализовывать интерфейс `Serializable`.

Еще одна возможность — использовать в качестве аргументов удаленные объекты, т. е. аргументы типа `Remote`. В этом случае на RMI-сервер передается ссылка на удаленный объект, которая на сервере преобразуется в заглушку. После этого RMI-сервер начинает играть роль клиента, работающего с заглушкой удаленного объекта.

Во всех остальных случаях выбрасывается исключение класса `MarshalException`.

Итак, аргументы простого или сериализуемого типа `Serializable` передаются на сервер по значению, а аргументы — удаленные объекты типа `Remote` передаются по ссылке. Передачу по ссылке можно использовать для изменения аргументов удаленным методом.

Те же способы передачи аргументов относятся и к результату, возвращаемому методом, но здесь добавляется еще одна возможность — метод может вернуть исключение. Оно сериализуется и передается как обычный объект.

Получив аргументы, RMI-сервер должен узнать определение их классов.

Различают три типа полученных аргументов: простые типы Java, объекты классов, которые RMI-сервер может найти с помощью своей переменной `CLASSPATH`, и остальные объекты.

Для первых двух типов определение класса отыскивается обычным образом, так же, как для всех объектов Java. Для третьего типа все определяется типом загрузчика классов.

Загрузчики классов `RMIClassLoader` или `AppletClassLoader` передают серверу вместе с пересылаемым объектом его местоположение, из которого сервер может узнать определение класса. Но если пересылаемый объект загружался обычным загрузчиком `ClassLoader`, то его местоположение неизвестно. В этом случае определение класса можно узнать только по его местоположению, указанному в системном свойстве `java.rmi.server.codebase`.

Протоколы системы RMI

Система удаленного выполнения методов RMI может использовать разные протоколы транспортного уровня, работающие по протоколу сетевого уровня IP. Основным протоколом служит TCP, но можно организовать передачу информации по протоколу UDP. Если система RMI функционирует совместно с CORBA, то используется протокол IIOP, тоже работающий поверх протокола TCP.

Кроме этих протоколов, в тех случаях, когда proxy-серверы не позволяют соединяться с сервером через сокеты, используется передача данных через протокол HTTP (tunneling).

Протокол JRMP

Как уже говорилось выше, система RMI создает для передачи данных сокеты потокового типа, работающие по протоколу TCP. Данные пересылаются по специально разработанному протоколу *JRMP* (Java Remote Method Protocol) в одном или нескольких пакетов, состоящих из *заголовка* и одного или нескольких *сообщений*.

Заголовок пакета начинается с четырех байтов 0x4a, 0x52, 0x4d, 0x49, образующих в кодировке ASCII слово "JRMI". Затем идут два байта с номером версии, сейчас это 0x00, 0x01. Завершает заголовок байт, называемый Protocol. Он определяет тип соединения и число сообщений, входящих в пакет. Это одно из трех чисел:

- 0x4c — SingleOpProtocol — в пакете только одно сообщение, которое идет непосредственно за заголовком. Такой пакет характерен для передачи информации, встроенной в протокол HTTP. Он не требует ответа от сервера.
- 0x4b — StreamProtocol — в пакете одно или несколько сообщений, передаваемых с помощью сокетов.
- 0x4d — MultiplexProtocol — в пакете одно или несколько сообщений, передаваемых в рамках одного соединения. Для передачи организуются виртуальные соединения. Такой пакет создается в том случае, когда клиент не может создать серверный сокет для приема входящих запросов, что характерно для апплетов.

В ответ на заголовки типа StreamProtocol и MultiplexProtocol сервер посылает подтверждение — байт 0x4e и так называемый EndpointIdentifier, содержащий DNS-адрес сервера и номер порта. Клиент в ответ на это обязан послать серверу EndpointIdentifier со своим адресом. После этого посылаются сообщения.

Сообщения, отправляемые клиентом, бывают трех типов:

- 0x50 — Call — выполнение метода. Далее идет идентификатор метода и сериализованные аргументы метода;
- 0x52— Ping — проверка серверного сокета;
- 0x54 — DgcAck — подтверждение для распределенной системы очистки памяти.

Перед сообщениями, отправляемыми клиенту, посылается один из двух байтов:

- 0x4f — ProtocolNotSupported — неверный протокол;
- 0x4e — ProtocolAck — протокол верный, далее идет результат выполнения метода.

Перед результатом выполнения метода посылается байт 0x51 — ReturnData.

В ответ на посылку байта 0x52 — Ping, посылается байт 0x53 — PingAck — байт подтверждения проверки серверного сокета.

RMI поверх HTTP

В тех случаях, когда proxy-сервер, установленный на сервере, не допускает прямого соединения через сокеты, клиентский сокет анализирует исключение, возникшее в результате попытки соединения. Если возникло исключение класса `NoRouteToHostException` или `UnknownHostException`, то клиентский сокет пытается установить соединение с сервером по протоколу HTTP методом `POST`. Конечно, на серверной машине должен работать HTTP-сервер.

Если proxy-сервер допускает обращение по любому порту, то соединение происходит по адресу

```
http://hostname:port/
```

Здесь указывается DNS-имя сервера и номер порта, прослушиваемого RMI-сервером. Серверный сокет RMI-сервера самостоятельно получает HTTP-запрос и извлекает RMI-пакет из `POST`-сообщения.

Если proxy-сервер не допускает обращения по произвольному порту, то для соединения клиентский сокет задает адрес

```
http://hostname:80/cgi-bin/java-rmi?forward=port
```

В этом случае работа происходит через CGI-скрипт `cgi-bin.cgi`, входящий в стандартную поставку J2SE SDK. Этот скрипт принимает HTTP-запрос и передает его серверному сокету, прослушивающему порт, указанный параметром `forward`.

Если серверный сокет получил запрос по протоколу HTTP, то свой ответ он упаковывает тоже по правилам протокола HTTP.

Можно запретить передачу информации по протоколу HTTP, установив значение системной переменной `java.rmi.server.disableHttp` равным `true`. По умолчанию это значение равно `false`.

Активизация удаленных объектов

Удаленный объект класса `UnicastRemoteObject` или его подкласса существует все время, пока работает создавший его процесс. Это хорошо, поскольку он всегда готов выполнять запрос, но он постоянно занимает ресурсы сервера. Кроме того, он перестает существовать, когда породивший его процесс завершается, что может произойти аварийно.

Система RMI позволяет сформировать удаленный объект, который создается при обращении к нему клиента и перестает существовать после выполнения запроса. Создание объекта и его экспорт производятся демоном активизации, прослушивающим по умолчанию порт 1098. Демон активизации запус-

кается утилитой rmid. При запуске, как правило, указывается файл rmid.policy с правами доступа демона к ресурсам машины:

```
rmid -J-Djava.security.policy=rmid.policy
```

Демон активизации в начале своей работы создает объект, реализующий интерфейс `Activator`. Этот объект выполняет метод `activate(ActivationID id, boolean force)`, с помощью которого демон находит активизируемый объект по его идентификатору `id` — объекту класса `ActivationID` — и активизирует найденный объект.

Активизируемый удаленный объект обычно расширяет абстрактный класс `Activatable` из пакета java.rmi.activation, который, в свою очередь, расширяет класс `RemoteServer`. В классе активизируемого удаленного объекта обязательно должен быть конструктор с двумя аргументами: один аргумент класса `ActivationID`, содержащий идентификатор объекта, второй аргумент класса `MarshalledObject`, содержащий передаваемые объекту данные в сериализованном виде. Этот конструктор вызывается при активизации объекта.

Конструктор активизируемого удаленного объекта обычно вызывает конструктор суперкласса, т. е. класса `Activatable`. В этом классе четыре конструктора, простейший из которых имеет вид

```
protected Activatable(ActivationID id, int port);
```

В нем задается идентификатор `id` активизируемого удаленного объекта и номер `port` порта, который прослушивается удаленным объектом. Если дать второму аргументу значение 0, то номер порта будет назначаться динамически во время обращения к объекту. Идентификатор объекта создается подсистемой активизации до создания объекта и передается конструктору. Конструктор регистрирует созданный объект в подсистеме активизации и экспортирует его.

Если не расширять класс `Activatable`, а создавать активизируемый удаленный объект как независимый класс, то регистрацию объекта в подсистеме активизации и его экспорт придется выполнять явно, статическим методом `register(ActivationDesc)` и/или одним из статических методов `exportObject()`.

С учетом сказанного активизируемый удаленный объект выглядит примерно так, как показано в листинге 15.6, в котором реализован удаленный интерфейс листинга 15.1 в виде активизируемого удаленного объекта.

Листинг 15.6. Активизируемый удаленный объект

```
import java.rmi.*;
import java.rmi.activation.*;
```

```
public class MyRemoteImpl extends Activatable
                              implements MyRemote{
   private String name;

   public String getName() throws RemoteException{
      return name;
   }

   public void setName(String name) throws RemoteException{
      this.name = name;
   }

   public MyRemoteImpl(ActivationID id, MarshalledObject obj)
            throws RemoteException{
      super(id, 0);

      // Если бы этот класс не расширял класс Activatable,
      // то вместо вызова конструктора суперкласса следовало
      // бы написать следующее:
      // Activatable.exportObject(this, id, 0);
   }
}
```

Активизируемый удаленный объект описывается классом `ActivationDesc`, в котором хранится информация, необходимая для активизации объекта: идентификатор группы, в которую входит объект, полное имя класса объекта, местоположение класса объекта, режим перезапуска объекта и сериализованные начальные данные объекта. Конструктор класса `Activatable` автоматически создает экземпляр класса `ActivationDesc`, используемый затем системой активизации объектов. Можно создать экземпляр класса и явно одним из четырех конструкторов, простейший из которых имеет вид

```
public ActivationDesc(String className,
   String location, MarshalledObject data);
```

Он создает в текущей группе описание объекта, активизируемого только по требованию. В листинге 15.7 применен такой конструктор для класса, выполняющего регистрацию удаленного объекта листинга 15.6 в системе активизации.

Листинг 15.7. Регистрация активизируемого удаленного объекта

```
import java.rmi.*;
import java.rmi.activation.*;

public class MyRemoteReg{

   public static void main(String[] args){

      if (System.getSecurityManager() == null)
        System.setSecurityManager(
             new RMISecurityManager());

      try{
         ActivationDesc ad = new ActivationDesc(
           "MyRemoteImpl", ".", null);

         MyRemoteImpl mri =
           (MyRemoteImpl)Activatable.register(ad);

         Naming.rebind("serverName", mri);

      }catch(Exception e){}
   }
}
```

Все исходные файлы готовы. Удаленный интерфейс, описанный в листинге 15.1, и клиент, описанный в листинге 15.4, остаются без изменения. Клиент даже не знает, каким образом активизируется вызываемый им удаленный объект.

Теперь надо откомпилировать файлы:

```
javac *.java
```

создать заглушку:

```
rmic MyRemoteImpl
```

запустить реестр:

```
rmiregistry&
```

и запустить демон активизации, прослушивающий порт номер 1098:

```
rmid -J-Djava.security.policy=rmid.policy&
```

Теперь можно зарегистрировать активизируемый удаленный объект в системе активизации, запустив на выполнение класс `MyRemoteReg`

```
java -Djava.server.codebase=http://hostname/myrmifiles/
     -Djava.security.policy=$HOME/policy     MyRemoteReg
```

Будьте внимательны — не оставляйте лишних пробелов и не забывайте заканчивать имя каталога слэшем.

Наконец, на клиентской машине компилируем и запускаем клиента

```
javac MyClient.java
java -Djava.security.policy=$HOME/policy  MyClient
```

Группа активизации

Активизируемый удаленный объект всегда входит в состав группы объектов, даже если в группе только один объект. Методы всей группы объектов выполняются отдельной виртуальной машиной Java. Группа активизируемых удаленных объектов описывается абстрактным классом `ActivationGroup` и создается автоматически при создании первого объекта класса `ActivationDesc` или явно статическим методом

```
public static ActivationGroup createGroup(ActivationGroupID gid,
    ActivationGroupDesc desc, long incarnation);
```

При создании группы подсистема активизации системы RMI уже должна знать идентификатор группы и информацию о ней.

Информация о группе объектов хранится в экземпляре класса `ActivationGroupDesc`. Он хранит имя и местоположение класса `ActivationGroup` этой группы и сериализованные начальные данные группы. Экземпляр класса создается конструктором

```
public ActivationGroupDesc(String className, String location,
   MarshalledObject data, Properties overrides,
   ActivationGroupDesc.CommandEnvironment cmd);
```

который переопределяет некоторые системные свойства, заданные аргументом `overrides`, и задает команду запуска новой виртуальной машины Java для выполнения методов группы, а также ее аргументы в виде объекта вложенного класса `CommandEnvironment`. Этот небольшой класс только упаковывает исполняемый файл Java и его аргументы в один объект конструктором

```
public ActivationGroupDesc.CommandEnvironment(
   String cmdpath, String[] args);
```

Второй конструктор

```
public ActivationGroupDesc(Properties overrides,
   ActivationGroupDesc.CommandEnvironment cmd);
```

использует имя класса и его местоположение по умолчанию.

Идентификатор группы создается при ее регистрации методом `registerGroup()` класса `ActivationSystem`. Экземпляр этого класса можно получить статическим методом `getSystem()` класса `ActivationGroup`.

Итак, создание группы активизации можно выполнить следующим образом:

```
ActivationGroupDesc desc =
   new ActivationGroupDesc(null, null);
ActivationGroupID gid =
   ActivationGroup.getSystem().registerGroup(desc);
ActivationGroup.createGroup(gid, desc, 0);
```

После этого создать описание объекта в новой группе:

```
ActivationDesc desc = new ActivationDesc(gid,
   "MyRemoteImpl", "http://hostname/myrmiclasses/", null);
```

Теперь можно обычным образом получить заглушку объекта и зарегистрировать ее в RMI-реестре:

```
MyRemote mr =(MyRemote)Activatable.register(desc);
LocateRegistry.getRegistry().rebind(serverName, mr);
```

Применение собственных сокетов

Для создания сокетов система RMI использует методы `createSocket(String host, int port)` и `createServerSocket(int port)` абстрактного класса `RMISocketFactory`, возвращающие объекты класса `Socket` и `ServerSocket` соответственно.

Экземпляр класса `RMISocketFactory` возвращается статическим методом `getSocketFactory()` того же класса. Перед использованием этого метода следует установить объект класса `RMISocketFactory` методом `setSocketFactory(RMISocketFactory)`. Если этого не сделать, то метод `getSocketFactory()` вернет `null`. В последнем случае следует воспользоваться методом `getDefaultSocketFactory()`, возвращающим ссылку на экземпляр класса `RMISocketFactory`, используемый системой RMI по умолчанию. Этот экземпляр создает и открывает стандартные сокеты Java потокового типа, работающие по протоколу TCP.

Установить экземпляр класса `RMISocketFactory` методом `setSocketFactory(RMISocketFactory)` можно только один раз и только в том случае, если такую установку допускает установленный диспетчер безопасности `SecurityManager`, иначе будет выброшено исключение `SecurityException`.

Методы

```
public Socket createSocket(String host, int port);
public ServerSocket createServerSocket(int port);
```

описаны интерфейсами `RMIClientSocketFactory` и `RMIServerSocketFactory` соответственно. Для применения в системе RMI собственных сокетов следует реализовать эти интерфейсы, например, так, как показано в листингах 15.8 и 15.9.

Листинг 15.8. Реализация фабрики клиентских сокетов

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;

public class MyClientSocketFactory implements Serializable,
                        RMIClientSocketFactory{
   public Socket createSocket(String host, int port)
                                 throws IOException{
      return new MySocket(host, port);
   }
}

class MySocket extends Socket{
   private InputStream is;
   private OutputStream os;

   public MySocket(){super();}

   public MySocket(String host, int port) throws IOException{
      super(host, port);
   }

   public InputStream getInputStream() throws IOException{
      if (is == null)
        is = new MyInputStream(super.getInputStream());

      return is;
   }
```

```
   public OutputStream getOutputStream() throws IOException{
      if (os == null)
        os = new MyOutputStream(super.getOutputStream());

      return os;
    }

   public synchronized void close() throws IOException {
      OutputStream o = getOutputStream();
      o.flush();
      super.close();
   }
}
```

Реализация входного потока `MyInputStream` и выходного потока `MyOutputStream` — задача разработчика. В этих потоках может происходить перекодировка, сжатие, проверка безопасности и другие действия, необходимые для правильного взаимодействия клиента и сервера.

Листинг 15.9. Реализация фабрики серверных сокетов

```
import java.rmi.*;
import java.rmi.server.*;

public class MyServerSocketFactory implements Serializable,
                        RMIServerSocketFactory{
   public Socket createServerSocket(int port)
                            throws IOException{
      return new MyServerSocket(port);
   }
}

class MyServerSocket extends ServerSocket{

   public MyServerSocket(int port) throws IOException{
       super(port);
   }

   public Socket accept() throws IOException{
      Socket s = new MySocket();
```

```
      implAccept(s);
      return s;
   }
}
```

Теперь надо установить объекты полученных классов в RMI-сервер класса, расширяющего класс `UnicastRemoteObject` конструктором

```
UnicastRemoteObject(int port, RMIClientSocketFactory csf,
   RMIServerSocketFactory ssf);
```

например, так, как написано в листинге 15.10.

Листинг 15.10. Удаленный объект, использующий собственные сокеты

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;

public class MyRemoteImpl extends UnicastRemoteObject
                implements MyRemote{
   private String name;

   public MyRemoteObject() throws RemoteException{
      super(0, new MyClientSocketFactory(),
               new MyServerSocketFactory());
   }

   public String getName() throws RemoteException{
      return name;
   }

   public void setName(String name) throws RemoteException{
      this.name = name;
   }
}
```

Для активизируемого удаленного объекта следует использовать конструктор класса `Activatable`, имеющий вид

```
Activatable(ActivationID id, int port, RMIClientSocketFactory csf,
   RMIServerSocketFactory ssf);
```

Если используются другие классы RMI-серверов, не наследующие класс `UnicastRemoteObject`, то для экспорта удаленных объектов следует применить статический метод

```
public static Remote exportObject(Remote obj, int port,
   RMIClientSocketFactory csf,
   RMIServerSocketFactory ssf);
```

Для экспорта активизируемых удаленных объектов, не наследующих класс `Activatable`, надо обратиться к статическому методу

```
public static Remote exportObject(Remote obj,
   ActivationID id, int port,
   RMIClientSocketFactory csf,
   RMIServerSocketFactory ssf);
```

класса `Activatable`.

Использование RMI-реестра

Реестр RMI, хранящий местоположение удаленных объектов, зарегистрированных в нем под произвольно заданными именами, и предоставляющий их клиентам — это удаленный объект, реализующий удаленный интерфейс `Registry` из пакета java.rmi.registry.

При обращении клиента к статическому методу `lookup(String)` класса `Naming` происходит обращение к RMI-реестру, адрес которого фиксирован и известен клиенту. Реестр пересылает клиенту свою заглушку, реализующую интерфейс `Registry`. Для этого используется один из статических методов `getRegistry()` класса `LocateRegistry`. Основной метод

```
public static Registry getRegistry(String host, int port,
   RMIClientSocketFactory csf);
```

Остальные методы задают по умолчанию параметры: `localhost`, порт номер 1099 и сокет по умолчанию.

Два статических метода создают экземпляр RMI-реестра

```
public static Registry createRegistry(int port);
public static Registry createRegistry(int port,
   RMIClientSocketFactory csf,
   RMIServerSocketFactory ssf);
```

Реестр системы RMI может работать не только самостоятельно, но и быть включенным в другую распределенную систему именования, например, JNDI.

Обратный вызов

Хотя система RMI основана на архитектуре клиент-сервер, ее легко можно превратить в симметричную систему, зарегистрировав на сервере клиентскую программу, которую следует реализовать как удаленный объект. После этого клиент и сервер могут обращаться к удаленным объектам друг друга.

Приведем для примера простейшую систему обмена сообщениями между клиентами, зарегистрированными в системе (chat). Удаленный интерфейс сервера этой системы приведен в листинге 15.11.

Листинг 15.11. Удаленный интерфейс chat-сервера

```
import java.rmi.*;

public interface MessageServer extends Remote{
    static int FAILURE = -1;
    static int SUCCESS = 0;

    void send(String name, String mess) throws RemoteException;

    int register(String name, MessageReceiver m)
           throws RemoteException;
}
```

Метод `register()` регистрирует клиента `m` типа `MessageReceiver` под именем `name`. Для сервера клиент — это удаленный объект. Поэтому в методе `register()` указывается не класс, а удаленный интерфейс `MessageReceiver` клиента. Метод `send()` посылает клиенту, зарегистрированному под именем `name`, сообщение `mess`.

Удаленный объект, реализующий этот интерфейс, совмещен для простоты со своим RMI-сервером. Он приведен в листинге 15.12.

Листинг 15.12. Удаленный объект chat-сервера

```
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class Server extends UnicastRemoteObject
                           implements MessageServer{
```

```
static Hashtable receivers = new Hashtable(5);

public Server() throws RemoteException{
   super();
}

public void send(String name, String mess)
                        throws RemoteException{

  Enumeration e = receivers.elements();

  while (e.hasMoreElements()){
    MessageReceiver m = (MessageReceiver)e.nextElement();
    m.print(name + ": " + mess);
  }
}

public int register(String name, MessageReceiver m){

  int retval = MessageServer.FAILURE;

  if ((name !=null) && (m != null)){
    if (receivers.get(name) == null){

       receivers.put(name, m);

       System.out.println("Добавлен " + name);
       retval = MessageServer.SUCCESS;

    }else System.out.println("Клиент " + name +
            " уже зарегистрирован");
  }
  return retval;
}

public static void main(String[] args){
   System.setSecurityManager(
       new RMISecurityManager());
```

```
      try{
         MessageServer srv = new Server();

         Registry reg = LocateRegistry.createRegistry(
             Registry.REGISTRY_PORT);

         reg.rebind("chat", srv);

         System.out.println("Сервер запущен.");
      }catch (Exception e){
         System.err.println(e);
      }
   }
}
```

Метод `send()` реализован так, что он обращается к методу `print()` удаленного, с его точки зрения, объекта `m`.

Теперь компилируем файлы на сервере:

```
javac *.java
```

создаем заглушку

```
rmic Server
```

и запускаем сервер

```
start java -Djava.security.policy=chat.policy  Server
```

Напишем клиентскую программу. Поскольку она будет удаленным объектом, создадим ее удаленный интерфейс. Он приведен в листинге 15.13.

Листинг 15.13. Удаленный интерфейс chat-клиента

```
import java.rmi.*;

public interface MessageReceiver extends Remote{

    void print(String mess) throws RemoteException;
}
```

В удаленном интерфейсе клиента всего один метод, выводящий сообщение `mess`.

Реализуем удаленный интерфейс клиента как графическое приложение Java. Оно приводится в листинге 15.14.

Листинг 15.14. Удаленный объект клиента

```
import java.awt.*;
import java.awt.event.*;
import java.rmi.*;
import java.rmi.server.*;
import javax.swing.*;

public class Client extends JFrame
        implements MessageReceiver, ActionListener{
    JTextField tf;
    JTextArea ta;
    static String name;
    static MessageServer server;

   public Client() throws RemoteException{
     setTitle(name);
     Container c = getContentPane();

     c.add(new JLabel("Обмен сообщениями", JLabel.CENTER),
        BorderLayout.NORTH);

     tf = new JTextField(30);
     tf.addActionListener(this);
     c.add(tf, BorderLayout.SOUTH);

     ta = new JTextArea(10, 30);
     c.add(ta);

     UnicastRemoteObject.exportObject(this);

     setSize(400, 300);
     setVisible(true);
     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
```

```
    public void actionPerformed(ActionEvent ae){
      try{
          server.send(name, tf.getText());
      }catch(Exception e){
          System.err.println(e);
      }
      tf.setText("");
    }

    public void print(String mess){
      ta.append(mess + "\n");
    }

    public static void main(String[] args){

      if (args.length < 1){
        System.out.println("Usage: java Client name");
        System.exit(-1);
      }

      name = args[0];

      if (System.getSecurityManager() == null)
         System.setSecurityManager(new RMISecurityManager());

      try{
         Client cl = new Client();

         server = (MessageServer)Naming.lookup(
             "//hostname/chat");

         int n = server.register(name, cl);

         if (n == MessageServer.FAILURE)
           throw new Exception("Couldn't connect to server");
      }catch(Exception e){
         System.err.println(e);
         System.exit(-1);
      }
   }
}
```

Поскольку класс `Client` уже расширяет класс `JFrame`, его нельзя сделать расширением класса `UnicastRemoteObject` или класса `Activatable`. Следовательно, регистрация удаленного объекта и его экспорт выполняются в конструкторе класса статическим методом `exportObject()`.

Метод `main()` создает экземпляр `cl` клиента, методом `lookup()` отыскивает сервер, получает его заглушку `server`, и регистрирует клиента на сервере методом `register()`. При регистрации на сервер пересылается заглушка клиента, с помощью которой сервер сможет обращаться к методам клиента, объявленным в интерфейсе `MessageReceiver`.

Теперь на каждой клиентской машине компилируем файлы клиента

```
javac *.java
```

создаем заглушку

```
rmic Client
```

и запускаем клиентскую программу

```
start java -Djava.security.policy=chat.policy
    Client nickname
```

После этого клиенты могут вести "разговор", отправляя и получая сообщения, посланные друг другу.

Глава 16

Обмен сообщениями между частями распределенного приложения

Все рассмотренные выше средства сетевой связи требуют, чтобы взаимодействующие сетевые программы находились в активном состоянии все время сеанса связи. Как говорят, это средства *синхронной* связи. Синхронную связь легко сделать двусторонней и многосторонней. Она отличается оперативностью и интерактивностью. Легко обеспечить надежность синхронной связи, просто подтверждая прием информации и повторяя передачу в случае потери сетевых пакетов.

Во многих случаях не удается или просто не нужно обеспечивать синхронную работу распределенных приложений. Тогда организуется *асинхронная* связь. При асинхронной связи одна программа посылает информацию, порция которой называется *сообщением* (message), другой программе, не будучи уверенной в том, что принимающая программа находится на связи. После отправки сообщения программа продолжает свою работу. Надежность передачи в этом случае должна обеспечивать *служба сообщений* (message service) — промежуточное программное обеспечение, принимающее сообщение, хранящее его, и при первой возможности передающее сообщение программе-получателю.

Служба сообщений может быть централизованной, работая как сервер на отдельной машине, или распределенной, встроенной в приложения, обменивающиеся сообщениями. В настоящее время разработчики стремятся сделать клиент "тонким" и чаще выбирают централизованную службу сообщений. Такая служба сообщений строится по одной из двух схем.

Первая схема организует непосредственную передачу сообщения от отправителя к получателю (point-to-point, PTP). По этой схеме служба сообщений организует в оперативной памяти или на диске *очереди сообщений* (message queues) для их временного хранения. Для каждого получателя сообщений организуется своя очередь, и только он может получить сообщения оттуда. Получатель может в любое время выйти на связь и извлечь сообщения из своей очереди. Сообщения передаются ему в том порядке, в котором они поступили в очередь. По этой схеме реализована электронная почта — один из примеров службы сообщений.

Вторая схема организует рассылку сообщений (publish-and-subscribe, Pub/Sub), *опубликованных* отправителями, всем *подписчикам* этих сообщений. Сообщения распределяются по нескольким *разделам* (topics). При подписке указывается один или несколько разделов, из которых подписчик хочет получать сообщения. Сообщения передаются ему в произвольном порядке. По этой схеме реализованы службы новостей, различные доски объявлений и рассылки.

Служба сообщений является частным случаем промежуточного программного обеспечения, ориентированного на обработку сообщений *MOM* (Message Oriented Middleware). Система MOM обеспечивает синхронную или асинхронную пересылку сообщений в распределенном приложении. Для связи клиентов с системой MOM она предоставляет API, не зависящий от операционной системы и сетевых протоколов.

Системы MOM сейчас быстро завоевывают популярность. Организован консорциум MOMA (Message Oriented Middleware Association — Ассоциация промежуточного программного обеспечения, ориентированного на сообщения), в который входят фирма IBM, выпускающая продукт MQSeries, корпорация Microsoft с продуктом MSMQ, фирма BEA, выпускающая MessageQ, корпорация Oracle, предоставляющая продукт AQ, фирма Sybase с системой сообщений dbQ и другие фирмы. Консорциум MOMA недавно переименован в IMWA (International Middleware Association — Международная ассоциация промежуточного программного обеспечения).

Технология Java содержит описание системы сообщений JMS (Java Message Service), входящее в стандартную поставку J2EE SDK. Интерфейсы и классы, составляющие это описание, содержатся в пакете javax.jms. Всякая реализация этих интерфейсов называется *поставщиком* (provider) JMS.

Первоначально поставщиками JMS служили системы MOM, в частности, MQSeries. Затем реализацию JMS стали включать в серверы приложений. Сейчас реализация JMS стала неотъемлемой частью J2EE (Java 2 Enterprise Edition) и всякий J2EE-сервер является поставщиком JMS.

Рассмотрим подробнее строение JMS. Начнем со структуры сообщения.

Сообщение *Message*

Сообщение системы JMS строится таким образом, чтобы оно могло быть передано большинству систем MOM и правильно обработано этими системами.

Каждое сообщение состоит из трех частей: заголовка (header), набора свойств (properties) и тела (body) сообщения. Тело сообщения может отсутствовать, у свойств могут быть пустые значения. Заголовок и свойства сообщения описываются интерфейсом `Message`.

Заголовок сообщения

Заголовок сообщения по-разному создается и понимается различными системами MOM, поэтому спецификация JMS определяет заголовок в самой общей форме. По этой спецификации заголовок может содержать десять полей.

Поле `JMSRedelivered` отмечает повторную доставку сообщения. Это поле устанавливается поставщиком JMS для повторной отправки сообщения получателю, если, например, получатель не подтвердил получение сообщения.

Шесть полей определяются при отправке сообщения методом `send()` или `publish()`. Это идентификатор сообщения — поле `JMSMessageID` и адресат сообщения — поле `JMSDestination`. Третье поле `JMSDeliveryMode` хранит один из режимов получения сообщения: `PERSISTENT`, означающий, что в случае краха поставщика услуг следует приложить дополнительные усилия по доставке сообщения клиенту, например, сохранить сообщение в базе данных, или `NON_PERSISTENT`, не требующий дополнительных усилий. Еще два поля — `JMSPriority` и `JMSTimestamp` — хранят приоритет сообщения, (приоритет возрастает от 0 до 9, значения 0—4 — это обычные приоритеты, 5—9 — повышенные приоритеты, приоритет по умолчанию 4) и время передачи сообщения поставщику. Шестое поле заголовка, определяемое при отправке сообщения — поле `JMSExpiration` — определяет срок хранения сообщения. Нулевой срок означает неопределенное время хранения.

Клиент определяет три поля заголовка. Поле `JMSCorrelationID` служит для связи сообщений, например, ответа с запросом, и имеет одно из трех значений. Это может быть идентификатор сообщения, назначаемый поставщиком JMS. Он начинается со строки `"ID:"`. Это может быть строка, содержание которой определяется приложением. Наконец, это может быть массив типа `byte[]`, но такой заголовок теряет переносимость. Второе поле, `JMSReplyTo`, определяет адресата, которому надо переслать ответ на сообщение. Третье поле, `JMSType`, определяет тип сообщения.

Интерфейс `Message` описывает методы `getXxx()` и `setXxx(xxx)` для получения значений каждого из этих десяти полей и установки значений полей заголовка.

Свойства сообщения

Свойства сообщения задают признаки, по которым поставщик услуг JMS может фильтровать и сортировать сообщения перед отправкой получателю. Свойства сообщения определяются отправителем. Они представляют собой пары "имя-значение". Их значения — простые типы Java: `boolean`, `byte`, `short`, `integer`, `long`, `float`, `double` и тип `String`. Интерфейс `Message` описывает методы `getXxxProperty(String name)` и `setXxxProperty(String`

`name, xxx value)` для каждого из этих восьми типов. Кроме того, описаны методы `getObjectProprety(String name)` и `setObjectProperty(String name, Object value)`. Их значения — объекты типов-оболочек `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` и типа `String`.

Спецификация JMS определяет несколько стандартных свойств сообщения. Их имена начинаются с префикса `"JMSX"`. Набор стандартных свойств определяется соединением, в котором создаются сообщения. Список имен стандартных свойств данного соединения можно получить методом

```
public Enumeration getJMSXPropertyNames();
```

описанном в интерфейсе `ConnectionMetaData`.

В настоящее время определено девять стандартных свойств:

- `JMSXAppID` — строка типа `String` — идентификатор программы-отправителя;
- `JMSXUserID` — строка типа `String` — идентификатор пользователя-отправителя;
- `JMSXGroupID` — строка типа `String` — идентификатор группы сообщений, к которой принадлежит сообщение;
- `JMSXGroupSeq` — число типа `int` — порядковый номер сообщения в группе, начинается с 1;
- `JMSXDeliveryCount` — число типа `int` — количество попыток доставки сообщения;
- `JMSXProducerTXID` — строка типа `String` — идентификатор транзакции, в рамках которой отправлено сообщение;
- `JMSXConsumerTXID` — строка типа `String` — идентификатор транзакции, в рамках которой получено сообщение;
- `JMSXRcvTimestamp` — число типа `long` — время отправки сообщения;
- `JMSXState` — состояние сообщения во временном хранилище: 1 — ожидает отправки, 2 — готово к отправке, 3 — время хранения сообщения истекло, 4 — сообщение сохранено.

Имена свойств, начинающиеся со строки `"JMS_"`, означают, что свойство определяется поставщиком JMS.

Имена всех свойств сообщения можно получить методом

```
public Enumeration getPropertyNames();
```

Удалить все свойства сообщения можно методом

```
public void clearProperties();
```

Тело сообщения

Система сообщений JMS представляет тело сообщения в одной из пяти форм. Они описаны пятью интерфейсами, расширяющими интерфейс `Message`.

Самая обычная и простая форма тела сообщения — текст в виде строки типа `String` — описана интерфейсом `TextMessage`. В этом интерфейсе всего два метода получения текста из сообщения и занесения текста в сообщение:

```
public String getText();
public void setText(String mess);
```

Вторая форма тела сообщения — поток, описанный интерфейсом `StreamMessage`, в который записываются простые типы данных Java, а также типы `String`, `Object` и массив типа `byte[]`. Запись выполняется методами `writeXxx(xxx value)`, а чтение данных — методами `readXxx()`.

Третья форма, описанная интерфейсом `MapMessage`, передает пары "имя—значение". Типы этих пар такие же, как в потоке `StreamMessage`, записываются они методами `writeXxx(String name, xxx value)`, их значения читаются методами `readXxx(String name)`. Получить имена всех пар можно методом

```
public Enumeration getMapNames();
```

Четвертая форма, описанная интерфейсом `ObjectMessage`, передает в сообщении сериализуемые объекты Java. Это делается двумя методами:

```
public Serializable getObject();
public void setObject(Serializable obj);
```

Наконец, пятая, самая общая, форма тела сообщения, описанная интерфейсом `BytesMessage`, передает просто набор байтов. Основной метод

```
public void writeBytes(byte[] mess, int offset, int length);
```

заносит в сообщение часть массива байтов `mess` длиной `length` байтов, начиная от индекса `offset`. Симметричный ему метод

```
public int readBytes(byte[] mess, int length);
```

читает массив байтов `mess`, содержащий тело сообщения. Методы

```
public void writeBytes(byte[] mess);
public int readBytes(byte[] mess);
```

записывают и читают весь массив целиком.

Прочие методы интерфейса `BytesMessage` читают и записывают в тело сообщения данные различных типов Java.

Удалить все тело сообщения любой формы можно методом

```
public void clearBody();
```

Сообщения создаются во время сессии, а сессия начинается после получения соединения с поставщиком JMS. Здесь соединение понимается не как процесс связи, а как объект, который надо получить.

Соединение *Connection*

В системе сообщений JMS соединение устанавливается либо для непосредственной пересылки сообщений получателю по схеме PTP, что описано интерфейсом `QueueConnection`, либо для рассылки сообщений подписчикам по схеме Pub/Sub. Этот тип соединения описан интерфейсом `TopicConnection`.

Интерфейс `QueueConnection` определяет метод

```
public QueueSession createQueueSession(
   boolean transacted, int ackMode);
```

создания сеанса типа `QueueSession`, а интерфейс `TopicConnection` — аналогичный метод

```
public TopicSession createTopicSession(
   boolean transacted, int ackMode);
```

создания сеанса типа `TopicSession`.

Если аргумент `transacted` равен `true`, то в течение такого сеанса создаются одна или несколько последовательных транзакций.

Транзакцию (transaction) образуют одно или несколько сообщений, которые отправляются все вместе или не отправляется ни одно из них. Это удобно в тех случаях, когда нужна гарантированная доставка всех сообщений получателю. Транзакцию можно завершить методом `commit()` интерфейса `Session`, после чего начинается следующая транзакция, или отказаться от ее выполнения методом `rollback()`.

Метод `commit()`, кроме того, посылает своему сеансу *подтверждение* (acknowledgment) приема всех сообщений, входящих в транзакцию. Второй аргумент `ackMode` при этом игнорируется. Метод `recover()` останавливает передачу сообщений и возобновляет передачу только неподтвержденных сообщений.

Если аргумент `transacted` равен `false`, то порядок подтверждения определяется вторым аргументом `ackMode`, который принимает одно из трех значений, описанных в интерфейсе `Session`:

- `AUTO_ACKNOWLEDGE` — получение сообщений подтверждается во время успешного выхода из метода `receive()` или после успешной обработки получения сообщения методом `onMessage()`;
- `CLIENT_ACKNOWLEDGE` — получатель сам подтверждает прием данного и всех предыдущих сообщений методом `acknowledge()` интерфейса `Message`;

- `DUPS_OK_ACKNOWLEDGE` — "вялое" (lazy) подтверждение, выражающееся в повторной доставке сообщений при ошибке.

Для того чтобы предоставить клиентам возможность соединения с поставщиком JMS, сам поставщик должен реализовать интерфейсы `QueueConnectionFactory` и `TopicConnectionFactory`. Методы

```
public QueueConnection createQueueConnection();
public QueueConnection createQueueConnection(
   String name, String password);
public TopicConnection createTopicConnection();
public TopicConnection createTopicConnection(
   String name, String password);
```

этих интерфейсов возвращают соединения (объекты, а не процессы) типа `QueueConnection` или `TopicConnection`.

Сами же объекты типа `QueueConnectionFactory` и `TopicConnectionFactory` поставщик JMS предоставляет всем желающим через какую-либо систему имен и каталогов, например, систему JNDI, описанную в следующей главе. Поставщик JMS, имеющийся в составе J2EE SDK, просто повторяет имена интерфейсов: `"QueueConnectionFactory"` и `"TopicConnectionFactory"`, а кроме того, связывает эти объекты с еще двумя именами: `"jms/QueueConnectionFactory"` и `"jms/TopicConnectionFactory"`.

Таким образом, установить соединение типа PTP для непосредственной передачи сообщений и создать сессию можно так:

```
Context ctx = new InitialContext();

QueueConnectionFactory qcf =
   (QueueConnectionFactory)ctx.lookup("QueueConnectionFactory");

QueueConnection qc = qcf.createQueueConnection();

QueueSession qs = qc.createQueueSession(
   false, Session.AUTO_ACKNOWLEDGE);
```

Установить соединение типа Pub/Sub для рассылки сообщений можно точно так же:

```
Context ctx = new InitialContext();

TopicConnectionFactory tcf =
   (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");

TopicConnection tc = tcf.createTopicConnection();
```

```
TopicSession ts = tc.createTopicSession(
   false, Session.CLIENT_ACKNOWLEDGE);
```

Для того чтобы получатель смог принимать сообщения, соединение следует предварительно запустить методом `start()`. Его всегда можно приостановить методом `stop()`, а потом снова запустить методом `start()`. После работы с соединением его следует закрыть методом `close()`, чтобы освободить занимаемые им ресурсы.

Сведения о соединении

Метод

```
public ConnectionMetaData getMetaData();
```

описанный в интерфейсе `Connection`, позволяет получить сведения о соединении в виде объекта типа `ConnectionMetaData`. Кроме уже упоминавшегося метода `getJMSXPropertyNames()`, в интерфейсе `ConnectionMetaData` описаны методы

```
public String getJMSVersion();
public int getJMSMajorVersion();
public int getJMSMinorVersion();
```

возвращающие версию системы сообщений JMS в виде строки, а также номер версии и ее модификации, и методы

```
public String getJMSProviderName();
public String getProviderVersion();
public int getProviderMajorVersion();
public int getProviderMinorVersion();
```

возвращающие имя и версию поставщика JMS в виде строки, а также номер версии и модификации.

Сеанс *Session*

Сеанс связи — центральное понятие системы сообщений, описанное интерфейсом `Session`. Именно в рамках сеанса создаются отправители (providers) и получатели (receivers) сообщений, очереди сообщений (message queues), разделы (topics) для рассылки сообщений, а также сами сообщения. Сеанс, как правило, существует все время данного соединения, хотя его можно завершить методом `close()`. Интерфейс `Session` расширяет интерфейс `Runnable`, следовательно, сеанс выполняется отдельным подпроцессом.

В рамках сеанса создаются сообщения, содержащие тело во всех пяти формах. Для этого используются методы

```
public BytesMessage createBytesMessage();
public MapMessage createMapMessage();
public ObjectMessage createObjectMessage();
public ObjectMessage createObjectMessage(Serilizable obj);
public StreamMessage createStreamMessage();
public TextMessage createTextMessage();
public TextMessage createTextMessage(String text);
```

Еще один метод

```
public Message createMessage();
```

создает сообщение без тела, содержащее только заголовок и свойства сообщения.

Сеансы, как и соединения, создаются для непосредственной передачи сообщений по схеме PTP или для рассылки сообщений по схеме Pub/Sub.

Сеансы первого типа описаны интерфейсом `QueueSession`, расширяющим интерфейс `Session`. Методы этого интерфейса создают очередь сообщений с именем `name`:

```
public Queue createQueue(String name);
```

временное хранилище сообщений:

```
public TemporaryQueue createTemporaryQueue();
```

а также отправителя и получателя сообщений из очереди `queue`:

```
public QueueSender createSender(Queue queue);
public QueueReceiver createReceiver(Queue queue);
public QueueReceiver createReceiver(Queue queue, String selector);
```

и объект, позволяющий просматривать очередь и ее характеристики:

```
public QueueBrowser createBrowser(Queue queue);
public QueueBrowser createBrowser(Queue queue, String selector);
```

Аргумент `selector` задает фильтр для отбора сообщений, отправляемых получателю.

Интерфейс `QueueBrowser` описывает метод просмотра очереди

```
public Enumeration getEnumeration();
```

который возвращает содержимое очереди в порядке поступления сообщений.

Кроме того, в интерфейсе `QueueBrowser` есть метод получения селектора сообщений, стоящих в очереди

```
public String getMessageSelector();
```

и метод получения очереди

```
public Queue createQueue(String name);
```

После работы с объектом типа `QueueBrowser` его следует закрыть методом `close()`.

Сеансы второго типа описаны интерфейсом `TopicSession`, тоже расширяющим интерфейс `Session`. Методы этого интерфейса создают раздел с именем `name`:

```
public Topic createTopic(String name);
```

временное хранилище сообщений:

```
public TemporaryTopic createTemporaryTopic();
```

отправителя и подписчика сообщений из раздела `topic`:

```
public TopicPublisher createPublisher(Topic topic);
public TopicSubscriber createSubscriber(Topic topic);
public TopicSubscriber createSubscriber(
   Topic queue, String selector, boolean noLocal);
```

Если аргумент `noLocal` равен `true`, то подписчик не будет получать сообщения, отправленные в рамках того же соединения.

Можно создать *неопределенного* (undefined) отправителя или издателя, не связанного с очередью или разделом, указав в методе `createSender(Queue)` или в методе `createPublisher(Topic)` аргумент `null`. Такой отправитель или издатель при посылке сообщения методом `send()` или `publish()` обязательно должен указывать очередь или раздел, в который отправлено сообщение.

Подписчик получает только те сообщения, которые поступили адресату во время активности подписчика. Система JMS позволяет создать *долговременного* (durable) подписчика, который получит и те сообщения, что поступили за то время, когда подписчик был не активен. Долговременного подписчика с именем `name` создают два метода

```
public TopicSubscriber createDurableSubscriber(
   Topic topic, String name);
public TopicSubscriber createDurableSubscriber(
   Topic topic, String name, String selector, boolean noLocal);
```

Удалить долговременного подписчика можно методом

```
public void unsubscribe(String name);
```

Отбор сообщений

У методов создания получателя есть аргумент `selector`, содержащий строку-фильтр для отбора сообщений, которые поставщик JMS передает получателю. Разные получатели могут получать разные сообщения из одной и той же очереди. Разные подписчики могут получать разные сообщения из одних и тех же разделов.

Сообщения можно отбирать по их заголовкам и свойствам, но не по телу сообщения.

Для того чтобы фильтр мог использоваться независимыми поставщиками JMS и системами MOM, строка `selector` создается по правилам составления условных выражений языка SQL92. Сообщение отправляется получателю, если это выражение будет истинно. Выражение разбирается слева направо с учетом приоритетов операций.

Запись условных выражений в языке SQL92 отличается от записи логических выражений в языке Java. При составлении таких выражений действуют следующие правила.

- Строки-константы заносятся в апострофы, а не в кавычки; если апостроф встречается в строке, то он удваивается, а не экранируется обратной наклонной чертой, например, `'Д''Артаньян'`. В константах различаются заглавные и строчные буквы. Два символа имеют особое значение: знак подчеркивания `'_'` используется для подстановки одного произвольного символа, знак процента `'%'` вызывает подстановку любого числа любых символов.
- Логические операции записываются словами `NOT`, `AND`, `OR`.
- В логических операциях используется трехзначная логика `TRUE`, `FALSE`, `UNKNOWN`.
- Добавлены операции `BETWEEN`, `LIKE`, `IN`, `IS`.
- Символ экранирования метасимволов `'_'` и `'%'` определяется операцией `ESCAPE`, например, выражение `"JMSXAppID LIKE '\_%' ESCAPE '\'"` будет истинно для любого значения переменной `JMSXAppID`, начинающегося со знака подчеркивания.
- Операция получения остатка от деления, `%`, не используется.
- Неравенство записывается знаками меньше и больше (<>).
- Арифметические операции и сравнения вещественных чисел с фиксированной точкой нельзя использовать в фильтре.
- Признак комментария SQL — два дефиса подряд — нельзя использовать в выражении.
- Служебные слова: `TRUE`, `NOT` и др., можно записывать и заглавными и строчными буквами.

Именами переменных в фильтре служат имена полей заголовка сообщения и имена свойств сообщения. Вот пример строки-фильтра:

```
String selector =
   "JMSXUserID = 'admin' AND content = 'info' AND length < 1024";
```

Адресат *Destination*

Поставщик JMS организует временные хранилища сообщений, полученных от отправителей перед отправкой их получателям. Они называются *адресатами* (destination) сообщений и описаны интерфейсом `Destination`. Интерфейс `Destination` не описывает методы и не определяет константы. Он служит просто признаком типа. Его расширяют два интерфейса.

При непосредственной отправке сообщений по схеме PTP адресатами служат *очереди* (queues) сообщений, описанные интерфейсом `Queue`.

При рассылке сообщений по схеме Pub/Sub адресатами служат *разделы* (topics), описанные интерфейсом `Topic`.

Интерфейсы `Queue` и `Topic` просто переопределяют метод `toString()` и описывают один метод получения имени очереди или раздела. Каждая очередь и каждый раздел при своем создании методами `createQueue(String)` или `createTopic(String)`, описанными в предыдущем разделе, получают имя, которое можно узнать методами

```
public String getQueueName();
public String getTopicName();
```

Хотя адресат — временное хранилище сообщений, он существует до окончания сеанса. Иногда возникает необходимость удалять адресата и создавать его вновь. Для этого интерфейсы `Queue` и `Topic` расширены интерфейсами `TemporaryQueue` и `TemporaryTopic` соответственно. Эти интерфейсы описывают временную очередь и временный раздел. Временные адресаты создаются в единственном экземпляре на каждый сеанс методами `createTemporaryQueue()` или `createTemporaryTopic()` интерфейсов `QueueSession` или `TopicSession`, описанными в предыдущем разделе. Временный адресат может удалить себя методом `delete()`. Если при этом существуют получатели, связанные с временным адресатом, то будет выброшено исключение класса `JMSException`.

Создание очередей и разделов осуществляется, как правило, поставщиком JMS. У каждого сервера приложений есть свои средства для этого. У J2EE-сервера, входящего в состав J2EE SDK, есть утилита администрирования j2eeadmin. Она работает из командной строки. Для создания очереди или раздела надо запустить утилиту с ключом `-addJmsDestination`.

Для создания очереди с именем `MyQueue` следует набрать командную строку

```
j2eeadmin –addJmsDestination MyQueue queue
```

а для создания раздела с именем `MyTopic` — набрать командную строку

```
j2eeadmin –addJmsDestination MyTopic topic
```

По умолчанию J2EE-сервер создает очередь с именем `jms/Queue` и раздел с именем `jms/Topic`. Посмотреть список созданных очередей и разделов можно командой

```
j2eeadmin –listJmsDestination
```

Отправители и получатели

После создания очередей или разделов, а также адресатов, можно приступить к созданию отправителей (producers, senders), издателей (publishers), получателей (consumers, receivers) и подписчиков (subscribers). Для этого служат описанные выше методы `createSender()`, `createReceiver()`, `createPublisher()`, `createSubscriber()`, `createDurableSubscriber()` интерфейсов `QueueSession` или `TopicSession`. При создании отправителей и получателей указывается их адресат, а для получателя можно указать фильтр сообщений.

Отправитель описывается интерфейсом `MessageProducer`. Методы этого интерфейса позволяют узнать и изменить значения по умолчанию некоторых полей заголовков сообщений:

- `setPriority(int prior)` — установить приоритет сообщений (по умолчанию 4);
- `setTimeToLive(long millisec)` — установить время хранения сообщений в миллисекундах (по умолчанию 0 мс);
- `setDisableMessageTimestamp(boolean disable)` — если аргумент равен `true`, то установить во всех сообщениях время равным 0;
- `setDisableMessageID(boolean disable)` — если аргумент равен `true`, то установить идентификатор всех сообщений `null`;
- `setDeliveryMode(int mode)` — установить режим доставки сообщений (по умолчанию `DeliveryMode.PERSISTENT`).

Соответствующие методы `getXxx()` позволяют узнать установки этих полей по умолчанию.

Отправитель любого типа может быть закрыт методом `close()`.

Интерфейс `MessageProducer` расширен двумя интерфейсами: `QueueSender` и `TopicPublisher`, описывающими отправителя и издателя сообщений.

Интерфейс `QueueSender` описывает несколько методов отправки сообщений. Основной метод

```
public void send(Queue queue, Message message,
   int deliveryMode, int priority, int timeToLive);
```

кроме посылки сообщения `message` в очередь `queue` задает три поля заголовка. Другие методы `send()` оставляют значения одного или нескольких полей по умолчанию. Если не указана очередь, то, по умолчанию, сообщение посылается в очередь, в которой создан отправитель этого сообщения. Неопределенный отправитель обязательно должен указать очередь, в которую посылается сообщение.

Кроме методов `send()`, интерфейс `QueueSender` описывает еще только один метод

```
public Queue getQueue();
```

возвращающий очередь, которая определила отправителя.

Интерфейс `TopicPublisher` описывает несколько методов публикации сообщений. Основной метод

```
public void publish(Topic topic, Message message,
   int deliveryMode, int priority, int timeToLive);
```

Этот метод и другие методы `publish()` аналогичны методам `send()`.

Кроме методов `publish()` интерфейс `TopicPublisher` описывает еще метод

```
public Topic getTopic();
```

возвращающий раздел, который определил издателя.

Получатель сообщений и подписчик рассылки описываются интерфейсом `MessageConsumer`. В нем сразу описаны методы получения сообщения:

- `public Message receive()` — получить следующее сообщение, как только оно будет доставлено адресату;
- `public Message receive(long timeout)` — ждать получения следующего сообщения `timeout` миллисекунд;
- `public Message receiveNoWait()` — получить следующее сообщение или вернуть `null`, если ни одно сообщение не доставлено адресату.

Если во время обращения к методу `receive()` без аргументов сообщение еще не поступило адресату, то приложение, обратившееся к этому методу, блокируется до поступления сообщения. Второй метод `receive(long)` ограничивает время ожидания, третий метод `receiveNoWait()` вообще не ждет поступления сообщения.

Если надо получать сообщения асинхронно, то нет нужды запускать подпроцесс для выполнения метода `receive()`. Для этого есть следующий метод интерфейса `MessageConsumer`:

```
public void setMessageListener(MessageListener listener);
```

Он устанавливает в объект-получатель сообщений экземпляр слушателя сообщений.

Слушатель сообщений — это класс, реализующий интерфейс `MessageListener`. В интерфейсе `MessageListener` описан всего один метод:

```
public void onMessage(Message message);
```

Поставщик JMS обращается к этому методу, как только адресату поступило сообщение. Сообщение передается как аргумент `message` метода `onMessage(Message)`. Реализация этого метода, заключающаяся в обработке полученного сообщения `message`, — это задача разработчика.

Получатель сообщения может быть закрыт методом `close()`.

Интерфейс `MessageConsumer` расширен двумя интерфейсами: `QueueReceiver` и `TopicSubscriber`, описывающими получателя сообщений и подписчика рассылки. Эти интерфейсы добавляют методы получения очереди и раздела соответственно:

```
public Queue getQueue();
public Topic getTopic();
```

Продолжим наш пример. Мы остановились на создании сеанса `qs` для непосредственной передачи сообщений по схеме Point-to-Point и сеанса `ts` для рассылки сообщений по схеме Pub/Sub.

После создания сеанса `qs` непосредственной передачи сообщений ищем объект-очередь с именем, например, `"StockQueue"`, с помощью системы имен JNDI:

```
Queue q = (Queue)ctx.lookup("StockQueue");
```

Создаем в сеансе `qs` экземпляр отправителя сообщений в очередь `q`:

```
QueueSender sender = qs.createSender(q);
```

Создаем сообщение `message` с телом `data` в текстовой форме:

```
TextMessage message = qs.createTextMessage();
String data = "Сообщение.";
message.setText(data);
```

Запускаем созданное ранее соединение `qc`:

```
qc.start();
```

Посылаем сообщение `data`:

```
sender.send(data);
```

Для получения сообщений из очереди `q`, в сеансе `qs` создаем экземпляр получателя:

```
QueueReceiver receiver = qs.createReceiver(q);
```

Получаем очередное сообщение, например, в форме потока:

```
StreamMessage message = (StreamMessage)receiver.receive();
```

После создания сеанса `ts` рассылки сообщений ищем объект-раздел с помощью, например, системы имен JNDI:

```
Topic t = (Topic)ctx.lookup("StockTopic");
```

В сеансе `ts` создаем экземпляр `publisher` издателя сообщений в раздел `t`:

```
TopicPublisher publisher = ts.createPublisher(t);
```

Запускаем ранее созданное соединение `tc`:

```
tc.start();
```

Посылаем сообщение:

```
publisher.publish(data);
```

Для получения подписки создаем в сеансе `ts` экземпляр подписчика:

```
TopicSubscriber subscriber = ts.createSubscriber(t);
```

Создаем экземпляр слушателя подписки и устанавливаем его в подписчик:

```
subscriber.setMessageListener(new MyListener());
```

Класс-слушатель подписки выглядит так:

```
public class MyListener implements MessageListener{
   public void onMessage(Message message){
     String data = message.getText();
   }
}
```

Обмен сообщениями в СДО

Приведем пример обмена сообщениями в системе дистанционного обучения. Администратор СДО посылает сообщения в раздел `"InfoTopic"`. В заголовке этих сообщений определен тип `JMSType`, равный `"AdminInfo"`. Кроме того, свойством сообщения `Recipient` помечается получатель сообщения: `"Student"` или `"Teacher"`.

В листинге 16.1 приведен отладочный вариант программы-отправителя — она запускается из командной строки, сообщения задаются как параметры командной строки и рассылаются по очереди то ученику, то учителю.

Листинг 16.1. Отправитель сообщений

```
import javax.naming.*;
import javax.jms.*;
import java.util.*;

public class InfoPublisher{

  public InfoPublisher(String[] mess){

    try{

      InitialContext context = new InitialContext();

      TopicConnectionFactory tcf = (TopicConnectionFactory)
         context.lookup("TopicConnectionFactory");

    }catch(Exception exc){
      System.err.println("From lookup: " + exc);
    }
    try{

     TopicConnection tc = tcf.createTopicConnection();

     TopicSession ts = tc.createTopicSession(false,
        Session.AUTO_ACKNOWLEDGE);

     Topic topic = (Topic)context.lookup("InfoTopic");

     TextMessage message = ts.createTextMessage();
     message.setJMSType("AdminInfo");

     String[] subs = new String[2];
     subs[0] = "Student";
     subs[1] = "Teacher";
```

```
      TopicPublisher tp = ts.createPublisher(topic);

      tc.start();

      for (int i = 0; i < mess.length; i++){
        message.setStringProperty("Recipient", subs[i % 2]);
        message.setIntProperty("messNumber", i);
        message.setText(mess[i]);
        System.out.println("From publ: " + message.getText());
        tp.publish(message);
      }
      tc.close();

    }catch(Exception e){
      System.err.println("From publ: " + e);
    }
  }

  public static void main(String[] args){

    if (args.length < 1){
       System.out.println(
          "Usage: java InfoPublisher <message>...");
       System.exit(-1);
    }
    InfoPublisher infoPub = new InfoPublisher(args);
  }
}
```

Получатель сообщений подписывается на сообщения из раздела `"InfoTopic"`. В программе листинга 16.2 тип получаемых сообщений `"Student"` или `"Teacher"` определяется при запуске программы из командной строки и используется в фильтре `selector` получаемых сообщений. В целях отладки сообщения просто анализируются и выводятся в стандартный вывод.

Листинг 16.2. Подписчик сообщений

```
import javax.jms.*;
import javax.naming.*;
import java.util.*;
```

```
public class InfoSubscriber implements MessageListener{

  public  boolean quitFromReceiving = false;

  public InfoSubscriber(String recipient){
    try{
      InitialContext ctx = new InitialContext();

      TopicConnectionFactory tcf = (TopicConnectionFactory)
          ctx.lookup("TopicConnectionFactory");

      TopicConnection tc = tcf.createTopicConnection();

      TopicSession ts = tc.createTopicSession(false,
          Session.AUTO_ACKNOWLEDGE);

      Topic topic = (Topic)ctx.lookup("InfoTopic");

      String selector =
        "JMSType = 'AdminInfo' AND Recipient = '" +
         recipient + "'";

       TopicSubscriber tr= ts.createDurableSubscriber(
          topic, "Info", selector, true);

       tr.setMessageListener(this);

       tc.start();

    }catch(Exception e){
       System.out.println(e);
    }
  }

  public void onMessage(Message message){
    try{
      System.out.println(message.getJMSMessageID());
```

```
    System.out.println("Идентификатор: " +
       message.getJMSMessageID() +
       " для адресата " + message.getJMSDestination());

    System.out.print("Срок хранения  ");

    if (message.getJMSExpiration() > 0)
        System.out.println(new Date(message.getJMSExpiration()));
    else
        System.out.println("не ограничен.");

    System.out.println("Приоритет: " + message.getJMSPriority());

    System.out.println("Режим: " + (
      message.getJMSDeliveryMode() == DeliveryMode.PERSISTENT ?
      "PERSISTENT" : "NON_PERSISTENT"));

    System.out.println("Ответ послать: " +
        message.getJMSReplyTo());

    System.out.println("Тип сообщения: " + message.getJMSType());

    if (message instanceof TextMessage){

      String messBody = ((TextMessage)message).getText();
      System.out.println("Тело сообщения: " + messBody);

    }else if (message instanceof ObjectMessage)
      System.out.println("Послан объект: " + message);

    quitFromReceiving = true;
    notifyAll();
  }
  catch(JMSException e){
    e.printStackTrace();
  }
}
```

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

    if (args.length < 1){
      System.out.println(
         "Usage: java InfoSubscriber Student|Teacher");
      System.exit(-1);
    }

   try{
     InfoSubscriber infoSub = new InfoSubscriber(args[0]);

     synchronized(infoSub){
       while (! infoSub.quitFromReceiving){
         try{
           infoSub.wait();
         }catch (InterruptedException ie){
           System.out.println(" Error:"+ ie);
           ie.printStackTrace();
         }
       }
     }

   }catch(Exception e){
     System.out.println("From main: " + e);
     e.printStackTrace();
   }
  }
}
```

Если в качестве поставщика JMS используется J2EE-сервер из поставки J2EE SDK, то надо его запустить:

```
j2ee –verbose
```

Потом создать раздел `InfoTopic` с помощью утилиты администрирования:

```
j2eeadmin –addJmsDestination InfoTopic topic
```

Проверить, какие разделы созданы:

```
j2eeadmin –listJmsDestination
```

После этого можно запускать программу-отправителя:

```
java -Djms.properties=$J2EE_HOME/config/jms_client.properties \
   InfoPublisher Message1 Message2 Message3
```

программу-подписчика, получающую сообщения, посланные ученику:

```
java -Djms.properties=$J2EE_HOME/config/jms_client.properties \
   InfoSubscriber Student
```

и программу-подписчика, получающую сообщения, посланные учителю:

```
java -Djms.properties=$J2EE_HOME/config/jms_client.properties \
   InfoSubscriber Teacher
```

Глава 17

Поиск распределенных объектов в сети

Распределенному приложению часто приходится обращаться к различным системам именования. Мы видели в *главе 15*, что система RMI отыскивает удаленные объекты с помощью RMI-реестра — небольшого сервера имен, связывающего имена удаленных объектов с их адресами. DNS-сервер связывает доменные имена вида host.some.com с IP-адресами. Информацией о разделенных ресурсах сети UNIX-машин управляет NIS-сервер, наделяющий ресурсы именами. Система распределенных объектов CORBA обладает развитой службой именования Naming Service. В каждой операционной системе есть файловая подсистема, связывающая имена файлов с их дескрипторами.

Все эти системы именования задают различные правила для составления имен и имеют различные методы создания, удаления, изменения имен, поиска объектов по их именам и получения сведений об объектах. Распределенное приложение должно иметь какие-то общие средства для работы с разными системами имен, образующие интерфейс с ними.

Технология Java предлагает свой интерфейс JNDI (Java Naming and Directory Interface) — набор средств связи с конкретными системами имен и работы в этих системах.

Систему JNDI можно рассматривать как обобщение файловой системы или доменной системы имен DNS. Каждый объект регистрируется в JNDI под некоторым *именем* (name). Эта операция называется *связыванием* (binding) имени и объекта. Она аналогична связыванию имени файла с его дескриптором в файловой системе или связыванию доменного имени, скажем, host.some.com, с IP-адресом, например, 145.223.15.22.

Каждое имя должно быть уникальным в пределах своего контекста. *Контекст* (context) — это множество *связываний* (binding), в котором все имена должны отличаться друг от друга. На контекст можно смотреть как на обобщение каталога файловой системы или домена DNS. В контексте могут содержаться имена других контекстов (subcontext). Так образуется иерархическая структура вложенных друг в друга контекстов. Для полного определения объекта надо указать его *составное имя* (compound name), в которое

входит последовательность имен вложенных контекстов, так же как в файловой системе указывается полный путь к файлу.

Во главе иерархии контекстов стоит *начальный контекст* (initial context). В системе JNDI нет единого корневого контекста и нет операции монтирования, подобно монтированию файловых систем. Поэтому во всем пространстве имен может быть несколько начальных контекстов.

Вся совокупность связанных между собой контекстов образует *систему именования* (naming system), а все множество имен — *пространство имен* (namespace).

Все эти понятия описаны интерфейсами и классами, входящими в пакет javax.naming. Имена объектов могут задаваться символьными строками класса `String` или объектами, описанными интерфейсом `Name`. Интерфейс `Name` реализован двумя классами. Составные имена, подобные путям к файлам /usr/include/sys/fs/f5.h, создаются и обрабатываются методами класса `CompoundName`. Имена, составленные по правилам сразу нескольких систем именования, подобно адресу URL http://www.some.com/cgi-bin/prog.pl, создаются и обрабатываются методами класса `CompositeName`. Понятие контекста и методы работы с ним описаны интерфейсом `Context`. Методы связывания имен реализованы в классе `Binding`, а начальный контекст — в классе `InitialContext`.

Совокупность всех этих классов и их методов работы с именами называется *службой имен* (name service).

Если объект системы JNDI является объектом Java, то он хранится в сериализованном виде. Если объект не может быть представлен объектом Java или не сериализуется, то на него создается *ссылка* (reference) — объект класса `Reference` — по которой отыскивается объект системы JNDI.

Среди контекстов выделяются *контексты каталогов* (directory context). Контексты каталогов снабжены средствами поиска объектов не только по имени, как обычный контекст, но и по другим признакам, например, по их содержимому. В простейшем случае создается множество *атрибутов* (attributes) объекта, по которым осуществляется поиск. Например, атрибутами файла могут служить его длина, имя владельца, дата модификации, права доступа к файлу. Контексты каталогов обычно создаются в иерархических системах именования, откуда произошло их название. Контекст каталога описан интерфейсом `DirContext`, расширяющим интерфейс `Context`, каждый атрибут — интерфейсом `Attribute`, а совокупность атрибутов — интерфейсом `Attributes`. Эти интерфейсы и реализующие их классы собраны в пакет javax.naming.directory.

Совокупность этих классов и их методов работы с контекстами каталогов называется *службой каталогов* (directory service).

Перед поиском объектов интерфейс JNDI нужно связать с конкретными системами именования и поиска: RMI-реестром, файловыми системами, системой COS Naming, входящей в CORBA, системой доменных имен DNS, сервером упрощенного доступа к каталогам LDAP (Lightweight Directory Access Protocol). Для связи с такими системами разработаны интерфейсы, образующие *интерфейс поставщика услуг* SPI (Service Provider Interface). Они составляют пакет javax.naming.spi. Эти интерфейсы реализуются для конкретных систем именования подобно тому, как интерфейс JDBC реализуется драйверами JDBC для конкретных систем управления базами данных. Реализации SPI называются *поставщиками услуг* (service providers) конкретной системы именования.

В состав Java 2 SDK версии 1.4 входят четыре реализации SPI:

- поставщик услуг RMI-реестра, вызываемый методами класса `RegistryContextFactory`;
- поставщик услуг доменной системы имен DNS, вызываемый методами класса `DnsContextFactory`;
- поставщик услуг COS Naming — сервера имен CORBA, — вызываемый методами класса `CMCtxFactory`;
- поставщик услуг протокола упрощенного доступа к каталогам LDAP, вызываемый методами класса `LdapCtxFactory`.

Эти классы находятся в подпакетах недокументированного пакета com.sun.jndi.

Другие реализации интерфейса SPI загружаются отдельно или создаются независимыми фирмами. Список таких реализаций можно посмотреть по адресу **http://java.sun.com/products/jndi/serviceproviders.html**.

Перед работой с системой JNDI надо загрузить поставщика услуг конкретной системы имен. О том, как это сделать, говорится в следующих разделах этой главы. Но сначала ознакомимся с правилами именования JNDI.

Имена

Правила записи имен JNDI такие же, как в большинстве файловых систем. Простое (atomic) имя системы JNDI — это строка символов. Составное имя состоит из простых имен, связанных *разделителем* (separator). Разделителем чаще всего служит наклонная черта — слэш, например, com/jdbc/myDB или точка, например, www.some.com, java.lang.String. В составном имени некоторые простые имена могут быть пустыми, например, имя /ABC//XYZ/ начинается с пустого имени, заканчивается пустым именем, кроме того, в его середине тоже пустое имя. Пустое имя — это пустая строка символов. Простые имена называются *компонентами* (components) составного имени.

Методы создания и удаления компонентов составного имени, выделения его частей и сравнения имен описаны интерфейсом `Name`. Этот интерфейс предписывает перенумеровывать компоненты составного имени, начиная с основного компонента, получающего номер 0. Количество компонентов возвращается методом `size()`. Компонент с номером `n`, который должен находиться в пределах от `0` до `size() – 1`, можно получить методом

```
public String get(int n);
```

Набор всех компонентов возвращает метод

```
public Enumeration getAll();
```

Часть имени, предшествующая компоненту с номером `n`, возвращается методом

```
public Name getPrefix(int n);
```

Часть имени от компонента с номером `n` включительно до последнего компонента возвращается методом

```
public Name getSuffix(int n);
```

Удалить компонент с номером `n` из имени можно методом

```
public Object remove(int n);
```

Добавить компонент перед указанной позицией `n` или в конец имени можно методами

```
public Name add(int n, String comp);
public Name add(String comp);
```

Второй из этих методов эквивалентен `add(this.size(), comp)`.

Оба метода возвращают старое имя, то, что было до добавления компонента.

Метод

```
public int compareTo(Object name);
```

сравнивает два имени одного и того же класса и возвращает отрицательное число, ноль или положительное число, если имя меньше, равно или больше имени `name` соответственно. Правила сравнения задаются реализацией метода, обычно это лексикографическое сравнение и выполнение метода сводится к применению одноименного метода класса `String`.

Интерфейс `Name` реализован двумя классами: `CompositeName` и `CompoundName`.

Имя класса *CompoundName*

Составные имена класса `CompoundName` отражают иерархическую структуру контекстов, каталогов, доменов в конкретных системах именования. В частности, иерархия может состоять только из одного уровня, образуя "плоское" (flat) пространство имен. Синтаксические элементы имени определяются

объектом класса `Hashtable` или его подкласса `Properties`, состоящим из системных свойств со следующими именами.

- `jndi.syntax.direction` — порядок следования компонентов имени. Значения `"right_to_left"`, `"left_to_right"`, `"flat"`. Последнее означает, что в имени есть только один компонент.
- `jndi.syntax.separator` — разделитель компонентов, обычно наклонная черта `"/"`. Это свойство необходимо определить, если значение свойства `jndi.syntax.direction` не равно `"flat"`.
- `jndi.syntax.separator2` — дополнительный разделитель. Если это свойство не определено, то дополнительный разделитель совпадает с основным разделителем, т. е. значением свойства `jndi.syntax.separator`.
- `jndi.syntax.ignorecase` — значение `"true"` предписывает игнорировать регистр букв при сравнении имен, значение `"false"` или отсутствие этого свойства — учитывать регистр.
- `jndi.syntax.escape` — строка символов экранирования, обычно обратная наклонная черта `"\"`.
- `jndi.syntax.beginquote` — начальный символ экранирования, обычно кавычка.
- `jndi.syntax.endquote` — конечный символ экранирования, обычно кавычка. Если это свойство не определено, то конечный символ совпадает с начальным символом, т. е. со значением свойства `jndi.syntax.beginquote`.
- `jndi.syntax.beginquote2` — дополнительный начальный символ экранирования, обычно апостроф.
- `jndi.syntax.endquote2` — дополнительный конечный символ экранирования, обычно апостроф. Если это свойство не определено, то конечный символ совпадает с начальным символом, т. е. со значением свойства `jndi.syntax.beginquote2`.
- `jndi.syntax.trimblanks` — значение `"true"` заставляет отбрасывать начальные и конечные *пробельные символы* при сравнении компонентов имени. Значение `"false"` или отсутствие этого свойства — учитывать пробельные символы при сравнении. Пробельные символы — это обычно пробел, символ горизонтальной табуляции и символ перевода строки.
- `jndi.syntax.separator.ava` — строка разделителей атрибутов имени, имеющих вид "ключ—значение" (attribute-value-assertion), например, запятая в строке `name=Иванов, age=35`.
- `jndi.syntax.separator.typeval` — строка, отделяющая ключ от значения (знак равенства в предыдущем примере).

Перед созданием экземпляра класса `CompoundName` надо определить эти свойства, например, так:

```
private Properties compProps = new Properties();

compProps.put("jndi.syntax.direction", "left_to_right");
compProps.put("jndi.syntax.separator", "/");
compProps.put("jndi.syntax.ignorecase", "false");
compProps.put("jndi.syntax.escape", "\\");
compProps.put("jndi.syntax.beginquote", "\"");
compProps.put("jndi.syntax.beginquote2", "'");
compProps.put("jndi.syntax.trimblanks", "false");
```

Затем создать экземпляр составного имени конструктором

```
CompoundName cname = new CompoundName("jdbc/myDB", compProps);
```

Этот путь представляется слишком сложным и, действительно, наиболее часто применяемый способ получить объекты-имена — извлечь их из контекста имени.

Например:

```
try{

   Context ctx = new InitialContext();

   NameParser parser = ctx.getNameParser("");

   Name compoundName = parser.parse("env/jdbc/myDB");

   for (int i = 0; i < compoundName.size(); i++)
      System.out.println(compoundName.get(i));

   ctx.close();

}catch (NamingException e){
   System.out.println(e);
}
```

Имя класса *CompositeName*

Компоненты составного имени класса `CompositeName` могут принадлежать разным системам именования, поэтому составное имя этого класса, как правило, разбивается на компоненты, после чего каждый компонент обраба-

тывается по правилам своей системы именования. Компоненты разделяются наклонной чертой и записываются слева направо, точнее говоря, при разбиении составного имени на компоненты применяются значения синтаксических элементов, которые определены в классе `CompositeName` следующим образом:

```
static private Properties compositeSyntax = new Properties();

static{
   compositeSyntax.put("jndi.syntax.direction", "left_to_right");
   compositeSyntax.put("jndi.syntax.separator", "/");
   compositeSyntax.put("jndi.syntax.ignorecase", "false");
   compositeSyntax.put("jndi.syntax.escape", "\\");
   compositeSyntax.put("jndi.syntax.beginquote", "\"");
   compositeSyntax.put("jndi.syntax.beginquote2", "'");
}
```

Кроме компонентов и слэшей в составном имени могут встречаться метасимволы: апострофы, кавычки и обратная наклонная черта — бэкслэш. Они предназначены для отмены специального значения слэша — быть разделителем простых имен в составном имени. Например, все три записи: `abc\/xyz`, `"abc/xyz"`, `'abc/xyz'` означают простое имя `abc/xyz`, содержащее слеш. Апострофы и кавычки должны встречаться в качестве метасимволов только парами — в начале и в конце компонента. Они отменяют специальное значение друг друга, например, чтобы отменить действие апострофа, применяем кавычки: `"a'bcd"` — это простое имя `a'bcd`. Его можно записать и так: `a\'bcd`. Чтобы отменить действие кавычки, применяем апострофы: `'a"bcd'`.

Составное имя класса `CompositeName` может быть пустым, оно создается конструктором по умолчанию:

```
CompositeName cname = new CompositeName();
```

Потом в него можно добавлять компоненты имени:

```
cname.add("jdbc");
cname.add("myDB");
```

Но удобнее сделать все сразу, задав имя в конструкторе:

```
CompositeName cname = new CompositeName("jdbc/myDB");
```

Класс `CompositeName` реализует все методы интерфейса `Name`, например:

```
String firstComp = cname.get(0);
for (int i = 0; i < cname.size(); i++)
   System.out.println(cname.get(i));
```

Или

```
Enumeration comps = cname.getAll();
while (comps.hasMoreElements())
   System.out.println(comps.nextElement());
```

Имя в виде строки URL

Система имен JNDI позволяет задавать имя в виде строки URL, первая часть которой, *схема*, указывает поставщика услуг конкретной системы именования:

- `"file:"` — адрес должен обрабатываться поставщиком услуг файловой системы;
- `"rmi:"` — адрес должен обрабатываться поставщиком услуг RMI-реестра;
- `"dns:"` — адрес должен обрабатываться поставщиком услуг DNS-сервера;
- `"ldap:"` — адрес должен обрабатываться поставщиком услуг LDAP-сервера;
- `"ior:"`, `"corbaname:"`, `"corbaloc:"`, `"iiopname:"`, `"iiop:"` — адрес должен обрабатываться поставщиком услуг COS Name Server;
- `"nds:"` — адрес должен обрабатываться поставщиком услуг службы каталогов NDS фирмы Novell;
- `"nis:"` — адрес должен обрабатываться поставщиком услуг сервера сетевой информационной системы NIS или NIS+;
- `"java:"` — логическое пространство имен J2EE;

Остальная часть адреса зависит от поставщика услуг и будет рассмотрена в соответствующих разделах этой главы. Рассмотрим подробнее только адрес со схемой `"java:"`.

Адрес URL со схемой `"java:"` определяет логическое пространство имен различных ресурсов, используемых компонентами приложения Java 2 Enterprise Edition — сервлетами, страницами JSP, компонентами EJB. Это логическое пространство имен создается контейнером, содержащим компоненты.

В пространстве имен `"java:"` создается корневой контекст `"comp"` для имен компонентов и их ресурсов, а в нем два вложенных контекста: `"env"` (environment) и `"UserTransaction"`. Первый контекст содержит имена ресурсов приложения, например, `"java:comp/env/EmpNames"`, а также вложенные контексты для имен компонентов определенного типа:

- `"ejb"` — имена компонентов EJB;
- `"jdbc"` — имена источников данных `DataSource` интерфейса JDBC;
- `"jms"` — соединения системы сообщений JMS;

- "mail" — соединения электронной почты JavaMail;
- "url" — соединения по адресам URL.

Например, база данных myDB, являющаяся источником данных, получает имя

```
"java:comp/env/jdbc/myDB"
```

Контекст "UserTransaction" связан с классом UserTransaction. Его используют компоненты, управляющие транзакцией.

Контекст

Методы работы с контекстом описаны интерфейсом Context. Все методы в этом интерфейсе встречаются парами — у одного метода аргумент типа Name, у другого — типа String. Основные методы осуществляют связывание имени name с объектом obj:

```
public void bind(Name name, Object obj);
public void bind(String name, Object obj);
public void rebind(Name name, Object obj);
public void rebind(String name, Object obj);
```

Методы bind() отличаются от методов rebind() только тем, что первые выбрасывают исключение класса NameAlreadyBoundException, если в контексте уже есть такое имя, а последние просто связывают с именем name другой объект.

Поиск объекта по имени name выполняет один из двух методов:

```
Object lookup(Name name);
Object lookup(String name);
```

Другие методы переименовывают объект:

```
public void rename(Name old, Name new);
public void rename(String old, String new);
```

удаляют имя из контекста:

```
public void unbind(Name name);
public void unbind(String name);
```

Методы

```
public NamingEnumeration list(Name ctxName);
public NamingEnumeration list(String ctxName);
```

возвращают список всех имен, входящих в контекст ctxName, в виде объекта, реализующего интерфейс NamingEnumeration — расширение интерфейса Enumeration.

Методы

```
public NamingEnumeration listBindings(Name ctxName);
public NamingEnumeration listBindings(String ctxName);
```

возвращают все связывания контекста, т. е. имена вместе со связанными с ними объектами.

Вложенный контекст с именем `ctxName` создается одним из методов:

```
public Context createSubcontext(Name ctxName);
public Context createSubcontext(String ctxName);
```

Перед созданием контекста можно определить несколько его свойств. Это системные свойства, имена которых начинаются с `java.naming`. В интерфейсе `Context` они описываются как строковые константы. Вот некоторые из них.

- `INITIAL_CONTEXT_FACTORY = "java.naming.factory.initial"`. Значение этого свойства — полное имя класса, создающего начальный контекст, обычно это поставщик услуг какой-нибудь системы именования.
- `PROVIDER_URL = "java.naming.provider.url"`. Это строка URL, содержащая адрес сервера имен системы именования, связанной с JNDI, или начальный контекст.
- `LANGUAGE = "java.naming.language"`. Это языки, используемые контекстом, они записываются в соответствии с рекомендацией RFC 1766 через двоеточие, например, `"en-US:ru:de"`.
- `SECURITY_PROTOCOL = "java.naming.security.protocol"`. Это защищенный протокол передачи данных, используемый JNDI, например, SSL.
- `SECURITY_AUTHENTICAION = "java.naming.security.authentication"`. Это уровень защищенности системы, одно из значений `"none"`, `"simple"`, `"strong"`.
- `SECURITY_PRINCIPAL = "java.naming.security.principal"`. Это имя владельца объектов в защищенной системе.
- `SECURITY_CREDENTIALS = "java.naming.security.credentials"`. Это пароль или иное средство проверки в защищенной системе.

По окончании работы с контекстом его рекомендуется закрыть методом `close()`, чтобы освободить занимаемые контекстом ресурсы.

Начальный контекст *InitialContext*

Интерфейс `Context` реализован классом `InitialContext`, создающим начальный контекст, в котором потом можно создавать вложенные контексты. Начальный контекст легко создать конструктором по умолчанию

```
InitialContext ctx = new InitialContext();
```

Потом заполнить контекст связываниями, задавая имена объектам:

```
ctx.bind("ivanov", st[0]);
ctx.bind("petrov", st[1]);
```

При необходимости в контексте создается вложенный контекст:

```
Context subctx = ctx.createSubcontext("emp");
```

Он тоже заполняется:

```
subctx.bind("ivanov", stud[0]);
subctx.bind("petrov", stud[1]);
```

После определения пространства имен любой клиент системы JNDI может искать в нем объект:

```
Student st = (Student)ctx.lookup("emp/ivanov");
```

Перед созданием начального контекста можно задать ему свойства, описанные в интерфейсе `Context`. Для этого надо создать объект класса `Hashtable` или `Properties`, записать в него свойства начального контекста и применить конструктор `InitialContext(Hashtable)`. Например, чтобы создать контекст JNDI, связанный с файловой системой локальной машины, надо задать реализацию SPI для файловой системы:

```
Properties prop = new Properties();

prop.put("INITIAL_CONTEXT_FACTORY",
   "com.sun.jndi.fscontext.RefFSContextFactory");

Context ctx = new InitialContext(prop);
```

После этого можно обращаться к файлам по их именам методом `lookup(String)`:

```
File ae = (File)ctx.lookup("C:\\autoexec.bat");
```

Надо сразу заметить, что обратиться к каталогу файловой системы таким же образом не удастся. Подробности изложены ниже в этой главе. Но можно просмотреть список файлов какого-либо каталога файловой системы:

```
NamingEnumeration list = ctx.list("adm\\log");

while (list.hasMore()){
   NameClassPair nc = (NameClassPair)list.next();
   System.out.println(nc);
}
```

Класс `NameClassPair` — это небольшой класс, содержащий пару "имя JNDI — имя класса Java". Имена JNDI и класса можно получить в виде строки `String` методами `getName()` и `getClassName()` этого класса.

У класса `NameClassPair` есть расширение — класс `Bindings`, хранящее пару "имя JNDI — объект". Кроме JNDI-имени и имени класса, которые можно получить унаследованными методами, этот класс позволяет получить еще и сам объект методом `getObject()`. Например:

```
NamingEnumeration bindings = ctx.listBindings("adm\\log");


while (bindings.hasMore()){
   Binding bd = (Binding)bindings.next();
   System.out.println(bd.getName() + ": " + bd.getObject());
}
```

Контекст каталога

Контекст каталога описан интерфейсом `DirContext`, расширяющим интерфейс `Context`. Поскольку контекст каталога отличается от простого контекста наличием атрибутов, многие методы интерфейса `Context` переопределены с учетом атрибутов контекста, описанных интерфейсом `Attributes`. Например:

```
public void bind(Name name, Object obj, Attributes attrs);
public void bind(String name, Object obj, Attributes attrs);
public void rebind(Name name, Object obj, Attributes attrs);
public void rebind(String name, Object obj, Attributes attrs);
public DirContext createSubcontext(Name ctxName, Attributes attrs);
public DirContext createSubcontext(String ctxName, Attributes attrs);
```

Каждый атрибут описан интерфейсом `Attribute`. Рассмотрим подробнее этот интерфейс.

Интерфейс *Attribute*

Каждый атрибут контекста каталога описан интерфейсом `Attribute`. У атрибута есть *имя* — символьная строка, описывающая его. В атрибуте содержится ноль или больше значений. Количество значений атрибута можно узнать методом `size()`. Значения могут быть упорядоченными или неупорядоченными. Упорядоченные значения могут дублироваться, различаясь номерами. Тип и представление значений атрибута определяются *схемой* (schema) атрибута. Это понятие детализируется реализациями интерфейса.

Имя атрибута возвращается методом

```
public String getID();
```

Значение добавляется в атрибут методом

```
public boolean add(Object value);
```

Если значения атрибута не упорядочены, и такое значение уже есть в атрибуте, то метод не делает ничего и возвращает `false`; если значения упорядочены, то новое значение добавляется в конец списка значений.

Если значения атрибута упорядочены, то новое значение можно вставить в определенное место списка значений методом

```
public void add(int ind, Object value);
```

Индекс `ind` должен лежать в диапазоне `[0, size()]`.

Заменить значение в упорядоченном списке значений можно методом

```
public Object add(int ind, Object value);
```

Этот метод возвращает старое значение атрибута.

Получить значение атрибута можно методами

```
public Object get();
public Object get(int ind);
```

Список всех значений возвращается методом

```
public NamingEnumeration getAll();
```

Удалить значение можно методом

```
public Object remove(int ind);
```

возвращающим удаленный объект, или методом

```
public boolean remove(Object value);
```

возвращающим `true`, если удаление сделано, и `false` в противном случае.

Удалить все значения атрибута можно методом

```
public void clear();
```

Класс *BasicAttribute*

Интерфейс `Attribute` реализован классом `BasicAttribute`. Экземпляры класса создаются его конструкторами. Два конструктора создают неупорядоченный или упорядоченный атрибут с именем `name` без значений:

```
public BasicAttribute(String name);
public BasicAttribute(String name, boolean ordered);
```

Другие два конструктора создают атрибут с именем `name` и со значением `value`:

```
public BasicAttribute(String name, Object value);
public BasicAttribute(String name, Object value, boolean ordered);
```

Методы класса просто реализуют соответствующие методы интерфейса `Attribute`.

Например:

```
BasicAttribute ba1 = new BasicAttribute("dns", "host.some.com");
BasicAttribute ba2 = new BasicAttribute("ip", "210.135.1.10");
BasicAttribute ba3 = new BasicAttribute("port", "1521");
ba3.add("1526");
```

Интерфейс *Attributes*

Интерфейс `Attributes` описывает множество всех атрибутов контекста каталога. Методы работы с этим множеством традиционны для коллекций. Два метода заносят атрибут в множество атрибутов:

```
public Attribute put(Attribute attr);
public Attribute put(String name, Object value);
```

Эти методы возвращают старое значение атрибута, или `null`, если такого значения не было.

Методы

```
public Attribute get(String name);
public NamingEnumeration getAll();
public NamingEnumeration getIDs();
```

возвращают один атрибут, множество всех атрибутов или множество идентификаторов всех атрибутов каталога контекста.

Метод

```
public Attribute remove(String name);
```

удаляет атрибут из множества, возвращая удаленное значение или `null`, если атрибута с именем `name` нет во множестве атрибутов.

Наконец, метод

```
public int size();
```

возвращает количество атрибутов.

Класс *BasicAttributes*

Класс `BasicAttributes` реализует интерфейс `Attributes`. Имена атрибутов множества этого класса могут сравниваться между собой с учетом регистра букв или без учета. Это определяется в конструкторе класса:

```
public BasicAttributes(boolean ignoreCase);
```

Конструктор по умолчанию `BasicAttributes()` и конструктор

```
public BasicAttributes(String name, Object value);
```

создают пустое множество атрибутов и множество с одним атрибутом, в котором регистр букв учитывается при сравнении имен атрибутов.

Еще один конструктор

```
public BasicAttributes(String name, Object value,
   boolean ignoreCase);
```

учитывает все эти параметры.

Методы класса просто реализуют методы интерфейса `Attributes`.

Вот пример создания и перебора атрибутов и их значений:

```
Attributes attrs = new BasicAttributes();

attrs.put(ba1);
attrs.put(ba2);
attrs.put(ba3);
attrs.put("database", "myDB");

NamingEnumeration at = attrs.getAll();

while (at.hasMore()){

   Attribute bd = (Attribute)at.next();

   System.out.print(bd.getID() + ": ");

   NamingEnumeration at1 = bd.getAll();

   while (at1.hasMore())
      System.out.print("  " + at1.next());

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

Действия с атрибутами

Интерфейс `DirContext` описывает методы получения атрибутов

```
public Attributes getAttributes(Name name);
public Attributes getAttributes(String name);
```

```
public Attributes getAttributes(Name name, String[] ids);
public Attributes getAttributes(String name, String[] ids);
```

возвращающие множество всех атрибутов объекта с именем `name` или некоторых его атрибутов, имена которых записаны в массив `ids`.

Методы

```
public void modifyAttributes(Name name, int op, Attributes attrs);
public void modifyAttributes(String name, int op, Attributes attrs);
```

изменяют множество атрибутов объекта с именем `name`. Вид изменения определяется вторым аргументом `op`, который принимает одно из трех значений:

- `ADD_ATTRIBUTE` — добавить атрибуты из множества `attrs` к множеству атрибутов объекта с именем `name`;
- `REPLACE_ATTRIBUTE` — заменить значения тех атрибутов объекта с именем `name`, которые входят в множество атрибутов `attrs`;
- `REMOVE_ATTRIBUTE` — удалить те атрибуты объекта с именем `name`, которые входят в множество атрибутов `attrs`.

Например:

```
try{
   Attributes repAttrs =
      new BasicAttributes("ip", "210.135.1.25");

   dirCtx.modifyAttributes("env/jdbc/myDB",
      DirContext.REPLACE_ATTRIBUTE, repAttrs);

}catch(NamingException ne){}
```

Методы

```
public void modifyAttributes(Name name, ModificationItem[] mods);
public void modifyAttributes(String name, ModificationItem[] mods);
```

последовательно выполняют модификации, занесенные в массив модификаторов, выраженных объектами класса `ModificationItem`. Это небольшой класс, содержащий пару "вид изменения — изменяемый атрибут", которая заносится в конструктор класса:

```
public ModificationItem(int op, Attribute attr);
```

Например:

```
ModificationItem[] mods = new ModificationItem[2];

mods[0] = new ModificationItem(
   DirContext.ADD_ATTRIBUTE,
   new BasicAttribute("name", "Иванов"));
```

```
mods[1] = new ModificationItem(
   DirContext.MODIFY_ATTRIBUTE,
   new BasicAttribute("ip", "210.135.1.25"));

dirCtx.modifyAttributes("env/jdbc/myDB", mods);
```

Поиск объектов в каталоге контекста

Хотя интерфейс `DirContext` наследует методы `lookup()` своего суперкласса `Context`, но для поиска объекта в контексте каталога `ctxName` по атрибутам `attrs` он описывает метод

```
public NamingEnumeration search(Name ctxName, Attributes attrs);
```

Этот метод и другие методы `search()` возвращают совокупность результатов поиска. Каждый результат представлен в виде объекта класса `SearchResult`. Это небольшой класс, расширяющий класс `Bindings`. Кроме объекта и его имени, которые хранятся в объекте класса `Bindings`, класс `SearchResult` хранит множество атрибутов объекта.

Метод

```
public NamingEnumeration search(Name ctxName,
   Attributes attrs, String[] retAttrs);
```

возвращает не все множество атрибутов найденных объектов, а только те атрибуты, которые указаны в массиве `retAttrs`.

Метод

```
public NamingEnumeration search(Name ctxName,
   String filter, SearchControls cons);
```

использует при поиске фильтр `filter` и условия поиска `cons`.

Фильтр задается строкой пар "атрибут—значение", оформленной согласно рекомендации RFC 2254, например, `"age=32"`. Кроме равенства можно указать приблизительное равенство (`~=`), больше или равно (`>=`), меньше или равно (`<=`). В качестве значения атрибута можно записать звездочку (`*`), означающую "любое значение". Каждая пара заключается в скобки, пары могут быть связаны логическими операциями конъюнкции (`&`), дизъюнкции (`|`) и отрицания (`!`). Знаки операций записываются перед операндами — используется обратная польская запись. Например:

```
String filter = "(&(name=И*)(age=60))";
```

Интерпретация операций и их выполнение зависят от поставщика услуг конкретной системы каталогов. Если поставщик услуг не может выполнить ту или иную операцию, то выбрасывается исключение класса `InvalidSearchFilterException`.

Условия поиска содержатся в объекте класса `SearchControls` и заносятся в конструктор класса

```
public SearchControls(int scope, long countlim, long timelim,
   String[] attrs, boolean retobj, boolean deref);
```

Здесь:

- `scope` — это одно из трех значений:
  - `OBJECT_SCOPE` — вернуть не более одного значения;
  - `ONELEVEL_SCOPE` — не просматривать подкаталоги;
  - `SUBTREE_SCOPE` — просмотреть рекурсивно все поддерево каталогов;
- `countlim` — наибольшее число возвращаемых объектов. Значение 0 указывает, что надо вернуть все найденные значения;
- `timelim` — наибольшее время в миллисекундах, выделенное на поиск. Ноль означает неограниченное время;
- `attrs` — набор атрибутов, которые надо вернуть в результате поиска. Значение `null` предписывает вернуть все атрибуты, пустое значение — не возвращать атрибуты;
- `retobj` — возвращать или не возвращать сам найденный объект;
- `deref` — не разрешать ссылки во время поиска.

Четвертый метод

```
public NamingEnumeration search(Name ctxName, String filter,
   Object[] args, SearchControls cons);
```

использует в аргументе `filter` выражения вида `{n}`, где `n` — неотрицательное число. Эти выражения перед поиском заменяются элементом массива `args[n]`.

Еще четыре таких же метода задают имя контекста `ctxName` не объектом, реализующим интерфейс `Name`, а строкой класса `String`.

Реализация контекста каталога

Интерфейс `DirContext` реализован классом `InitialDirContext`, который расширяет класс `InitialContext`. Поэтому работа с этим классом аналогична работе с классом `InitialContext`.

Экземпляры класса создаются конструктором по умолчанию `InitialDirContext()` или конструктором

```
public InitialDirContext(Hashtable env);
```

Методы класса унаследованы от класса `InitialContext` или реализуют методы интерфейса `DirContext`, не добавляя к ним ничего нового.

Листинг 17.1 показывает большой пример работы с контекстом каталога JNDI. В нем используется поставщик услуг LDAP.

Листинг 17.1. Работа с контекстом каталога

```
import javax.naming.*;
import javax.naming.directory.*;
import java.util.*;

public class SimpleDirSearch{

  DirContext context = null;
  Properties properties = new Properties();

  public SimpleDirSearch(Properties p){
    properties = p;
  }

  public DirContext getInitialContext()
                   throws NamingException{

    if (context == null){
      System.out.println("Properties :" + properties);

      context = new InitialDirContext(properties);
      System.out.println("Контекст: " + context);
    }
    return context;
  }

  public static void printSearchResults(Attributes attrs)
    throws NamingException{

      if (attrs == null)
        System.out.println("Атрибуты отсутствуют. ");
      else{
        NamingEnumeration ne = attrs.getAll();
        while (ne.hasMore()){
```

```
        Attribute attr = (Attribute)ne.next();
        String id = attr.getID();

        NamingEnumeration vals = attr.getAll();
        while (vals.hasMore())
          System.out.println(id + ": " + vals.next());
      }
    }
}

public static void printSearchResults(NamingEnumeration enum)
                                  throws NamingException{
  while (enum != null && enum.hasMore()){
    SearchResult sr = (SearchResult)enum.next();

    System.out.println("Результат: " + sr.getName() );

    Attributes attrs = sr.getAttributes();

    if(attrs == null)
      System.out.println("Атрибуты отсутствуют.");
    else{
      NamingEnumeration ne = attrs.getAll();
      while (ne.hasMore()){

        Attribute attr = (Attribute)ne.next();
        String id = attr.getID();

        NamingEnumeration vals = attr.getAll();
        while (vals.hasMore())
          System.out.println(id + ": " + vals.next());
      }
    }
    System.out.println();
  }
}

public void searchUsingFilter(String filter){
  try{
    DirContext dirCtx = getInitialContext();
```

```
    SearchControls sc = new SearchControls();
    String[] resAttrs = {"cn", "sn", "telephonenumber"};

    sc.setReturningAttributes(resAttrs);

    sc.setCountLimit(10);

    NamingEnumeration res =
      dirCtx.search("ou=People ", filter, sc);
    printSearchResults(res);

  }catch(NamingException ne){
    System.err.println(ne);
  }
}

public void addANewAttribute(){
  try{
    DirContext dirCtx = getInitialContext();

    Attribute cl = new BasicAttribute("objectclass");

    cl.add("top");
    cl.add("person");
    cl.add("organizationalPerson");
    cl.add("inetOrgPerson");

    Attribute cn = new BasicAttribute("cn", "Иван Петров");
    Attribute sn = new BasicAttribute("sn", "Петров");
    Attribute telnumb = new BasicAttribute(
      "telephonenumber", "7654321");
    Attribute givenNames =
        new BasicAttribute("givenname", "Иван");

    String dn = "uid=petr, ou=People";

    Attributes org = new BasicAttributes();
    org.put(cl);
```

```
      org.put(cn);
      org.put(sn);
      org.put(telnumb);
      org.put(givenNames);

      dirCtx.createSubcontext(dn, org);

    }catch(NamingException ne){
      System.err.println(ne);
    }
  }

  public void replaceAttributes(){
    try{
      DirContext dirCtx = getInitialContext();

      ModificationItem modItems[] =
         new ModificationItem[2];

      modItems[0] = new ModificationItem(
        DirContext.REPLACE_ATTRIBUTE,
        new BasicAttribute("sn", "Сидоров"));

      modItems[1] = new ModificationItem(
        DirContext.REPLACE_ATTRIBUTE,
        new BasicAttribute("telephonenumber", "1234567"));

      dirCtx.modifyAttributes(
         "uid=sid, ou=People ", modItems);

      String filter = "(&(sn=Сидоров)(email=*))";

      searchUsingFilter(filter);

    }catch(NamingException ne){
      System.err.println(ne);
    }
  }
```

```
public void removeAttributes(){
  try{
    DirContext dirCtx = getInitialContext();

    ModificationItem modItems[] =
       new ModificationItem[1];

    modItems[0] = new ModificationItem(
      DirContext.REMOVE_ATTRIBUTE,
      new BasicAttribute("telephonenumber"));

    dirCtx.modifyAttributes("uid=petr, ou=People ",
      modItems);

    String filter = "(&(sn=Doe)(givenname=*))";

    searchUsingFilter(filter);

  }catch(NamingException ne){
    System.err.println(ne);
  }
}

public void getSelectedAttributesUsingGetAttributes(){
  try{
    DirContext dirCtx = getInitialContext();

    String resAttrs[] =
      {"uid", "givenname", "sn", "telephonenumber"};

    Attributes res =  dirCtx.getAttributes(
       "uid=sid,ou=People ", resAttrs);

    printSearchResults(res);

  }catch(NamingException ne){
    System.err.println(ne);
  }
}
```

```
public void getEveryAttribute(){
  try{
    DirContext dirCtx = getInitialContext();

   Attributes res =
      dirCtx.getAttributes("uid=sid, ou=People");
   printSearchResults(res);

  }catch(NamingException ne){
    System.err.println(ne);
  }
}

public void getAttrs(){
  try{
    DirContext dirCtx = getInitialContext();

    Attributes matchAttrs = new BasicAttributes(true);
    matchAttrs.put(new BasicAttribute("uid", "sid"));
    matchAttrs.put(new BasicAttribute("givenname"));

    NamingEnumeration searchResults =
       dirCtx.search("ou=People", matchAttrs);

    printSearchResults(searchResults);

  }catch(NamingException ne){
    System.err.println(ne);
  }
}

public void getSelectedAttributes(){
  try{
    DirContext dirCtx = getInitialContext();

    BasicAttribute atr1 =
       new BasicAttribute("uid", "sid");

    BasicAttribute atr2 =
        new BasicAttribute("givenname");
    Attributes matchAttrs = new BasicAttributes(true);
```

```
      matchAttrs.put(atr1);
      matchAttrs.put(atr2);

      String resAttrs[] =
        {"uid", "givenname", "sn", "telephonenumber"};

      NamingEnumeration res =
         dirCtx.search("ou=People", matchAttrs, resAttrs);
      printSearchResults(res);

    }catch(NamingException ne){
      System.err.println(ne);
    }
  }

  public static void main(String[] args){

    Properties env = System.getProperties();

    env.put(Context.INITIAL_CONTEXT_FACTORY,
           "com.sun.jndi.ldap.LdapCtxFactory");

    env.put(Context.PROVIDER_URL,
       "ldap://some.firm.com:389/o=firm.com");

    SimpleDirSearch sds =
      new SimpleDirSearch(env);

    sds.getEveryAttribute();
    sds.getSelectedAttributes();

    String filter = "(&(sn=petr)(mail=*))";
    sds.searchUsingFilter(filter);

    sds.getAttrs();

    sds.addANewAttribute();
    sds.removeAttributes();
    sds.replaceAttributes();
  }
}
```

Ссылки на объекты

Если объект не входит в текущую систему имен или каталогов или он настолько велик, что система JNDI не может его хранить, то система JNDI хранит ссылку на него. Каждый такой объект должен обеспечить возможность ссылки на него, реализовав метод

```
public Reference getReference();
```

интерфейса `Referenceable`.

Класс `Reference`, экземпляр которого должен вернуть этот метод, содержит все сведения, необходимые для поиска объекта: упорядоченный список адресов объекта, полное имя класса объекта `className`, имя класса-фабрики, создающего объект `factory`, и местоположение этого класса-фабрики `factoryLoc`. Эти сведения заносятся в конструктор класса

```
public Reference(String className, RefAddr addr,
   String factory, String factoryLoc);
```

Как видно из описания аргументов конструктора, адрес объекта `addr` задается экземпляром класса `RefAddr`. Это абстрактный класс, содержащий кроме адреса объекта еще и тип адреса. Для получения объекта класс `RefAddr` содержит абстрактный метод

```
public abstract Object getContent();
```

который реализован в подклассах `BinaryRefAddr` и `StringRefAddr` класса `RefAddr`.

Класс `BinaryRefAddr` содержит двоичный адрес, например, MAC-адрес сетевой карты, в виде массива байтов `src`, задаваемого в конструкторе

```
public BinaryRefAddr(String addrType, byte[] src);
```

Класс `StringRefAddr` содержит адрес в виде строки символов `addr`, задаваемой в конструкторе

```
public StringRefAddr(String addrType, String addr);
```

Например:

```
public class MyRefObject implements Referenceable{

   public Reference getReference() throws NamingException{

      StringRefAddr sra =
         new StringRefAddr("URL", "file:/java/myclasses");
```

```
        return new Reference("MyRefObject", sra, null, null);
    }

    // Прочие методы
}
```

Теперь можно включить ссылку на объект `MyRefObject` в систему имен JNDI, как обычно, методом `rebind()`:

```
MyRefObject mr = new MyRefObject();
ctx.rebind("myobject", mr);
```

Затем объект можно получить следующим образом:

```
MyRefObject newmr = (MyRefObject)ctx.lookup("myobject");
```

В том случае, когда адрес состоит из одной строковой ссылки, как, например, адрес URL или имя типа `CompositeName`, записанное в виде строки, вместо класса `Reference` удобнее использовать его расширение — класс `LinkRef`. Этот класс хранит один адрес типа "LinkAddress". В его конструкторах

```
public LinkRef(Name addr);
public LinkRef(String addr);
```

просто задается ссылка `addr` в виде строки или объекта типа `Name`.

Поставщик услуг RMI

Реализация SPI для работы с RMI-реестром входит в стандартную поставку J2SE SDK. Для ее активизации следует определить системное свойство `java.naming.factory.initial` следующим образом:

```
Properties env = new Properties();

env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.registry.RegistryContextFactory");
```

Кроме того, надо задать адрес RMI-реестра:

```
env.put(Context.PROVIDER_URL, "rmi://server:1099");
```

Если это свойство не определено, то будет использован адрес `"rmi:"`, означающий RMI-реестр, работающий на той же самой машине `localhost` и прослушивающий порт с номером 1099.

Вместо того чтобы определять эти свойства, можно всякий раз указывать полный адрес вида "rmi://[server]:[port][/[object]]" или "rmi:[/][object]" с теми же правилами умолчания.

Пространство имен, зарегистрированных в RMI-реестре, "плоское" (flat), поэтому все эти имена рассматриваются системой JNDI как объекты типа CompositeName, но простые, состоящие из одного компонента. При сравнении имен учитывается регистр букв. Таким образом, установлены следующие значения синтаксических элементов:

```
private Properties props = new Properties();

props.put("jndi.syntax.direction", "flat");
props.put("jndi.syntax.ignorecase", "false");
props.put("jndi.syntax.escape", "\\");
props.put("jndi.syntax.beginquote", "\"");
props.put("jndi.syntax.beginquote2", "'");
```

Поэтому точки, слэши и прочие спецсимволы, встречающиеся в именах RMI-реестра, следует экранировать обратной наклонной чертой, апострофами или кавычками. По той же причине нельзя создавать вложенные контексты.

При выполнении методов bind(), lookup() и прочих методов, описанных интерфейсом Context, поставщик услуг RMI обращается к соответствующим методам класса Registry системы RMI. Следовательно, каждый объект, хранящийся в контексте RMI, должен реализовывать интерфейс Remote. В контексте можно хранить еще и ссылки на объекты, реализующие интерфейс Referenceable.

Само пространство имен RMI-реестра всегда реализует интерфейс Referenceable, поэтому можно создать ссылку на RMI-реестр в каком-нибудь контексте JNDI. Такая ссылка содержит адреса класса StringRefAddr. Тип каждого адреса — "URL". Он должен быть строкой с адресом URL вида "rmi://host:port/object" и содержать адрес RMI-реестра или удаленного объекта, зарегистрированного в нем. Например:

```
Context regCtx = (Context)ctx.lookup("rmi://host");
```

Поставщик услуг файловой системы

Фирма Sun Microsystems разработала реализацию SPI для работы с файловой системой. Эта реализация ко времени написания книги не входит в стандартную поставку Java 2 SDK, но может быть свободно получена с адреса **http://java.sun.com/products/jndi/serviceproviders.html**. В поставку входят два

файла fscontext.jar и providerutil.jar, которые надо поместить в каталог $JAVA_HOME/jre/lib/ext или занести путь к этим файлам в переменную `CLASSPATH`.

Для начала работы с поставщиком услуг надо определить системное свойство `java.naming.factory.initial` следующим образом:

```
Properties env = new Properties();


env.put(Context.INITIAL_CONTEXT_FACTORY,
       "com.sun.jndi.fscontext.FSContextFactory");
```

Класс `FSContextFactory` создает начальный контекст для доступа к файловой системе. Это доступ только для чтения, т. е. для просмотра файловой системы. Если нужен поиск файлов и хранение объектов Java в файловой системе, то вместо класса `FSContextFactory` надо указать его подкласс `RefFSContextFactory`.

Вместо определения этого системного свойства можно всякий раз задавать адрес объекта строкой URL со схемой `"file:"`.

Системное свойство `java.naming.provider.url` задает каталог, который будет служить текущим контекстом аналогично текущему каталогу файловой системы. Например, после:

```
env.put(Context.PROVIDER_URL, "file:D:\\tmp");
Context ctx = new InitialContext(env);
File f = (File)ctx.lookup("admlog.txt");
```

будет происходить поиск файла D:\tmp\admlog.txt.

Если это свойство не определено, то текущим каталогом будет корневой каталог файловой системы.

Эта реализация SPI файловой системы определяет каталог файловой системы как контекст JNDI, т. е., объект типа `Context`, а подкаталог — как вложенный контекст. Она осуществляет доступ к файлам через объекты класса `java.io.File`, следовательно, во-первых, может работать только с файловыми системами, к которым есть доступ через этот класс. Во-вторых, это означает, что объектами, хранящимися в контекстах, будут объекты класса `File`. В-третьих, разделителем компонентов следует ставить строку, хранящуюся в статическом поле `File.separator`. Например:

```
File f = (File)ctx.lookup(
   "log" + java.io.File.separator + "report.txt");
```

Это обеспечит правильную запись составных имен в любой файловой системе.

Поставщик услуг DNS

Реализация SPI для доменной системы имен входит в стандартную поставку J2SE SDK. Для начала работы с этой реализацией надо определить системное свойство `java.naming.factory.initial` следующим образом:

```
Properties env = new Properties();

env.put(Context.INITIAL_CONTEXT_FACTORY,
       "com.sun.jndi.dns.DnsContextFactory");
```

Вместо определения этого системного свойства можно обращаться к объектам, записывая адрес строкой URL вида

```
"dns:[//host:[port]][/domain]"
```

В этой строке по умолчанию принимается значение компьютера `localhost`, порт с номером 53 и домен текущего контекста.

Второе системное свойство `java.naming.provider.url` определяет адреса начальных DNS-серверов и начальные контексты для него, например:

```
env.put(Context.PROVIDER_URL,
   "dns://serv1.some.com/ dns://serv2.some.com/");
```

Адреса записываются через пробелы, следовательно, в самих адресах пробелов быть не должно.

Если это свойство не определено, то устанавливается строка `"dns:"`.

Данная реализация SPI представляет каждое доменное имя в виде объекта, реализующего интерфейс `Name`. Компоненты имени перенумерованы справа налево и разделены точкой. Символом экранирования служит обратная наклонная черта. При сравнении имен регистр букв не принимается во внимание. Точнее говоря, установлены следующие свойства:

```
private Properties props = new Properties();

props.put("jndi.syntax.direction", "right_to_left");
compProps.put("jndi.syntax.separator", ".");
compProps.put("jndi.syntax.ignorecase", "true");
compProps.put("jndi.syntax.escape", "\\");
```

Каждый узел DNS представляется объектом типа `DirContext`. Записи ресурсов DNS-узла RR (resource records), сделанные по рекомендациям RFC 1035, RFC 1886, RFC 2168 и RFC 2782, становятся атрибутами контекста каталога. Например, IP-адрес 217.110.1.27 — это запись RR типа "A". Он будет храниться как атрибут с именем "A" и значением "217.110.1.27". IP-адрес версии IPv6 — это запись RR типа "AAAA", он будет храниться как атрибут с именем "AAAA". Имена атрибутов сравниваются без учета регистра букв.

Таким образом, чтобы получить атрибуты контекста, надо указать их имена согласно перечисленным рекомендациям RFC, например:

```
DirContext dctx = new InitialDirContext();

Attributes attrs = dctx.getAttributes(
   "dns://server.some.com/host.some.com",
    new String[] {"A", "AAAA", "MX"});
```

Следующий пример показывает просмотр корневого каталога сайта **host.firm.com**. Обратите внимание на завершающую наклонную черту — имя корневого каталога:

```
NamingEnumeration list = ctx.list("dns://host.firm.com/");
```

Последний пример — поиск файла на сайте **host.firm.com**:

```
File f = (File)ctx.lookup(
   "dns://host.firm.com/public/html/main/html");
```

Поставщик услуг NIS

Реализация SPI для сетевой информационной системы NIS (Network Information Service) не входит в стандартную поставку J2SE SDK, но ее можно свободно загрузить с адреса **http://java.sun.com/products/jndi/serviceproviders.html**. Файл nis.jar надо поместить в каталог $JAVA_HOME/jre/lib/ext/ или занести путь к этому файлу в переменную `CLASSPATH`.

Для начала работы с поставщиком услуг NIS надо определить системное свойство `java.naming.factory.initial` следующим образом:

```
Properties env = new Properties();

env.put(Context.INITIAL_CONTEXT_FACTORY,
       "com.sun.jndi.nis.NISCtxFactory");
```

и задать адрес NIS-сервера:

```
env.put(Context.PROVIDER_URL, "nis://some.firm.com/some.domain");
```

Данная реализация хранит объекты NIS в виде `DirContext`. В начальном контексте есть вложенный контекст с именем `system`, хранящий связывания с объектами NIS. Для имени `system/passwd.byname` создается псевдоним `user`, а для имени `system/hosts.byname` — псевдоним `host`.

Информационная система NIS не предназначена для изменения данных, поэтому методы типа `bind()` и `createSubcontext()` не выполняются в этой реализации SPI NIS.

Поставщик услуг LDAP

В конце 80-х годов прошлого столетия был разработан стандарт ISO системы каталогов X.500. Это большой и сложный стандарт, который ввел понятие *схемы* (schema) — набора правил, определяющих тип объектов, хранящихся в каталоге, возможность и способ создания подкаталогов, атрибуты каталога, их значения и пр. Стандарт X.500 определяет, в частности, протокол доступа к каталогам *DAP* (Directory Access Protocol). Это протокол прикладного уровня над TCP или UDP, тоже сложный и запутанный, как и весь стандарт X.500.

В середине 90-х годов попытка упростить протокол DAP привела к созданию облегченного протокола *LDAP* (Lightweight Directory Access Protocol), который получил большое распространение. На этом протоколе основана служба каталогов NDS (Netscape Directory Server). Служба каталогов NDS (Novell Directory Services) фирмы Novell основана на протоколе X.500 и совместима с LDAP.

Протокол LDAP определяет дифференцированные имена DN (distinguished names) вида

```
"cn=Иван Петрович, sn=Сидоров, o=Газпром"
```

состоящие из относительных (relative) дифференцированных имен RDN, разделенных запятыми. Каждое RDN имеет вид

```
"имя=значение"
```

и хранится в каталоге как атрибут контекста с именем, например, `cn` и значением `"Иван Петрович"`. Атрибуты пронумерованы справа налево.

Имена атрибутов — это строки символов, в которых не различаются регистры букв. Некоторые имена атрибутов общеприняты:

- `c` (country) — страна;
- `o` (organization) — организация;
- `ou` (organization unit) — подразделение;
- `cn` (common name) — имя объекта;
- `sn` (surname) — фамилия.

Значения атрибутов — это или символьные строки или массивы байтов.

Таким образом, имя DN протокола LDAP хорошо описывается интерфейсом `DirContext`.

Услуги системы каталогов LDAP предоставляются LDAP-серверами, работающими по протоколу TCP и прослушивающими по умолчанию порт с номером 389.

Третья версия протокола LDAPv3, описанная рекомендациями RFC 2251—RFC 2256, предоставляет дополнительные услуги ("extended" operations). Например, LDAP-сервер может вернуть результат запроса в отсортированном виде или добавить к результату время выполнения запроса. Для использования дополнительных услуг протокола LDAPv3 разработан пакет интерфейсов и классов javax.naming.ldap. Его основа — интерфейс `LdapContext`, расширяющий интерфейс `DirContext`. В интерфейс `LdapContext` добавлены методы запроса и получения дополнительных услуг, предоставляемых LDAP-сервером. Этот интерфейс реализован классом `InitialLdapContext`.

Мы не можем в этой книге подробно заниматься работой с системой каталогов LDAP, во-первых, потому, что это сильно увеличит объем книги, во-вторых, потому, что документация JNDI дает много примеров работы с LDAP по которым можно подробно ознакомиться с этой системой.

Поставщик услуг LDAP входит в стандартную поставку J2SE SDK. Он полностью реализует протокол LDAPv3. Для работы с ним достаточно определить системное свойство `java.naming.factory.initial` следующим образом:

```
Properties env = new Properties();

env.put(Context.INITIAL_CONTEXT_FACTORY,
       "com.sun.jndi.ldap.LdapCtxFactory");
```

и задать адрес LDAP-сервера примерно так:

```
env.put(Context.PROVIDER_URL,
       "ldap://some.firm.com:389/o=firm.com");
```

Эта реализация SPI представляет имя LDAP как объект класса `CompositeName`, в котором само имя является первым, левым, компонентом, а остальные компоненты, расположенные после разделителя — наклонной черты — относятся к другим системам именования.

Поставщик услуг COS Naming

Служба имен Naming Service, входящая в состав основных служб COS (Common Object Services) системы распределенных объектов CORBA, очень хорошо сочетается с системой имен и каталогов JNDI, потому что она тоже создавалась для работы с конкретными системами именования: X.500 DAP, NIS+, LDAP. Она тоже определяет составные имена объектов CORBA и хранит связывания имен с этими объектами в контекстах.

В середине 90-х годов прошлого века фирмы Sun и IBM разработали расширение службы имен COS Naming: переносимую службу имен *INS* (Interoperable Naming Service), описанную спецификацией INS. Имена в INS задаются в виде строки URL со схемой `corbaloc:` или `corbaname:`.

Схема `corbaloc:` позволяет обратиться к стандартным службам CORBA, например, адрес службы TraderService, расположенной по адресу **some.firm.com**, выглядит так: `"corbaloc:iiop:some.firm.com:2345/TraderService"`.

Схема `corbaname:` используется для обращения к объектам CORBA, например, строка `"corbaname::some.firm.com:2345#personal/emp"` задает обращение к объекту по имени `personal/emp` в корневом контексте.

Кроме этих двух схем сохраняются устаревшие схемы `iiop:` и `iiopname:`, задающие имена вида `"iiop://[host:[port]][/corbaObject]"` и `"iiopname://[<addr_list>][/corbaObject]"`. По умолчанию задается `localhost`, порт с номером 900 для `iiop:`, порт 9999 для `iiopname:` и корневой контекст.

Реализация SPI для сервера имен входит в стандартную поставку J2SE SDK. Для начала работы с этой реализацией надо определить системное свойство `java.naming.factory.initial` следующим образом:

```
Properties env = new Properties();

env.put(Context.INITIAL_CONTEXT_FACTORY,
       "com.sun.jndi.cosnaming.CNCtxFactory");
```

Адрес посредника ORB и начальный контекст можно задать системным свойством `java.naming.provider.url`, например, так:

```
env.put(Context.PROVIDER_URL,
       "iiop://some.firm.com:900/");
```

В этой реализации поставщика услуг имена INS представляются составными именами типа `CompositeName`, в которых разделителем простых имен служит наклонная черта, а простое имя делится точкой на имя и вид. При сравнении имен учитывается регистр букв.

Глава 18

Средства электронной почты

Система обмена сообщениями JMS, рассмотренная в *главе 16*, позволяет передавать сообщения разным частям распределенного приложения и другим приложениям Java, связанным с поставщиком услуг JMS. В составе J2EE SDK есть еще одна аналогичная система — система электронной почты *JavaMail*. Она позволяет получать и отправлять стандартные почтовые сообщения, описанные в рекомендациях RFC 822, RFC 2045—2047 и оформленные по правилам обычных почтовых протоколов SMTP, POP3, IMAP4.

Службы электронной почты состоят из двух частей: почтового клиента, называемого по стандарту X.400 *почтовым агентом пользователя* MUA (Mail User Agent) и почтового сервера, называемого *агентом пересылки почты* MTA (Mail Transfer Agent).

Почтовый клиент принимает и отправляет сообщения. Для создания и чтения электронных сообщений он или вызывает текстовый редактор, или содержит текстовый редактор в своем составе. У него обычно графический интерфейс, позволяющий правильно написать или прочитать адрес и оформить заголовок письма.

Почтовый сервер принимает письма от клиента, сортирует их по адресам, пакует и отправляет адресатам.

Деление программ электронной почты на MUA и MTA еще более условно, чем деление приложений на клиентскую и серверную части. Многие MUA выполняют некоторые функции почтового сервера, храня и сортируя почту. К таким развитым почтовым агентам пользователя относятся известные программы-"мейлеры" The Bat!, Outlook Express, Eudora, PM-Mail. Классический почтовый сервер — это sendmail. Кроме него, популярны почтовые серверы Qmail, Postfix, Exim.

Система электронной почты JavaMail, входящая в состав J2EE SDK, содержит набор интерфейсов и классов, позволяющий создать почтового клиента с некоторыми возможностями почтового сервера. Эти интерфейсы и классы собраны в четыре пакета.

Пакет javax.mail содержит интерфейсы и абстрактные классы, описывающие понятия сообщения `Message`, его частей `Part`, его заголовка `Header` и тела

`BodyPart`, адреса `Address`, сеанса связи `Session`, реализации протокола `Transport`. Этот пакет предоставляет каркас для создания своей реализации почтовой системы, ориентированной на какие-то конкретные протоколы.

Пакет javax.mail.internet содержит расширения классов из пакета javax.mail, реализующие стандарт *MIME* (Multipurpose Internet Mail Extensions) — классы `MimeMessage`, `ContentType`, `MimeBodyPart`, `MimeMultipart` и т. д. Эти полностью реализованные классы можно использовать в своем приложении.

Компоненты электронной почты JavaMail сигнализируют о своих действиях событиями классов `MailEvent`, `StoreEvent`, `FolderEvent`, `TransportEvent` и другими, собранными в пакет javax.mail.event. С их помощью можно получить сведения о процессе доставки письма, его хранении и пересылки, и как-то отреагировать на эти сведения.

Четвертый пакет, javax.mail.search, содержит средства поиска писем в папках и отбора их для посылки получателю. К ним относятся классы `HeaderTerm`, `BodyTerm`, `DateTerm`, `AddressTerm`, `SubjectTerm`, `SearchTerm` и другие классы, задающие разные критерии поиска.

Рассмотрим средства создания, хранения, отправки и получения писем, предоставляемые этими пакетами, но сначала сделаем некоторые замечания.

Замечания о кодировке

Протокол *SMTP*, по рекомендации RFC 821, пересылает только 7-битные сообщения длиной не более 1000 байтов, используя восьмой бит для контроля четности или нечетности. Поэтому, по рекомендации RFC 822, почтовое сообщение должно быть записано в символьном виде с использованием только 7-битных символов US-ASCII, т. е. исходная строка Java должна содержать только символы Unicode в диапазоне \u0000—\u007F.

Это слишком сильное ограничение постарались устранить рекомендации RFC 2045—2047, описывающие стандарт MIME. По этому стандарту в письме можно пересылать не только текст, но и изображения, музыку и вообще любые бинарные сообщения. Каждый тип сообщения имеет несколько подтипов, конкретизирующих этот тип, например, текстовое сообщение можно записать в формате HTML, SGML и множестве других форматов. У подтипа могут быть параметры, например, кодировка текста.

Получатель должен знать тип и подтип пришедшего сообщения, чтобы вызвать для его просмотра соответствующую программу: текстовый редактор, графический редактор, музыкальный проигрыватель или что-то еще. Для этого в заголовок письма введено поле `Content-Type`. Рекомендация RFC 2046 ввела семь типов сообщений — значений этого поля.

- `text` — текстовое сообщение, его подтипы `plain` — неразмеченный текст, который может просмотреть любой текстовый редактор,

richtext — текст, размеченный по стандарту SGML (Standard Generalized Markup Language), html, xml — частные случаи размеченного текста, и другие подтипы. Для указания кодировки байтовых символов используется параметр подтипа charset.

- image — изображение, его подтипы gif, jpeg, tiff и др.
- audio — звук, его подтипы basic, au, wav, x-midi и др.
- video — фильм, его подтипы mpeg, quicktime и др.
- application — просто последовательность байтов, подтипы octet-stream, msword, rtf, pdf, x-tar, x-gtar, x-cpio, zip, x-tex, x-latex и множество других.
- multipart — в сообщении содержится несколько сообщений разных типов. Подтип mixed означает, что эти сообщения не зависят друг от друга, подтип alternative — что это одно и то же сообщение, записанное в разных типах. Подтип parallel указывает, что все сообщения надо просматривать одновременно, например, текст и музыку к нему. Подтип digest говорит о том, что тип всех сообщений message/rfc822. Другие подтипы: http, news, partial, report и т. п.
- message — сообщение, которое по разным причинам, указываемым подтипом, не может быть передано обычным образом: подтип partial указывает на то, что это только часть большого сообщения, подтип external-body говорит о том, что это ссылка на какое-то большое сообщение, подтип rfc822 соответствует письму, составленному по рекомендации RFC 822.

В записи типов и подтипов не различаются регистры букв. Подтип отделяется от типа наклонной чертой и должен указываться обязательно. Если подтип может быть любым, то записывается звездочка, например, text/*. Значение параметра подтипа отделяется от его имени знаком равенства. В значениях параметров регистр букв, как правило, различается. Если значение содержит спецсимволы, то его надо записывать в кавычках. Обычно на принимающей стороне есть таблица нескольких вариантов записи значения каждого параметра.

Значение атрибута заголовка отделяется от его имени двоеточием, а параметры подтипа отделяются от него точкой с запятой, например:

```
Content-Type: text/plain; charset=koi8-r
```

Количество типов и подтипов постоянно растет. Все новые типы и подтипы должны быть зарегистрированы в организации IANA (Internet Assigned Numbers Authority) по правилам, указанным в рекомендации RFC 2048. Допускается использование незарегистрированных типов и подтипов, они должны начинаться с буквы x и дефиса, например, x-rgb, x-cgi-movie.

Если в заголовке письма нет поля `Content-Type`, то по умолчанию понимается

```
Content-Type: text/plain; charset=us-ascii
```

Все старые почтовые системы работают с 7-битными кодами. Поэтому все 8-битные сообщения должны быть перекодированы в 7-битный код US-ASCII. Есть несколько способов перекодировки: uuencode, base64, QP (quoted-printable). В письме, кроме типа и подтипа сообщения, записанного с использованием всех восьми битов каждого байта, следует указать способ обратного восстановления восьмого бита. Эта информация приводится в поле заголовка `Content-Transfer-Encoding`. Значения `7bit`, `8bit`, `binary` этого поля означают, что перекодировка не нужна. Стандартные способы кодировки записываются значениями `uuencode`, `base64` или `quoted-printable`. Они описаны в рекомендации RFC 2045. Кодировка пользователя отмечается значением `x-token`. Итак, поле перекодировки может иметь такой вид:

```
Content-Transfer-Encoding: base64
```

Перекодировка символов Unicode, используемых в Java, в 7-битный код, выполняется в два этапа. Сначала строка типа `String` преобразуется в массив байтов в какой-либо байтовой кодировке, например, так:

```
String s = "Какой-то текст.";
byte[] text = s.getBytes("KOI8_R");
```

Потом массив байтов переводится в 7-битную кодировку с помощью кодировки base64 (обозначается буквой B) или кодировки quoted-printable (обозначается буквой Q). Кодировка `Q` применяется, если у большинства байтов массива старший бит равен нулю, кодировка `B` — в противном случае. Кодирование выполняется методами класса `MimeUtility`.

В классе `MimeUtility` собраны статические методы кодировки и декодировки.

Основной метод кодировки слова (`word`), которое трактуется рекомендацией RFC 822, как текст, не содержащий спецсимволов, выглядит так:

```
public static String encodeWord(
   String word, String charset, String encoding);
```

Этот метод сначала проверяет, есть ли в строке `word` символы вне диапазона \u0000—\u007F. Если таких символов нет, то строка возвращается без изменений. Если в строке есть символы вне этого диапазона, то метод преобразует строку `word` в массив байтов с помощью байтовой кодировки `charset`, потом кодирует этот массив кодом, указанным в аргументе `encoding`, который принимает значения `"Q"` или `"B"`, и переводит его снова в строку Unicode, добавляя к каждому байту нулевой байт.

Второй метод

```
public static String encodeWord(String word);
```

использует байтовую кодировку, установленную в виртуальной машине Java по умолчанию и автоматически выбирает кодировку "Q", если в строке word большинство символов US-ASCII, или кодировку "B" в противном случае. Фактически этот метод обращается к методу

```
encodeWord(word, null, null);
```

Для кодировки текста (text), соответствующего определению рекомендации RFC 822, применяются аналогичные методы

```
public static String encodeText(
   String text, String charset, String encoding);
public static String encodeText(String text);
```

Декодирование слова и текста выполняется методами

```
public static String decodeWord(String word);
public static String decodeText(String text);
```

Два метода кодируют выходной поток и декодируют входной поток:

```
public static OutputStream encode(
   OutputStream os, String encoding);
public static InputStream encode(
   InputStream os, String encoding);
```

Второй аргумент этих методов encoding может принимать значения "7bit", "8bit", "binary", "base64", "quoted-printable", "uuencode" и "x-uuencode".

Письмо *Message*

По стандарту MIME письмо может состоять из нескольких частей (multipart). Каждая часть письма состоит из заголовка, содержащего набор полей, и тела письма. Общие свойства всех частей письма описаны интерфейсом Part. Две его константы задают признак части письма:

- ATTACHMENT — присоединенная часть письма;
- INLINE — встроенная часть письма.

Интерфейс описывает методы заполнения содержимого данной части письма. Метод

```
public void setContent(Multipart content);
```

заносит в данную часть письма объект content, в который вложены другие части письма. Метод

```
public void setContent(Object content, String type);
```

заносит произвольный объект `content` MIME-типа `type`. Метод

```
public void setText(String text);
```

заносит строку текста `text` MIME-типа `text/plain`.

Два метода добавляют заголовок `name` этой части письма со значением `value`:

```
public void setHeader(String name, String value);
```

и новое значение в существующий заголовок с именем `name`:

```
public void addHeader(String name, String value);
```

Метод `getSize()` возвращает длину этой части письма в байтах.

Остальные методы `setXxx()` и `getXxx()` задают и предоставляют дополнительную информацию о данной части письма.

Интерфейс `Part` расширен интерфейсом `MimePart`, добавляющим методы, специфичные для стандарта MIME. Для русскоязычного программиста интересны два метода. Метод

```
public void setContentLanguage(String[] languages);
```

записывает в эту часть письма список `languages` использованных в ней языков. Они записываются двухбуквенными обозначениями по рекомендации RFC 1766, например, `"ru"`, `"en-US"`. Метод

```
public void setText(String text, String charset);
```

заносит содержимое данной части письма `text` типа `text/plain`, заданное в виде строки символов Unicode. Строка преобразуется в массив байтов по кодовой таблице, заданной строкой `charset`, например, `"koi8-r"` или `"windows-1251"`, а потом перекодируется в строку, содержащую только 7-битные символы в диапазоне Unicode от \u0000 до \u007F, как объяснялось выше.

Интерфейс `Part` частично реализован абстрактными классами `BodyPart` и `Message`.

Класс `BodyPart` описывает одну из частей письма, входящую в письмо MIME-типа `multipart`. Он добавляет к методам интерфейса `Part` всего один метод

```
public Multipart getParent();
```

возвращающий тело сообщения, в котором находится эта часть, или `null`, если оно неизвестно.

Абстрактный класс `BodyPart` должен быть расширен для полного формирования какой-либо части письма. В почтовую систему JavaMail входит одно такое расширение — класс `MimeBodyPart`, реализующий интерфейс `MimePart`.

В классе `MimeBodyPart` три конструктора. Кроме конструктора по умолчанию `MimeBodyPart()`, создающего пустой объект, есть конструктор

```
public MimeBodyPart(InputStream is);
```

читающий содержимое объекта из входного потока, и конструктор

```
public MimeBodyPart(InternetHeaders headers, byte[] content);
```

собирающий эту часть письма из атрибутов заголовка `headers` и содержимого `content` в виде массива байтов.

Методы класса просто реализуют методы интерфейсов `Part` и `MimePart`.

Другой абстрактный класс, частично реализующий интерфейс `Part` — класс `Message`, — описывает общие методы составления письма. Это методы: `setFrom(Address)`, записывающий адрес отправителя, `setSubject(String)`, записывающий заголовок письма, `setSentDate(Date)`, ставящий дату отправки, и др. Для практического использования класс `Message` должен быть расширен. Одно его расширение содержится в стандартной поставке JavaMail. Это класс `MimeMessage`, реализующий интерфейс `MimePart`. Как следует из названия, он определяет письмо, соответствующее стандарту MIME. Экземпляр класса `MimeMessage` создается в рамках сеанса связи конструктором

```
public MimeMessage(Session session);
```

или конструктором, получающим содержимое письма из входного потока:

```
public MimeMessage(Session session, InputStream is);
```

Класс `MimeMessage` реализует методы интерфейса `MimePart` в соответствии со стандартом MIME. Вот наиболее важные методы определения некоторых полей заголовка. Методы

```
public void setFrom();
public void setFrom(Address addr);
```

записывают в заголовок поле `"From"` — адрес отправителя. Первый метод записывает адрес того клиента, кто написал письмо, второй — адрес, указанный аргументом `addr`. Методы

```
public void setSubject(String subj);
public void setSubject(String subj, String charset);
```

заполняют поле `"Subject"` — заголовок письма. Первый из этих методов используется для заголовков, записанных 7-битными символами в кодировке US-ASCII. Второй метод следует использовать при записи заголовка строкой `subj`, содержащей символы с кодами Unicode больше \u007F. Строка `subj` будет преобразована методом

```
String encSubj = MimeUtility.encodeText(subj, charset, null);
```

и полученная строка `encSubj` будет занесена в заголовок письма.

Метод

```
public void setReplyTo(Address[] addrs);
```

задает поле "`Reply-To`" — список адресов, по которым надо разослать ответ на письмо.

Итак, перед созданием письма надо начать сеанс связи с почтовым сервером. Сеанс связи определяется классом `Session`.

Сеанс связи *Session*

Всякий поставщик услуг JavaMail может создать несколько одновременно проводимых сеансов связи. В каждом сеансе может одновременно работать несколько почтовых клиентов, но все они будут работать под одним именем и паролем. Один сеанс — сеанс по умолчанию — создает сам поставщик JavaMail, другие создаются почтовыми клиентами. В рамках каждого сеанса можно обратиться к нескольким почтовым ящикам.

У окончательного (final) класса `Session` нет конструкторов, его экземпляры создаются статическими фабричными методами

```
public static Session getDefaultInstance(
   Properties props, Authenticator auth);
public static Session getInstance(
   Properties props, Authenticator auth);
```

Первый из этих методов возвращает ссылку на сеанс по умолчанию, создаваемый системой JavaMail. Если такого объекта нет, то он создается вторым методом `getInstance()`; при этом используются свойства `props`. Поскольку к объекту по умолчанию может подключиться любой клиент, выполняющийся той же JVM, необходима проверка клиента с помощью второго аргумента — объекта `auth`. Если этот аргумент равен `null`, то к сеансу по умолчанию может подключиться любой клиент, обращающийся к методу `getDefaultInstance()` со вторым аргументом `null`.

Класс `Authenticator` абстрактный, он описывает в общем виде методы проверки клиента. При его расширении следует определить только метод аутентификации клиента:

```
protected PasswordAuthentication getPasswordAuthentication();
```

Определение метода заключается в создании объекта класса `PasswordAuthentication`, хранящего имя и пароль пользователя почтовой системы, конструктором

```
public PasswordAuthentication(String name, String password);
```

Свойства `props`, задаваемые перед началом сеанса, определяют следующие характеристики сеанса:

- `mail.store.protocol` — протокол доступа к хранилищу писем на почтовом сервере, по умолчанию первый подходящий протокол из конфигурационного файла javamail.default.providers, находящегося в архиве $J2EE_HOME/lib/j2ee.jar;
- `mail.transport.protocol` — протокол передачи писем, по умолчанию первый подходящий протокол из того же конфигурационного файла;
- `mail.host` — адрес почтового сервера, по умолчанию `localhost`;
- `mail.user` — имя пользователя, по умолчанию значение системного свойства `user.name`;
- `mail.from` — адрес отправителя, по умолчанию `username@host`;
- `mail.`*`protocol`*`.host` — адрес почтового сервера для указанного протокола *`protocol`*, по умолчанию совпадает со значением свойства `mail.host`;
- `mail.`*`protocol`*`.user` — имя пользователя для указанного протокола *`protocol`*, по умолчанию совпадает со значением свойства `mail.user`;
- `mail.debug` — если `true`, то задает режим отладки для начинающегося сеанса, по умолчанию `false`.

Спецификация JavaMail API Design Specification рекомендует явно определять первые пять свойств, несмотря на правила умолчания, поскольку не все почтовые системы могут придерживаться их.

Кроме этих стандартных свойств, клиент может определить свои свойства. Более того, кроме стандартного файла javamail.default.providers можно написать еще файл javamail.providers с другим описанием почтовых протоколов и поместить этот файл в каталог $JAVA_HOME/lib.

Для получения сведений о реализованных протоколах система JavaMail создает объекты класса `Provider`. Это простой класс, не имеющий конструктора, предоставляющий сведения о программах, реализующих почтовые протоколы, методами `getProtocol()`, `getVendor()`, `getVersion()`, `getClassName()`, `getType()`. Объект класса `Provider` не создается конструкторами, а возвращается методом

```
public Provider getProvider(String protocol);
```

а все такие объекты — методом

```
public Provider[] getProviders();
```

Итак, самый простой способ создать сеанс выглядит примерно так:

```
Properties props = System.getProperties();
props.put("mail.smtp.host", "smtp.somefirm.com");

Session sess = Session.getDefaultSession(props, null);
```

После этого можно создать письмо и записать его заголовок и текст:

```
MimeMessage mess = new MimeMessage(sess);
mess.setSubject("О вреде табака", "koi8-r");
mess.setText("Милостивые государыни ...", "koi8-r");
```

Вместо того чтобы указывать кодировку текста отдельно для заголовка и для тела письма, можно записать тип письма в заголовке:

```
mess.setHeader("Content-Type", "text/plain; charset=koi8-r");
```

Следующий шаг — отправка письма — требует определения адресов получателей письма и транспортного средства пересылки письма.

Адрес *Address*

Абстрактный класс `Address` описывает только абстрактные методы `getType()`, `equals(Object)` и `toString()`, требующие переопределения в подклассах. Основные методы формирования адреса и получения его частей реализованы в подклассах `InternetAddress` и `NewsAddress` класса `Address`.

Класс `InternetAddress` определяет адрес получателя или отправителя почты в своих конструкторах. Основной конструктор

```
public InternetAddress(String address, String person, String charset);
```

создает объект, содержащий адрес `address`, имя пользователя `person` и то же имя пользователя, перекодированное методом

```
String encName = MimeUtility.encodeWord(person, charset, null);
```

Почтовый адрес `address` следует записывать по обычным правилам RFC 822 в виде

```
username@host.network
```

Второй конструктор

```
public InternetAddress(String address, String person);
```

создает объект, не содержащий перекодированное имя пользователя.

В третьем конструкторе

```
public InternetAddress(String address);
```

аргумент `address` содержит адрес и имя пользователя в одной строке

Наконец, конструктор по умолчанию `InternetAddress()` создает пустой объект. В него можно затем занести все сведения методами

```
public void setAddress(String addr);
public void setPersonal(String person);
public void setPersonal(String person, String charset);
```

Остальные методы класса позволяют получить различные части адреса.

Класс `NewsAddress` описывает объект, содержащий адрес группы новостей Usenet по рекомендации RFC 1036. Основной конструктор

```
public NewsAddress(String newsgroup, String host);
```

создает объект, содержащий имя группы новостей `newsgroup` и адрес машины `host`, на которой расположена группа новостей.

Второй конструктор

```
public NewsAddress(String newsgroup);
```

не определяет адрес машины. Третий конструктор — конструктор по умолчанию `NewsAddress()` — создает пустой объект. Недостающие сведения можно добавить потом методами

```
public void setHost(String host);
public void setNewsgroup(String newsgroup);
```

Остальные методы класса предоставляют имя группы новостей и адрес машины.

После того как определены адреса получателей почты, можно занести эти адреса в письмо. Это выполняется методами

```
public void setRecipient(Message.RecipientType type, Address addr);
public abstract void setRecipients(
    Message.RecipientType type, Address[] addrs);
```

Первый аргумент этих методов `type` указывает тип получателя в виде объекта вложенного статического класса `RecipientType`. Этот простой класс содержит три константы:

- `TO` — получатель письма;
- `CC` — получатель копии письма (carbon copy);
- `BCC` — получатель "слепой" копии письма (blind carbon copy).

В любое время можно добавить новых адресатов письма методами

```
public void addRecipient(Message.RecipientType type, Address addr);
public abstract void addRecipients(
    Message.RecipientType type, Address[] addrs);
```

Теперь письмо готово к отправке. Осталось установить связь с почтовым сервером.

Связь *Service*

Связь с "почтовым отделением" описывает абстрактный класс `Service`. Его основу составляет метод связи с сервером `connect()`. При обращении к ме-

тоду надо указать адрес `host` и порт `port` сервера, имя клиента `user` и его пароль `password`:

```
public void connect(String host, int port,
   String user, String password);
```

Второй метод

```
public void connect(String host, String user, String password);
```

связывается с портом по умолчанию. Для протокола SMTP номер порта по умолчанию 25, для протокола POP3 — номер 110, для протокола IMAP4 — номер порта 143.

Третий метод `connect()` использует все эти сведения по умолчанию.

Класс `Service` расширен двумя классами. Абстрактный класс `Store` описывает связь для получения почты из почтового ящика, а абстрактный класс `Transport` описывает связь для отправки почты серверу, обычно по протоколу SMTP.

Несмотря на то, что класс `Transport` абстрактный, в нем определены статические методы, которые можно использовать непосредственно. Метод

```
public static void send(Message mess);
```

посылает письмо `mess` по адресам, записанным в заголовке письма `mess`. Метод сам предварительно устанавливает соединение с сервером, адрес которого определен в начале сеанса значением свойства `mail.host`.

Второй статический метод

```
public static void send(Message mess, Address[] addrs);
```

сам устанавливает адреса, указанные в списке `addrs`, в письмо `mess`. Адреса, записанные в заголовке письма, при этом игнорируются.

Итак, задать адресата и послать сообщение `mess`, которое мы определили выше, можно так:

```
InternetAddress[] addr =
   {new InternetAddress("john@some.firm.com")};
mess.setRecipients(Message.RecipientType.TO, addr);
Transport.send(mess);
```

Другой способ связаться с почтовым сервером — воспользоваться начавшимся сеансом связи и получить экземпляр класса `Transport`, методы которого полностью определены на сервере одним из следующих методов класса `Session`:

```
public Transport getTransport(Address addr);
public Transport getTransport(String protocol);
public Transport getTransport();
```

```
public Transport getTransport(Provider provider);
public Transport getTransport(URLName url);
```

Класс `URLName`, объект которого задается в последнем методе, — это простой класс, аккумулирующий сведения об адресе почтового сервера, заданном в виде URL. Его экземпляры создаются конструктором

```
public URLName(String protocol, String host, int port,
   String file, String username, String password);
```

или конструкторами

```
public URLName(String url);
public URLName(URL url);
```

Методы класса `URLName` — это методы `getXxx()`, выдающие сведения, имеющиеся в объекте класса.

В листинге 18.1 приведен полный пример почтового клиента с графическим интерфейсом. Связь с сервером установлена при помощи класса `Transport`, в котором вид этого клиента показан на рис. 18.1.

Листинг 18.1. Отправка письма

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
import javax.mail.*;
import javax.mail.event.*;
import javax.mail.internet.*;

public class MailClient extends JFrame{

  JTextField dest = new JTextField(50),
           source = new JTextField(50),
             subj = new JTextField(50),
           status = new JTextField(50);

  JTextArea content = new JTextArea(5, 50);
  JButton send = new JButton("Отправить");

  public MailClient(){
    super("  Почтовый клиент");
```

```
Container c = getContentPane();

JPanel p1 = new JPanel(false);
p1.setLayout(new BoxLayout(p1, BoxLayout.X_AXIS));
c.add(p1, BorderLayout.NORTH);

JPanel p3 = new JPanel(false);
p3.setLayout(new GridLayout(0, 1));
p1.add(p3);

p3.add(new JLabel("From: ", JLabel.RIGHT));
p3.add(new JLabel("To: ", JLabel.RIGHT));
p3.add(new JLabel("Subiect: ", JLabel.RIGHT));
p3.add(new JPanel());
p3.add(new JLabel("Content: ", JLabel.RIGHT));

JPanel p4 = new JPanel(false);
p4.setLayout(new GridLayout(0, 1));
p1.add(p4);

p4.add(source);
p4.add(dest);
p4.add(subj);

c.add(content);

JPanel p = new JPanel();
send.setActionCommand("Send");
p.add(send);
p4.add(p);

c.add(status, BorderLayout.SOUTH);

send.addActionListener(new ButtonHandler());
JMenuBar mb = new JMenuBar();
JMenu fl = new JMenu("Файл");
JMenuItem exit = new JMenuItem("Выход");
exit.setActionCommand("Exit");
```

```
    exit.addActionListener(new MenuItemHandler());
    fl.add(exit);
    mb.add(fl);
    setJMenuBar(mb);

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(350, 450);
    setVisible(true);
  }

  void sendMessage(){
    Properties props = new Properties();
    props.put("mail.smtp.host", "mail.ru");
    props.put("mail.from", source.getText());

    Session session = Session.getInstance(props, null);
    try{
      Message message = new MimeMessage(session);

      InternetAddress[] addr =
        {new InternetAddress(dest.getText())};

      message.setRecipients(Message.RecipientType.TO, addr);
      message.setFrom(new InternetAddress(source.getText()));
      message.setSubject(subj.getText());
      message.setContent(content.getText(),
         "text/plain; charset=windows-1251");

      Transport tr = session.getTransport("smtp");
      tr.addConnectionListener(new ConnectionHandler());
      tr.addTransportListener(new TransportHandler());
      tr.connect();
      tr.sendMessage(message, addr);

    }catch(Exception e){
      status.setText(e.toString());
    }
  }
```

```
  public static void main(String[] args){
    new MailClient();
  }

  class ConnectionHandler implements ConnectionListener{
    public void opened(ConnectionEvent e){
      status.setText("Связь установлена.");
    }
    public void disconnected(ConnectionEvent e){
      status.setText("Связь не установлена.");
    }
    public void closed(ConnectionEvent e){
      status.setText("Связь закрыта.");
    }
  }

  class TransportHandler implements TransportListener{
    public void messageDelivered(TransportEvent e){
      status.setText("Письмо доставлено.");
    }
    public void messageNotDelivered(TransportEvent e){
      status.setText("Письмо не отправлено.");
    }
    public void messagePartiallyDelivered(TransportEvent e){
      status.setText("Письмо доставлено не всем адресатам.");
    }
  }

  class ButtonHandler implements ActionListener{
    public void actionPerformed(ActionEvent ev){
      String s=ev.getActionCommand();
      if (s.equals("Send")) sendMessage();
    }
  }

  class MenuItemHandler implements ActionListener{
    public void actionPerformed(ActionEvent ev){
      if (ev.getActionCommand() == "Exit") System.exit(0);
    }
  }
}
```

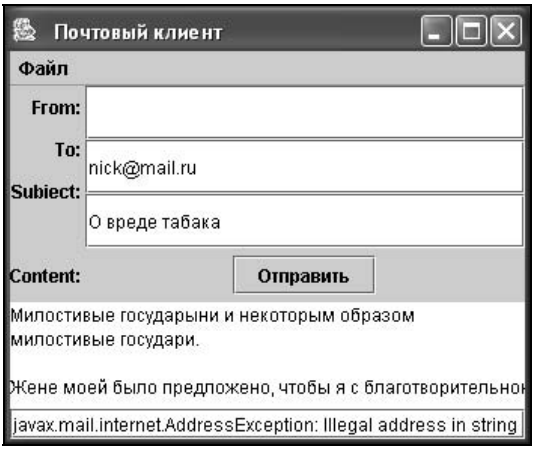


Рис. 18.1. Почтовый клиент

В листинге 18.1 проведена обработка событий соединения и пересылки, возникающих при установлении связи с почтовым сервером и пересылке письма на сервер. Она производится точно так же, как и в графических библиотеках AWT и Swing. Типы событий и методы их обработки очевидны из приведенного примера.

Хранилище *Store*

Абстрактный класс `Store` — расширение класса `Service` — описывает связь с почтовым сервером, установленную для получения почты из почтового ящика клиента. Экземпляр класса можно получить из начавшегося сеанса с указанием протокола. Обычно используется аргумент `"pop3"`, если для получения почты используется протокол POP3, или `"imap"`, если применяется протокол IMAP4:

```
Store store = session.getStore("pop3");
```

Метод `getStore()` без аргументов использует протокол, установленный для данного сеанса по умолчанию. Еще два метода

```
public Store getStore(Provider provider);
public Store getStore(URLName url);
```

определяют хранилище по сведениям `provider` или адресу `url` так же, как при получении экземпляра класса `Transport`.

Далее надо установить связь с сервером. При этом, как правило, следует указать имя пользователя и его пароль. Для этого используется один из методов `connect()`, описанных выше, например:

```
store.connect("ivan@mail.ru", "ivan", "edf5Rns");
```

Кроме унаследованных методов, позволяющих установить связь с сервером, класс `Store` содержит методы получения доступа к начальной папке клиента, описанной классом `Folder`.

Метод `getDefaultFolder()` возвращает папку по умолчанию с именем `INBOX`. Для протокола POP3 это единственная доступная клиенту папка. Протокол IMAP4 позволяет получить и другие папки.

Метод `getFolder(String name)` возвращает папку с именем `name`, метод `getFolder(URLName url)` возвращает папку, которая отыскивается по адресу `url`.

Абстрактный класс `Folder` описывает папку — хранилище писем. Есть несколько типов папок, определенных следующими константами класса `Folder`:

- `HOLDS_FOLDERS` — папка может хранить вложенные папки;
- `HOLDS_MESSAGES` — папка может хранить письма.

Дизъюнкция этих констант `HOLDS_FOLDERS | HOLDS_MESSAGES` определяет папку, в которой можно хранить и письма, и папки.

Папка, в которой хранятся письма, перед началом работы должна быть открыта. Открытие производится методом `open(int mode)` только для чтения или для чтения и записи. Режим открытия определяется аргументом `mode`, принимающим одно из двух значений:

- `READ_ONLY` — письма в папке можно только читать;
- `READ_WRITE` — можно добавлять и удалять письма, изменять параметры папки.

В папку, открытую для записи, можно добавлять письма, если папка может хранить письма методом

```
public void appendMessages(Message[] mess);
```

и создавать вложенные папки, если данная папка может хранить папки, с помощью метода

```
public boolean create(int type);
```

Аргумент `type` этого метода принимает значения `Folder.HOLDS_FOLDERS`, `Folder.HOLDS_MESSAGES` или их дизъюнкцию.

Получение писем

После открытия папки можно получить все хранящиеся в ней письма, например, так:

```
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
Message[] mess = folder.getMessages();
```

Письма в папке пронумерованы. Можно получить диапазон писем методом

```
public Message[] getMessages(int begin, int end);
```

с номерами от `begin` до `end` включительно. Можно получить письма с разными номерами, записав эти номера в массив `msgNums` и обратившись к методу

```
public Message[] getMessages(int[] msgNums);
```

Кроме того, можно получить письма, отобранные в результате поиска, методами

```
public Message[] search(SearchTerm term);
public Message[] search(SearchTerm term, Message[] msgs);
```

Второй из этих методов отбирает письма только из заданного массива `msgs`. Все письма из этого массива должны лежать в данной папке. Критерий поиска задается объектом `term` абстрактного класса `SearchTerm` или объектами его подклассов `AddressTerm`, `NotTerm`, `AndTerm`, `OrTerm`, `StringTerm`, `FlagTerm`, `ComparisionTerm` и подклассов этих классов.

Основу классов `XxxTerm` составляет логический метод

```
public boolean match(Xxx);
```

по-разному реализованный в разных классах. Множество этих классов, собранных в пакет javax.mail.search, позволяет организовать отбор писем по самым разным критериям.

Например, чтобы отобрать письма с заголовком `"Заказ"`, пришедшие в папку `folder` от отправителя `ivan@mail.ru`, надо сделать следующее:

```
SearchTerm st = new AddTerm(
   new FromStringTerm("ivan@mail.ru"), new SubjectTerm("Заказ"));
Message[] orders = folder.search(st);
```

После того как массив сообщений `mess` получен тем или иным способом, его можно просмотреть, например, так:

```
BufferedReader br = new BufferedReader(
   new InputStreamReader(System.in));

int n = mess.length;
for (int i = 0; i < n; i++){

  System.out.println(i + ": " + mess[i].getFrom()[0]
    + "\t" + mess[i].getSubject());

  System.out.println("Читать/Пропустить/Выйти? Y/N/Q");
  String line = br.readLine();
```

```
  if (line.equalsIgnoreCase("Y"))
       mess[i].writeTo(System.out);
  else if (line.equalsIgnoreCase("N")) continue;
  else if (line.equalsIgnoreCase("Q")) break;
}
```

Пометки писем

Письма в папке, открытой для записи, можно пометить различными флажками. Стандартные флажки описаны как следующие константы класса `Flags.Flag`:

- `ANSWERED` — на письмо уже дан ответ;
- `DELETED` — письмо надо удалить;
- `DRAFT` — черновой вариант письма;
- `FLAGGED` — какая-то пометка на усмотрение клиента;
- `RECENT` — новое письмо, полученное после последнего просмотра папки;
- `SEEN` — письмо доставлено получателю;
- `USER` — эта пометка означает, что папка позволяет пользователю помечать письма.

Пометка `flag` устанавливается в письмо или удаляется из него методом

```
public void setFlag(Flafs.Flag flag, boolean set);
```

класса `Message`. Если аргумент `set` равен `true`, то пометка устанавливается, если `false` — снимается.

Другой способ пометить письмо — собрать сначала все пометки в объект класса `Flags`. Для этого, прежде всего, создается пустой объект конструктором `Flags()`, или объект с одним флагом конструктором `Flags(Flags.Flag)`. Затем в этот объект добавляются другие флаги методом `add(Flags.Flag)` класса `Flags`.

После того как объект класса `Flags` сформирован, применяется метод

```
public void setFlags(Flafs flag, boolean set);
```

Не всякая реализация почтового протокола реагирует на эти пометки. Например, протокол POP3 работает только с флагом `DELETED`. Он не отмечает новые письма, при необходимости это должен сделать почтовый клиент.

Письма, помеченные `DELETED`, удаляются из папки операцией *вычеркивания* (expung). Вычеркивание выполняется методом

```
public Message[] expunge();
```

возвращающим массив удаленных писем, или при закрытии папки методом `close(true)`.

Посмотреть все пометки письма `mess` можно так:

```
Flags fls = mess.getFlags();
String userFlags = fls.getUserFlags();
Flags.Flag[] flag = fls.getSystemFlags();
for (int k = 0; k < flag.length; k++)
   if (flag[k] == Flags.Flag.SEEN)
      mess.setFlag(Flags.Flag.DELETED, true);
```

Работа с присоединенной информацией

Очень часто вместе с письмом посылается *(attach)* дополнительная информация: документ, чертеж, рисунок, произвольный объект. Такая информация обычно загружается из файла. Всякая почтовая система должна уметь посылать и получать присоединенную часть письма.

С точки зрения почтовой системы JavaMail письмо с присоединенной информацией — это письмо из нескольких частей (multipart). Одна часть письма — сообщение, остальные — присоединенная информация. Для присоединения к письму информации необходимо явно использовать систему активизации JAF (JavaBeans Activation Framework), составляющую пакет javax.activation.

Сначала информация, полученная из файла, описывается классом `FileDataSource` из этого пакета, а информация, полученная с адреса URL, — классом `URLDataSource`. Затем создается дескриптор полученного объекта, описанный классом `DataHandler`. Этот-то дескриптор и включается в письмо как одна из его частей `BodyPart`.

Весь процесс показан в листинге 18.2.

Листинг 18.2. Посылка присоединенного файла

```
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class AttachSend{

   public AttachSend(String host, String from,
            String to, String filename) throws Exception{
```

```
    attSend(host, from, to, filename);
  }

  public void attSend(String host, String from,
          String to, String filename) throws Exception{

   Properties props = System.getProperties();

   props.put("mail.smtp.host", host);

   Session session = Session.getInstance(props, null);

   MimeMessage msg = new MimeMessage(session);
   msg.setFrom(new InternetAddress(from));
   msg.addRecipient(Message.RecipientType.TO,
         new InternetAddress(to));
   msg.setSubject("Заявление",
     "text/plain; charset=windows-1251");

   MimeBodyPart bp = new MimeBodyPart();

   bp.setText("К заявлению прилагаю...", "windows-1251");

   Multipart mp = new MimeMultipart();

   mp.addBodyPart(bp);

   // Присоединенная часть письма:

   bp = new MimeBodyPart();

   DataSource source = new FileDataSource(filename);

   bp.setDataHandler(new DataHandler(source));

   bp.setFileName(filename);

   mp.addBodyPart(bp);
```

```
    msg.setContent(mp);

    Transport.send(msg);
  }

  public static void main(String args[]){

    if (args.length < 4){
      System.out.println(
         "Usage: java AttachSend <host> <from> <to> <filename>");
      System.exit(0);
    }
    try{
       new AttachSend(args[0], args[1], args[2], args[3]);
    }catch(Exception e){
       System.err.println(e);
    }
  }
}
```

Отделение присоединенной информации от полученного письма `mess` сводится к разбору письма на отдельные части и определению типа каждой части:

```
Multipart mp = (Multipart)mess.getContent();

int n=mp.getCount();

for (int i = 0; i < n; i++){

  Part part = mp.getBodyPart(i);

  String disp = part.getDisposition();

  if (disp != null)
    if (disp.equals(Part.ATTACHMENT)){
       // Присоединенная часть письма.
    }else if (disp.equals(Part.INLINE)){
       // Встроенная часть письма.
  }
```

Отправка HTML-файла

Почтовая система JavaMail не содержит специальных средств интерпретации языка HTML. Решение этой задачи передается почтовому клиенту получателя. Большинство почтовых клиентов, таких как The Bat!, Outlook Express, хорошо справляются с этой задачей.

Отправка HTML-файла не представляет никаких трудностей, надо только указать тип содержимого `text/html`. Очень часто HTML-файл включает изображения тегом `<img>`. Их можно отправить вместе с файлом. Отправка такого файла очень похожа, хотя и с некоторыми особенностями, на отправку присоединенной информации, пример которой приведен в листинге 18.2.

Во-первых, тип письма, как и ранее, `multipart`, но подтип — `related`, что надо указать явно. Во-вторых, тип сообщения `text/html`. В-третьих, в теге `<img>` надо указать ссылку на часть письма с изображением с помощью URL специальной схемы cid: (content identification, идентификация содержимого). Этот URL ссылается на значение поля `Content-ID` заголовка описываемой части письма.

Пример такого письма приведен в листинге 18.3.

Листинг 18.3. Отправка HTML-файла с рисунком

```
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class HtmlImage{

  public HtmlImage(String host, String from,
           String to, String imagefile){
     try{

        sendHtml(host, from, to, imagefile);

     }catch(Exception e){
        System.err.println(e);
     }
  }
```

```
public void sendHtml(String host, String from,
         String to, String imagefile) throws Exception{

  Properties props = System.getProperties();

  props.put("mail.smtp.host", host);

  Session session = Session.getDefaultInstance(props, null);

  MimeMessage mess = new MimeMessage(session);

  mess.setSubject("Анкета","windows-1251");
  mess.setFrom(new InternetAddress(from));
  mess.addRecipient(Message.RecipientType.TO,
     new InternetAddress(to));

  BodyPart bp = new MimeBodyPart();

  String htmlText = "<H1>Анкета</H1>" +
    "Моя фотография: " +
    "<img src=\"cid:photo\">";

  bp.setContent(htmlText,
     "text/html; charset=windows-1251");

  MimeMultipart mp = new MimeMultipart("related");

  mp.addBodyPart(bp);

  bp = new MimeBodyPart();

  DataSource fds = new FileDataSource(imagefile);
  bp.setDataHandler(new DataHandler(fds));

  bp.setHeader("Content-ID", "photo");

  mp.addBodyPart(bp);
```

```
    mess.setContent(mp);

    Transport.send(mess);
  }

  public static void main(String args[]){

    if (args.length < 4){
      System.out.println(
         "Usage: java HtmlImage <host> <from> <to> <filename>");
      System.exit(0);
    }
    new HtmlImage(args[0], args[1], args[2], args[3]);
  }
}
```

Глава 19

Связь компонентов Java с архитектурой CORBA

Как было показано в предыдущих главах этой части книги, в технологии Java есть все средства сетевой связи различных частей распределенного приложения. Но иногда приходится обеспечивать взаимодействие с приложениями или объектами, использующими другие технологии: COM, DCOM, CORBA. Для совместной работы с такими объектами в технологию Java введены средства связи с наиболее общей, универсальной технологией CORBA.

Архитектура CORBA, как следует из ее названия — Common Object Request Broker Architecture, — предназначена для взаимодействия объектов самого разного вида, написанных на разных объектно-ориентированных и процедурных языках, созданных разными объектно-ориентированными системами, и работающих на разных платформах. Она обобщает и удаленный вызов процедур RPC, и службы сообщений типа MOM, и даже хранимые процедуры баз данных. Спецификации CORBA описывают общие стандарты для создания распределенных объектов. Всякий объект, удовлетворяющий этим стандартам, может работать в среде CORBA.

Для описания объектов CORBA с соблюдением всех стандартов, создан специальный язык *IDL* (Interface Definition Language). Вначале объект описывается на языке IDL, затем это описание компилируется в программу на конкретном языке программирования: C, C++, Ada, Smalltalk, Java. Для каждого конкретного языка должен быть создан свой компилятор. В технологии Java стандартный компилятор, переводящий IDL-описание в исходный модуль на языке Java, называется idlj. Он входит в поставку J2SE SDK.

Программа, полученная после компиляции, требует конкретизации, поскольку первоначальное описание IDL не содержит деталей реализации объекта, а только интерфейс взаимодействия объекта со своими клиентами и с серверами CORBA. После записи всех деталей программа компилируется обычным для используемой платформы образом, создается объект и вовлекается в работу распределенной системы CORBA.

Объекты CORBA общаются через посредников — небольшие серверы, управляющие связью распределенных объектов, называемые *ORB* (Object Request Broker). *Посредник запросов объектов*, ORB, должен работать на каж-

дой машине, вовлеченной в распределенную систему CORBA, или быть доступным для объектов по сети. Он принимает запрос клиента, общаясь с другими ORB, находит машину, на которой расположен нужный объект и передает запрос посреднику, находящемуся на той удаленной машине. Дальнейшую работу проводит ORB удаленной машины. Он активизирует найденный объект, передает ему запрос, ждет результатов и отправляет их обратно.

Есть множество реализаций ORB, среди которых наибольшей популярностью пользуются VisiBroker фирмы Borland International и Orbix фирмы IONA Technologies. Посредник запросов входит в каждый сервер приложений. В поставку J2SE SDK входит Java ORB — реализация ORB фирмы Sun Microsystems, и фоновая программа, названная orbd (ORB Daemon). Она запускает ORB и останавливает ORB, содержит в себе реализацию системы именования CORBA Naming Service и диспетчер долговременных (persistent) объектов, Server Manager.

Посредники запросов в поисках удаленных объектов обмениваются информацией по специально разработанному транспортному протоколу *IIOP* (Internet Inter-ORB Protocol). Этот протокол — одна из реализаций общего протокола *GIOP* (General Inter-ORB Protocol), предназначенного для работы поверх любого транспортного протокола, а не только TCP, по которому работает IIOP.

ORB хранит сведения о распределенных объектах в своем хранилище (repository). У каждого объекта есть уникальный *идентификатор объекта* (Object ID), по которому объекты различаются между собой во всем распределенном приложении. Посредник обращается к хранящемуся у него распределенному объекту через *адаптер объекта* (Object Adapter) — программу, активизирующую объект, передающую ему запрос и принимающую результаты запроса. Всякий ORB должен реализовывать хотя бы *базовый адаптер объекта, BOA* (Basic Object Adapter). Спецификации адаптера BOA были проработаны не полностью, и его реализации в различных посредниках стали сильно отличаться друг от друга. Это привело к ухудшению переносимости ORB. Поэтому в дальнейшем была разработана спецификация *переносимого адаптера объекта, POA* (Portable Object Adapter). В настоящее время большинство ORB реализуют адаптер POA, хотя и сохраняют адаптер BOA для выполнения старых приложений.

Адаптер POA, кроме того, что он одинаков в различных реализациях ORB и позволяет создавать переносимые приложения, обладает дополнительными возможностями. Он может создавать долговременные (persistent) объекты, продолжающие существовать после перезапуска сервера. Он создает объект-*прислужник* (servant), выполняющий обращения к методам распределенного объекта. "Прислужник" — это реализация объекта CORBA на конкретном языке. В реализации Java "прислужники" настолько тесно связаны с распределенными объектами, которые они обслуживают, что можно их просто

отождествлять. Иногда говорят, что "прислужник" — это распределенный объект в активном состоянии.

Кроме посредников запросов, ORB, серверы CORBA реализуют одну или несколько *служб* (services) CORBA, расширяющих возможности ORB. С одной такой службой — службой именования, COS Naming — мы уже встречались в *главе 17*.

В поставку J2SE SDK, кроме реализации посредника запросов объектов, Java ORB, демона ORBD и компилятора `idlj`, входят интерфейсы и классы, обеспечивающие создание приложений, способных работать в среде CORBA. Эти средства связи с CORBA образуют часть стандартной поставки J2SE SDK, называемую *Java IDL*. Она образует пакет org.omg.CORBA и множество пакетов вспомогательных классов.

С другой стороны, система вызова удаленных объектов RMI, которая была описана в *главе 15*, расширена так, что она может использовать транспортный протокол IIOP. Обращение к удаленным объектам по протоколу IIOP получило название *RMI-IIOP*. Рассмотрим подробнее эту возможность.

Использование RMI-IIOP

Клиент системы вызова удаленных объектов RMI, использующий транспортный протокол IIOP, может обращаться к любому удаленному объекту CORBA, написанному не только на языке Java, но и на других языках. Надо лишь, чтобы удаленный объект реализовал интерфейс `Remote`. Есть еще несколько ограничений.

- Константы, определенные в удаленных интерфейсах, т. е. интерфейсах, расширяющих интерфейс `Remote`, должны быть простого типа Java или типа `String`.
- Имена Java должны учитывать правила интерпретации имен CORBA. Вкратце, лучше не использовать в именах знак доллара и знак подчеркивания и избегать имен, отличающихся только регистром букв. Допускается модная тенденция называть переменную именем типа, но строчными буквами, скажем, `String string = "My String"`.
- Не следует использовать классы `RMISocketFactory`, `UnicastRemoteObject`, `Unreferenced` и систему распределенного "сбора мусора" DGC.

Подробно эти ограничения и другие правила соответствия языков Java и CORBA изложены в документе "Java Language to IDL Mapping Specification", доступном по адресу **http://www.omg.org/cgi-bin/doc?formal/01-06-07.pdf**.

С учетом этих ограничений программы, созданные в расчете на протокол JRMP, подобные тем, что приводились в *главе 15*, требуют некоторого изменения при переходе на протокол IIOP.

Во-первых, вместо RMI-реестра протокол IIOP использует систему именования COS Naming. Программы Java связываются с ней через систему именования JNDI с помощью соответствующей реализации SPI. Следовательно, у клиента и у сервера надо заменить методы `bind()`, `rebind()` и `lookup()` класса `java.rmi.Naming` на соответствующие методы класса `javax.naming.InitialContext`, предварительно создав объект этого класса, как объяснялось в *главе 17*, и связав его с поставщиком услуг COS Naming, например:

```
Properties props = new Properties();

props.put(Context.INITIAL_CONTEXT_FACTORY,
         "com.sun.jndi.cosnaming.CNCtxFactory");
props.put(Context.PROVIDER_URL, "iiop://<hostname>:1050");

InitialContext ic = new InitialContext(props);

ic.rebind("MyRemote", myRemote);
```

Свойства можно задать позднее, при запуске клиента и сервера, например:

```
java -Djava.naming.factory.initial=\
          com.sun.jndi.cosnaming.CNCtxFactory
     -Djava.naming.provider.url=iiop://<hostname>:1050
         <appl_class>
```

Второй способ именования — прямо использовать службу именования COS Naming без посредства JNDI. Для этого в поставку J2SE SDK входит утилита orbd и сохраняется старая утилита tnameserv.

Утилита tnameserv запускает сервер имен COS Naming, прослушивающий, по умолчанию, порт с номером 900. Другой номер порта можно задать при запуске сервера имен:

```
tnameserv -ORBInitialPort 1050&
```

Этот сервер, как следует из названия утилиты (Transient Naming Service), обеспечивает только кратковременное хранение имен. После перезапуска сервера имена теряются.

Утилита orbd запускает сервер имен COS Naming, который обеспечивает и кратковременное, и долговременное (Persistent Naming Service) хранение имен. Такие имена сохраняются после краха и перезапуска сервера. Для обеспечения долговременности объектов утилита orbd запускает Bootstrap Service (Служба начальной загрузки) и Server Manager, которые активизируют долговременные объекты и управляют их поведением.

Во-вторых, классы удаленных объектов следует наследовать не от класса `UnicastRemoteObject`, а от класса `PortableRemoteObject` из пакета javax.rmi

и поменять все ссылки на класс `UnicastRemoteObject`, сделав их ссылками на класс `PortableRemoteObject`. Если вместо наследования классов используется метод `exportObject()`, то это должен быть метод класса `PortableRemoteObject`.

После изменения кода программы и ее компиляции, следует создать заглушки, ориентированные на IIOP, с помощью компилятора `rmic` с ключом `-iiop`. Если используется адаптер POA, то надо добавить ключ `-poa`.

Потом следует запустить систему именования COS Naming, например, с помощью демона orbd.

При использовании базового адаптера BOA больше ничего делать не надо.

Адаптер BOA

Перепишем пример из листингов 15.1—15.4 для работы по протоколу IIOP с использованием адаптера BOA.

Удаленный интерфейс листинга 15.1 остается без изменений. Перепишем его в листинг 19.1.

Листинг 19.1. Удаленный интерфейс

```
import java.rmi.*;


public interface MyRemote extends Remote{

   String getName() throws RemoteException;
   void setName(String name) throws RemoteException;
}
```

При реализации этого удаленного интерфейса изменим листинг 15.2, расширив не класс `UnicastRemoteObject`, а класс `PortableRemoteObject`. Получим листинг 19.2.

Листинг 19.2. Удаленный объект

```
import java.rmi.*;
import javax.rmi.*;

public class MyRemoteImpl extends PortableRemoteObject
                            implements MyRemote{
   private String name;
```

```
   public String getName() throws RemoteException{
      return name;
   }

   public void setName(String name) throws RemoteException{
      this.name = name;
   }

   public MyRemoteImpl() throws RemoteException{
      super();
   }

   public MyRemoteImpl(String name) throws RemoteException{
      this.name = name;
   }
}
```

При изменении RMI-сервера листинга 15.3 создаем в начальном контексте системы именования JNDI имя "MyServer", связанное с удаленным объектом mri. (листинг 19.3).

Листинг 19.3. RMI-IIOP-сервер

```
import javax.naming.*;
import java.util.*;

public class MyRemoteServer{

  public static void main(String[] args){
    try{
      MyRemoteImpl mri = new MyRemoteImpl();

      Properties props = new Properties();

      props.put(Context.INITIAL_CONTEXT_FACTORY,
               "com.sun.jndi.cosnaming.CNCtxFactory");
      props.put(Context.PROVIDER_URL, "iiop://<host>:1050");

      InitialContext ic = new InitialContext(props);
```

```
      ic.rebind("MyServer", mri);

    }catch(Exception e){
      System.err.println("Ошибка RMI: " + e);
    }
  }
}
```

После этого компилируем исходные файлы на сервере:

```
javac MyRemote.java MyRemoteImpl.java MyServer.java
```

Создаем заглушку и связку (tie), рассчитанные на протокол IIOP и на адаптер BOA: Для этого запускаем компилятор `rmic` с ключом `-iiop`:

```
rmic -iiop MyRemoteImpl
```

Заглушка записывается в файл _MyRemote_Stub.class, имя которого составляется по правилу: `_<имя интерфейса>_Stub.class`, а связка — в файл _MyRemoteImpl_Tie.class, имя которого составляется по правилу: `_<имя объекта>_Tie.class`.

Связка (tie) — это объект, распределяющий запросы между различными экземплярами удаленных объектов.

Запускаем демон, создающий ORB и управляющий им:

```
orbd -ORBInitialPort 1050&
```

Посредник будет прослушивать порт с номером 1050.

Наконец, запускаем RMI-сервер, создающий удаленный объект и регистрирующий его в системе именования CORBA Naming Service:

```
java MyServer&
```

На сервере все готово, удаленный объект создан и ждет обращения к его методам. Теперь займемся клиентом.

Листинг 15.4 тоже требует изменения. Теперь удаленный объект отыскивается с помощью системы именования CORBA, Naming Service, с которой клиент связывается через систему именования JNDI. Измененный клиент приведен в листинге 19.4.

Листинг 19.4. Клиент системы RMI-IIOP

```
import java.rmi.*;
import javax.naming.*;
import java.util.*;
```

```
public class MyClient{

  public static void main(String[] args){

   try{
    Properties props = new Properties();

    props.put(Context.INITIAL_CONTEXT_FACTORY,
       "com.sun.jndi.cosnaming.CNCtxFactory");
    props.put(Context.PROVIDER_URL, "iiop://<host>:1050");

    InitialContext ic = new InitialContext(props);

    MyRemote mr = (MyRemote)ic.lookup("MyServer");

    mr.setName("Иванов");

    System.out.println("Фамилия: " + mr.getName());

   }catch(Exception e){
      System.out.println("Ошибка RMI: " + e);
   }
  }
}
```

На клиентской машине компилируем клиента:

```
javac MyClient.java
```

и запускаем его

```
java MyClient
```

Происходит обращение к удаленным методам, они выполняются, и клиент получает результат.

Использование COS Naming

Для прямого использования службы именования COS Naming, без посредства JNDI, в сервере и клиенте надо изменить код связывания с системой имен. Прямую связь с COS Naming обеспечивают интерфейсы и классы из пакета org.omg.CosNaming.

После инициализации экземпляра посредника, например:

```
ORB orb = ORB.init(args, props);
```

получаем начальный контекст с именем "NameService":

```
NamingContextExt ctx =
   NamingContextExtHelper.narrow(orb.resolve_initial_references(
            "NameService"));
```

Использованный здесь интерфейс NamingContextExt описывает расширенную систему имен INS (Interoperable Naming Service), допускающую имена в форме URL со схемами corbaloc: и corbaname:. Система имен INS была изложена в *главе 17*.

Затем преобразуем ссылку на контекст ctx в объектную ссылку, потом преобразуем имя, заданное строкой, например, "MyServer", в имя, заданное массивом компонентов, и связываем это имя с объектной ссылкой:

```
org.omg.CORBA.Object objref = ctx;
NameComponent[] myServ= ctx.to_name("MyServer");
ctx.rebind(myServ, objref);
```

Клиентская программа тоже получает экземпляр посредника и начальный контекст, например:

```
Properties props = new Properties();

props.put("org.omg.CORBA.ORBInitialPort", "1050");
props.put("org.omg.CORBA.ORBInitialHost", "name.server.com");

ORB orb = ORB.init(args, props);

NamingContextExt ctx =
   NamingContextExtHelper.narrow(orb.resolve_initial_references(
       "NameService"));
```

После этого можно обращаться к удаленным объектам по их именам:

```
org.omg.CORBA.Object servRef = ctx.resolve_str("MyServer");
```

Если использовать имя в форме URL, то получить объектную ссылку можно проще:

```
org.omg.CORBA.Object servRef = orb.string_to_object(
    "corbaname::name.server.com:1050#MyServer");
```

Для приведения типа используется объект класса `XxxHelper`, созданный компилятором `idlj`, например:

```
MyRemoteImpl mri = MyRemoteHelper.narrow(servRef);
```

После этого можно обращаться к методам удаленного объекта:

```
mri.setName("Петров");
```

Адаптер POA

Использование переносимого адаптера POA требует изменения RMI-IIOP-сервера, записанного в листинге 19.3. Теперь, кроме создания удаленного объекта и регистрации его в системе именования JNDI или COS Naming, надо создать объект адаптера POA, активизировать его, создать "прислужника" (servant), активизировать его, создать ссылку на удаленный объект и, наконец, запустить ORB, посредника запросов.

Рассмотрим подробнее все эти действия.

Вначале создается ORB статическим методом

```
public static ORB init(String[] args, Properties props);
```

Аргументы этого метода `args` и `props` будут переданы RMI-серверу, когда он будет создан.

Один ORB может создать несколько адаптеров POA. Они образуют иерархическую структуру, во главе которой стоит корневой адаптер. Корневой адаптер регистрируется в системе именования COS Naming в корневом контексте под именем `"RootPOA"`. Для регистрации и получения объектной ссылки используется метод

```
public org.omg.CORBA.Object resolve_initial_references(String name);
```

класса `ORB`. Созданный CORBA-объект, как правило, хотя и необязательно, отыскивается, проверяется и приводится к типу `POA` методом

```
public static POA narrow(Object obj);
```

класса `POAHelper`. Обычно все операции по созданию корневого адаптера выполняются двумя действиями:

```
ORB orb = ORB.init(args, null);
POA rPoa =
   POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
```

Корневой адаптер редко используется непосредственно для работы с распределенными объектами, потому что его поведение невозможно изменить. Он служит для создания вложенных адаптеров методом

```
public POA create_POA(String name, POAManager mgr, Policy[] policy);
```

здесь аргумент `name` дает имя адаптеру, отличающее его от других адаптеров, созданных в контексте корневого адаптера, аргумент `mgr` задает объект, управляющий адаптером, аргумент `policy` — список параметров, определяющих поведение создаваемого адаптера.

Каждый параметр адаптера определяется объектом класса, реализующего интерфейс `Policy`. Эти классы собраны в пакет org.omg.PortableServer. При создании объектов этих классов задаются значения, специфичные для каждого класса. Вот список параметров адаптера. Значение Thread Policy принимается по умолчанию, именно оно заложено в корневой адаптер.

- Thread Policy: `ORB_CTRL_MODEL`, `SINGLE_THREAD_MODEL` — многопоточная или однопоточная, последовательная, обработка запросов.
- Lifespan Policy: `TRANSIENT`, `PERSISTENT` — адаптер создает кратковременные объекты или долговременные объекты, продолжающие существовать после завершения работы адаптера.
- Object Id Uniqueness Policy: `UNIQUE_ID`, `MULTIPLE_ID` — "прислужники", создаваемые адаптером, обслуживают один или несколько объектов.
- Id Assignment Policy: `SYSTEM_ID`, `USER_ID` — идентификаторы объектов назначаются адаптером или могут назначаться приложением.
- Servant Retention Policy: `RETAIN`, `NON_RETAIN` — ссылки на "прислужников" хранятся в оперативной памяти, в специальной области, называемой Active Object Map, или могут выгружаться на диск.
- Request Processing Policy: `USE_ACTIVE_OBJECT_MAP_ONLY`, `USE_DEFAULT_SERVANT`, `USE_SERVANT_MANAGER` — ищутся только активные объекты, отыскивается, кроме активных, объект по умолчанию или для поиска используется диспетчер "прислужников".
- Implicit Activation Policy: `IMPLICIT_ACTIVATION`, `NON_IMPLICIT_ACTIVATION` — внутренняя или внешняя активизация "прислужников".

Способы задания параметров адаптера показаны в листинге 19.5.

После создания адаптера в корневом или другом контексте методом

```
POA tPoa = rPoa.create_POA("MyServer", null, null);
```

в котором заданы параметры адаптера по умолчанию и второй аргумент равен `null`, надо активизировать его диспетчер — объект типа `POAManager`, управляющий одним или несколькими адаптерами POA. Для этого сначала методом

```
public POAManager the_POAManager();
```

описанным в интерфейсе `POAOperations`, создается объект типа `POAManager`, затем он активизируется своим методом `activate()`. Обычно все это делается одним оператором, например:

```
tPoa.the_POAManager().activate();
```

Теперь можно создать экземпляр RMI-сервера и работающего на него "прислужника" — объект класса `Servant`. Имя класса-прислужника составляется из имени класса удаленного объекта, например, `RemoteImpl` со знаками подчеркивания до и после него и добавлением слова `Tie` следующим образом: `_RemoteImpl_Tie`. Экземпляр этого класса создается позднее компилятором `rmic`, а получить его в программе можно статическим методом

```
public static Tie getTie(Remote obj);
```

класса `Util` из пакета javax.rmi.CORBA. Этот метод был создан для получения связки — объекта, реализующего интерфейс `Tie`, но сейчас он переработан для получения "прислужника", надо только привести тип, как показано ниже:

```
RemoteImpl ri = new RemoteImpl();
_RemoteImpl_Tie servant = (_RemoteImpl_Tie)Util.getTie(ri);
```

Следующий шаг — снабдить "прислужника" идентификатором. Идентификатор — это просто любой массив байтов `id`, который надо связать с "прислужником" методом

```
public void activate_object_with_id(byte[] id, Servant servant);
```

Затем, методом

```
public Object create_reference_with_id(byte[] id, String ref);
```

создается объектная ссылка, которая регистрируется в системе именования JNDI или COS Naming. Для регистрации ссылки методом `bind()` надо сначала получить список идентификаторов и адаптеров методом

```
public String[] _all_interfaces(POA poa, byte[] id);
```

и связать с именем какой-нибудь из них, допустим, первый, методом `bind()`, например, так:

```
tPoa.create_reference_with_id(id,
          servant._all_interfaces(tPoa, id)[0]));
```

Наконец, запускаем адаптер POA методом `run()`, и сервер готов к приему запросов.

Все изменения RMI-IIOP-сервера при использовании адаптера POA приведены в листинге 19.5.

Листинг 19.5. RMI-IIOP-сервер с адаптером POA

```
import javax.naming.*;
import com.sun.corba.se.internal.POA.POAORB;
import org.omg.PortableServer.*;
import java.util.*;
```

```
import org.omg.CORBA.*;
import javax.rmi.CORBA.*;

public class MyRemotePOAServer{

  public static void main(String[] args){

    try{
      Properties p = new Properties();

      p.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.cosnaming.CNCtxFactory");
      p.put(javax.naming.Context.PROVIDER_URL,
        "iiop://<host>:1050");
      p.put("org.omg.CORBA.ORBClass",
        "com.sun.corba.se.internal.POA.POAORB");
      p.put("org.omg.CORBA.ORBSingletonClass",
        "com.sun.corba.se.internal.corba.ORBSingleton");

      ORB orb = ORB.init(args, p);

      POA rootPOA =
        (POA)orb.resolve_initial_references("RootPOA");

      Policy[] pol = new Policy[3];

      pol[0] = rootPOA.create_lifespan_policy(
        LifespanPolicyValue.TRANSIENT);
      pol[1] = rootPOA.create_request_processing_policy(
        RequestProcessingPolicyValue.USE_ACTIVE_OBJECT_MAP_ONLY);
      pol[2] = rootPOA.create_servant_retention_policy(
        ServantRetentionPolicyValue.RETAIN);

      POA tPOA = rootPOA.create_POA(
                  "MyTransientPOA", null, pol);

      tPOA.the_POAManager().activate();

      MyRemoteImpl mri = new MyRemoteImpl();
```

```
      _MyRemoteImpl_Tie servant =
            (_MyRemoteImpl_Tie)Util.getTie(mri);

      String myId = "myremote";
      byte[] id = myId.getBytes();

      tPOA.activate_object_with_id(id, servant);

      InitialContext ic = new InitialContext(p);

      ic.rebind("MyServer",
          tPOA.create_reference_with_id(id,
              servant._all_interfaces(tPOA, id)[0]));

      orb.run();

    }catch(Exception e){
      System.err.println("Ошибка RMI: " + e);
    }
  }
}
```

Перед компиляцией кода листинга 19.5 создадим новую заглушку (stub), а вместо связки (tie) создадим "прислужника" (servant). Для этого предварительно удалим файлы _MyRemote_Stub.class и _MyRemoteImpl_Tie.class, если они есть, и выполним команду

```
rmic -iiop -poa MyRemoteImpl
```

"Прислужник" будет записан в файл с тем же именем _MyRemoteImpl_Tie.class, что и связка, которая создавалась ранее для адаптера BOA, а "прислужник" будет расширять класс `Servant`.

Откомпилируем RMI-сервер:

```
javac MyRemotePOAServer.java
```

Теперь запустим ORB с помощью утилиты orbd:

```
orbd -ORBInitialPort 1050&
```

и запустим RMI-сервер

```
java MyRemotePOAServer
```

На клиентской стороне ничего менять не надо, просто запускаем клиента:

```
java MyClient
```

Использование IDL

Вместо того чтобы сразу писать код программы на языке Java, можно сначала создать описание распределенных объектов на языке IDL, а потом откомпилировать это описание в файлы Java с помощью компилятора idlj.

В листинге 19.6 приведено описание удаленного объекта листинга 19.1 на языке IDL. Предположим, что оно записано в файл MyRemote.idl.

Листинг 19.6. Описание распределенного объекта на IDL

```
module student{
   interface MyRemote{
      wstring getName();
      void setName(in wstring name);
   };
};
```

На сервере компилируем этот файл с ключом `-fserver`:

```
idlj -fserver MyRemote.idl
```

Компилятор создает пакет student, соответствующий ему подкаталог student и в нем три файла.

Файл MyRemoteOperations.java содержит интерфейс, названный `MyRemoteOperations`. Он соответствует интерфейсу листинга 19.1, но его методы не выбрасывают исключение класса `RemoteException`.

Файл MyRemote.java содержит один пустой интерфейс. Этот интерфейс, названный `MyRemote`, расширяет интерфейс `MyRemoteOperations`, а также интерфейс `Object` из пакета org.omg.CORBA и интерфейс `IDLEntity`.

Файл MyRemotePOA.java содержит "прислужника" — класс `MyRemotePOA`, расширяющий класс `Servant`, а также реализующий интерфейс `MyRemoteOperations` и интерфейс `InvokeHandler`. Таким образом, "прислужник" совмещен с распределенным объектом. Класс `MyRemotePOA` абстрактен, поскольку мы еще не реализовали методы распределенного объекта. Он должен быть расширен для реализации методов.

Описанные файлы называются *каркасами* (skeletons) серверной части.

На клиентской машине компилируем файл MyRemote.idl без ключа или с ключом `-fclient`:

```
idlj MyRemote.idl
```

Компилятор создает пять файлов. Кроме MyRemoteOperations.java и MyRemote.java, это файл _MyRemoteStub.java, содержащий заглушку, и файлы MyRemoteHelper.java и MyRemoteHolder.java, содержащие классы со вспомогательными методами чтения, записи, поиска. Эти методы используются заглушкой для сборки (marshalling) запроса, но могут применяться и клиентом и сервером, поэтому классы `XxxHelper` и `XxxHolder` часто создаются и на сервере.

Можно создать сразу все шесть файлов, если вызвать компилятор с ключом `-fall`:

```
idlj -fall MyRemote.idl
```

Итак, компилятор `idlj` уже создал пакет student, в нем клиентскую заглушку `_MyRemoteStub` и "прислужника" `MyRemotePOA`, совмещенного с пока еще не реализованным распределенным объектом.

Полученная модель, в которой распределенный объект получается как расширение своего "прислужника", называется *моделью наследования* переносимого "прислужника" (Portable Servant Inheritance Model).

Если распределенный объект уже расширяет какой-либо класс, то следует воспользоваться моделью делегирования, называемой здесь *моделью связки* (Tie Model). Для применения этой модели надо вызвать компилятор с флагом `-fallTie`:

```
idlj -fallTie MyRemote.idl
```

В этом случае создается седьмой файл MyRemotePOATie.java, содержащий класс-связку `MyRemotePOATie`, расширяющий класс `MyRemotePOA`. Конструктор класса `MyRemotePOATie` требует в качестве аргументов распределенный объект и экземпляр адаптера, пусть это будет `tPOA`:

```
MyRemoteOperations myObject = new MyRemoteImpl();
MyRemotePOATie tie = new MyRemotePOATie(myObject, tPOA);
MyRemote myServant = tie._this(orb);
MyServant.setName("Иванов");
```

Здесь реализация `MyRemoteImpl` интерфейса `MyRemoteOperations` не расширяет класс "прислужника" `MyRemotePOA`. Она делегируется связке `tie`, из которой уже получаем распределенный объект, связанный с адаптером `tPOA`.

После получения всех вспомогательных классов остается написать код реализации распределенного объекта `MyRemoteImpl`, код CORBA-сервера и код клиента.

В листинге 19.7 приведен код реализации распределенного объекта в модели наследования.

Листинг 19.7. Распределенный объект CORBA

```
public class MyRemoteInhImpl extends student.MyRemotePOA{

   private String name;

   public String getName(){
      return name;
   }

   public void setName(String name){
      this.name = name;
   }
   public MyRemoteInhImpl(){
      this.name = name;
   }

   public MyRemoteInhImpl(String name){
      this.name = name;
   }
}
```

Листинг 19.8 содержит код сервера, создающего экземпляры распределенного объекта.

Листинг 19.8. CORBA-сервер

```
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.CosNaming.*;
import java.util.*;
import student.*;

public class MyRemoteInhServer{

  public static void main(String args[]){
    try{
      Properties props = new Properties();
```

```
      props.put("org.omg.CORBA.ORBInitialPort", "1050");

      ORB orb = ORB.init(args, props);

      POA rootPOA =
         (POA)orb.resolve_initial_references("RootPOA");

      POA tPOA = rootPOA.create_POA("MyPOA", null, null);
      tPOA.the_POAManager().activate();

      MyRemoteInhImpl impl = new MyRemoteInhImpl();

      org.omg.CORBA.Object ref =
          tPOA.servant_to_reference(impl);

      MyRemote href = MyRemoteHelper.narrow(ref);

      org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");

      NamingContextExt ncRef =
        NamingContextExtHelper.narrow(objRef);

      NameComponent[] path = ncRef.to_name("MyInheritIdl");
      ncRef.rebind(path, href);

      orb.run();

    }catch (Exception e){
       System.err.println("Ошибка: " + e);
    }
  }
}
```

После компиляции этих файлов запускаем демон orbd:

```
orbd -ORBInitialPort 1050&
```

и запускаем сервер

```
java MyRemoteInhServer
```

Теперь напишем клиента. Его текст приведен в листинге 19.9.

Листинг 19.9. Клиент CORBA

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import java.util.*;
import student.*;

public class MyIdlClient{

  public static void main(String args[]){
    try{
      Properties props = new Properties();

      props.put("org.omg.CORBA.ORBInitialPort", "1050");
      props.put("org.omg.CORBA.ORBInitialHost", "<nameserver>");

      ORB orb = ORB.init(args, props);

      org.omg.CORBA.Object objRef =
         orb.resolve_initial_references("NameService");

      NamingContextExt ncRef =
         NamingContextExtHelper.narrow(objRef);

      MyRemote mr = MyRemoteHelper.narrow(
             ncRef.resolve_str("MyInheritIdl"));

      mr.setName("Иванов");
      System.out.println("Фамилия: " + mr.getName());

    }catch(Exception e){
       System.out.println("Ошибка: " + e);
    }
  }
}
```

Откомпилировав код клиента, запустим его

```
javac MyIdlClient.java
java MyIdlClient
```

Клиент отправляет запрос посреднику запросов, ORB, связавшись с сервером имен и получив от него адрес ORB. Посредник принимает запрос, отыскивает распределенный объект, активизирует его "прислужника", дожидается выполнения методов распределенного объекта и отправляет результаты клиенту.

Долговременные объекты

Адаптер POA допускает создание долговременных (persistent) объектов, продолжающих существовать после завершения работы адаптера. В случае получения запроса после останова адаптера, демон посредника orbd перезапускает адаптер, и сам восстанавливает его состояние, активизируя долговременные объекты.

Для создания долговременного объекта надо изменить только код сервера, например, код листинга 19.8.

Во-первых, вместо `tPOA` надо создать и активизировать POA с соответствующей политикой:

```
Policy[] pol = {rootPOA.create_lifespan_policy(
   LifespanPolicyValue.PERSISTENT)};


POA persPOA = rootPOA.create_POA("MyPersistentPOA", null, pol);


persPOA.the_POAManager().activate();
```

Во-вторых, созданный объект, который будет долговременным, например:

```
MyRemoteInhImpl impl = new MyRemoteInhImpl();
```

надо зарегистрировать в посреднике под каким-нибудь именем:

```
byte[] name = "SomeName".getBytes();
persPOA.activate_object_with_id(name, impl);
```

Далее код сервера остается прежним.

Затем серверная программа обычным образом компилируется и запускается. Для некоторых реализаций ORB этого достаточно для хранения долговременных объектов. Для реализации Java ORB, входящей в стандартную поставку J2SE SDK, долговременный объект надо еще зарегистрировать у демона orbd. Для этого после запуска демона и сервера

```
orbd -ORBInitialPort 1050
java MyRemoteInhServer
```

надо запустить утилиту servertool

```
servertool -ORBInitialPort 1050
```

Увидев приглашение утилиты: `servertool >`, надо набрать команду регистрации

```
servertool > register -server MyRemoteInhServer -classpath .\
   -applicationName Serv1
```

Последний параметр `-applicationName Serv1` необязателен, это только регистрационное имя.

После этого можно посмотреть список зарегистрированных CORBA-серверов

```
servertool > list
```

Чтобы выйти из утилиты, надо набрать команду `quit`

```
servertool > quit
```

Заключение

Эта глава — только краткое введение в средства Java, позволяющие реализовать спецификации CORBA. За ее границами осталось много деталей создания, активизации и управления поведением объектов CORBA. Достаточно сказать, что классы, посвященные архитектуре CORBA, составляют около тридцати пакетов. Для их описания нужна целая книга. При этом технология Java не реализует полностью даже CORBA версии 2.3.1, не говоря уже о CORBA 3.0.

Может быть, такая реализация и не потребуется. Информационные технологии развиваются стремительно, быстро сменяя друг друга. Технология WWW, появившаяся всего несколько лет назад, оказалась очень плодотворной, на ее базе появляются все новые и новые решения. Одно из последних нововведений — службы Web (Web services) — решает те же задачи, что и CORBA — взаимодействие распределенных объектов, созданных с использованием разных технологий.

При взаимодействии объектов средствами служб Web запросы оформляются просто как документы XML, что обеспечивает их переносимость. Для пересылки запросов и ответов разработан протокол *SOAP* (Simple Object Access Protocol) со своими правилами сборки (marshalling) и разборки запросов и ответов — протокол прикладного уровня, который обычно работает по HTTP, хотя может работать и по другому протоколу. Можно рассматривать его как аналог протокола IIOP в технологии CORBA. Взаимодействующие объекты должны удовлетворять интерфейсам, записанным первоначально на языке WSDL (Web Services Description Language) — аналоге языка описания

интерфейсов IDL. Службы Web публикуют интерфейсы своих распределенных объектов на языке WSDL. Для публикации создается реестр *UDDI* (Universal Description and Discovery Interface) — всемирный Web-каталог распределенных объектов.

Запрос от клиента принимает SOAP-сервер, просто расшифровывающий и передающий его программе, которая может интерпретировать и выполнить запрос. Поскольку запрос сформулирован на языке XML, нужна просто программа, умеющая интерпретировать этот язык, а такие программы легко написать на любом языке программирования и выполнить на любой платформе.

Таким образом, службы Web сочетают простоту и универсальность взаимодействия распределенных объектов, что, возможно, приведет к их повсеместному применению. Поживем — увидим.

Часть V

Серверная часть распределенного приложения Java

Глава 20

Сервлеты и JSP

Первоначально перед Web-серверами стояла простая задача: найти и отправить клиенту файл, указанный в полученном от клиента запросе. Запрос составлялся тоже очень просто в специально придуманной форме URL.

Потом понадобилось сделать на сервере какую-либо небольшую предварительную обработку отправляемого файла. Web-сервер усложнился — появились включения на стороне сервера SSI (Server Side Include) и различные приемы динамической генерации страниц HTML.

Затем возникла необходимость выполнять на сервере процедуры. В запрос URL вставили вызов процедур, а на сервере реализовали технологию CGI (Common Gateway Interface). Теперь в запросе URL указывается процедура, которую надо выполнить на сервере, и записываются аргументы этой процедуры, 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 `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 стандартный вывод. Неожиданную популярность получил язык Perl. Оказалось, что на нем удобно писать CGI-программы. Возникли специальные языки: PHP, ASP, серверный 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.

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.

Один контейнер может управлять работой нескольких установленных в него сервлетов. При этом один контейнер может 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. Некоторые адреса фирм — производителей сервлетных контейнеров — официально они называются Servlet Container Providers, приведены в *главе 1*. Точное распределение обязанностей между Web-сервером и контейнером сервлетов осуществляется их производителями.

В стандартную поставку J2EE SDK входит контейнер сервлетов Tomcat как составная часть J2EE-сервера приложений.

Как водится в технологии Java, понятие "сервлет" описывается интерфейсом `Servlet`. Рассмотрим его подробнее.

Интерфейс *Servlet*

Выше 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. В старых версиях сервлетных контейнеров конфигурационный файл назывался servlet.properties и содержал пары "имя—значение". В новых версиях контейнеров конфигурационный файл записывается на языке XML и называется web.xml. Конфигурационный файл 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 все чаще используется построитель `ant`, написанный на языке Java. Он разработан Apache Software Foundation в рамках проекта Jakarta.

В стандартную поставку J2EE SDK входит графическая утилита установки Application Deployment Tool, вызываемая из командной строки набором команды `deploytool`.

В листинге 20.1 показан фрагмент конфигурационного файла web.xml, созданного для контейнера Tomcat. Разъяснения элементов XML даны в комментариях.

Листинг 20.1. 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 соответствует 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", то есть 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", то есть 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 тега `<servlet-name>` конфигурационного файла, набор начальных параметров, взятых из тегов `<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 20.2 приведен простейший сервлет, отправляющий клиенту свое имя и начальные параметры.

Листинг 20.2. Начальные параметры сервлета

```
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 20.2 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 20.2 компилируется обычным образом, а затем полученный сервлет `InfoServlet.class` устанавливается в контейнер. Процедура установки выполняется утилитой установки, входящей в состав контейнера сервлетов или сервера приложений, либо make-файлом. Для утилиты deploytool этот процесс описан в *главе 1*. Напомним только, что J2EE-сервер уже должен быть запущен, а в процессе установки надо на странице **Web Context** заполнить поле **Context Root** — виртуальный каталог, в котором будет расположен сервлет, например, "InfoAppl". Кроме того, в процессе установки надо задать начальные параметры.

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, другим.

> ***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 команды **Update and Redeploy** меню **Tools**.

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) сервлета составляют каталоги и файлы, описывающие Web-приложение. Они содержат, в частности, код сервлета, изображения, чертежи, конфигурационный файл, 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 J2EE 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 можно определить атрибуты, значениями которых служат 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);
```

С помощью атрибутов удобно 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, хранящуюся в запросе клиента. Кроме того, контейнер 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()` и `getRemoteHost()` возвращают IP-адрес и полное DNS-имя отправителя запроса, а методы `getServerName()` и `getServerPort()` возвращают имя и номер порта сервера, принявшего запрос.

Метод `getScheme()` возвращает схему запроса: `http:`, `https:`, `ftp:` и т. д., а метод `getProtocol()` возвращает 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()` возвращает 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 20.5. 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 пока не будет прочитано `length` символов. 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 известного класса `BufferedReader` из пакета java.io. Он описан, например, в книгах [1, 11, 15, 18]. 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)` устанавливает длину тела ответа.

Тело ответа передается 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 20.2.

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 оставаться в спящем состоянии, ожидая следующего запроса, или выгружаться, 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 соблюдайте 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.

Если параллельная работа неудобна для сервлета, то можно организовать последовательное выполнение запросов. Для этого сервлет должен реализовать интерфейс `SingleThreadModel`. Это пустой интерфейс-пометка без констант и методов, он служит только флагом, указывающим контейнеру, что следует организовать очередь последовательно выполняющихся запросов к сервлету. Этот вариант использован в листинге 20.6, описывающем сервлет, который должен последовательно посылать клиенту изображения одно за другим.

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 20.2. Упростим его, не определяя метод `init()`, а прямо используя реализацию методов интерфейса `ServletConfig`, сделанную в классе `GenericServlet`. Этот вариант приведен в листинге 20.3. В него добавлено еще получение контекста сервлета.

Листинг 20.3. Упрощенное чтение начальных параметров сервлета

```
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.

Адрес URL от названия схемы `http` до вопросительного знака возвращает метод

```
public String getRequestURL();
```

Часть этого адреса, показывающая путь к сервлету, получается методом

```
public String getServletPath();
```

Часть пути, определяющая контекст сервлета, возвращается методом

```
public String getContextPath();
```

Часть URI после вопросительного знака извлекается 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 извлечь 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 cookie, хранящиеся в браузере клиента, в виде массива объектов класса `Cookie` методом

```
public Cookie[] getCookies();
```

В пакете javax.servlet.http есть класс `HttpServletRequestWrapper` — прямая реализация интерфейса `HttpServletRequest`, который расширяет класс `ServletRequestWrapper`. Объект этого класса создается конструктором

```
public HttpServletRequestWrapper(HttpServletRequest req);
```

и обладает всеми методами интерфейса `HttpServletRequest`. Для получения дополнительных возможностей объекта, содержащего HTTP-запрос, например, для написания фильтра, разработчики 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);
```

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`.

Обычно код ответа заносится в сообщение об ошибке `message`, посылаемое методом

```
public void sendError(int sc, String message);
```

Если надо послать стандартное сообщение об ошибке, например, "404 Not Found", то применяется метод

```
public void sendError(int sc);
```

Этот метод использован в листинге 20.6.

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`. Для получения дополнительных возможностей объекта, содержащего запрос, разработчики 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()`. Этот метод уже определен, причем он 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 один из методов, соответствующий этому HTTP-методу

```
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 просто посылают сообщение о том, что они не реализованы. Чаще всего приходится переопределять методы `doGet()` и `doPost()`.

Выполним с помощью сервлета регистрацию ученика дистанционной системы обучения СДО, которая была реализована апплетом в листингах 14.1—14.2. 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 20.4 приведена HTML-форма регистрации ученика (см. рис. 14.1).

Листинг 20.4. Форма регистрации ученика 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>
```

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 20.5. Страница подтверждения регистрации выглядит так же, как на рис. 14.2.

Листинг 20.5. Сервлет регистрации ученика СДО

```
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 20.5 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 JServer с контейнером сервлетов Apache/JServ. Можно установить сервлет прямо в Oracle Server и использовать для соединения с базой серверный драйвер JDBC с именем `kprb`. Об этом драйвере говорилось в *главе 11*. 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 20.6 приведен пример сервлета, позволяющего клиенту просматривать изображения типа GIF и JPEG в каталоге, заданном начальным параметром сервлета.

Листинг 20.6. Сервлет, отправляющий изображения клиенту

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ImageServlet extends HttpServlet
                      implements SingleThreadModel{

  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 имеет 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 20.5 завязывается диалог. Ученик регистрируется и посылает све-

дения о себе на сервер СДО. Сервер предлагает ученику выбрать учебный курс. Ученик выбирает курс и посылает его название серверу. Сервер должен занести имя курса в учетную карточку ученика, а для этого ему надо знать регистрационный номер ученика.

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 `setXxx()` позволяют добавить в объект остальные сведения, а методы `getXxx()` — 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 при чтении 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 этого 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 cookie, то формируется идентификатор клиента в строке URL и т. д.

Основу средств создания сеанса связи с клиентом составляет интерфейс `HttpSession`. Объект типа `HttpSession` создается 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 не существует, 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, 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 для 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 значения атрибута — 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. У разработчика есть возможность 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 тоже 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 производится методом `doFilter()`, который получает объекты `req` и `resp`, изменяет их и передает управление следующему фильтру в цепочке с помощью аргумента `chain`.

Организует цепочку контейнер, интерфейс `FilterChain` описывает только один метод

```
public void doFilter(ServletRequest req, ServletResponse resp);
```

Для того чтобы передать управление следующему фильтру 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 возникает необходимость определить кодировку параметров запроса и переслать 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`. В листинге 20.7 приведен пример такой реализации в контейнере сервлетов Tomcat. Оцените качество кодирования и посмотрите, почему, используя 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 20.7. 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 20.7, 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 он берет кодировку из своего начального параметра `"encoding"`. После этого 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 таким образом.

```
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);

      // ...

      chain.doFilter(mreq, mresp);
   }

   public void destroy(){
      mreq = null;
      mresp = null;
   }
}
```

После того как класс-фильтр написан и откомпилирован, 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>`. Эту операцию можно выполнить 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 к сервлетам, но и к 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.

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 другому ресурсу. Если требуемый ресурс находится в том же контексте, что и сервлет, который его вызывает, то для получения ресурса надо 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 надо 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).

Страницы JSP

Как видно из приведенных листингов, большую часть сервлета занимают операторы вывода в выходной поток тегов HTML, формирующих результат — страницу HTML. Эти операторы повторяются из сервлета в сервлет. Возникла идея не записывать теги HTML в операторах Java, а наоборот, записывать операторы Java в странице HTML с помощью тегов специального

вида. Затем обработать полученную страницу препроцессором, распознающим все теги и преобразующим их в код сервлета Java.

Так получился язык разметки *JSP* (JavaServer Pages), расширяющий язык HTML тегами вида `<% имя тега атрибуты %>`. С помощью тегов можно не только внести описания, выражения и операторы Java, но и вставить в страницу файл с текстом или изображением, вызвать объект Java или компонент JavaBean или компонент EJB.

Программист может даже расширить язык JSP своими собственными, как говорят, *заказными тегами* (custom tags).

В листинге 20.8 приведен пример страницы "Hello World" с текущей датой, записанной на языке JSP.

Листинг 20.8. 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 20.8, в сервлет. Препроцессор обрабатывает страницу JSP автоматически при первом обращении к ней. 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, сохранить ее в файле с расширением jsp и установить файл в контейнер, так же как и страницу 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. Для совместимости с ним практически все теги JSP продублированы элементами XML с именами из пространства имен `jsp:`. Например, страницу JSP, записанную в листинге 20.8, можно написать так, как это сделано в листинге 20.9.

Листинг 20.9. Простейшая страница JSP в форме XML

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
   <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>
```

Некоторые теги — `<jsp:forward>`, `<jsp:include>`, `<jsp:plugin>`, `<jsp:useBean>`, `<jsp:getProperty>`, `<jsp:setProperty>` — всегда записывались в форме XML.

Напомним, 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>
```

Учтите, что 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).

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 `jsp` и вставку элементов `CDATA`. После приведения получается документ XML, официально называемый *представлением XML* (XML View).

Теги JSP

Набор 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.

Комментарий отмечается тегом

```
<%-- Комментарий --%>
```

Это единственный тег, не имеющий XML-аналога.

Объявления полей и методов Java записываются в теге

```
<%! Объявления %>
<jsp:declaration> Объявления </jsp:declaration>
```

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, а не выражение.

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>
```

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 20.8 и 20.9 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 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, а выбрасывает исключение, 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, есть 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 XML.

Включение файла на этапе выполнения или включение результата выполнения сервлета, который может быть представлен страницей JSP, выполняется тегом

```
<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 `%= выражение %`. 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 тега):

```
<jsp:include Атрибуты >
   Теги <jsp:param>
</jsp:include>
```

В теле тега 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 включаемому ресурсу как его начальные параметры, и их имена, разумеется, должны совпадать с именами начальных параметров ресурса.

Встроенные объекты JSP

Каждая страница JSP может пользоваться в выражениях и скриптлетах девятью готовыми встроенными объектами, создаваемыми контейнером JSP при выполнении сервлета, полученного после компиляции страницы JSP. Выше уже использовались 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 20.10 приведена страница JSP, использующая скриптлеты и встроенные объекты для организации запроса к базе СДО.

Листинг 20.10. 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 компонентам JavaBeans с помощью тега

```
<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 вторым 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`.
- 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 тега 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 начальным параметром страницы JSP, 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 страницы JSP вплоть до совпадения регистров букв.

Для получения свойств уже вызванного компонента JavaBean с именем `"myBean"` существует тег

```
<jsp:getProperty name="myBean" property="имя свойства" />
```

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
>
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 похожи на параметры 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 можно поместить 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" />
```

в котором указывается 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>
```

Заказные теги

Разработчик страниц JSP может расширить набор тегов JSP, создав свои собственные, как говорят, *заказные теги* (custom tags). Заказные теги создаются сразу целой библиотекой, даже если в нее входит только один тег. Описание каждой библиотеки хранится в отдельном XML-файле с расширением tld, называемом *описателем библиотеки тегов TLD* (Tag Library Descriptor). Этот файл хранится в каталоге WEB-INF конкретного Web-приложения или в его подкаталоге. Если Web-приложение упаковано в JAR-архив, то TLD-файлы, описывающие библиотеки заказных тегов этого приложения, находятся в каталоге архива META-INF.

В листинге 20.11 приведен пример простого TLD-файла, описывающего библиотеку заказных тегов с одним тегом `head`, реализуемым классом `HeadTag`.

Листинг 20.11. Описатель TLD библиотеки тегов

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
   PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
   "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-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 заказных 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="1.2"
>
   Страница JSP
</jsp:root>
```

Как видите, к каждой странице JSP всегда подключается библиотека с префиксом тегов `jsp`. Стандартные (core) теги JSP входят в эту библиотеку.

Префиксы `jsp`, `jspx`, `java`, `javax`, `servlet`, `sun`, `sunw` зарезервированы фирмой Sun Microsystems, их нельзя употреблять в заказных библиотеках 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 20.11. 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, в их каталогах META-INF.

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 заказного тега требует предварительной обработки, то надо использовать расширение интерфейса `IterationTag` — интерфейс `BodyTag`. Эти интерфейсы собраны в пакет javax.servlet.jsp.tagext. В этом пакете есть и готовые реализации указанных интерфейсов — класс `TagSupport`, реализующий интерфейс `IterationTag`, и его расширение — класс `BodyTagSupport`, реализующий интерфейс `BodyTag`.

Итак, для создания тега без тела или с телом, 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, хранится в TLD-файле, описывающем библиотеку, в XML-элементе `<tag>`. Во вложенном в него 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 заказной тег `<sdo:reg>`, например:

```
<sdo:reg name="%=request.getParameter(\"name\") %" />
```

то для обработки этого тега будет создан объект класса `RegTag` из пакета `sdotags`.

Класс-обработчик заказного тега

Как уже было сказано выше, класс-обработчик тега, не имеющего тела, или тело которого не требует преобразований, должен реализовать интерфейс `Tag` или расширить класс `TagSupport`. Если тело тега перед посылкой клиенту надо преобразовать, то классу-обработчику надо реализовать интерфейс `BodyTag` или расширить класс `BodyTagSupport`.

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();
```

выполняющему действия, завершающие обработку заказного 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 заказного тега. Если этот метод возвращает константу `EVAL_BODY_AGAIN` интерфейса `IterationTag`, то тело элемента будет обработано еще раз, если константу `SKIP_BODY` — обработка тела не будет повторяться.

Интерфейс `BodyTag` позволяет буферизовать выполнение тела элемента. Буферизация выполняется, если метод `doStartTag()` возвращает константу `EVAL_BODY_BUFFERED` интерфейса `BodyTag`. В этом случае перед обработкой тела тега контейнер обращается к методу

```
public void doInitBody();
```

который может выполнить различные предварительные действия.

У описанных 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` этот объект можно получить методом `getPageContext()` класса `JspFactory`, предварительно получив объект класса `JspFactory` его собственным статическим методом `getDefaultFactory()`. Сложность этих манипуляций — еще один аргумент для того, чтобы не реализовывать интерфейсы, а расширять класс `TagSupport` или класс `BodyTagSupport`.

Итак, для создания заказного тега без тела или с телом, не требующим обработки, удобнее всего расширить класс `TagSupport`, а для создания заказного тега с обработкой тела — расширить класс `BodyTagSupport`. При этом разработчик получает в свое распоряжение объект класса `PageContext` просто в виде защищенного поля `pageContext`.

Описанные выше 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 заказной тег без аргументов и без тела, просто отправляющий клиенту сообщение:

```
<sdo:info />
```

Реализующий его класс может выглядеть так, как показано в листинге 20.12.

Листинг 20.12. Класс простейшего заказного 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 20.12 надо откомпилировать обычным образом и установить в контейнер так же, как устанавливается сервлет. Проследите за соответствием пакетов Java и каталогов файловой системы: в каталоге WEB-INF/classes должен быть подкаталог sdotags с файлом InfoTag.class.

Заказной тег с атрибутами

Для каждого атрибута открывающего тега надо определить свойство JavaBean, т. е. поле с именем, совпадающим с именем атрибута, и методы доступа `getXxx()` и `setXxx()`. Например, немного выше мы определили заказной тег

```
<sdo:reg name="имя" />
```

с одним атрибутом `name`. Класс `RegTag`, содержащийся в листинге 20.13, реализует этот тег.

Листинг 20.13. Заказной 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;
   }
}
```

Заказной тег с телом

Если у заказного 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>` может быть еще одно значение, равное `tagdependent`. Оно применяется, если содержимое тела тега написано не на языке JSP, а на каком-то другом языке, например, это запрос на языке SQL.

Если содержимое тела тега не надо 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 заказной 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 20.14.

Листинг 20.14. Заказной 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 заказной 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 заказного тега

Если тело заказного 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 заказного тега. После выполнения метода `doInitBody()` 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.

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, 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 20.15 приведен фрагмент обработчика этого тега.

Листинг 20.15. Заказной 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 заказные теги, расположенные на одной странице JSP, должны взаимодействовать друг с другом. Например, тело тега может содержать другие, вложенные, теги JSP. Они могут быть независимыми от внешнего тега 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 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 можно обращаться 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 заказных тегов есть свои средства. Внешний и вложенный тег 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, необязательно непосредственно содержащего 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 заказной тег `connection`, устанавливает соединение с источником данных, как показано в листинге 20.16.

Листинг 20.16. 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 20.17.

Листинг 20.17. 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 — сделать его атрибутом 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 другой метод

```
public void setAttribute(String name, Object obj, int scope);
```

того же класса. Третий аргумент этого метода задает область действия атрибута — это одна из констант `PAGE_SCOPE`, `APPLICATION_SCOPE` или `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 — записать 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>`. Этот объект создается как расширение класса `TagExtraInfo`. Вот как выглядит это расширение для примера, приведенного выше:

```
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 использовать для проверки атрибутов тега на этапе компиляции страницы JSP.

Обработка исключений в заказных тегах

Выше уже говорилось о том, что обработку исключения, возникшего на странице JSP, можно перенести на другую страницу, указанную атрибутом `errorPage` тега `<%@ page %>`. В заказных 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>` можно использовать методы 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();
    }
}
```

Стандартная библиотека тегов JSTL

Несмотря на недолгую историю языка JSP, его возможности для создания заказных тегов были использованы многими фирмами и отдельными разработчиками. Уже создано множество библиотек заказных тегов. Их обзор можно посмотреть, например, на сайте **http://www.servletsuite.com/jsp.htm**. В рамках уже не раз упоминавшегося проекта Jakarta создается библиотека заказных тегов Struts, которую можно скопировать с сайта **http://jakarta.apache.org/builds/jakarta-struts/**.

Фирма Sun Microsystems создает стандартные библиотеки заказных тегов JSTL (JSP Standard Tag Library). Они составляют подпакеты пакета javax.servlet.jsp.jstl. Ко времени написания книги библиотеки JSTL еще не входили в стандартную поставку J2EE SDK, но их можно получить по адресу **http://java.sun.com/jstl/**.

В состав JSTL сейчас входят четыре библиотеки заказных тегов: core, xml, fmt и sql.

- Библиотека core описывается на странице JSP тегом

  ```
  <%@ taglib uri="/jstl-core" prefix="c" %>
  ```

  В нее входят теги, создающие подобие языка программирования: `<c:expr>`, `<c:set>`, `<c:forEach>`, `<c:forEachToken>`, `<c:if>`, `<c:choose>`, `<c:when>` и др. Их действие понятно из названий.

- Библиотека xml, описываемая тегом

  ```
  <%@ taglib uri="/jstl-xml" prefix="x" %>
  ```

  содержит аналогичные управляющие теги `<x:expr>`, `<x:set>`, `<x:forEach>`, `<x:if>`, `<x:choose>`, `<x:when>` и др., а также теги `<x:parse>`, `<x:param>`, `<x:transform>`, интерпретирующие документ XML.

- Библиотека fmt содержит теги, помогающие в интернационализации страниц JSP. Она описывается так:

  ```
  <%@ taglib uri="/jstl-fmt" prefix="xml" %>
  ```

  В нее входят теги `<fmt:locale>`, `<fmt:timeZone>`, `<fmt:formatDate>`, `<fmt:parseDate>` и др.

- Библиотека sql, описываемая тегом

  ```
  <%@ taglib uri="/jstl-sql" prefix="sql" %>
  ```

  содержит теги связи с базами данных: `<sql:driver>`, `<sql:query>`, `<sql:update>`, `<sql:param>`, `<sql:transaction>`.

Как видно из этого краткого описания, цель библиотек JSTL — создать так называемый *язык выражений* (Expression Language), чтобы действия можно было записывать как выражения алгоритмического языка, примерно так:

```
<c:if test="count > 0" >
   <c:set var="name" value="theName" />
</c:if>
```

Создатели JSTL стремятся сделать выражения независимыми от языка, который их интерпретирует. С этой целью, кроме обычного варианта библиотек, называемого JSTL-RT (Runtime), параллельно создается вариант JSTL-EL (Expression Language), содержащий те же теги. Названия TLD-файлов этого варианта содержат префикс тегов библиотеки, например, `"c.tld"`. В названиях TLD-файлов первого варианта встречается суффикс `"-rt"`, например, "c-rt.tld".

Глава 21

Компоненты на сервере

В предыдущей главе было показано, как с помощью Web-компонентов — сервлетов и страниц JSP — можно значительно расширить возможности Web-сервера, снабженного Web-контейнерами. В частности, сервлетами и страницами JSP можно реализовать промежуточный слой (middleware) распределенного приложения. Это делается по следующей схеме.

Сервлет или страница JSP, создающая промежуточный слой, принимает запрос от клиента с помощью Web-сервера. Сервлет анализирует запрос и обрабатывает его своими методами. При этом сервлет обращается к другим сервлетам или страницам JSP, если это нужно для обработки. В процессе обработки запроса сервлеты извлекают необходимую информацию из баз данных через драйверы JDBC. Ответ на запрос отправляется клиенту в виде одной или нескольких страниц HTML или документов XML тем же самым сервлетом, который принимал запрос.

Даже из такого краткого описания видно, что эта схема накладывает слишком большие обязанности на Web-компоненты. Сервлет становится громоздким и запутанным, замедляет работу приложения и захватывает много ресурсов на сервере. Страница JSP заполняется скриптлетами, а вы, наверное, уже заметили, что программный код, записанный скриптлетами, выглядит чужеродным элементом на странице JSP. Само строение языка JSP как командного языка отторгает появление программного кода на страницах JSP.

Следует еще учесть, что бóльшую часть работы по созданию промежуточного слоя распределенного приложения выполняют контейнеры сервлетов. Они обеспечивают безопасность работы приложения, создают транзакции и следят за их выполнением, формируют запросы от клиента в удобной форме и возвращают ответ клиенту.

Применение компонентов JavaBeans не спасает ситуацию. Они могут без изменений работать в разных сервлетах и легко применяются на страницах JSP, но не связаны с Web-контейнером и могут выполнять только вспомогательные функции.

Фирма Sun Microsystems нашла выход, разработав компоненты специального вида — *компоненты EJB* (Enterprise JavaBeans). Хотя в их названии есть слово "JavaBeans", они не похожи на компоненты JavaBeans. Общее у них толь-

ко то, что компоненты EJB, так же как и компоненты JavaBeans, переносимы, они могут выполняться любым сервером приложений, следующим спецификации EJB. Можно легко расширить возможности сервера приложений, установив в него компоненты EJB.

Компонент EJB состоит из нескольких интерфейсов и одного или нескольких классов, решающих задачу обработки информации, как говорят, реализующих бизнес-логику приложения. Компонент EJB, так же как и сервлет, работает под управлением специального программного слоя, *EJB-контейнера*. Один или несколько EJB-контейнеров вместе со связующими программами составляют *EJB-сервер*. Чтобы обеспечить связь с контейнером, компонент EJB должен реализовать несколько интерфейсов, методы которых выполняются EJB-контейнером.

Каждый компонент EJB должен быть установлен (deploy) в EJB-контейнер, подобно тому, как сервлет должен быть установлен в контейнер сервлетов. В процессе установки или до нее, для каждого компонента EJB создается конфигурационный XML-файл, содержащий параметры компонента. Этот конфигурационный файл называется "Deployment Descriptor", мы будем называть его *DD-файл*.

Клиент не может обратиться непосредственно к компоненту EJB, он обращается к EJB-контейнеру. Контейнер может обслуживать одновременно несколько клиентов и обращаться к нескольким компонентам. При обращении клиента EJB-контейнер выполняет необходимые проверки, начинает транзакцию, отыскивает нужный компонент EJB, загружает его и передает ему запрос клиента. Все обращения компонента EJB к источнику данных, другому компоненту или клиенту тоже выполняются через EJB-контейнер.

Для связи компонента EJB с EJB-контейнером создаются методы, осуществляющие обратный вызов (callback), применяется система именования JNDI, описанная в *главе 17*, или используются объекты, реализующие специально разработанный интерфейс `EJBContext`. Спецификация EJB описывает строгие правила взаимодействия компонента со своим контейнером и интерфейсы, образующие контракт компонента (Component Contract).

Компоненты EJB образуют еще один промежуточный слой, располагающийся между Web-компонентами и источником данных. В этом слое сосредотачивается вся бизнес-логика распределенного приложения. Компоненты EJB получают запрос от Web-компонентов, которые играют роль их клиентов, обрабатывают запрос, обращаясь для этого к источнику данных, и передают результат обратно Web-компонентам. Web-компоненты — сервлеты и страницы JSP — только принимают запрос от клиента, расшифровывают его и передают компонентам EJB.

Таким образом, распределенное приложение, использующее компоненты EJB, состоит из четырех слоев: клиента, Web-слоя, EJB-слоя и источника данных.

- Клиентом распределенного приложения может быть приложение Java, апплет, объект CORBA. Клиент взаимодействует с Web-слоем приложения.
- Web-слой составляют Web-компоненты приложения — сервлеты и страницы JSP. Они принимают запрос от клиента и передают его для обработки EJB-слою.
- EJB-слой состоит из компонентов EJB. Компоненты EJB обрабатывают запрос, обращаясь для этого к источнику данных.
- Источником данных могут служить базы данных, хранилища данных, файловые системы.

Фирма Sun выпустила спецификацию "Enterprise JavaBeans Specification", решающую все вопросы, связанные с компонентами EJB. Спецификация EJB содержит более пятисот страниц. Она подробно описывает строение компонентов EJB, процесс их создания и установки в контейнер, и даже расписывает роли создателей компонентов.

Роли создателей компонентов EJB

Спецификация EJB предлагает распределить весь процесс создания, установки и обслуживания компонентов EJB между шестью исполнителями.

- *Производитель компонентов EJB* (Enterprise Bean Provider) создает классы Java, образующие компоненты EJB. Результат его работы — содержащий компоненты JAR-архив, который спецификация называет *ear-jar-файлом*, и конфигурационный DD-файл для их установки в контейнер (deployment descriptor). Производитель компонентов должен хорошо знать технологию Java, а также разбираться в предметной области распределенного приложения и его структуре.
- *Сборщик приложения* (Application Assembler) получает от производителей ear-jar-файлы и конфигурационные DD-файлы, и объединяет содержащиеся в архивах компоненты EJB между собой и с другими компонентами приложения, например, Web-компонентами. При этом он дополняет DD-файлы связующей информацией. В результате сборки получается один или несколько JAR-архивов и конфигурационных DD-файлов. Сборщик приложения не обязан знать тонкости реализации компонентов EJB, но должен хорошо понимать их связь с клиентами и другими компонентами распределенного приложения.
- *Настройщик* (Deployer) получает от производителя или от сборщика JAR-архивы с компонентами EJB и устанавливает их в EJB-сервер или контейнер, настраивая конфигурационные DD-файлы и, возможно, создавая дополнительные классы для взаимодействия контейнера с компонентами EJB. В своей работе настройщик обычно использует какую-либо утилиту,

поставляемую фирмой — создателем контейнера. Настройщик должен хорошо знать особенности применяемого EJB-сервера или сервера приложений и уметь сконфигурировать его для устанавливаемого приложения.

- *Поставщик EJB-сервера* (EJB Server Provider) — представитель фирмы, выпускающей применяемый EJB-сервер, или специалист, хорошо разбирающийся в промежуточном программном обеспечении. Обычно этот специалист играет ту же роль, что и поставщик EJB-контейнера.
- *Поставщик EJB-контейнера* (EJB Container Provider) производит программное обеспечение для работы компонентов EJB в составе сервера приложений. На контейнер возлагается обеспечение безопасности, отслеживание транзакций, создание сеанса связи с клиентом и другие услуги.
- *Администратор системы* (System Administrator) следит за работой установленных компонентов EJB и обеспечивает безотказную работу распределенного приложения, используя утилиты администратора, создаваемые поставщиками EJB-сервера и EJB-контейнера.

Эти роли могут играть не только люди, но и программные продукты, а также люди, использующие программные продукты. Трудно рассчитывать на то, что представитель фирмы — поставщика программного обеспечения будет вместе с вами участвовать в разработке вашего распределенного приложения. Роли поставщиков, скорее всего, будут играть созданные ими продукты и утилиты управления программными продуктами — EJB-контейнерами, EJB-серверами или целиком серверами приложений. Некоторые роли могут быть совмещены, и выполняться одним исполнителем. В реальной жизни так и происходит, но спецификация считает, что разные роли исполняют разные работники.

Интерфейсы компонентов EJB

Кроме спецификации EJB, фирма Sun выпустила набор интерфейсов, устанавливающих правила взаимодействия компонентов с контейнером. Эти интерфейсы образуют пакет javax.ejb, входящий в стандартную поставку J2EE SDK.

Сам компонент EJB описывается очень просто — это набор классов, каждый из которых реализует интерфейс `EnterpriseBean`.

Интерфейс `EnterpriseBean` расширяет интерфейс `Serializable`, значит, каждый объект компонента EJB должен быть спроектирован таким образом, чтобы его можно было сериализовать. Как и интерфейс `Serializable`, интерфейс `EnterpriseBean` не определяет ни констант, ни методов. Его "реализация" служит пометкой, указывающей EJB-контейнеру на то, что данный класс является компонентом EJB.

Компоненты EJB подразделяются на три типа. Они описываются тремя интерфейсами, расширяющими интерфейс `EnterpriseBean`.

- Интерфейс `SessionBean` описывает так называемые *компоненты сеанса*. Мы будем называть их *session-компонентами*. Компоненты сеанса устанавливают сеанс связи с одним клиентом, сохраняя свое состояние между запросами клиента. Состояние запоминается только в оперативной памяти, поэтому после краха системы связь не восстанавливается. Есть более простой вариант session-компонентов, которые, вопреки своему названию, не сохраняют состояние между запросами. Как правило, они существуют только на время выполнения своих методов. С помощью session-компонентов можно выполнять самые разные действия: прием запросов, процедуры обработки данных, извлечение данных из хранилищ и их обновление, обращение к другим компонентам EJB, в том числе компонентам данных. Можно сказать, что session-компоненты играют роль "глаголов" в изложении бизнес-логики приложения.
- Интерфейс `EntityBean` описывает *компоненты данных*, представляющие информацию, извлеченную из источника данных, в виде объектов Java, удобных для обработки компонентами сеанса. Мы будем называть их *entity-компонентами*. Код SQL для работы с источником данных может записываться в entity-компоненте или генерироваться контейнером. Entity-компоненты поддерживают связь одновременно с несколькими клиентами и восстанавливаются EJB-контейнером после краха системы. Они играют роль "существительных" в изложении бизнес-логики приложения. Entity-компоненты могут существовать столько же, сколько существует хранилище данных.
- Интерфейс `MessageDrivenBean` описывает *компоненты, управляемые сообщениями*. Они выполняют те же функции, что и компоненты сеанса, но не поддерживают связь с клиентом, а начинают выполнение своих методов асинхронно после получения первого сообщения от клиента. Мы будем называть их *MDB-компонентами*. Компонент, управляемый сообщениями, может одновременно обрабатывать несколько сообщений. В настоящее время передача сообщений происходит под управлением системы JMS, описанной в *главе 16*, в дальнейшем предполагается внедрить и другие системы сообщений. В системе JMS MDB-компонент играет роль слушателя и получателя сообщений, поступивших адресату.

Связь с клиентом

С точки зрения клиента session-компонент — это кратковременный объект, выполняющий обработку информации, а entity-компонент — это объектно-ориентированное представление информации, извлеченной из источника данных. С точки зрения компонента EJB клиент может быть *локальным* (lo-

cal) или *удаленным* (remote). Локальный клиент обязательно работает под управлением той же виртуальной машины Java, что и компонент EJB. Удаленный клиент может выполняться той же самой или другой виртуальной машиной. Чаще всего он работает на другом компьютере.

Конструкция удаленного клиента не зависит от его расположения. Удаленный объект всегда использует механизм RMI для передачи аргументов, даже если он работает на той же виртуальной машине Java. Поэтому, в частности, аргументы методов, доступных удаленному клиенту, и их результаты передаются по значению. Кроме того, они должны быть сериализуемыми. Удаленным клиентом может служить апплет, приложение Java, сервлет, документ JSP, клиент CORBA, другой компонент EJB, установленный в тот же или другой контейнер.

Локальный клиент тесно связан с компонентом EJB, он работает на той же виртуальной машине Java. Аргументы и результаты методов, к которым обращается локальный клиент, передаются по ссылке. Поэтому при их разработке следует учитывать побочные эффекты, возникающие из-за того, что аргументы могут изменяться во время выполнения методов. Локальным клиентом компонента EJB может быть другой компонент EJB или Web-компонент. Локальные клиенты очень удобны в тех случаях, когда они работают в том же контейнере, что и компонент EJB, особенно необходим обмен большими объемами информации.

Для обеспечения связи с контейнером производитель session-компонента и entity-компонента должен создать еще по два интерфейса для локального и удаленного клиента. Эти интерфейсы должны расширять стандартные интерфейсы из пакета javax.ejb.

Интерфейсы локального клиента расширяют интерфейсы `EJBLocalHome` и `EJBLocalObject`, интерфейсы удаленного объекта — интерфейсы `EJBHome` и `EJBObject`. Для реализации связи удаленного клиента с компонентом стандартно применяется механизм RMI, описанный в *главе 15*, поэтому интерфейсы `EJBHome` и `EJBObject` расширяют интерфейс `Remote`. Производитель компонента не должен беспокоиться о реализации интерфейсов — она выполняется EJB-контейнером.

Интерфейсы типа `EJBLocalObject` и `EJBObject`, называемые *remote-интерфейсами*, описывают методы, предоставляемые клиенту компонентом EJB. Интерфейс типа `EJBObject`, предназначенный для удаленного клиента, играет роль удаленного интерфейса системы RMI, поэтому все методы этого интерфейса должны выбрасывать исключение класса `RemoteException`.

Интерфейсы типа `EJBLocalHome` и `EJBHome`, называемые *home-интерфейсами*, используются контейнером по запросу клиента для создания и удаления экземпляров компонента EJB или для поиска существующего объекта типа `EntityBean`. Они обеспечивают жизненный цикл компонента. Все методы ин-

терфейса, расширяющего интерфейс `EJBHome`, должны выбрасывать исключение класса `RemoteException`, поскольку они используются системой RMI.

Для MDB-компонентов ни remote- ни home-интерфейсы не нужны.

Состав компонента EJB

Итак, производитель компонента EJB, рассчитанного на использование удаленными клиентами, должен, как минимум, создать класс, реализующий один из интерфейсов `SessionBean`, `EntityBean` или `MessageDrivenBean`, в котором написать бизнес-методы обработки информации. Для session- и entity-компонентов надо еще оформить два интерфейса, расширяющих интерфейсы `EJBObject` и `EJBHome`. Если компонент будет работать с локальными клиентами, то надо разработать еще два интерфейса, расширяющих интерфейсы `EJBLocalObject` и `EJBLocalHome`. После этого производитель должен написать конфигурационный DD-файл компонента.

Обратите внимание на то, что компонент EJB не реализует home- и remote-интерфейсы, хотя эти интерфейсы содержат сигнатуры методов компонента. Реализацией интерфейсов занимается EJB-контейнер, а реализует он их не в виде объекта, выполняющего методы, описанные в интерфейсе, а в виде заглушек (stubs), обращающихся к объекту EJB. Заглушки remote- и home-интерфейсов будут пересланы удаленному клиенту, как положено в системе RMI. Удаленный клиент на самом деле обращается к методам заглушек, заглушки связываются с EJB-контейнером, а тот создает экземпляр компонента EJB для выполнения запрошенных методов.

Для работы с entity-компонентом производитель компонента создает еще один класс, описывающий в виде объекта так называемый *первичный ключ* (primary key) компонента, однозначно определяющий entity-компонент. Такое название взято потому, что очень часто entity-компонент представляет строку таблицы, в таком случае удобно взять для идентификации первичный ключ таблицы.

Спецификация EJB вводит соглашение об именах классов и интерфейсов, составляющих компонент EJB. Если мы решили создать компонент `"Cool"`, то спецификация предлагает:

- компонент назвать `CoolEJB`;
- класс компонента назвать `CoolBean`;
- home-интерфейс назвать `CoolHome`;
- remote-интерфейс назвать `Cool`;
- локальный home-интерфейс назвать `LocalCoolHome`;
- локальный remote-интерфейс назвать `LocalCool`;
- первичный ключ, если он нужен, назвать `CoolPK`.

В результате, если мы не собираемся работать с локальными клиентами, то получаем файлы с именами CoolBean.java, CoolHome.java, Cool.java и CoolPK.java, содержащие исходный текст компонента EJB.

Полученные после их компиляции файлы CoolBean.class, CoolHome.class, Cool.class и CoolPK.class упаковываются в JAR-архив, называемый, как уже говорилось выше, ear-jar-файлом (EAR — Enterprise ARchive). В этом архиве создается каталог META-INF, содержащий конфигурационный DD-файл, обычно именуемый ear-jar.xml. Файл ear-jar и DD-файл можно создать вручную или в процессе установки с помощью утилиты установки.

В листинге 21.1 приведен конфигурационный DD-файл компонента `SimpleEAR`, написанного в *главе 1*. Он создан утилитой установки deploytool.

Листинг 21.1. Конфигурационный DD-файл ear-jar.xml-компонента EJB

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE ejb-jar PUBLIC
    '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN'
    'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>
  <display-name>Ejb1</display-name>
  <enterprise-beans>
    <session>
      <display-name>SimpleEJB</display-name>
      <ejb-name>SimpleEJB</ejb-name>
      <home>SimpleHome</home>
      <remote>Simple</remote>
      <ejb-class>SimpleEJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
      <security-identity>
        <description></description>
        <use-caller-identity></use-caller-identity>
      </security-identity>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <method-permission>
```

```
<unchecked />
<method>
  <ejb-name>SimpleEJB</ejb-name>
  <method-intf>Home</method-intf>
  <method-name>remove</method-name>
  <method-params>
    <method-param>java.lang.Object</method-param>
  </method-params>
</method>
<method>
  <ejb-name>SimpleEJB</ejb-name>
  <method-intf>Remote</method-intf>
  <method-name>getHandle</method-name>
  <method-params />
</method>
<method>
  <ejb-name>SimpleEJB</ejb-name>
  <method-intf>Home</method-intf>
  <method-name>remove</method-name>
  <method-params>
    <method-param>javax.ejb.Handle</method-param>
  </method-params>
</method>
<method>
  <ejb-name>SimpleEJB</ejb-name>
  <method-intf>Home</method-intf>
  <method-name>getHomeHandle</method-name>
  <method-params />
</method>
<method>
  <ejb-name>SimpleEJB</ejb-name>
  <method-intf>Remote</method-intf>
  <method-name>getPrimaryKey</method-name>
  <method-params />
</method>
<method>
  <ejb-name>SimpleEJB</ejb-name>
  <method-intf>Home</method-intf>
```

```
        <method-name>getEJBMetaData</method-name>
        <method-params />
      </method>
      <method>
        <ejb-name>SimpleEJB</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>remove</method-name>
        <method-params />
      </method>
      <method>
        <ejb-name>SimpleEJB</ejb-name>
        <method-intf>Home</method-intf>
        <method-name>create</method-name>
        <method-params />
      </method>
      <method>
        <ejb-name>SimpleEJB</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>getResult</method-name>
        <method-params />
      </method>
      <method>
        <ejb-name>SimpleEJB</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>isIdentical</method-name>
        <method-params>
          <method-param>javax.ejb.EJBObject</method-param>
        </method-params>
      </method>
      <method>
        <ejb-name>SimpleEJB</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>getEJBHome</method-name>
        <method-params />
      </method>
    </method-permission>
  </assembly-descriptor>
</ejb-jar>
```

Для каждого приложения создается только один содержащий его ear-jar-файл. Описания компонентов EJB входят в этот файл как отдельные модули или входят в состав других модулей. Каждый модуль описывается своим DD-файлом. Модули, входящие в ear-jar-файл, описываются DD-файлом с именем application.xml, лежащим в каталоге META-INF верхнего уровня. Пример такого файла приведен в листинге 21.2.

Например, утилита установки deploytool, входящая в стандартную поставку J2EE SDK, упаковывает class-файлы компонента EJB в ear-jar-файл с именем ejb-jar-ic.jar, образующий один из модулей приложения. В каталоге META-INF этого архива она создает конфигурационный DD-файл с именем ejb-jar.xml, в котором перечисляются имена интерфейсов и классов, составляющих компонент EJB, а также их методы. Все модули приложения утилита deploytool упаковывает в JAR-архив, который спецификация предлагает назвать по имени компонента EJB, в нашем примере Cool.ear. Файл application.xml, находящийся в каталоге META-INF этого архива, описывает модули, включенные в приложение.

Клиентские программы, входящие в распределенное EJB-приложение, тоже требуют установки в EJB-контейнер. В процессе установки создается отдельный модуль для каждого клиента. Утилита deploytool дает ему имя app-client-ic.jar, а входящий в его каталог META-INF конфигурационный DD-файл называет application-client.xml.

Наконец, во время установки можно создать отдельный ear-jar-файл для клиентской программы, содержащий все необходимое для работы клиента на машине, не имеющей EJB-сервера. Сервер, после того как клиент связался с ним, извлекает из этого ear-jar-файла нужные компоненты. В частности, клиентский ear-jar-файл содержит реализацию remote- и home-интерфейсов, сделанную контейнером — заглушки с именами, в нашем примере, _Cool_Stub.class и _CoolHome_Stub.class.

В листинге 21.2 показан конфигурационный DD-файл application.xml клиентского ear-jar-файла компонента `SimpleEAR` из *главы 1*. Он создан утилитой deploytool.

Листинг 21.2. Конфигурационный DD-файл application.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE application PUBLIC
   '-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN'
   'http://java.sun.com/dtd/application_1_3.dtd'>

<application>
  <display-name>Simple</display-name>
```

```
  <description>Application description</description>
  <module>
    <java>app-client-ic.jar</java>
  </module>
  <module>
    <ejb>ejb-jar-ic.jar</ejb>
  </module>
</application>
```

Разновидности компонентов EJB

Компоненты EJB имеют несколько разновидностей. Проще всего устроены session-компоненты. Различают две их разновидности: session-компоненты без сохранения состояния (stateless), их обозначают аббревиатурой *SLSB*, и session-компоненты, сохраняющие измененное состояние и передающие его от вызова к вызову (stateful). Их обозначают аббревиатурой *SFSB*. Разновидность session-компонента указывается тегом `<session-type>` в DD-файле (см. листинг 21.1), но определяется строением класса компонента.

Session-компонент без сохранения состояния можно рассматривать просто как распределенный объект, предоставляющий клиенту набор методов реализации бизнес-логики приложения. Его удобно применять для выполнения расчетов, синтаксического разбора запросов, аутентификации клиента, сортировки, перебора строк выборки и других вспомогательных операций, часто выполняемых клиентами. Текущее состояние компонента не сохраняется, не запоминается и не передается от одного запроса другому. Это позволяет одному экземпляру компонента работать последовательно со многими клиентами. Контейнер, как правило, создает пул SLSB-компонентов и предоставляет клиенту компонент из пула. Один и тот же клиент может при следующем запросе получить другой экземпляр SLSB-компонента из пула.

Session-компонент с сохранением состояния, как следует из его названия, способен организовать сеанс связи с клиентом. Каждый экземпляр компонента работает только с одним клиентом. Состояние экземпляра сохраняется между запросами клиента, но не на диске, а только в оперативной памяти. SLSB-компоненты удобны для предоставления клиенту результатов последовательных действий, например, строк выборки. Кроме того, компоненты типа SFSB удобны для организации взаимодействия с другими компонентами и объектами, работающими на сервере. Компонент не восстанавливается после краха EJB-сервера.

Например, если вам, как клиенту, надо получать какие-то простые числа одно за другим, то вы создаете SLSB-компонент, вычисляющий простое число, и обращаетесь к нему, запрашивая каждый раз одно число. Если же

вам надо получать последовательные простые числа, то вы создаете SFSB-компонент, который хранит текущее простое число и выдает вам при каждом запросе следующее простое число.

Entity-компонент связан с источником данных, в котором сохраняет свое текущее состояние. Он автоматически восстанавливается после краха системы с тем состоянием, которое было у компонента после последней завершенной транзакции. Его удобно применять для представления сложной информации, извлеченной из источника данных, например, строки таблицы базы данных, в виде объектов Java. С компонентом могут работать одновременно несколько клиентов.

Эти особенности усложняют строение entity-компонента. Он должен создавать и поддерживать соединение с источником данных, обращаться к нему с запросами, своевременно обновлять данные, участвовать в распределенных транзакциях. Для облегчения написания entity-компонентов часть этих функций можно возложить на EJB-контейнер. Такие entity-компоненты, сохранность которых обеспечивается контейнером, называются *CMP-компонентами* (Container Managed Persistence). В отличие от них entity-компоненты, сами обеспечивающие свою сохранность, называются *BMP-компонентами* (Bean Managed Persistence).

CMP-компоненты, конечно, удобнее для производителя компонентов EJB, но они пока еще не предоставляют всех возможностей, которые обеспечиваются BMP-компонентами. Поэтому BMP-компоненты остаются в арсенале разработчика компонентов EJB.

MDB-компоненты отличаются от всех других разновидностей компонентов EJB. Пока это просто слушатели и получатели сообщений системы JMS.

Контекст компонента EJB

Контейнер, в котором работает компонент EJB, может предоставить ему некоторые сведения об окружении компонента. Эти сведения описаны интерфейсом `EJBContext`, реализуемым EJB-контейнером. Его методы

```
public EJBHome getEJBHome();
public EJBLocalHome getEJBLocalHome();
```

предоставляют ссылки на удаленный и локальный home-интерфейсы. Методы

```
public UserTransaction getUserTransaction();
public boolean getRollbackOnly();
```

предоставляют BMP-компоненту сведения о транзакции, в которой он участвует.

Метод

```
public Principal getCallerPrincipal();
```

дает сведения о вызвавшем компонент клиенте.

Интерфейс `EJBContext` расширен тремя интерфейсами `SessionContext`, `EntityContext` и `MessageDrivenContext` соответственно трем типам компонентов EJB. Расширения для session- и entity-компонентов, `SessionContext` и `EntityContext`, добавляют методы получения ссылки на remote-интерфейсы:

```
public EJBObject getEJBObject();
public EJBLocalObject getEJBLocalObject();
```

Интерфейс `EntityContext` содержит еще метод получения ссылки на первичный ключ:

```
public Object getPrimaryKey();
```

Объект, реализующий контекст, устанавливается в компонент каждого типа одним из методов

```
public void setSesionContext(SessionContext ctx);
public void setEntityContext(EntityContext ctx);
public void setMessageDrivenContext(MessageDrivenContext ctx);
```

описанных в интерфейсах `SessionBean`, `EntityBean` и `MessageDrivenBean` соответственно.

Итак, чтобы воспользоваться методами контекста, надо в классе компонента `XxxBean` определить поле, которое будет содержать ссылку на контекст, полученную методом `setSessionContext()`:

```
private SessionContext ctx;

setSessionContext(SessionContext ctx){
   this.ctx = ctx;
}
EJBHome home = ctx.getEJBHome();
```

Метаданные компонента

EJB-контейнер хранит сведения об установленных в нем компонентах EJB. Эти сведения описаны интерфейсом `EJBMetaData`. Удаленный клиент может получить ссылку на home-интерфейс методом

```
public EJBHome getEJBHome();
```

классы интерфейсов и первичного ключа следующими методами:

```
public Class getHomeInterfaceClass();
public Class getRemoteInterfaceClass();
public Class getPrimaryKeyClass();
```

Методом

```
public boolean isSession();
```

можно узнать, не является ли компонент session-компонентом, а методом

```
public boolean isStatelessSession();
```

уточнить, не является ли он, кроме того, SLSB-компонентом.

Идентификаторы компонента EJB

При создании компонента контейнер сохраняет ссылку на него в двух объектах, связанных с remote- и home-интерфейсами, — remote- и home-идентификаторах.

Remote-идентификатор — это объект, реализующий интерфейс `Handle`. Интерфейс `Handle` описывает только один метод

```
public EJBObject getEJBObject();
```

возвращающий remote-интерфейс компонента. Клиент может получить ссылку на удаленный идентификатор компонента методом

```
public Handle getHandle();
```

описанным в интерфейсе `EJBObject`.

Home-идентификатор реализует интерфейс `HomeHandle`. Интерфейс `HomeHandle` тоже описывает только один аналогичный метод

```
public EJBHome getEJBHome();
```

возвращающий home-интерфейс компонента. Клиент может получить ссылку на локальный идентификатор компонента методом

```
public HomeHandle getHomeHandle();
```

описанным в интерфейсе `EJBHome`.

Идентификаторы компонента используются для ссылки компонента на себя самого в тех же случаях, в каких применяется ссылка `this` в локальных системах.

Клиентские заглушки

Теперь, после того как мы рассмотрели типы и структуру компонентов EJB, можно познакомиться внимательнее с тем, что же получает клиент компонента EJB.

Итак, удаленный клиент в соответствии с правилами системы RMI получает две заглушки — объекты, реализующие некоторые расширения интерфейсов `EJBObject` и `EJBHome`. С точки зрения системы RMI это удаленные интерфейсы, расширяющие интерфейс-пометку `Remote`. Все методы этих интерфейсов выбрасывают, кроме всего прочего, исключение класса `RemoteException`.

Все заглушки устроены одинаково для любого типа компонентов EJB: Session Beans, Entity Beans и MessageDriven Beans, но некоторые их методы применимы только для компонентов определенного типа. Попытка применения таких методов к компонентам другого типа приводит к возникновению исключительной ситуации.

Всякое расширение интерфейса `EJBObject`, называемое remote-интерфейсом, описывает бизнес-методы компонента, предоставляемые удаленному клиенту. Они должны выбрасывать исключения класса `RemoteException`, типы их аргументов должны удовлетворять требованиям RMI/IIOP, описанным в *главе 15*. Кроме того, remote-интерфейс наследует методы интерфейса `EJBObject`:

- `public EJBHome getEJBHome()` — возвращает ссылку на home-интерфейс;
- `public Handle getHandle()` — возвращает ссылку на remote-идентификатор;
- `public Object getPrimaryKey()` — возвращает ссылку на первичный ключ, этот метод применим только к entity-компонентам;
- `public void remove()` — удаляет компонент;
- `public boolean isIdentical(EJBObject obj)` — сравнивает remote-интерфейсы.

Расширения интерфейса `EJBHome`, называемые home-интерфейсами, добавляют к своему суперинтерфейсу методы `createXxx()` создания компонента. Об этих методах мы будем говорить в следующих двух главах. Кроме того, они наследуют методы:

- `public HomeHandle getHomeHandle()` — возвращает ссылку на home-идентификатор;
- `public EJBMetaData getEJBMetaData()` — возвращает ссылку на метаданные компонента;
- `public void remove(Handle handle)` — удаляет компонент, указанный remote-идентификатором;

- `public void remove(Object pk)` — удаляет компонент, указанный первичным ключом, этот метод только для entity-компонентов.

Локальный клиент получает меньше возможностей. Кроме бизнес-методов компонента, локальный remote-интерфейс — расширение интерфейса `EJBLocalObject`, — наследует методы:

- `public EJBLocalHome getEJBLocalHome()` — возвращает ссылку на локальный home-интерфейс;
- `public Object getPrimaryKey()` — возвращает ссылку на первичный ключ, только для entity-компонентов;
- `public void remove()` — удаляет компонент;
- `public boolean isIdentical(EJBLocalObject obj)` — сравнивает локальные remote-интерфейсы.

Локальный home-интерфейс — расширение интерфейса `EJBLocalHome`, — кроме методов `createXxx()` создания компонента, описывает только один унаследованный метод удаления компонента

```
public void remove();
```

Переменные окружения компонента

В процессе установки компонента EJB или вручную в DD-файле можно определить значения переменных окружения компонента, подобно тому, как это делалось с сервлетами. Определение записывается в элементах `<ejb-entry>`, вложенных в описание компонента, в роли которого может выступать элемент `<session>`, `<entity>` или `<message-driven>`.

Частным случаем переменных окружения служат ссылки на другие компоненты EJB, точнее говоря, на их home- и remote-интерфейсы. Такие ссылки называются *EJB-ссылками* (EJB references). Они описываются в элементе `<ejb-ref>`, если описывается ссылка на удаленные интерфейсы, или в элементе `<ejb-local-ref>`, если ссылка делается на локальные интерфейсы. Эти элементы тоже вкладываются в описание компонента.

В листинге 21.3 приведен пример описания переменных окружения и EJB-ссылок.

Листинг 21.3. Фрагмент DD-файла с EJB-ссылкой и переменными окружения

```
<session>
   <ejb-name>VerifyEJB</ejb-name>
   <ejb-class>sdo.ejb.VerifyBean</ejb-class>
```

```
   <env-entry>
      <description>Наибольшее число попыток</description>
      <env-entry-name>max</env-entry-name>
      <env-entry-type>java.lang.Integer</env-entry-type>
      <env-entry-value>5</env-entry-value>
   </env-entry>
   <env-entry>
      <env-entry-name>min</env-entry-name>
      <env-entry-type>java.lang.Integer</env-entry-type>
      <env-entry-value>1</env-entry-value>
   </env-entry>
   <env-entry>
      <env-entry-name>name</env-entry-name>
      <env-entry-type>java.lang.String</env-entry-type>
      <env-entry-value>Иванов</env-entry-value>
   </env-entry>
   <ejb-ref>
      <description>Регистрация</description>
      <ejb-ref-name>ejb/reg</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
      <home>sdo.ejb.RegHome</home>
      <remote>sdo.ejb.Reg</remote>
   </ejb-ref>
</session>
```

Переменные окружения компонентов EJB хранятся в контексте `java:/comp/env` системы именования JNDI и во вложенных в него контекстах. EJB-ссылки спецификация рекомендует хранить в контексте `java:/comp/env/ejb` или во вложенных в него контекстах.

Чтобы получить значения определенных в листинге 21.3 переменных и EJB-ссылки, в классе компонента надо проделать примерно такие действия, которые приведены в листинге 21.4.

Листинг 21.4. Компонент, использующий EJB-ссылку и переменные окружения

```
package sdo.ejb;

import javax.ejb.*;
import javax.naming.*;
import javax.rmi.*;
```

```
public class VerifyBean implements SessionBean{

   public void list(int attempts){

      InitialContext ctx = new InitialContext();

      Object res = ctx.lookup("java:comp/env/ejb/reg");

      RegHome home = (RegHome)PortableRemoteObject.narrow(
            res, RegHome.class);

      Reg reg = home.findByPrimaryKey("123");

      if (nameCheck(attempts, reg.getName())){

      // Продолжение метода
      }
   }

   public boolean nameCheck(int attempts, String validName){

      InitialContext ctx = new InitialContext();

      Context env = (Context)ctx.lookup("java:comp/env");

      Integer max = (Integer)env.lookup("max");
      Integer min = (Integer)env.lookup("min");

      if (attempts > max.intValue() ||
          attempts < min.intValue())
        return false;

      String name = (String)env.lookup("name");

      if (name.equals(validName)) return true;
      else return false;
   }
}
```

Ограничения компонентов EJB

Поскольку компоненты EJB должны быть переносимы между EJB-серверами, они не могут использовать средства, специфичные для EJB-сервера. Это приводит к следующим ограничениям.

- Компоненты EJB не могут создавать новые подпроцессы (threads) и управлять существующими подпроцессами, приостанавливать их, менять приоритет или завершать их.
- Компоненты EJB не могут синхронизировать подпроцессы для синхронизации работы нескольких экземпляров компонента.
- Компоненты EJB не могут использовать изменяемые статические поля. Каждое поле типа `static` должно быть окончательным, `final`.
- Компоненты EJB не могут применять графические библиотеки для связи с консолью.
- Компонентам EJB запрещено использовать потоки ввода/вывода для связи с файловой системой.
- Компонентам EJB следует избегать применения сокетов.
- Компоненты EJB не должны пользоваться средствами исполняющей системы Java: создавать загрузчики классов, устанавливать менеджеры безопасности, изменять стандартные потоки ввода/вывода.
- Компонентам EJB нельзя загружать native-библиотеки.
- Компоненты EJB не должны добавлять классы в стандартные пакеты.
- Компонентам EJB следует проявлять осторожность при использовании ссылки `this`; безопаснее пользоваться методами `getEJBObject()`, описанными в интерфейсах `SessionContext` и `EntityContext`.

Глава 22

Создание Session и MessageDriven Beans

Как уже говорилось в предыдущей главе, у компонентов Session Beans есть две разновидности: session-компоненты с сохранением состояния (stateful) и session-компоненты без сохранения состояния (stateless). Первые сокращенно обозначаются SFSB, вторые — SLSB. Разновидность session-компонента указывается в элементе `<session-type>` конфигурационного DD-файла компонента, вложенном в описание данного компонента, т. е. в элемент `<session>`. EJB-контейнер руководствуется не только этой записью, а еще и наличием или отсутствием в классе компонента сериализуемых полей, точнее, наличием или отсутствием *режима диалога* (conversational state) компонента с клиентом. Режим диалога SFSB-компонента обеспечивают не только сериализуемые поля класса компонента, а и открытые сокеты, соединения с источником данных, открытые курсоры.

Создание и удаление session-компонента

Для создания session-компонента home-интерфейс, расширяющий интерфейс `EJBHome`, должен предоставить удаленному клиенту один или несколько методов `createXxx()`. Эти методы не описаны в интерфейсе `EJBHome`, их должен описать в расширении этого интерфейса производитель компонента EJB. Общие требования к этим методам следующие: они должны выбрасывать исключение класса `RemoteException` и типы их аргументов должны удовлетворять требованиям RMI/IIOP, т. к. это методы распределенного объекта системы RMI. Кроме того, методы `createXxx()` должны выбрасывать исключение класса `CreateException`.

Все остальное остается на долю производителя компонента. Производитель определяет сигнатуру метода и тип возвращаемого значения. Метод должен возвращать ссылку на remote-интерфейс компонента. Производитель расписывает каждый метод `createXxx()` в классе создаваемого компонента, соблюдая количество, типы и порядок следования его аргументов. Пусть, например, home-интерфейс листинга 22.1 описывает четыре метода `createXxx()`.

Листинг 22.1. Методы `createXxx()` home-интерфейса

```
import java.rmi.*;
import javax.ejb.*;

public interface SomeHome extends EJBHome{

   Some create()
      throws RemoteException, CreateException;

   Some create(String name, long id)
      throws RemoteException, CreateException;

   Some create(long id)
      throws RemoteException, CreateException;

   Some createNew(String name, long id)
      throws RemoteException, CreateException;
}
```

При записи методов `createXxx()` в классе компонента префикс их имен меняется, имена в нем имеют вид `ejbCreateXxx()`. Методы `ejbCreateXxx()` обязательно должны быть открытыми, `public`, иначе они не будут видны удаленному клиенту. Они не могут быть окончательными, `final`, или статическими, `static`.

Напомним, что класс компонента не реализует методы интерфейса, поэтому, в частности, методы `ejbCreateXxx()` возвращают значение типа `void` и не выбрасывают `RemoteException`. Класс компонента может выглядеть так, как показано в листинге 22.2.

Листинг 22.2. Методы `ejbCreateXxx()` класса session-компонента

```
import java.rmi.*;
import javax.ejb.*;

public class SomeBean implements SessionBean{

   public String name;
   public long id;
   public boolean news;
```

```
   public SomeBean(){}

   // Другие поля и бизнес-методы класса.

   // Пустая реализация означает, что EJB-контейнер будет
   // использовать свои методы создания компонента.
   public void ejbCreate(){}

   public void ejbCreate(String name, long id){
      this.name = name;  this.id = id;
   }

   public void ejbCreate(long id){
      name = "No name";  this.id = id;
   }

   public void ejbCreateNew(String name, long id){
      news = true; this.name = name; this.id = id;
   }

   // Реализация методов интерфейса. Пустая реализация
   // означает, что контейнер будет использовать заложенную
   // в него реализацию.
   public void setSessionContext(SessionContext ctx){}
   public void ejbRemove(){}
   public void ejbActivate(){}
   public void ejbPassivate(){}
}
```

Методы `createXxx()` с аргументами нужны только SFSB-компоненту для задания своего начального состояния. У session-компонентов без сохранения состояния, т. е. компонентов типа SLSB, в home-интерфейсе можно и нужно описать только один метод `create()` без аргументов. Он должен возвращать значение типа remote-интерфейса компонента (см. листинг 1.16). Пустая реализация метода `ejbCreate()` (см. листинг 1.17) показывает контейнеру, что следует создавать объект session-компонента своими методами.

Производитель SLSB-компонента может сделать непустую реализацию метода `ejbCreate()`, даже определить в нем значения полей класса компонента, но эти значения не будут сохраняться между вызовами методов компо-

нента. Для облегчения работы контейнера поля SLSB-компонента, кроме поля-ссылки на `SessionContext`, можно пометить как `transient`. Это предотвратит попытку их сериализации.

Создавая session-компонент, контейнер выполняет три действия.

1. Определяет класс компонента и выполняет метод `newInstance()` класса `Class`, который создает экземпляр класса с помощью конструктора по умолчанию класса контейнера.
2. Создает объект типа `SessionContext` и устанавливает его в компонент методом `setSessionContext(SessionContext)`.
3. Выполняет указанный ему клиентом метод `ejbCreateXxx()`.

Как видно из этого описания, контейнер использует только один конструктор session-компонента — конструктор по умолчанию.

Обратная операция удаления session-компонента выполняется методом

```
public void remove(Handle handle);
```

описанным в интерфейсе `EJBHome`. Этот метод реализуется контейнером, клиенту достаточно просто обратиться к нему.

Попытка применения второго метода `remove(Object pk)` интерфейса `EJBHome` вызовет исключение класса `RemoteException`, поскольку этот метод использует первичный ключ `pk` entity-компонента.

В процессе удаления компонента контейнер обращается к методу

```
public void ejbRemove();
```

интерфейса `SessionBean`, содержащемуся в классе компонента. В листинге 22.2 сделана пустая реализация этого метода, но производитель session-компонента может занести в него какие-то действия, связанные с закрытием и освобождением ресурсов.

Клиент, для того чтобы создать session-компонент и воспользоваться его методами, должен сначала отыскать ссылку на home-интерфейс компонента с помощью системы именования JNDI:

```
InitialContext ctx = new InitialContext();

SomeHome home = (SomeHome)PortableRemoteObject.narrow(
   ctx.lookup("java:comp/env/ejb/some"), SomeHome.class);
```

В процессе установки контейнер создает контекст `java:comp/env/ejb` системы JNDI, в который помещает JNDI-имена компонентов EJB.

После этого можно создавать экземпляры компонента EJB и обращаться к их методам:

```
Some s1 = home.create();
Some s2 = home.create("Иванов", 123L);
```

```
Some s3 = home.create(456L);
Some s4 = home.createNew("Иванов", 123L);
```

Активизация SFSB-компонента

Интерфейс `SessionBean` описывает еще два метода

```
public void ejbPassivate();
public void ejbActivate();
```

которые обязательно должны быть реализованы в классе session-компонента, хотя бы пустым образом.

Контейнер обращается к методу `ejbPassivate()`, когда он решает, что пора сохранить SFSB-компонент, который долгое время не используется. Обычно SFSB-компонент при сохранении сериализуется, поэтому все сохраняемые поля и ссылки должны быть сериализуемы. Сразу же после активизации сохраненного SFSB-компонента контейнер вызывает метод `ejbActivate()`. Таким образом, session-компоненты типа SLSB без сохранения состояния должны иметь пустую реализацию этих методов, а компоненты типа SFSB с сохранением состояния могут и должны в методе `ejbPassivate()` освободить временные несериализуемые ресурсы, например, закрыть соединения JDBC. Поля компонента, которые не нужно сохранять, можно пометить как `transient`.

Кроме этих методов и методов `ejbCreateXxx()` класс компонента содержит реализацию бизнес-методов, описанных в remote-интерфейсе компонента, а также поля и другие методы, необходимые компоненту для его работы.

Пример SFSB-компонента

Переложим часть работы, выполняемой сервлетом листинга 20.5, на session-компонент. Пусть session-компонент обращается к базе данных SDO, заносит в нее сведения о новом ученике, извлекает из нее сведения о читаемых курсах и вновь зарегистрированных учениках.

Сначала напишем remote-интерфейс, описывающий извлечение из базы данных SDO сведений о последнем зарегистрированном ученике и сведений о читаемых курсах. Этот интерфейс приведен в листинге 22.3.

Листинг 22.3. Remote-интерфейс методов регистрации ученика

```
import java.rmi.*;
import java.util.*;
import javax.ejb.*;
```

```
public interface RegPrep extends EJBObject{

   public String getNewStud() throws RemoteException;

   public Vector getCourses() throws RemoteException;
}
```

Затем напишем home-интерфейс. Опишем в нем один метод, как показано в листинге 22.4.

Листинг 22.4. Home-интерфейс методов регистрации ученика

```
import java.rmi.*;
import javax.ejb.*;

public interface RegPrepHome extends EJBHome{

   RegPrep create(String surname, String name,
                  String secname, String addr)
            throws CreateException, RemoteException;
}
```

Теперь напишем класс session-компонента. Он приведен в листинге 2.5.

Листинг 22.5. Класс компонента регистрации ученика

```
import java.sql.*;
import javax.naming.*;
import java.util.*;
import javax.ejb.*;

public class RegPrepBean implements SessionBean{

   private Connection con;
   private String driver;
   private String url;
   private String admin;
   private String password;

   public SessionContext ctx;
```

```
public void setSessionContext(SessionContext ctx){
   this.ctx = ctx;
}

public void ejbCreate(String surname, String name,
                      String secname, String addr){
  try{
    InitialContext ctx = new InitialContext();

    Context env = (Context)ctx.lookup("java:comp/env");

    driver   = (String)env.lookup("driver");
    url      = (String)env.lookup("url");
    admin    = (String)env.lookup("name");
    password = (String)env.lookup("password");

    Class.forName(driver);

    if (surname.length() * name.length() *
        secname.length() * addr.length() == 0){

        throw new UnsupportedOperationException();
     }

     String fullname = surname.trim() + " " +
         name.trim() + " " + secname.trim();

     con = DriverManager.getConnection(
         url, admin, 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();
```

```
    }catch(Exception e){
      System.err.println(e);
    }
  }

  public String getNewStud(){

    String fullname = null;

    try{
      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();

    }catch(Exception e){
      System.err.println(e);
    }
    return fullname;
  }

  public Vector getCourses(){

     Vector v = new Vector();

    try{
      Statement st = con.createStatement();

      ResultSet rs =
        st.executeQuery("SELECT course_name FROM courses");
```

```
        while (rs.next())
           v.addElement(rs.getString(1));

        rs.close();

      }catch(Exception e){
        System.err.println(e);
      }
      return v;
    }

    public void ejbPassivate(){
       con = null;
    }

    public void ejbActivate(){

      try{
         con = DriverManager.getConnection(
             driver, admin, password);
      }catch(Exception e){
        System.err.println(e);
      }
    }

    public void ejbRemove(){
      try{
        con.close();
      }catch(Exception e){
        System.err.println(e);
      }
    }
}
```

Компилируем полученные файлы RegPrep.java, RegPrepHome.java и RegPrepBean.java и устанавливаем компонент в EJB-контейнер. При этом получаем или пишем вручную DD-файл. В листинге 22.6 приведен DD-файл, созданный утилитой deploytool.

Листинг 22.6. Конфигурационный DD-файл session-компонента

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE ejb-jar PUBLIC
   '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN'
   'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>
  <display-name>Ejb2</display-name>
  <enterprise-beans>
    <session>
      <display-name>RegPrepBean</display-name>
      <ejb-name>RegPrepBean</ejb-name>
      <home>RegPrepHome</home>
      <remote>RegPrep</remote>
      <ejb-class>RegPrepBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Bean</transaction-type>
      <env-entry>
        <env-entry-name>driver</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>
           oracle.jdbc.driver.OracleDriver
        </env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>url</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>
           jdbc:oracle:thin:@homexp:1521:SDO
        </env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>admin</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>sdoadmin</env-entry-value>
      </env-entry>
```

```
    <env-entry>
      <env-entry-name>password</env-entry-name>
      <env-entry-type>java.lang.String</env-entry-type>
      <env-entry-value>sdoadmin</env-entry-value>
    </env-entry>
    <security-identity>
      <description></description>
      <use-caller-identity></use-caller-identity>
    </security-identity>
  </session>
</enterprise-beans>
<assembly-descriptor>
  <method-permission>
    <unchecked />
    <method>
      <ejb-name>RegPrepBean</ejb-name>
      <method-intf>Home</method-intf>
      <method-name>create</method-name>
      <method-params>
        <method-param>java.lang.String</method-param>
        <method-param>java.lang.String</method-param>
        <method-param>java.lang.String</method-param>
        <method-param>java.lang.String</method-param>
      </method-params>
    </method>
    <method>
      <ejb-name>RegPrepBean</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>getHandle</method-name>
      <method-params />
    </method>
    <method>
      <ejb-name>RegPrepBean</ejb-name>
      <method-intf>Home</method-intf>
      <method-name>remove</method-name>
      <method-params>
        <method-param>javax.ejb.Handle</method-param>
      </method-params>
```

```
    </method>
    <method>
      <ejb-name>RegPrepBean</ejb-name>
      <method-intf>Home</method-intf>
      <method-name>getHomeHandle</method-name>
      <method-params />
    </method>
    <method>
      <ejb-name>RegPrepBean</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>getNewStud</method-name>
      <method-params />
    </method>
    <method>
      <ejb-name>RegPrepBean</ejb-name>
      <method-intf>Home</method-intf>
      <method-name>getEJBMetaData</method-name>
      <method-params />
    </method>
    <method>
      <ejb-name>RegPrepBean</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>isIdentical</method-name>
      <method-params>
        <method-param>javax.ejb.EJBObject</method-param>
      </method-params>
    </method>
    <method>
      <ejb-name>RegPrepBean</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>getEJBHome</method-name>
      <method-params />
    </method>
    <method>
      <ejb-name>RegPrepBean</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>remove</method-name>
      <method-params />
    </method>
```

```
      <method>
        <ejb-name>RegPrepBean</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>getPrimaryKey</method-name>
        <method-params />
      </method>
      <method>
        <ejb-name>RegPrepBean</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>getCourses</method-name>
        <method-params />
      </method>
      <method>
        <ejb-name>RegPrepBean</ejb-name>
        <method-intf>Home</method-intf>
        <method-name>remove</method-name>
        <method-params>
          <method-param>java.lang.Object</method-param>
        </method-params>
      </method>
    </method-permission>
  </assembly-descriptor>
</ejb-jar>
```

Надо проследить, чтобы в процессе установки был создан JAR-архив со всей информацией, необходимой для работы клиента, в частности, в нем расположены классы-заглушки, реализующие home- и remote-интерфейсы в системе RMI. Имя этого архива обычно составляется из имени компонента EJB и слова `"Client"`. В утилите deploytool для создания клиентского JAR-архива надо поставить флажок в поле **Return Client Jar**. Будет создан файл RegPrepClient.jar, который следует применять вместе с библиотеками классов клиента.

Клиенты компонента EJB

Компонент готов к использованию. Теперь надо создать удаленного клиента и обратиться к компоненту. Удаленным клиентом может быть отдельное приложение Java, апплет, сервлет, страница JSP, другой компонент EJB или приложение Java, связанное с компонентом EJB и установленное в контейнер.

Отдельное приложение

Создадим сначала отдельное, самостоятельное приложение Java, которое будет обращаться к компоненту EJB по его JNDI-имени "reg", данному компоненту при его установке в контейнер. Это приложение показано в листинге 22.7. Чтобы не увеличивать объем листинга, приведен только простейший консольный вариант.

Листинг 22.7. Приложение в качестве клиента session-компонента

```
import javax.naming.*;
import java.util.*;
import javax.rmi.*;

public class RegPrepAppl{

   public static void main(String[] args){

     if (args.length != 4){
       System.err.println("Usage: java RegPrepClient"
             <surname> <name> <secname> <address>");
       System.exit(-1);
     }

     try{
       InitialContext ctx = new InitialContext();

       Object ref = ctx.lookup("reg");

       RegPrepHome home =
          (RegPrepHome)PortableRemoteObject.narrow(
             ref, RegPrepHome.class);

       RegPrep reg =
         home.create(args[0], args[1], args[2], args[3]);

       String fullname = reg.getNewStud();

       System.out.println("Последний зарегистрированный ученик " +
           fullname);
```

```
        Vector v = reg.getCourses();

        for (int i = 0; i < v.size(); i++)
           System.out.println(v.get(i));

      }catch(Exception e){
        System.out.println("From client: " + e);
      }
    }
}
```

После компиляции клиентской программы ее надо вызвать на выполнение, подключив клиентский JAR-архив следующим образом:

```
java -classpath "$CLASSPATH:RegPrepClient.jar" RegPrepAppl
   Иванов Петр Сидорович "ivan@some.com"
```

Сервлет

Сервлет применяется в качестве клиента компонентов EJB наиболее часто. Как правило, сервлет передает запрос, полученный от клиента, на обработку компонентам EJB и принимает от них обработанную информацию для передачи клиенту. В этом случае сервлет вместе с используемыми компонентами EJB образует единое J2EE-приложение.

Используем еще раз в качестве примера сервлет листинга 20.5, освободив его от обязанности связываться с базой данных. Переложим эту обязанность на созданный компонент EJB. Результат показан в листинге 22.8.

Листинг 22.8. Сервлет в качестве клиента session-компонента

```
import java.io.*;
import java.util.*;
import javax.naming.*;
import javax.rmi.*;
import javax.servlet.http.*;

public class RegPrepServ extends HttpServlet{

  private RegPrepHome home;

  public void init(){
```

```
  try{
     InitialContext ctx = new InitialContext();
     Object ref = ctx.lookup("reg");
     home = (RegPrepHome)PortableRemoteObject.narrow(
               ref, RegPrepHome.class);

  }catch(Exception e){
     System.err.println("From init exc: " + 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;
}

RegPrep reg = home.create(surname, name, secname, addr);

String fullname = reg.getNewStud();

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("Выберите учебный курс:<br> ");

pw.println("<form method=post action=" +
"\"http://homexp:8000/InfoAppl/servlet/CoursesServlet\">");
pw.println("<select size=5 name=courses>");

Vector v = reg.getCourses();
int n = v.size();

for (int i = 0; i < n; i++)
   pw.println("<option>" + v.get(i));

pw.println("</option></form></body></html>");
pw.flush();
pw.close();
```

```
      }catch(Exception e){
         System.err.println("From doPost(): " + e);
      }
   }
}
```

После компиляции сервлета его вместе с HTML-файлом regprep.html (см. листинг 20.4) следует установить в контейнер, указав EJB-ссылку на компонент `RegPrepBean`. Потом можно загружать файл regprep.html в браузер и вызывать сервлет, заполнив HTML-форму.

Страница JSP

Поскольку страница JSP компилируется в сервлет, создание клиента — страницы JSP ничем не отличается от создания клиента-сервлета.

Листинг 22.9 показывает страницу JSP, просто повторяющую сервлет листинга 22.8.

Листинг 22.9. Страница JSP в качестве клиента session-компонента

```
<%@ page import="javax.rmi.*, javax.naming.*, java.util.*"
    contentType="text/html;charset=windows-1251" %>

<%
 try{
   InitialContext ctx = new InitialContext();
   Object ref = ctx.lookup("reg");
   RegPrepHome home = (RegPrepHome)PortableRemoteObject.narrow(
           ref, RegPrepHome.class);

   String surname = request.getParameter("surname");
   String name    = request.getParameter("name");
   String secname = request.getParameter("secname");
   String addr    = request.getParameter("addr");
   if (surname.length() * name.length() *
       secname.length() * addr.length() == 0){
%>

<html><head>
<title>Продолжение регистрации</title>
```

```
</head><body>
<h2 align=center>Дистанционная система обучения СДО</h2>
<h3>Замечание:</h3>
Заполните, пожалуйста, все поля.<br>
</body></html>

<%
   }else{

     RegPrep reg = home.create(surname, name, secname, addr);
     String fullname = reg.getNewStud();

     StringTokenizer sttok = new StringTokenizer(fullname);
     sttok.nextToken();
     name = sttok.nextToken();
     secname = sttok.nextToken();
%>

  <html><head><title>Регистрация</title>
  </head><body>
  <h2 align=center>Дистанционная система обучения СДО</h2>
   Добро пожаловать, <%= name %>  <%= secname %>!<br>
   Вы зарегистрированы в СДО.<br>
   Выберите учебный курс:<br>

   <form method=post action=
     "http://homexp:8000/InfoAppl/servlet/CoursesServlet">
   <select size=5 name=courses>

<%
    Vector v = reg.getCourses();
    int n = v.size();

    for (int i = 0; i < n; i++){
%>
       <option> <%= v.get(i) %>
<% } %>

    </option></form></body></html>
```

```
<% }
  }catch(Exception e){
    System.err.println("From JSP: " + e);
  }
%>
```

Другой компонент EJB

Компоненты EJB очень часто обращаются друг к другу, один компонент служит клиентом другого компонента EJB. Если компонент служит удаленным клиентом, то в его построении и установке нет ничего нового по сравнению с клиентом-сервлетом.

В следующей главе мы освободим компонент `RegPrep` от несвойственной для session-компонента работы по обеспечению связи с источником данных и оформим его как клиента entity-компонента, которому передадим всю работу с источником данных.

Клиент, связанный с приложением J2EE

В листинге 22.7 приведено отдельное самостоятельное приложение Java, обращающееся как клиент к компоненту EJB по его JNDI-имени "`reg`". Есть другой вариант — создать клиента, который будет частью приложения J2EE, одним из его компонентов. Такой клиент будет работать в своем контейнере — контейнере клиента. Это дает возможность клиенту пользоваться всеми услугами J2EE-сервера, а не только одного компонента EJB.

Чтобы сделать клиентское приложение Java одним из компонентов приложения J2EE, надо установить его в контейнер подобно установке Web-компонентов и компонентов EJB. В утилите установки deploytool для этого выбирается пункт меню **New | Application Client** В процессе определения компонента указывается имя файла с главным классом клиентского приложения, например, RegPrepClient.class, и определяются ссылки на другие компоненты приложения, необходимые для работы клиента. В начале установки следует включить флажок **Return Client Jar**.

В результате установки создается JAR-архив с именем, повторяющим имя главного класса клиента, в нашем примере RegPrepClient.jar. Его вместе с файлом приложения RegPrep.ear надо скопировать на клиентскую машину. При запуске клиента следует загрузить библиотеки системы сообщений JMS и системы безопасности. Для облегчения этого процесса в состав J2EE SDK включена утилита runclient, которая используется примерно так:

```
runclient -client RegPrep.ear -name RegPrepClient
```

Открывается окно, в поля которого надо ввести имя, по умолчанию `guest`, и пароль, по умолчанию `guest123`.

Компоненты, управляемые сообщениями

Компоненты EJB типа MDB (Message Driven Beans) не поддерживают связь с клиентом. Они активизируются контейнером, как только поступит сообщение. Поэтому MDB-компоненты устроены наиболее просто. У них нет home- и remote-интерфейсов, поскольку они не работают под управлением системы RMI и клиент не вызывает их даже посредством контейнера.

С точки зрения клиента MDB-компонентов вообще не существует. Клиент взаимодействует со службой сообщений, посылая сообщения адресату (destination), которым может быть очередь сообщений типа `Queue` или раздел подписки типа `Topic`, как описано в *главе 16*.

Интерфейс, описывающий MDB-компоненты, т. е. интерфейс `MessageDrivenBean`, расширяющий интерфейс `EnterpriseBean`, предоставляет всего два метода — метод

```
public void setMessageDrivenContext(MessageDrivenContext ctx);
```

устанавливающий контекст компонента, и метод

```
public void ejbRemove();
```

удаляющий компонент.

Интерфейс `MessageDrivenContext` устроен еще проще — он ничего не добавляет к своему предку — интерфейсу `EJBContext`.

В настоящее время MDB-компоненты основаны на системе сообщений JMS. Поскольку каждый J2EE-сервер обязан иметь в своем составе службу сообщений JMS, реализовать MDB-компоненты очень легко.

На сервере уже созданы очереди сообщений типа `Queue` и разделы рассылки типа `Topic`. Клиент посылает сообщение не MDB-компоненту, а адресату в одну из очередей или в какой-нибудь раздел рассылки. С каждой очередью или разделом связан один MDB-компонент. Эта связь определяется при установке компонента в контейнер и записывается в DD-файл в элемент `<message-driven-destination>`. Тип адресата `Queue` или `Topic` указывается во вложенном элементе `<destination-type>`.

Компонент, связанный с разделом рассылки типа `Topic`, может быть определен как долговременный (durable) подписчик и получать сообщения, пришедшие адресату в то время, когда подписчик не был активен. Это определение выполняется вложенным элементом `<subscription-durability>`, имеющим значение `Durable`. Если этот элемент не определен, или его значение `NonDurable`, то MDB-компонент будет служить кратковременным подписчиком, т. е. не будет получать сообщения, пришедшие адресату в то время, когда подписчик не был активен.

Контейнер, получив сообщение, вызывает MDB-компонент подобно session-компоненту типа SLSB, без сохранения состояния. Это означает, что у MDB-компонента может и должен быть только один метод `ejbCreate()` без аргументов. Чаще всего контейнер создает пул MDB-компонентов и одновременно обрабатывает несколько сообщений, выбирая MDB-компоненты из пула. В этом большое преимущество MDB-компонентов перед обычными получателями системы сообщений JMS — создание пула MDB-компонентов, подпроцессов для их одновременной работы, слежение за ними — вся рутинная работа возлагается на контейнер.

Для того чтобы откликнуться на вызов контейнера, MDB-компонент должен стать слушателем системы сообщений JMS, т. е. реализовать интерфейс `MessageListener`, записав обработку сообщения в его метод `onMessage(Message)`. Контейнер, создав экземпляр MDB-компонента или взяв его из пула MDB-компонентов, обращается к этому методу и передает ему сообщение для обработки.

В дальнейшем фирма Sun предполагает внедрить кроме JMS и другие системы сообщений. Тогда MDB-компонент будет реализовывать не интерфейс `MessageListener`, а интерфейс нужной ему системы сообщений.

Итак, MDB-компонент:

- реализует интерфейсы `MessageDrivenBean` и `MessageListener`;
- должен содержать конструктор по умолчанию, хотя бы унаследованный от суперкласса;
- может реализовать метод `setMessageDrivenContext( MessageDrivenContext)` и пользоваться полученным контекстом;
- в единственном методе `ejbCreate()` без аргументов записывает начальные действия;
- в методе `onMessage(Message)` обрабатывает полученное сообщение, реализуя бизнес-логику компонента;
- в методе `ejbRemote()` записывает завершающие действия.

Следующий пример дается только для того, чтобы показать, насколько проще MDB-компонент отдельного подписчика системы сообщений JMS. В листинге 22.10 приведен MDB-компонент системы дистанционного обучения СДО, выполняющий те же действия, что и класс-подписчик системы сообщений JMS (см. листинг 16.2). Убедитесь, насколько проще стала программа.

Листинг 22.10. MDB-компонент, получающий сообщения

```
import javax.ejb.*;
import javax.jms.*;
import java.util.*;
```

```
public class InfoSubscriberMDB
   implements MessageDrivenBean, MessageListener{

  public void onMessage(Message message){

    try{
      System.out.println(message.getJMSMessageID());

      System.out.println("Идентификатор: " +
         message.getJMSMessageID() +
         " для адресата " + message.getJMSDestination());

      System.out.print("Срок хранения  ");

      if (message.getJMSExpiration() > 0)
          System.out.println(new Date(message.getJMSExpiration()));
      else
          System.out.println("не ограничен.");

      System.out.println("Приоритет: " + message.getJMSPriority());

      System.out.println("Режим: " + (
        message.getJMSDeliveryMode() == DeliveryMode.PERSISTENT ?
        "PERSISTENT" : "NON_PERSISTENT"));

      System.out.println("Ответ послать: " +
          message.getJMSReplyTo());

      System.out.println("Тип сообщения: " + message.getJMSType());

      if (message instanceof TextMessage){

        String messBody = ((TextMessage)message).getText();
        System.out.println("Тело сообщения: " + messBody);

      }else if (message instanceof ObjectMessage)
        System.out.println("Послан объект: " + message);
```

```
    }catch(Exception e){
      System.err.println(e);
    }
  }

  public void ejbCreate(){}
  public void setMessageDrivenContext(MessageDrivenContext mdc){}
  public void ejbRemove(){}
}
```

В классе `InfoSubscriberMDB` понадобилось реализовать только метод `onMessage()`. Все остальное, делавшееся "вручную" в классе `InfoSubscriber` (см. листинг 16.2), взял на себя контейнер.

Все сведения о разделе рассылки сообщений и фильтре сообщений определяются в процессе установки MDB-компонента в контейнер, некоторые из них записываются в DD-файл. Он приведен в листинге 22.11.

Листинг 22.11. DD-файл MDB-компонента

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE ejb-jar PUBLIC
   '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN'
   'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>
  <display-name>Ejb1</display-name>
  <enterprise-beans>
    <message-driven>
      <display-name>InfoSubscriberMDB</display-name>
      <ejb-name>InfoSubscriberMDB</ejb-name>
      <ejb-class>InfoSubscriberMDB</ejb-class>
      <transaction-type>Bean</transaction-type>
      <message-selector>
         JMSType = 'AdminInfo' AND Recipient ='Teacher'
      </message-selector>
      <acknowledge-mode>
         Auto-acknowledge
      </acknowledge-mode>
```

```
      <message-driven-destination>
        <destination-type>
             javax.jms.Topic
        </destination-type>
        <subscription-durability>
            Durable
        </subscription-durability>
      </message-driven-destination>
      <security-identity>
        <description></description>
        <run-as>
          <description></description>
          <role-name></role-name>
        </run-as>
      </security-identity>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

Создание раздела рассылки `InfoTopic` и запуск программы-отправителя `InfoPublisher` (см. листинг 16.1) выполняется точно так же, как описано в конце *главы 16*.

Приведем еще один более содержательный пример. MDB-компонент, описанный в листинге 22.12, получает сообщение, содержащее имя учебного курса системы дистанционного обучения СДО, находит в базе данных SDO описание курса с этим именем и отправляет его обратно.

Листинг 22.12. MDB-компонент, отправляющий ответ на сообщение

```
import java.sql.*;
import javax.naming.*;
import javax.ejb.*;
import javax.jms.*;

public class CoursesMDB
     implements MessageDrivenBean, MessageListener{

   private java.sql.Connection con;
   private MessageDrivenContext ejbContext;
   private InitialContext ctx;
```

```
  public void setMessageDrivenContext(MessageDrivenContext mdc){

   ejbContext = mdc;

   try{
     ctx = new InitialContext();

   }catch(NamingException ne){
     throw new EJBException(ne);
   }
 }

  public void ejbCreate(){

    try{
     Context env = (Context)ctx.lookup("java:comp/env");

     String driver = (String)env.lookup("driver");
     String url    = (String)env.lookup("url");
     String admin   = (String)env.lookup("admin");
     String password = (String)env.lookup("password");

     Class.forName(driver);

     con = DriverManager.getConnection(
           url, admin, password);

    }catch(Exception e){
      System.err.println(e);
    }
  }

  public void onMessage(Message message){

    try{
      String name = ((TextMessage)message).getText();

      PreparedStatement pst = con.prepareStatement(
         "SELECT description FROM courses WHERE course_name = ?");
```

```
      pst.setString(1, name);
      ResultSet rs = pst.executeQuery();

      rs.next();

      String descr = rs.getString(1);

      replyMess(message, descr);

      rs.close();

    }catch(Exception e){
      System.err.println(e);
    }
  }

  public void replyMess(Message msg, Object reply)
                    throws NamingException, JMSException{

    Queue queue = (Queue)msg.getJMSReplyTo();

    QueueConnectionFactory factory = (QueueConnectionFactory)
        ctx.lookup("java:comp/env/jms/QueueConnectionFactory");

    QueueConnection connect = factory.createQueueConnection();

    QueueSession session = connect.createQueueSession(true,0);

    QueueSender sender = session.createSender(queue);

    ObjectMessage message = session.createObjectMessage();
    message.setObject(reply);

    sender.send(message);

    connect.close();
  }
```

```
   public void ejbRemove(){

     try{
       con.close();
       ctx.close();
       ejbContext = null;

     }catch(Exception e){
       System.err.println(e);
     }
   }
}
```

Глава 23

Создание Entity Beans

Как уже было сказано в *главе 21*, entity-компоненты делают выборку из источника данных и представляют выбранную информацию в виде объектов Java, удобных для использования клиентом. В дальнейшем они поддерживают постоянную связь с источником данных, обновляя содержащуюся в нем информацию. Entity-компоненты могут существовать все время, пока существует источник данных. После краха J2EE-сервера entity-компоненты автоматически восстанавливаются с информацией, соответствующей последней завершенной транзакции.

В классическом случае entity-компонент связан с одной или несколькими таблицами базы данных, а каждый его экземпляр хранит одну строку выборки из этих таблиц.

У каждого entity-компонента должен быть так называемый *первичный ключ* (primary key), определяемый явно или неявно. Им может быть любой объект Java, удовлетворяющий требованиям RMI-IIOP. Чаще всего это уникальная для каждого entity-компонента строка символов класса `String` или целое значение класса `Integer` или `Long`, но первичным ключом может быть специально созданный класс. Значением первичного ключа entity-компонента часто служит первичный ключ таблицы, с которой связан компонент. Если первичный ключ таблицы состоит из одного атомарного значения, то первичным ключом entity-компонента будет один из стандартных классов Java, соответствующих типу атомарного значения. Если первичный ключ таблицы содержит несколько полей, то для первичного ключа entity-компонента надо создать класс, поля которого будут соответствовать полям первичного ключа таблицы.

Класс первичного ключа определяется при установке компонента в контейнер, а его конкретное значение — при заполнении компонента информацией из источника данных. Компонент может получить первичный ключ методом

```
public Object getPrimaryKey();
```

описанным в интерфейсе `EJBObject`.

Home-методы

В *главе 21* отмечалось, что home- и remote-интерфейсы для удаленного и локального клиента одинаковы для всех типов EJB-компонентов: Session Beans, Entity Beans и Message Driven Beans. Но home-интерфейс entity-компонента отличается тем, что в нем можно описать некоторые бизнес-методы компонента, кроме тех, что описаны в его remote-интерфейсе. Бизнес-методы, описанные в home-интерфейсе entity-компонента, называются в спецификации EJB *home-методами*. Каждый home-метод удаленного клиента, как и всякий удаленный метод системы RMI, должен выбрасывать исключение класса `RemoteException`. Итак, home-интерфейс entity-компонента в самом простом виде может выглядеть так, как показано в листинге 23.1.

Листинг 23.1. Home-интерфейс entity-компонента с бизнес-методом

```
import javax.ejb.*;
import java.rmi.*;

public interface SimpleEntityHome extends EJBHome{

   public int calculate() throws RemoteException;

   SimpleEntity findByPrimaryKey(Object primkey)
      throws FinderException, RemoteException;
}
```

Здесь описан только один home-метод `calculate()` и нет ни одного метода `create()`.

Home-метод, описанный в home-интерфейсе под именем `xxx()`, должен быть реализован в классе компонента с теми же аргументами и типом возвращаемого значения, но с именем `ejbHomeXxx()`. Он будет выполнен контейнером, когда удаленный клиент обратится к методу `xxx()`. Класс компонента, соответствующий home-интерфейсу листинга 23.1, может выглядеть так, как показано в листинге 23.2.

Листинг 23.2. Класс компонента, содержащий home-метод

```
import javax.ejb.*;

public class SimpleEntityBean implements EntityBean{
```

```
   private String info = "123456";

   public String getInfo(){
      return info;
   }

   public int ejbHomeCalculate(){
      return Integer.parseInt(info);
   }

   public void setEntityContext(EntityContext ctx){}
   public void unsetEntityContext(){}
   public void ejbLoad(){}
   public void ejbStore(){}
   public void ejbRemove(){}
   public void ejbActivate(){}
   public void ejbPassivate(){}
}
```

Home-методы аналогичны статическим методам обычных классов Java. Они тоже действуют сразу во всем entity-компоненте, а не в отдельном его экземпляре. Дело в том, что home-методы могут выполняться и в том случае, когда экземпляр entity-компонента находится в пассивном состоянии в пуле экземпляров. В этом состоянии экземпляр не связан с источником данных и не содержит данных, извлеченных из него. Поэтому в home-методах нельзя обращаться к переменным экземпляра, связанным с источником данных.

Интерфейс *EntityBean*

Для выполнения поставленных перед entity-компонентами задач интерфейс `EntityBean`, расширяющий интерфейс `EnterpriseBean`, описывает методы обновления и сохранения entity-компонента. Все методы интерфейса `EntityBean` перечислены в листинге 23.2, там сделана их пустая реализация.

Контейнер обращается к методу

```
public void ejbLoad();
```

интерфейса `EntityBean`, когда он обновляет содержимое entity-компонента новой выборкой из источника данных. Когда контейнер сохраняет содержимое entity-компонентов в источнике данных, он обращается к методу

```
public void ejbStore();
```

Перед использованием компонента, еще до его связи с источником данных, контейнер создает пул экземпляров entity-компонента. Экземпляры entity-компонента, хранятся в пуле "пустыми", они не содержат информации из источника данных. При выборе экземпляра компонента из пула контейнер выполняет метод `ejbLoad()`, заполняя выбранный экземпляр.

Сразу после выбора экземпляра из пула контейнер выполняет метод

```
public void ejbActivate();
```

в котором можно определить ресурсы, необходимые во время работы экземпляра, но ненужные в пассивном его состоянии.

Перед переводом экземпляра в пассивное состояние и сохранением его в пуле контейнер выполняет метод

```
public void ejbPassivate();
```

в котором можно освободить эти ресурсы.

Остальные методы интерфейса `EntityBean` традиционны для компонентов EJB.

Метод

```
public void ejbRemove();
```

выполняемый контейнером, отсоединяет экземпляр компонента от источника данных, очищает экземпляр, и возвращает его в пул экземпляров компонента.

Контейнер выполняет метод

```
public void setEntityContext(EntityContext ctx);
```

во время создания экземпляра entity-компонента, сразу после выполнения метода `newInstance()` класса `Class`. Этот метод устанавливает контекст компонента, который будет связан с компонентом в течении времени его жизни. В этом методе обычно сохраняется ссылка на контекст, которую будет использовать экземпляр, а также определяются ресурсы, связанные с самим entity-компонентом, а не с его экземпляром, например, соединение с базой данных. После выполнения метода `setEntityContext()` созданный экземпляр помещается в пул.

Метод

```
public void unsetEntityContext();
```

удаляет контекст из компонента, освобождая связанные с ним ресурсы, например, закрывая соединение с базой данных.

Методы создания экземпляра

Кроме перечисленных методов интерфейса `EntityBean`, которые должен реализовать каждый entity-компонент, в классе компонента могут присутствовать методы `ejbCreateXxx()`, по одному для каждого соответствующего метода `createXxx()`, описанного в home-интерфейсе компонента.

В отличие от session-компонента, наличие методов `createXxx()` в entity-компоненте необязательно. Если в классе entity-компонента нет ни одного метода `ejbCreateXxx()`, значит, в его home-интерфейсе нет методов `createXxx()`. Это, в свою очередь, означает, что клиент не может создать такой entity-компонент, а может обращаться только к home-методам entity-компонента, описанным в его home-интерфейсе. Это один из способов ограничить доступ клиента к источнику данных через компонент.

Возвращаемым типом методов `ejbCreateXxx()` должен быть первичный ключ компонента, который явно или неявно определяется в методе. При этом в BMP-компоненте возвращается значение первичного ключа, но в CMP-компоненте операндом операторов `return` должно служить значение `null`. Это странное правило позволяет BMP-компоненту быть наследником CMP-компонента.

Сразу же по завершении метода `ejbCreateXxx()` выполняется SQL-оператор `INSERT`, заносящий определенные в методе значения созданного entity-компонента в источник данных.

Для каждого метода `ejbCreateXxx()` в классе entity-компонента должен быть определен, хотя бы с пустым телом, метод с именем `ejbPostCreateXxx()`, который выполняется контейнером сразу же после соответствующего метода `ejbCreateXxx()`. У метода `ejbPostCreateXxx()` должны быть те же аргументы, что и у метода `ejbCreateXxx()`, но тип возвращаемого значения `void`. В этом методе уже известно значение первичного ключа компонента, его можно получить методом `getPrimaryKey()` и использовать для завершения инициализации компонента.

В листингах 23.3 и 23.4 дан пример описания и реализации методов `createXxx()`.

Листинг 23.3. Home-интерфейс с методами `createXxx()`

```
import java.rmi.*;
import javax.ejb.*;

public interface SomeHome extends EJBHome{

   Some create()
      throws RemoteException, CreateException;
```

```
   Some create(String name, long id)
      throws RemoteException, CreateException;

   Some create(long id)
      throws RemoteException, CreateException;

   Some createNew(String name, long id)
      throws RemoteException, CreateException;

   Some findByPrimaryKey(long primkey)
      throws FinderException, RemoteException;
}
```

Листинг 23.4. Методы `ejbCreateXxx()` в классе entity-компонента

```
import java.rmi.*;
import javax.ejb.*;

public class SomeBean implements EntityBean{

   public String name;
   public long id;
   public boolean news;

   public SomeBean(){}

   public Long ejbCreate()
                     throws CreateException{

      this.name = "No Name";  this.id = 0L;
      insertRow(name, id);
      return new Long(0);
      // В CMP-компоненте return null;
   }

   public Long ejbCreate(String name, long id)
                     throws CreateException{
```

```
    this.name = name;  this.id = id;
    insertRow(name, id);
    return new Long(this.id);
    // В CMP-компоненте return null;
}

public Long ejbCreate(long id)
                    throws CreateException{

    name = "No name";  this.id = id;
    insertRow(name, id);
    return new Long(this.id);
   // В CMP-компоненте return null;
}

public Long ejbCreateNew(String name, long id)
                    throws CreateException{

    news = true; this.name = name; this.id = id;
    insertRow(name, id);
    return new Long(this.id);
    // В CMP-компоненте return null;
}

Long ejbFindByPrimaryKey(long primkey)
                    throws FinderException{

    findRow(primkey);
    return new Long(primkey);
};

public void ejbPostCreate(String name, long id){}
public void ejbPostCreate(long id){}
public void ejbPostCreate(){}
public void ejbPostCreateNew(String name, long id){}

public void setEntityContext(EntityContext ctx){}
public void unsetEntityContext(){}
```

```
    public void ejbLoad(){}
    public void ejbStore(){}
    public void ejbRemove(){}
    public void ejbActivate(){}
    public void ejbPassivate(){}
}
```

В листинге 23.4 не показан метод `insertRow()`, заносящий в источник данных значения, определенные клиентом, и метод `findRow()`, отыскивающий строку.

Методы поиска

Еще одна группа методов, которые можно описать в home-интерфейсе entity-компонента, — это *методы поиска* (finder methods) с именами `findXxx()`. Эти методы позволяют отыскать в источнике данных информацию, которая будет занесена в компонент. Поиск ведется по признакам, определенным аргументами метода `findXxx()`. Результатом поиска будет один или несколько экземпляров компонента. Возвращаемым типом методов поиска должен служить тип remote-интерфейса компонента, если результат поиска однозначен, или коллекция ссылок на найденную информацию типа `Collection` или `Set`, если результат поиска неоднозначен.

Один из методов поиска — метод `findByPrimeryKey()` обязательно должен быть описан в каждом home-интерфейсе каждого entity-компонента. У него один аргумент — первичный ключ компонента, и одно возвращаемое значение (см. листинги 23.1 и 23.3).

В классе BMP-компонента методы поиска реализуются под именами `ejbFindXxx()` с теми же аргументами. Тип возвращаемого значения совпадает с первичным ключом компонента, если результат поиска однозначен, например, поиск ведется оператором `SELECT`, выбирающим одну строку таблицы. Если же результат поиска неоднозначен, например, выбирается несколько строк, то метод поиска возвращает коллекцию первичных ключей компонента.

В классе CMP-компонента методы поиска не только не реализуются, но даже не записываются. Они определяются во время установки CMP-компонента в контейнер с помощью языка EJB QL или другого языка запросов, "понимаемого" контейнером EJB.

В листинге 23.5 описано несколько методов поиска, а в листинге 23.6 — набросок их реализации в BMP-компоненте.

Листинг 23.5. Home-интерфейс с методами поиска

```
import java.util.*;
import java.rmi.*;
import javax.ejb.*;

public interface SomeHome extends EJBHome{

   Collection findByName(String name)
      throws FinderException, CreateException;

   Collection findById(long id)
      throws FinderException, CreateException;

   Some findByPrimaryKey(long primkey)
      throws FinderException, RemoteException;
}
```

Листинг 23.6. Методы поиска в классе entity-компонента

```
import java.util.*;
import java.rmi.*;
import javax.ejb.*;

public class SomeBean implements EntityBean{

   public String name;
   public long id;
   public boolean news;

   public SomeBean(){}

   Collection findByName(String name)
               throws FinderException{

      return findRows(name);
   };
```

```
   Collection findById(long id)
               throws FinderException{

      return findRows(id);
   }

   Long ejbFindByPrimaryKey(long primkey)
               throws FinderException{

      findRow(primkey);
      return new Long(primkey);
   };

   public void setEntityContext(EntityContext ctx){}
   public void unsetEntityContext(){}
   public void ejbLoad(){}
   public void ejbStore(){}
   public void ejbRemove(){}
   public void ejbActivate(){}
   public void ejbPassivate(){}
}
```

Состояния экземпляра entity-компонента

Итак, каждый экземпляр entity-компонента может находиться в двух состояниях: или в пуле экземпляров (pooled state) или быть готовым к выполнению бизнес-методов (ready state).

Контейнер создает экземпляр последовательным применением методов `newInstance()` и `setEntityContext()` и помещает его в пул экземпляров еще до обращения клиента к entity-компоненту. В этом состоянии выполняются home-методы, методы поиска `ejbFindXxx()`, в CMP-компонентах еще и методы выбора `ejbSelectXxx()`.

Перевод экземпляра в состояние готовности происходит при выполнении методов `ejbCreate()` и `ejbActivate()`, обратный переход выполняется методами `ejbRemove()` и `ejbPassivate()`.

В состоянии готовности выполняются бизнес-методы, методы `ejbLoad()`, `ejbStore()`, а в CMP-компонентах еще и методы `ejbSelectXxx()`.

Сохранность информации, содержащейся в entity-компоненте, может обеспечиваться контейнером, если это CMP-компонент, или самим entity-компонентом, если это BMP-компонент.

CMP-компонент

Entity-компонент, сохранность которого обеспечивается контейнером, с точки зрения производителя компонента устроен проще. Мы будем называть такой entity-компонент *CMP-компонентом* (Container Managed Persistence).

Те поля CMP-компонента, которые надо сохранять в источнике данных, называются *CMP-полями* (cmp-fields). Они не определяются в классе CMP-компонента, а указываются в DD-файле элементом `<cmp-field>`. В классе же CMP-компонента записываются только заголовки методов доступа к этим полям `getXxx()/setXxx()`, подобных методам доступа компонентов JavaBeans. Контейнер сам обращается к источнику данных и заполняет CMP-поля.

Поскольку тело методов доступа к CMP-полям не записывается, методы доступа к таким полям абстрактны и должны быть помечены `abstract`. Следовательно, и класс CMP-компонента, содержащий CMP-поля, становится абстрактным и должен быть помечен `abstract`. Реализация абстрактных методов доступа выполняется контейнером.

Приведем пример. Освободим session-компонент `RegPrepBean` (см. листинг 22.5), от несвойственной ему обязанности связываться с базой данных SDO и передадим эту обязанность двум CMP-компонентам: CMP-компоненту `StudentsCMP`, связанному с таблицей `students`, и CMP-компоненту `CoursesCMP`, связанному с таблицей `courses`. Облегченный session-компонент `RegPrepBean` будет их локальным клиентом.

Сначала мы запишем home- и remote-интерфейсы локального клиента CMP-компонента `StudentsCMP` в листинги 23.7 и 23.8.

Листинг 23.7. Home-интерфейс локального клиента регистрации

```
import javax.ejb.*;

public interface StudentsCMPHome extends EJBLocalHome{

   StudentsCMP create(String name, String addr)
      throws CreateException;

   StudentsCMP findByPrimaryKey(Object primkey)
      throws FinderException;
}
```

Листинг 23.8. Remote-интерфейс локального клиента регистрации

```
import javax.ejb.*;

public interface StudentsCMP extends EJBLocalObject{

   public String getNewStud();
}
```

Потом, в листинге 23.9, напишем класс CMP-компонента.

Листинг 23.9. Класс CMP-компонента регистрации ученика СДО

```
import javax.ejb.*;

public abstract class StudentsCMPBean implements EntityBean{

   // Методы доступа к cmp-полям.
   public abstract int getID(); // Первичный ключ.
   public abstract void setID(int id);

   public abstract String getName();
   public abstract void setName(String name);

   public abstract String getAddr();
   public abstract void setAddr(String addr);

   public Object ejbCreate(String name, String addr)
        throws CreateException{

      setName(name);
      setAddr(addr);

      return null;
   }

   public void ejbPostCreate(String name, String addr)
        throws CreateException{}

   public String getNewStud(){
```

```
      return getName();
   }

   public void setEntityContext(EntityContext ctx){}
   public void unsetEntityContext(){}
   public void ejbLoad(){}
   public void ejbStore(){}
   public void ejbRemove(){}
   public void ejbActivate(){}
   public void ejbPassivate(){}
}
```

Второй CMP-компонент `CoursesCMP` хранит и предоставляет клиенту с помощью метода `findAll()` список курсов, читаемых в системе дистанционного обучения СДО. В листингах 23.10—23.12 записаны его интерфейсы и класс.

Листинг 23.10. Home-интерфейс локального клиента списка курсов СДО

```
import javax.ejb.*;
import java.util.*;

public interface CoursesCMPHome extends EJBLocalHome{

   CoursesCMP create(String name, String descr)
      throws CreateException;

   Collection findAll() throws FinderException;

   CoursesCMP findByPrimaryKey(Object primkey)
      throws FinderException;
}
```

Листинг 23.11. Remote-интерфейс локального клиента списка курсов СДО

```
import javax.ejb.*;

public interface CoursesCMP extends EJBLocalObject{

   public String[] getCourses();
}
```

Листинг 23.12. Класс CMP-компонента списка курсов СДО

```
import javax.ejb.*;
import java.util.*;

public abstract class CoursesCMPBean implements EntityBean{

   // Методы доступа к cmp-полям.
   public abstract String getCourse_name();
   public abstract void setCourse_name(String course_name);

   public abstract String getDescription();
   public abstract void setDescription(String description);

   public Object ejbCreate(String name, String descr)
        throws CreateException{

      setCourse_name(name);
      setDescription(descr);

      return null;
   }

   public void ejbPostCreate(String name, String descr)
        throws CreateException{}

   public abstract Collection ejbSelectAllCourses()
      throws FinderException;


   public String[] getCourses(){

     String[] s = null;

     try{
        Collection c = ejbSelectAllCourses();
        Iterator it = c.iterator();

        int n = c.size();
        if (n == 0) return null;
```

```
        s = new String[n];

        for (int k = 0; k < n; k++)
           s[k] = (String)it.next();

     }catch(Exception e){
        System.err.println(e);
     }
     return s;
   };

   public void setEntityContext(EntityContext ctx){}
   public void unsetEntityContext(){}
   public void ejbLoad(){}
   public void ejbStore(){}
   public void ejbRemove(){}
   public void ejbActivate(){}
   public void ejbPassivate(){}
}
```

Локальным клиентом созданных CMP-компонентов теперь будет session-компонент (см. листинг 22.5), у которого обращения к базе данных заменены обращениями к методам CMP-компонентов `StudentsCMP` и `CoursesCMP`. Облегченный session-компонент `RegPrepBean` описан в листинге 23.13.

Листинг 23.13. Session-компонент — локальный клиент CMP-компонента

```
import javax.naming.*;
import javax.rmi.*;
import java.util.*;
import javax.ejb.*;

public class RegPrepBean implements SessionBean{

   private StudentsCMP reg;

   public void ejbCreate(String surname, String name,
                         String secname, String addr){
     try{
       InitialContext ctx = new InitialContext();
```

```
    Object ref = ctx.lookup(
      "java:comp/env/ejb/StudentsCMP");

    StudentsCMPHome home =
       (StudentsCMPHome)PortableRemoteObject.narrow(
          ref, StudentsCMP.class);

    reg = home.create(
       surname + " " + name + " " + secname, addr);

  }catch(Exception e){
    System.err.println(e);
  }
}

public String getNewStud(){

  String fullname = null;

  try{
     fullname = reg.getNewStud();

  }catch(Exception e){
    System.err.println(e);
  }
  return fullname;
}

public Vector getCourses(){

   Vector v = new Vector();

  try{
    InitialContext ctx = new InitialContext();

    Object ref = ctx.lookup(
      "java:comp/env/ejb/CoursesCMP");
```

```
        CoursesCMPHome home =
           (CoursesCMPHome)PortableRemoteObject.narrow(
              ref, CoursesCMP.class);

        Collection c = home.findAll();
        Iterator it = c.iterator();

        while (it.hasNext())
           v.addElement(it.next());

      }catch(Exception e){
        System.err.println(e);
      }
      return v;
    }

    public void setSessionContext(SessionContext ctx){}
    public void ejbPassivate(){}
    public void ejbActivate(){}
    public void ejbRemove(){}
}
```

Отношения, управляемые контейнером

Поскольку entity-компоненты часто представляют таблицы реляционных баз данных, между ними могут существовать отношения, соответствующие ER-отношениям (Entity-Relationship) между таблицами. Такова, например, связь между таблицами по первичному ключу одной таблицы и внешнему ключу другой таблицы.

Отношения между CMP-компонентами выражаются виртуальными полями, аналогичными CMP-полям. Они называются *отношениями, управляемыми контейнером* (Container Managed Relationship). Мы будем называть их *CMR-полями*. Как и CMP-поля, CMR-поля тоже указываются в DD-файле, но другим элементом — элементом `<cmr-field>`. В классе CMP-компонента для них также записываются только заголовки абстрактных методов доступа, ничем не отличающиеся от заголовков методов доступа к CMP-полям.

Как известно, ER-отношения между таблицами могут быть однонаправленными и двунаправленными. При реализации однонаправленного отношения абстрактные методы доступа записываются в одном из двух CMP-компонентов, участвующих в отношении, а именно, в том CMP-

компоненте, из которого исходит отношение. Двунаправленные отношения реализуются абстрактными методами поиска, записанными в оба CMP-компонента.

Таблицы, участвующие в ER-отношении, могут составлять отношения "один-к-одному", "один-к-многим", "многие-к-одному" и "многие-ко-многим". В абстрактных методах доступа сторона отношения "один" задается аргументом метода `setXxx()` или возвращаемым значением метода `getXxx()` в виде одного объекта, реализующего remote-интерфейс. Сторона отношения "многие" задается аргументом или возвращаемым значением типа `Collection`, если значения могут повторяться, или значением типа `Set`, если все значения различны.

Например, отношение "ученики изучают курсы" между таблицами `students` и `courses` базы данных SDO — это двунаправленное отношение "многие-ко-многим", реализованное в базе данных SDO таблицей `studied_courses`.

Пусть таблица `students` представлена CMP-компонентом `StudentsCMP`, (см. листинги 23.7—23.9), а таблица `courses` представлена CMP-компонентом `CoursesCMP` (см. листинги 23.10—23.12). Занесем в классы этих компонентов абстрактные методы доступа, реализующие отношение "ученики изучают курсы", как показано в листингах 23.14, 23.15.

Листинг 23.14. Абстрактные методы доступа к CMR-полям в `StudentsCMP`

```
import javax.ejb.*;
import java.util.*;

public abstract class StudentsCMPBean implements EntityBean{

   // Методы доступа к CMR-полям.
   public abstract Collection getCourses();
   public abstract void setCourses(Collection courses);

   public abstract String getDescription();
   public abstract void setDescription(String description);
}
```

Листинг 23.15. Абстрактные методы доступа к CMR-полям в `CoursesCMP`

```
import javax.ejb.*;
import java.util.*;

public class CoursesCMPBean implements EntityBean{
```

```
   // Методы доступа к CMR-полям.
   public abstract Collection getStudents();
   public abstract void setStudents(Collection students);
}
```

Методы выбора

Методы выбора — это частный случай методов поиска `ejbFindXxx()`, но у них есть некоторые отличия.

- Методы выбора определяются только в CMP-компонентах, но не в BMP-компонентах.
- Методы выбора не описываются в home-интерфейсах, следовательно, недоступны клиенту.
- Методы выбора описываются в классе CMP-компонента как абстрактные методы, т. е. без реализации и с пометкой `abstract`. Класс компонента, естественно, тоже становится абстрактным и помечается ключевым словом `abstract`.

Есть два вида методов выбора. Первый вид связан с самим CMP-компонентом, такие методы описываются именами `ejbSelectXxx()`. Второй вид связан с отдельным экземпляром CMP-компонента, имена таких методов `ejbSelectXxxInEntity()`. Например:

```
public abstract class SomeBean extends EntityBean{

   public abstract Collection ejbSelectAllNames(Date date)
      throws FinderException;

   public abstract Collection ejbSelectAllNamesInEntity(Date date)
      throws FinderException;
}
```

Методы выбора полезны для выбора значений CMP-полей и CMR-полей.

Методы выбора, так же как и методы поиска, определяются во время установки CMP-компонента в контейнер с помощью языка запросов EJB QL.

Язык запросов EJB QL

Как говорилось выше, методы поиска `findXxx()`, в том числе и обязательный метод `findByPrimaryKey()`, не реализуются в CMP-компонентах, а только описываются в home-интерфейсе. Они даже не упоминаются в классе CMP-компонента. Методы выбора `ejbSelectXxx()`, наоборот, описыва-

ются не в home-интерфейсе, а в классе CMP-компонента как абстрактные методы.

Вместо реализации методов поиска и выбора в классе компонента, во время установки CMP-компонента в контейнер записывается оператор запроса к источнику данных, который и будет определять условия поиска или выбора. Оператор запроса включается в DD-файл как элемент `<ejb-ql>`, относящийся к нему метод поиска или выбора — элемент `<query-method>`. Эти элементы вкладываются в элемент `<query>`.

Оператор запроса записывается на специально разработанном языке запросов EJB QL (EJB Query Language), независимом от диалекта языка SQL или другого языка, используемого источником данных. Полное описание языка EJB QL приведено в спецификации EJB. Некоторые контейнеры EJB используют другие языки запросов, определяющие методы поиска и выбора. Например, популярный сервер приложений WebLogic Server фирмы BEA использует свой собственный язык запросов WLQL.

Язык запросов EJB QL пока определяет всего один оператор `SELECT`, имеющий вид

```
SELECT [DISTINCT] OBJECT(var)
FROM AbstractSchemaName [AS] var, IN(var.cmrField) [AS] coll
[WHERE coll.cmpField = ?1]
```

Необязательные элементы записаны в квадратных скобках.

Как видно из этого схематичного описания, в операторе три части: `SELECT`, `FROM` и `WHERE`, причем последняя часть необязательна.

В части `SELECT` записывается необязательное слово `DISTINCT`, означающее, что в выборку войдет только одна из повторяющихся строк. Операция `OBJECT()` указывает выбираемый объект с помощью имени переменной, в примере это имя `var`. Вместо этой операции можно записать выражение, значением которого будет выбираемый объект. Объект должен иметь тип remote-интерфейса или `Collection`, как возвращаемый тип методов поиска в home-интерфейсе.

В части `FROM` через запятую перечисляются *элементы коллекции* (collection members), и/или *переменные-диапазоны* (range variable). Элементы коллекции используются в тех случаях, когда выборка делается из одного объекта, а переменные-диапазоны пробегают целую коллекцию объектов, из которой и будет сделана выборка.

Каждый элемент коллекции состоит из *имени абстрактной схемы* (abstract schema name) и имени переменной (identification variable). В примере, приведенном выше, элемент коллекции — это

```
AbstractSchemaName [AS] var
```

Здесь имя абстрактной схемы `AbstractSchemaName` чаще всего просто повторяет имя remote-интерфейса CMP-компонента. Оно определяется во время установки CMP-компонента в контейнер и записывается в DD-файл в элементе `<abstract-schema-name>`. Это имя занесено в переменную `var`.

Каждая переменная-диапазон состоит из операции `IN`, отношения, записанного в скобках и переменной, содержащей результат вычисления отношения в виде коллекции. Диапазон переменных в примере — это строка

```
IN(var.cmrField) [AS] coll
```

Он обращается к другому CMP-компоненту, связанному с компонентом, занесенным в переменную `var`, CMR-полем `cmrField`, которое устанавливает отношение "один-ко-многим". Результат этого отношения заносится в переменную `coll`. Более привычной была бы запись `col IN var.cmrField`, которая применяется в языке WLQL сервера приложений WebLogic.

Здесь применена интересная, но странная синтаксическая конструкция: запись `var.cmrField`, где `cmrField` — это CMR-поле, являющееся ссылкой на другой компонент EJB, — означает этот самый другой компонент. Мы можем обратиться к его CMP-полю `cmpField` так: `var.cmrField.cmpField`. Надо отдавать себе отчет в том, что первая точка в этой записи — это оператор перехода к другому компоненту, а вторая точка — просто уточнение имени поля именем экземпляра класса.

В необязательной части `WHERE` содержится условное выражение с трехзначной логикой, т. е. принимающее значения `TRUE`, `FALSE` или `UNKNOWN`, составленное по правилам, аналогичным правилам языка SQL в интерпретации JDBC. Это означает, что строки символов записываются в апострофах, а не в кавычках, неравенство записывается знаками "меньше" (`<`) и "больше" (`>`), логические операции записываются словами `NOT`, `AND` и `OR`, есть операции `[NOT] BETWEEN`, `IN`, `LIKE`, `IS [NOT] NULL`, `IS [NOT] EMPTY`, функции `ABS()`, `SQRT()`, `CONCAT()`, `SUBSTRING()`, `LOCATE()`, `LENGTH()`.

В условном выражении среди прочих переменных могут встречаться переменные с именами `?1`, `?2`, и т. д. Значениями этих переменных служат значения аргументов соответствующего метода `findXxx()`. Значение первого по порядку аргумента заносится в переменную `?1`, второго — в переменную `?2` и т. д.

Например, методу

```
public abstract Collection ejbSelectByRange(int from, int to)
   throws FinderException;
```

может соответствовать оператор запроса

```
SELECT OBJECT(p) FROM RegPrep p
WHERE p.id BETWEEN ?1 AND ?2
```

Как видно из этого краткого описания, язык запросов EJB QL находится в зачаточном состоянии, он, несомненно, будет развиваться и усложняться.

BMP-компонент

Возможность передать контейнеру все хлопоты по сохранению данных, обеспечиваемая CMP-компонентами, сильно облегчает работу производителя entity-компонентов. Кроме того, CMP-компоненты переносимы, их можно установить в любой контейнер EJB, задав при установке конкретные параметры источника данных и записав операторы запроса. При смене источника данных достаточно переустановить CMP-компонент, ничего не меняя в его коде.

Однако CMP-компоненты пока еще предоставляют не все средства, необходимые производителю entity-компонентов, да и реализация контейнера EJB часто оставляет желать лучшего. В тех случаях, когда производитель хочет добиться максимальной производительности, ему приходится самому обеспечивать сохранность данных, ориентируясь на конкретный источник данных. Вот тогда-то и приходится создавать BMP-компоненты.

В BMP-компонентах нет CMP- и CMR-полей, нет методов выбора `ejbSelectXxx()`, не используется язык запросов EJB QL, контейнер не определяет отношения между entity-компонентами. Класс BMP-компонента не может быть абстрактным.

В BMP-компонентах следует реализовать все методы, обращающиеся к источнику данных: `ejbCreateXxx()`, `ejbPostCreateXxx()`, `ejbLoad()`, `ejbStore()`, `ejbFindXxx()`, `ejbRemove()`, явно записав в них все операторы JNDI-поиска, соединения с источниками данных с помощью JDBC или других средств, извлечения и сохранения в источнике данных необходимой информации. Грубо говоря, в методах `ejbCreateXxx()` и `ejbPostCreateXxx()` следует выполнять SQL-операторы `INSERT`, в методах `ejbFindXxx()` и `ejbLoad()` — операторы `SELECT`, в методе `ejbStore()` — операторы `UPDATE`, в методе `ejbRemove()` — операторы `DELETE`.

Для лучшей переносимости можно создать специальные классы, которые будут содержать методы, специфичные для конкретного источника данных, а в BMP-компоненте только обращаться к методам этих классов. Такие классы называются DAO (Data Access Object).

Перепишем CMP-компоненты `StudentsCMP` и `CoursesCMP` (см. листинги 23.9 и 23.12) в виде BMP-компонентов, непосредственно обращающихся к базе данных SDO. Назовем их `StudentsBMP` и `CoursesBMP`. У них те же home- и remote-интерфейсы, а их классы приведены в листингах 23.16 и 23.17.

Листинг 23.16. Класс BMP-компонента регистрации ученика СДО

```
import java.sql.*;
import javax.naming.*;
import java.util.*;
import javax.ejb.*;

public class StudentsBMPBean implements EntityBean{

   private Connection con;
   private String driver;
   private String url;
   private String admin;
   private String password;

   // Сохраняемые поля
   private int id;
   private String name;
   private String addr;

   public EntityContext ctx;

   public void setEntityContext(EntityContext ctx){
      this.ctx = ctx;
   }

   public Integer ejbCreate(String name, String addr){

     this.id = 0;
     this.name = name;
     this.addr = addr;

     try{
       InitialContext ctx = new InitialContext();

       Context env = (Context)ctx.lookup("java:comp/env");

       driver   = (String)env.lookup("driver");
       url      = (String)env.lookup("url");
```

```
    admin    = (String)env.lookup("name");
    password = (String)env.lookup("password");

    Class.forName(driver);

     con = DriverManager.getConnection(
         url, admin, password);

    PreparedStatement pst = con.prepareStatement(
        "INSERT INTO students (id, name, address) " +
        "VALUES(reg_seq.NEXTVAL, ?, ?)");
    pst.setString(1, name);
    pst.setString(2, addr);

    int count = pst.executeUpdate();
    pst.close();

    pst = con.prepareStatement(
       "SELECT id FROM students ORDER BY id DESC");
    ResultSet rs = pst.executeQuery();
    rs.next();
    id = rs.getInt(1);

  }catch(Exception e){
    System.err.println(e);
  }
  return new Integer(id);
}

public void ejbPostCreate(String name, String addr){}

public String getNewStud(){

  String fullname = null;

  try{
    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();

    }catch(Exception e){
      System.err.println(e);
    }
    return fullname;
  }

  public void ejbLoad(){

   try{
      con = DriverManager.getConnection(
           driver, admin, password);

      PreparedStatement pst = con.prepareStatement(
        "SELECT id, name, address FROM students ORDER BY id DESC");

      ResultSet rs = pst.executeQuery();
      rs.next();
      id = rs.getInt(1);
      name = rs.getString(2);
      addr = rs.getString(3);

    }catch(Exception e){
      System.err.println(e);
    }
  }

  public void ejbStore(){

   try{
      con = DriverManager.getConnection(
           driver, admin, password);
```

```
    PreparedStatement pst = con.prepareStatement(
      "UPDATE students SET name = ?, address = ? WHERE id = ?");

    pst.setString(1, name);
    pst.setString(2, addr);
    pst.setInt(3, id);
    int n = pst.executeUpdate();

  }catch(Exception e){
    System.err.println(e);
  }
}

public Integer ejbFindByPrimaryKey(int primkey)
                          throws FinderException{

  try{
    PreparedStatement pst = con.prepareStatement(
       "SELECT id FROM students ORDER BY id DESC");

    ResultSet rs = pst.executeQuery();
    if (!rs.next()) throw new ObjectNotFoundException();

  }catch(Exception e){
    System.err.println(e);
  }

  return new Integer(primkey);
}

public void ejbPassivate(){
   con = null;
}

public void ejbActivate(){

  try{
     con = DriverManager.getConnection(
         driver, admin, password);
```

```
      }catch(Exception e){
        System.err.println(e);
      }
    }

    public void ejbRemove(){
      try{
        con.close();

      }catch(Exception e){
        System.err.println(e);
      }
    }

    public void unsetEntityContext(){}
}
```

Листинг 23.17. Класс BMP-компонента списка курсов СДО

```
import java.sql.*;
import javax.naming.*;
import java.util.*;
import javax.ejb.*;

public class CoursesBMPBean implements EntityBean{

    public EntityContext ctx;
    private String driver;
    private String url;
    private String admin;
    private String password;

    // Сохраняемые поля.
    private String course;
    private String description;
    private Connection con;

    public void setEntityContext(EntityContext ctx){
       this.ctx = ctx;
    }
```

```
public String ejbCreate(String name, String descr){

  course = name;
  description = descr;

  try{
    InitialContext ctx = new InitialContext();

    Context env = (Context)ctx.lookup("java:comp/env");

    driver   = (String)env.lookup("driver");
    url      = (String)env.lookup("url");
    admin    = (String)env.lookup("name");
    password = (String)env.lookup("password");

    Class.forName(driver);

    con = DriverManager.getConnection(
         url, admin, password);

    PreparedStatement st = con.prepareStatement(
       "INSERT INTO courses VALUES (?, ?)");

    st.setString(1, name);
    st.setString(2, descr);
    int n = st.executeUpdate();

  }catch(Exception e){
    System.err.println(e);
  }
  return name;
}

public void ejbPostCreate(String name, String addr){}

public String[] getCourses(){

  Vector courses = ejbFindAll();
```

```
    int n = courses.size();
    if (n == 0) return null;

    String[] s = new String[n];

    for (int k = 0; k < n; k++)
       s[k] = (String)courses.get(k);

    return s;
  }

  public Vector ejbFindAll(){

    Vector v = new Vector();

    try{
      Statement st = con.createStatement();

      ResultSet rs =
        st.executeQuery("SELECT course_name FROM courses");

      while (rs.next())
         v.add(rs.getString(1));

      rs.close(); st.close();

    }catch(Exception e){
      System.err.println(e);
    }
    return v;
  }

  public String ejbFindByPrimaryKey(String primkey)
                            throws FinderException{

    try{
      con = DriverManager.getConnection(
           driver, admin, password);
```

```
      PreparedStatement st =
         con.prepareStatement(
          "SELECT course_name, description" +
          " FROM courses WHERE courses_name = ?");
      st.setString(1, primkey);

      ResultSet rs = st.executeQuery();
      rs.next();

      rs.close(); st.close();

    }catch(Exception e){
      System.err.println(e);
    }
    return primkey;
  }

  public void ejbLoad(){

   try{
      con = DriverManager.getConnection(
           driver, admin, password);

      PreparedStatement st =
         con.prepareStatement(
          "SELECT course_name, description" +
          " FROM courses WHERE courses_name = ?");
      st.setString(1, course);

      ResultSet rs = st.executeQuery();
      rs.next();

      course = rs.getString("courses_name");
      description = rs.getString("desription");

      rs.close(); st.close();

    }catch(Exception e){
      System.err.println(e);
```

```
    }
  }

  public void ejbStore(){

    try{
      PreparedStatement st = con.prepareStatement(
        "UPDATE courses SET description = ?"
            + " where course_name = ?");

      st.setString(1, description);

      st.executeUpdate();

    }catch (Exception e){
      System.err.println(e);
    }
  }

  public void ejbPassivate(){
     con = null;
  }

  public void ejbActivate(){

    try{
       con = DriverManager.getConnection(
           driver, admin, password);

    }catch(Exception e){
      System.err.println(e);
    }
  }

  public void ejbRemove(){

    try{
      PreparedStatement st = con.prepareStatement(
       "DELETE FROM courses WHERE courses_name = ?");
```

```
        st.setString(1, course);
        int n = st.executeUpdate();

        con.close();

      }catch(Exception e){
        System.err.println(e);
      }
    }

    public void unsetEntityContext(){}
}
```

Глава 24

Транзакции

Как известно, *транзакцию* (transaction) составляет серия действий по обработке информации, которая должна быть проведена до конца или не выполняться вообще. Неполное выполнение всей серии приведет к невосполнимой потере информации. Классический пример — банковская транзакция по переводу денег с одного счета на другой. Если деньги будут сняты с одного счета, но не занесены на другой счет, они будут потеряны. Чтобы избежать подобных неприятностей, первоначальное состояние информации сохраняется во временном хранилище на время выполнения всей серии действий, т. е. транзакции. В случае неудачного выполнения какой-либо операции, входящей в транзакцию, просто восстанавливается начальное состояние информации, как говорят, происходит *откат* (rollback) транзакции. После этого транзакцию можно повторить. После успешного выполнения всех действий, входящих в транзакцию, происходит *фиксация* (commit) транзакции, при этом измененные во время транзакции данные заносятся в постоянное хранилище, а временное хранилище освобождается.

Таким образом, в серии действий, образующих транзакцию, должно быть помечено ее начало каким-нибудь методом `begin()`, который запоминает состояние информации на момент начала транзакции. Фиксация транзакции отмечается методом `commit()`, сохраняющим измененную транзакцией информацию в постоянном хранилище и освобождающим временные ресурсы. При неудаче вызывается метод `rollback()`, восстанавливающий состояние информации, сохраненное методом `begin()`.

Ясно, что транзакция не может быть короткой, она должна включать все необходимые действия. Она не должна быть очень длинной, вероятность краха длинной транзакции будет слишком велика. Каким же образом объединить несколько операций в транзакцию? Теория баз данных сформулировала ответ на этот вопрос в виде четырех правил, названных ACID (Atomicity, Consistency, Isolation, Durability).

- *Атомарность* (atomicity) транзакции означает, что должны быть выполнены или все операции, входящие в транзакцию, или ни одна из них. Транзакция — это одна неделимая операция, самая мелкая единица обработки данных.

- *Согласованность* (consistency) транзакции означает, что она переводит одно согласованное состояние информации в другое согласованное состояние, причем во время транзакции согласованность нарушается. Например, во время банковской операции переноса денег с одного счета на другой общая сумма денег, хранящихся в банке, уменьшается, пока деньги, снятые с одного счета, не будут зачислены на другой счет.
- *Изоляция* (isolation) одной транзакции от другой, параллельно выполняемой транзакции, позволяет одной транзакции увидеть только результат выполнения другой транзакции, но не промежуточное состояние изменяемой ею информации. Условие изоляции требует синхронизации работы одновременно выполняемых транзакций.
- *Достоверность* (durability) означает, что результат завершенной транзакции сохраняется даже после краха системы. Фиксация или откат должны быть проведены до конца или завершены в процессе восстановления системы.

Выполнение условий ACID гарантирует сохранность и согласованность информации после завершения каждой транзакции.

Транзакции приобретают особенное значение в распределенных системах, в которых несколько компонентов могут одновременно обратиться к одной и той же информации или, наоборот, один компонент может изменять информацию сразу в нескольких хранилищах данных. Такая транзакция называется *распределенной* (distributed transaction). Системы управления хранилищами данных, например, драйверы, файловые системы, СУБД, носят общее название *диспетчеры ресурсов* RM (Resource Managers). Разумеется, фиксация распределенной транзакции должна выполняться сразу всеми диспетчерами ресурсов, вовлеченных в нее. Для слежения за выполнением распределенной транзакции во всех участвующих в ней компонентах создается еще один программный компонент, называемый *диспетчером транзакции* TM (Transaction Manager). Диспетчер начинает транзакцию по запросу приложения, которое служит его клиентом. Диспетчер транзакции с помощью диспетчеров ресурсов регистрирует у себя состояние всех хранилищ данных и все его изменения. Диспетчер отдает команду на фиксацию распределенной транзакции.

Фиксация распределенной транзакции проходит две фазы (two-phase commit). Вначале клиент обращается к диспетчеру транзакции с запросом на фиксацию транзакции методом `commit()`. Тот осуществляет два действия.

1. Диспетчер транзакции дает указание всем диспетчерам ресурсов, участвующим в транзакции, подготовить (prepare) изменяемую информацию к сохранению в постоянном хранилище. Диспетчеры ресурсов выполняют это указание и докладывают диспетчеру транзакции о том, успешно ли прошел этот процесс.
2. Диспетчер транзакции просматривает доклады диспетчеров ресурсов. Если все диспетчеры ресурсов успешно сохранили информацию, то диспет-

чер транзакции делает у себя отметку о завершении транзакции и посылает диспетчерам ресурсов указание на сохранение измененных данных и освобождение временных хранилищ данных. Если хотя бы один диспетчер ресурсов доложил о том, что информацию сохранить не удалось, то диспетчер транзакции делает у себя отметку об откате транзакции и посылает диспетчерам ресурсов указание о восстановлении первоначальной информации.

Стандарты распределенных транзакций

Создание и выполнение распределенной транзакции требует точного согласования действий приложения, участвующего в транзакции, диспетчера транзакции и диспетчеров ресурсов, вовлеченных в транзакцию. Поскольку приложения и диспетчеры создаются разными независимыми фирмами, необходим стандарт, описывающий их взаимодействие.

Сообщество Open Group создало стандарт X/Open DTP (Distributed Transaction Processing), описывающий интерфейсы взаимодействия между участниками распределенной транзакции.

Взаимодействие между приложением и диспетчером транзакции определяется интерфейсом TX. В нем описываются методы открытия `tx_open()` и закрытия `tx_close()` диспетчера транзакций, начала `tx_begin()`, фиксации `tx_commit()` и отката `tx_rollback()` транзакции и другие методы, реализуемые диспетчером транзакций.

Взаимодействие диспетчера транзакции и диспетчеров ресурсов определяется интерфейсом XA. Диспетчеры ресурсов реализуют методы `xa_start()` вовлечения данных в транзакцию и `xa_end()` освобождения ресурсов, метод `xa_prepare()`, вызываемый диспетчером транзакции в первой фазе двухфазной фиксации, метод `xa_commit()`, вызываемый диспетчером транзакции во второй фазе двухфазной фиксации, метод отката `xa_recover()` и другие методы. Диспетчер транзакций реализует метод `ax_reg()` регистрации диспетчера ресурсов и метод `ax_unreg()` отмены регистрации.

Интерфейсы X/Open DTP реализованы во многих мониторах транзакций и системах промежуточного программного обеспечения. Технология Java предлагает свой вариант этих интерфейсов, названный JTA (Java Transaction Architecture). Он описан в интерфейсах, составляющих пакеты javax.transaction и javax.transaction.xa. Эти интерфейсы должен реализовать каждый J2EE-сервер.

Интерфейсы JTA объединены в несколько групп.

- Интерфейсы `Transaction`, `TransactionManager`, `Status` и `Synchronization` составляют группу управления транзакцией (JTA Transaction Management). Они реализуются и используются диспетчером транзакций.

- Интерфейс `UserTransaction` относится к группе прикладных интерфейсов (JTA Application Interfacing). Он реализуется диспетчером транзакции и используется приложением для работы с группой управления транзакцией.
- Интерфейсы `XAResource` и `Xid` относятся к группе управления ресурсами (JTA Resource Management). Они реализуются диспетчерами ресурсов, например, драйверами JDBC, и используются диспетчером транзакции для взаимодействия с диспетчерами ресурсов.

Интерфейсы JTA не реализуются напрямую. Их реализация основана на еще одном наборе интерфейсов, называемом JTS (Java Transaction Service). Дело в том, что система взаимодействия распределенных объектов CORBA расширила и детализировала интерфейсы X/Open DTP в своей службе транзакций OTS (Object Transaction Service). Поскольку технология Java взаимодействует с CORBA, она конкретизировала интерфейсы CORBA OTS в своем наборе интерфейсов JTS. Эти интерфейсы нижнего уровня собраны в пакеты org.omg.CosTransactions и org.omg.CosTSPortability, входящие в состав J2EE SDK. Они применяются для взаимодействия J2EE-серверов с CORBA, а, кроме того, с их помощью реализован диспетчер транзакции, входящий в состав J2EE-сервера.

Таким образом, интерфейсы JTA можно считать открытыми интерфейсами, предназначенными для использования клиентами диспетчера транзакции и взаимодействия диспетчера транзакции с диспетчерами ресурсов, а интерфейсы JTS — это закрытые, внутренние интерфейсы, предназначенные для реализации диспетчера транзакции в составе J2EE-сервера.

Интерфейсы управления транзакцией

Диспетчер транзакции реализует интерфейсы `Transaction`, `TransactionManager`, `Status` и `Synchronization`.

Сама транзакция, разумеется, описывается интерфейсом `Transaction`. Объект, реализующий этот интерфейс, создается во время создания транзакции и хранит сведения о ней, образуя контекст транзакции. Интерфейс описывает методы фиксации `commit()` и отката `rollback()` транзакции, метод `registerSynchronization(Synchronization)`, регистрации синхронизирующего объекта. Метод `setRollbackOnly()` задает транзакции режим, при котором транзакцию можно только откатить, но не фиксировать. Метод

```
public int getStatus();
```

показывает текущее состояние транзакции в виде одной из констант, описанных в интерфейсе `Status`.

- `STATUS_ACTIVE` — транзакция выполняется.
- `STATUS_COMMITED` — транзакция фиксирована.

- `STATUS_COMMITING` — транзакция фиксируется.
- `STATUS_MARKED_ROLLBACK` — транзакция будет отменена, вероятно, в результате действия метода `setRollbackOnly()`.
- `STATUS_NO_TRANSACTION` — объект не участвует в транзакции.
- `STATUS_PREPARED` — закончена первая фаза фиксации транзакции.
- `STATUS_PREPARING` — транзакция проходит первую фазу фиксации.
- `STATUS_ROLLEDBACK` — выполнен откат транзакции.
- `STATUS_ROLLING_BACK` — транзакция в состоянии отката.
- `STATUS_UNKNOWN` — состояние транзакции неизвестно.

Наконец, к транзакции можно приписать предварительно созданный объект связи с диспетчером ресурсов `res` методом

```
public boolean enlistResource(XAResource res);
```

или убрать этот объект методом

```
public boolean delistResource(XAResource res, int flag);
```

Аргумент `flag` принимает одно из трех значений `TMSUCCESS`, `TMSUSPEND` или `TMFAIL`. Это константы, описанные в интерфейсе `XAResource`. Методы возвращают `true`, если операция прошла успешно, и `false` в противном случае.

Управление транзакцией со стороны диспетчера транзакции описано методами интерфейса `TransactionManager`. Метод `begin()` создает и начинает транзакцию. Методы `commit()` и `rollback()` управляют ее завершением. Метод `getStatus()` информирует о состоянии транзакции точно так же, как аналогичный метод интерфейса `Transaction`. Метод

```
public void Transaction getTransaction();
```

предоставляет ссылку на объект-транзакцию. Такой ссылкой можно воспользоваться в методе `resume(Transaction)` для перевода подпроцесса в другую транзакцию.

Метод

```
public Transaction suspend();
```

приостанавливает транзакцию, а метод

```
public void setTransactionTimeout(int seconds);
```

изменяет максимальное время выполнения транзакции, установленное по умолчанию или предыдущим обращением к этому методу. Нулевой аргумент метода означает возврат ко времени по умолчанию. По истечении установленного времени транзакция откатывается.

Время выполнения транзакции по умолчанию устанавливается обычно в конфигурационном файле J2EE-сервера. Для сервера, входящего в поставку

J2EE SDK, это значение параметра `transaction.timeout`, записанного в файле $J2EE_HOME/config/default.properties.

Нулевое значение этого параметра означает неограниченное время выполнения транзакции.

Диспетчер транзакции может уведомлять всех заинтересованных клиентов о завершении транзакции, обращаясь для этого к двум методам, описанном в интерфейсе `Synchronization`. Это метод

```
public void beforeCompletion();
```

выполняемый перед завершением транзакции, и метод

```
public void afterCompletion(int status);
```

выполняемый сразу же после завершения транзакции ее фиксацией или откатом. Аргумент `status` дает код завершения транзакции, равный одной из констант интерфейса `Status`, описанных выше.

Прикладной интерфейс

Интерфейс `UserTransaction` реализуется диспетчером транзакции. Диспетчер создает объект, реализующий этот интерфейс, и передает ссылку на него клиенту. Клиент обращается к его методам для управления транзакцией. Методы интерфейса `UserTransaction` просто дублируют одноименные методы `begin()`, `commit()`, `rollback()`, `setRollbackOnly()`, `getStatus()` и `setTransactionTimeout(int)` интерфейса `TransactionManager`.

Для того чтобы клиент мог получить ссылку на объект, реализующий интерфейс `UserTransaction`, J2EE-сервер создает JNDI-имя `java:comp/UserTransaction`. Таким образом, клиент может управлять транзакцией примерно так:

```
Context ctx = new InitialContext();

UserTransaction utx =
   (UserTransaction)ctx.lookup("java:comp/UserTransaction");

utx.begin();

// Выполняем транзакцию.

if (/* Все в порядке */) utx.commit();
else utx.rollback();
```

У компонентов EJB есть еще одна возможность обратиться к объекту типа `UserTransaction` — использовать метод `getUserTransaction()` интерфейса `EJBContext`.

Интерфейсы управления ресурсами

Два интерфейса управления ресурсами: `XAResource` и `Xid` из пакета javax.transaction.xa реализуются диспетчерами ресурсов. Их реализуют, например, драйверы JDBC и поставщики услуг системы сообщений JMS. Методы этих интерфейсов предоставляются диспетчеру транзакции для управления ресурсами и оповещения диспетчеров ресурсов об изменении состояния транзакции. Эти интерфейсы недоступны клиенту диспетчера транзакции, но могут использоваться клиентами диспетчеров ресурсов, например, клиент JDBC может получить ссылку на объект, реализующий интерфейс `XAResource`, методом `getXAResource()` интерфейса `XAConnection`. Клиент системы сообщений JMS может получить ссылку на объект типа `XAResource` методом `getXAResource()` интерфейса `XASession`.

По стандарту X/Open DTP каждая транзакция снабжается идентификатором XID, состоящим из трех частей: глобального идентификатора, формата и идентификатора той части распределенной транзакции, которая связана с данным диспетчером ресурсов. Интерфейс `Xid` описывает методы получения всех трех частей идентификатора:

```
public byte[] getGlobalTransactionId();
public int getFormatId();
public byte[] getBranchQualifier();
```

Максимальное число байтов в возвращаемых массивах задается константами `MAXGTRIDSIZE` и `MAXBQUALSIZE`.

Интерфейс `XAResource` описывает связь между диспетчером транзакции и диспетчером ресурса. Его реализует диспетчер ресурса и предоставляет методы интерфейса диспетчеру транзакции.

Интерфейс `XAResource` использует объект типа `Xid` в своих методах для идентификации транзакции.

Метод

```
public void start(Xid id, int flag);
```

подсоединяет данный ресурс к транзакции с идентификатором `id`. Аргумент `flag` принимает одно из трех значений, выраженных следующими константами:

- `TMNOFLAGS` — флаги отсутствуют;
- `TMJOIN` — подсоединить ресурс к существующей транзакции;
- `TMRESUME` — восстановить связь с приостановленной транзакцией.

Метод

```
public void end(Xid id, int flag);
```

отсоединяет ресурс. Аргумент `flag` принимает одно из трех значений, выраженных следующими константами:

- `TMSUCCESS` — отсоединить ресурс от транзакции;
- `TMFAIL` — отсоединить ресурс и подготовить транзакцию к откату;
- `TMSUSPEND` — приостановить связь с транзакцией.

Метод

```
public void commit(Xid id, boolean onePhase);
```

оповещает диспетчер ресурсов о завершении транзакции с идентификатором `id`. Если аргумент `onePhase` равен `true`, то завершение выполняется в одну фазу, в противном случае произойдет двухфазное завершение.

Метод

```
public int prepare(Xid id);
```

сообщает диспетчеру ресурсов о наступлении первой фазы завершения транзакции. Метод возвращает одно из двух значений:

- `XA_RDONLY` — транзакция не меняла данные и может быть зафиксирована;
- `XA_OK` — транзакция может быть зафиксирована.

Если диспетчер ресурсов считает, что транзакция должна быть отменена, то метод должен выбросить исключение класса `XAException`.

Метод

```
public void rollback(Xid id);
```

оповещает диспетчер ресурсов об откате транзакции с идентификатором `id`.

Метод

```
public Xid[] recover(int flag);
```

запрашивает список ресурсов, необходимый для отката транзакции. Аргумент `flag` принимает значение `TMNOFLAGS` или:

- `TMSTARTRSCAN` — начать подготовку к откату;
- `TMENDRSCAN` — завершить подготовку к откату.

Метод

```
public boolean setTransactionTimeout(int seconds);
```

устанавливает время выполнения транзакции для данного ресурса. Нулевое значение аргумента восстанавливает время по умолчанию. Метод возвращает `true`, если операция прошла удачно, `false` в противном случае.

Метод

```
public int getTransactionTimeout();
```

информирует о времени выполнения транзакции, установленном для данного ресурса.

Наконец, метод

```
public boolean isSameRM(XAResource res);
```

сравнивает данный объект с объектом `res`. Этот метод используется вместо обычного метода `equals()`.

Транзакции в апплетах и приложениях

Спецификация JTA не допускает управления транзакциями в апплетах и отдельных приложениях Java. Всякая попытка воспользоваться в них методами интерфейсов JTA или интерфейсов JTS вызовет сообщение об ошибке.

Транзакции в Web-компонентах

Контекст транзакции, в которой участвует сервлет, устанавливается уже после его инициализации, т. е. после выполнения метода `init()`. Поэтому транзакция должна начинаться и заканчиваться в методе `service()`. Всякая транзакция, не завершенная к моменту окончания метода `service()`, будет отменена.

Поскольку страницы JSP компилируются в сервлеты, для них действует то же правило: методы управления транзакцией надо применять так, чтобы все они выполнялись из метода `service()` того сервлета, в который будет откомпилирована страница JSP.

Прежде чем привести пример, следует напомнить, что клиенты JDBC и системы сообщений JMS могут управлять транзакцией в источнике данных с помощью методов `commit()` и `rollback()`, описанных в интерфейсах `Connection` и `Session` соответственно. Эти методы ни в коем случае нельзя использовать вместе с одноименными методами интерфейсов JTA. Методы JTA следует применять в *распределенных* транзакциях, например, когда идет одновременная обработка информации в нескольких базах данных.

Приведем пример. Возьмем сервлет регистрации ученика СДО из листинга 20.5 и организуем в нем транзакцию, заносящую в базу данных SDO сведения о новом ученике, чтобы потом извлечь сведения уже из измененной базы. В листинге 24.1 показана только часть кода, связанная с транзакцией.

Листинг 24.1. Транзакция в сервлете

```
import java.io.*;
import java.sql.*;
import java.util.*;
import javax.servlet.http.*;
import javax.naming.*;
import javax.transaction.*;

public class RegPrepServlet extends HttpServlet{

  public void doPost(HttpServletRequest req, HttpServletResponse resp){

    try{
       // Начало метода…

       Connection con = DriverManager.getConnection(
           url, user, password);

       InitialContext ctx = new InitialContext();

       UserTransaction ut = (UserTransaction)
            ctx.lookup("java:comp/UserTransaction");

       ut.begin();

       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();

       ut.commit();

       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);

        // Продолжение метода...

        rs.close();

        con.close();

      }catch(Exception e){
         System.err.println(e);
      }
   }
}
```

В распределенном приложении Web-компоненты играют представительскую роль, осуществляя связь с клиентом. Такая роль обычно не связана с транзакциями. Транзакции важны в бизнес-логике приложения, они чаще всего связаны с хранилищами данных и больше подходят для EJB-компонентов.

Транзакции в EJB-компонентах

Поскольку EJB-компоненты выполняются в контейнере, они могут передать ему управление транзакцией (container-managed transaction demarcation). Для entity-компонентов это обязательное правило, они не могут самостоятельно управлять транзакцией. Session- и MDB-компоненты могут выбирать — или предоставить управление транзакцией контейнеру, или получить ссылку на объект типа `UserTransaction` и воспользоваться его методами `begin()`, `commit()`, `rollback()` (bean-managed transaction demarcation). Выбранный вариант сообщается контейнеру при установке session- или MDB-компонента в контейнер. Его значения, `"Bean"` или `"Container"`, записываются в DD-файл в элемент `<transaction-type>`.

Транзакции, управляемые компонентом

Повторим еще раз, что в entity-компонентах нельзя использовать интерфейсы JTA. Их можно применять только в session- и MDB-компонентах.

J2EE-сервер не допускает вложенные транзакции. Прежде чем начать новую транзакцию, следует завершить предыдущую.

Если решено управлять транзакцией в session-компоненте, то следует учитывать, что транзакция в SFSB-компоненте должна начинаться и заканчиваться в пределах одного бизнес-метода. В листинге 24.2 приведена схема такого метода. Если транзакция начинается, но не заканчивается до выхода из метода, то контейнер откатывает транзакцию и выбрасывает удаленному клиенту исключение класса `RemoteException`, а локальному клиенту — исключение класса `EJBException`.

Листинг 24.2. Транзакция в бизнес-методе SLSB-компонента

```
import javax.transaction.*;
import java.sql.*;
import javax.sql.*;
import javax.ejb.*;
import javax.naming.*;

public class SomeStatelessBean implements SessionBean{

   SessionContext ejbCtx;

   public void setSessionContext(SessionContext ctx){

      ejbCtx = ctx;
   }

   public void someMethod(...){

      InitialContext ctx = new InitialContext();

      DataSource ds1 = (DataSource)
         ctx.lookup("java:comp/env/jdbc/Database1");

      Connection con1 = ds1.getConnection();
      Statement stmt1 = con1.createStatement();

      DataSource ds2 = (DataSource)
         ctx.lookup("java:comp/env/jdbc/Database2");

      Connection con2 = ds2.getConnection();
      Statement stmt2 = con2.createStatement();
```

```
        UserTransaction ut = ejbCtx.getUserTransaction();

        ut.begin();

        // Какие-то действия с источниками данных
        stmt1.executeQuery(...);
        stmt1.executeUpdate(...);
        stmt2.executeQuery(...);
        stmt2.executeUpdate(...);
        stmt1.executeUpdate(...);
        stmt2.executeUpdate(...);

        ut.commit();

        stmt1.close(); stmt2.close();
        con1.close(); con2.close();
    }
    // Другие методы…
}
```

Транзакция в SFSB-компоненте может начинаться в одном методе и заканчиваться в другом, но обязательно перед выходом из компонента. В листинге 24.3 приведена схема управления транзакцией в SFSB-компоненте. Транзакция начинается в методе `method1()`, а заканчивается в методе `method3()`.

Листинг 24.3. Транзакция в SFSB-компоненте

```
import javax.transaction.*;
import java.sql.*;
import javax.sql.*;
import javax.ejb.*;
import javax.naming.*;

public class SomeStatefulBean implements SessionBean{

    SessionContext ejbCtx;
    Connection con1, con2;
    UserTransaction ut;

    public void setSessionContext(SessionContext ctx){
```

```
    ejbCtx = ctx;
  }

  public void method1(...){

    InitialContext ctx = new InitialContext();

    ut = ejbCtx.getUserTransaction();

    ut.begin();

    DataSource ds1 = (DataSource)
       ctx.lookup("java:comp/env/jdbc/Database1");

    con1 = ds1.getConnection();
    Statement stmt = con1.createStatement();

    stmt.executeUpdate(...);
    stmt.executeUpdate(...);
  }

  public void method2(...){

    InitialContext ctx = new InitialContext();

    DataSource ds2 = (DataSource)
       ctx.lookup("java:comp/env/jdbc/Database2");

    con2 = ds2.getConnection();
    Statement stmt = con2.createStatement();

    stmt.executeUpdate(...);
    stmt.executeUpdate(...);
  }

  public void method3(...){

    Statement stmt = con1.createStatement();
    stmt.executeUpdate(...);
```

```
        stmt = con2.createStatement();
        stmt.executeUpdate(...);

        ut = ejbCtx.getUserTransaction();

        ut.commit();

        stmt.close(); con1.close(); con2.close();
    }

    // Прочие методы…
}
```

Транзакция в MDB-компоненте не должна выходить за пределы метода `onMessage()`. Если транзакция начата, но не закончена в методе `onMessage()`, то контейнер откатывает транзакцию и удаляет экземпляр MDB-компонента.

В листинге 24.4 приведен MDB-компонент, регистрирующий постояльца гостиницы и обращающийся для этого к двум entity-компонентам. Все действия объединены в одну транзакцию.

Листинг 24.4. Транзакция в MDB-компоненте

```
import javax.jms.*;
import java.util.*;
import javax.ejb.*;
import javax.naming.*;
import javax.transaction.*;

public class ReservationBean
       implements MessageDrivenBean, MessageListener{

  MessageDrivenContext ejbCtx;
  Context jndiContext;

  public void setMessageDrivenContext(MessageDrivenContext mdc){

     ejbCtx = mdc;
```

```
    try{
       jndiContext = new InitialContext();

    }catch(NamingException ne){
       System.err.println(ne);
    }
 }

 public void ejbCreate(){}

 public void onMessage(Message message){

   try{
     UserTransaction ut = ejbCtx.getUserTransaction();

     ut.begin();

     ReservationHome resHome = (ReservationHome)
       jndiContext.lookup("java:comp/env/ejb/Reservation");

     Reservation res =
      resHome.create(customer, cruise, cabin, price, new Date());

     Object ref = jndiContext.lookup
               ("java:comp/env/ejb/Payment");

     PaymentHome pHome = (PaymentHome)
        PortableRemoteObject.narrow(ref, PaymentHome.class);

     Payment proc = pHome.create();
     proc.byCredit(customer, card, price);

    ut.commit();

   }catch(Exception e){
     System.err.println(e);
   }
 }
```

```
  public void ejbRemove(){

    try{
       jndiContext.close();
       ejbCtx = null;

    }catch(NamingException ne){
        System.err.println(ne);
    }
  }
}
```

Транзакции, управляемые контейнером

Компоненты EJB любых типов могут передать управление транзакцией своему контейнеру, причем для entity-компонентов это единственный способ. При этом в текстах файлов компонента нигде не используются интерфейсы JTA. При установке компонента в контейнер указывается тип управления транзакцией "Container-managed transaction" и могут быть заданы атрибуты транзакции, содержащие пожелания производителя компонента контейнеру.

Контейнер включает в транзакцию весь метод целиком, он не может выделить для формирования транзакции какие-то участки кода из метода. Поэтому атрибуты транзакции задаются для каждого метода отдельно. Session-компоненты требуют атрибутов только для бизнес-методов, entity-компоненты — еще и для методов `createXxx()`, `remove()` и методов поиска, MDB-компоненты — для метода `onMessage()`. Для каждого из таких методов при установке компонента в контейнер определяется атрибут транзакции, который записывается в DD-файл в элементе `<trans-attribute>`.

Всего имеется шесть атрибутов транзакции: `NonSupported`, `Required`, `Supports`, `RequiresNew`, `Mandatory` и `Never`. В MDB-компонентах применяются только атрибуты `NonSupported` и `Required`. Для session- и entity-компонентов допустимы атрибуты `Required`, `RequiresNew` и `Mandatory`. Остальные атрибуты необязательны и могут не учитываться контейнером.

Дадим краткое описание каждого атрибута.

- `NonSupported`. Метод выполняется без транзакции. Если клиент уже начал транзакцию до обращения к методу, то она приостанавливается, а после выполнения метода возобновляется. Этот атрибут удобен в работе с диспетчером ресурсов, который не обеспечивает транзакции или не может управляться J2EE-сервером.

- `Required`. Метод выполняется в транзакции, уже производящейся клиентом. Если клиент не начинал транзакцию, то контейнер начинает новую транзакцию для выполнения метода. Новая транзакция всегда начинается для метода `onMessage()` MDB-компонента, у которого нет клиента. По окончании метода контейнер пытается завершить транзакцию. Если транзакция завершилась откатом, то контейнер выбрасывает клиенту исключение класса `RollbackException`.
- `Supports`. Метод выполняется в транзакции, уже осуществляющейся клиентом, или выполняется без транзакции, если клиент еще не начал транзакцию. Этот атрибут требует осторожности — ведь метод должен выполняться и с атрибутом `NonSupported` и с атрибутом `Required`. Лучше всего, если код метода не будет зависеть от способа управления транзакцией.
- `RequiresNew`. Для выполнения метода контейнер начинает новую транзакцию, приостановив транзакцию клиента, если она была начата. По окончании метода контейнер пытается зафиксировать транзакцию. Этот атрибут удобен для методов, выполняющих отдельную работу, которая должна быть выполнена, даже если транзакция будет отменена, например, запись в журнал.
- `Mandatory`. Метод выполняется только в рамках транзакции клиента. Если клиент не начинал транзакцию, то удаленному клиенту выбрасывается исключение класса `TransactionRequiredException`, а локальному клиенту — исключение класса `TransactionRequiredLocalException`. Этот атрибут применяется в тех случаях, когда важно, чтобы метод выполнялся в рамках транзакции клиента.
- `Never`. Метод выполняется только без транзакции. Если клиент уже начал транзакцию, то удаленному клиенту выбрасывается исключение класса `RemoteException`, а локальному клиенту — исключение класса `EJBException`. Этот атрибут применяется в тех случаях, когда метод не приемлет никакой транзакции.

Документация, описывающая транзакции в технологии Java, дает краткую рекомендацию — при всякой возможности применять транзакции, управляемые контейнером, а в этих транзакциях устанавливать атрибут `Required`, если метод не требует определенного атрибута.

Глава 25

Технология Java и язык XML

На протяжении всей книги мы встречались с языком разметки XML (eXtensible Markup Language). Он широко применяется в технологии Java для записи конфигурационных файлов, но его можно использовать и для хранения и передачи информации через WWW.

Язык XML возник как расширение гипертекстового языка разметки HTML (HyperText Markup Language). Набор тегов языка HTML строго фиксирован, он не расширяем и должен одинаково пониматься всеми браузерами. Ограниченные возможности языка HTML быстро перестали удовлетворять разработчиков. Введение таблиц стилей CSS (Cascading Style Sheet) и включений на стороне сервера SSI (Server Side Include) лишь ненадолго уменьшило их недовольство.

Выход есть. Еще в 1986 году стал стандартом язык создания языков разметки SGML (Standard Generalized Markup Language), с помощью которого и был создан язык HTML. Язык SGML определяет структурную, а не визуальную разметку документа. Он показывает с помощью тегов, как документ разбивается на главы, параграфы и абзацы, но ничего не говорит о том, как они будут отображаться на листе бумаги или на экране дисплея. Каждый конкретный набор тегов, созданный по правилам SGML, снабжается описанием DTD (Document Type Definition), разъясняющим связь тегов между собой и правила их применения. Специальная программа — драйвер принтера или SGML-браузер — руководствуется этим описанием при печати или отображении документа на экране дисплея.

Язык SGML оказался слишком сложным, требующим тщательного и объемистого описания тегов. Он применяется только в крупных проектах, например, для создания единой системы документооборота крупной фирмы.

Язык XML оказался золотой серединой между языками SGML и HTML. Он достаточно широк, чтобы можно было создать все нужные теги, и достаточно прост, чтобы можно было быстро их описать.

Создавая описание документа на языке XML надо, прежде всего, продумать его структуру. Например, мы хотим описать учебные курсы системы дистанционного обучения СДО.

В описание входят название курса и его темы. Для каждой темы указывается ее номер и название:

```
Кулинария
1. Варка.
2. Соление
Фотография
1. Экспозиция
2. Проявка
3. Печать
```

Затем мы составляем описание учебных курсов на языке XML, как показано в листинге 25.1.

Листинг 25.1. Пример XML-документа

```
<?xml version="1.0" encoding="Windows-1251"?>
<!DOCTYPE mydocs SYSTEM "courses.dtd">
<courses-list>
   <course>
      <course-name>Кулинария</course-name>
      <topic>
         <topic-number>1 </topic-number>
         <topic-name>Варка</topic-name>
      </topic>
      <topic>
         <topic-number>2 </topic-number>
         <topic-name>Соление</topic-name>
      </topic>
   </course>
   <course>
      <course-name>Фотография</course-name>
      <topic>
         <topic-number>1 </topic-number>
         <topic-name>Экспозиция</topic-name>
     </topic>
      <topic>
         <topic-number>2 </topic-number>
         <topic-name>Проявка</topic-name>
      </topic>
```

```
      <topic>
         <topic-number>3 </topic-number>
         <topic-name>Печать</topic-name>
      </topic>
   </course>
</courses-list>
```

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 "/>".

Для описания нам понадобились открывающие теги <courses-list>, <course>, <course-name>, <topic>, <topic-number>, <topic-name> и соответствующие им закрывающие теги, помеченные наклонной чертой. Теперь необходимо дать их описание DTD. В описании указываются только самые общие признаки логической взаимосвязи элементов.

- Элемент <courses-list> может содержать в себе только ноль или больше элементов <course> и больше ничего.
- Элемент <course> содержит одну или несколько пар элементов <course-name> и <topic>.
- Элемент <course-name> содержит только одну текстовую строку и ни одного вложенного элемента.
- Элемент <topic> содержит одну и только одну пару элементов <topic-number> и <topic-name>.
- Элемент <topic-number> содержит одно целое число.
- Элемент <topic-name> содержит одну текстовую строку.

Это словесное описание формализуется в описании DTD (листинг 25.2).

Листинг 25.2. Описание DTD документа XML

```
<!ELEMENT courses-list (course)*>
<!ELEMENT course (course-name, topic)+>
<!ELEMENT courses-name (#PCDATA)>
<!ELEMENT topic (topic-number, topic-name)>
<!ELEMENT topic-number (#PCDATA)>
<!ELEMENT topic-name (#PCDATA)>
```

Следует всегда учитывать, что язык XML, в отличие от HTML, различает регистры букв.

Как видите, описание DTD почти очевидно. Первое слово — `ELEMENT` — означает, что элемент может содержать тело с вложенными элементами. Вместо него может стоять слово `EMPTY`, означающее пустой элемент. Слова `ANY` и `MIXED` предупреждают о том, что элемент может содержать и простые данные и/или вложенные элементы. Слово `ENTITY` указывает на обозначение или адрес данных, включенный в описание DTD. Слово `ATTLIST` начинает описание списка атрибутов элемента.

После имени элемента в скобках записываются вложенные элементы и тип данных в теле элемента. Тип `PCDATA` (Parsed Character DATA) означает строку символов Unicode. Тип `CDATA` означает строку символов Unicode, которую не следует интерпретировать.

Звездочка после скобки означает "ноль или более вхождений", а плюс — "одно или более вхождений". Вопросительный знак означает "ноль или один раз".

Описание DTD можно занести в отдельный файл, например, courses.dtd, указав его имя в заголовке, как показано во второй строке листинга 25.1. Можно включить описание в заголовок XML-файла, заключив его в квадратные скобки:

```
<!DOCTYPE mydocs [ Описание DTD ]>
```

После того как создано описание DTD тегов нашей реализации XML и написан документ с этими тегами, следует проверить правильность их написания. Для этого есть специальные программы — *проверяющие анализаторы* (validating parsers).

В открывающих тегах любого элемента XML могут встречаться *атрибуты*. Их применение хорошо известно из языка HTML. Например:

```
<font size = "5" color = "blue">
```

В отличие от языка HTML в языке XML значения атрибутов обязательно надо заключать в кавычки или в апострофы. Атрибуты описываются в DTD элементом `ATTLIST` примерно так:

```
<!ATTLIST font
    size CDATA #IMPLIED color CDATA #IMPLIED>
```

Еще одна конструкция языка XML — *инструкции по обработке* (processing instructions) — позволяет передать анализатору или другой программе — обработчику документа дополнительные сведения для обработки. Инструкция по обработке выглядит так:

```
<?application data ?>
```

Следующую конструкцию языка XML — *пространство имен* — опишем подробнее.

Пространство имен тегов

Поскольку в разных языках-реализациях XML могут встретиться одни и те же имена тегов, имеющие совершенно разный смысл, а в документе они могут быть смешаны, анализатору надо дать возможность их как-то различать. Для этого имена тегов снабжают префиксом, одинаковым для всех тегов одной и той же реализации XML. Все имена тегов с один и тем же префиксом образуют одно *пространство имен* (namespace). Префикс определяется атрибутом `xmlns` и применяется следующим образом:

```
<sdo:sometag xmlns:sdo = "http://some.firm.com/2002/sdoml">
<sdo:course>Кулинария</sdo:course>
```

Адрес URL — значение атрибута `xmlns` с определением `sdo` — не имеет никакого значения и может не соответствовать никакому действительному адресу. Смотрите на него просто как на строку символов, идентифицирующую пространство имен. Обычно указывается URL фирмы, создавшей данную реализацию XML, или имя файла с описанием DTD.

По правилам SGML 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.

Появление имени без префикса в документе, использующем пространство имен, означает, что имя принадлежит пространству имен по умолчанию. Например, язык XHTML допускает применение тегов HTML и элементов XML в одном документе. Допустим, мы определили элемент 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:sdo = "http://some.firm.com/2002/sdoml">
<head><title>Мои книги</title></head>
<body>
   <sdo:book>
      <sdo:title>Технология Java и XML</sdo:title>
   </sdo:book>
</body></html>
```

Первая часть работы закончена. Документы XML написаны. Теперь надо подумать о том, каким образом они будут отображаться на экране дисплея, на листе бумаги, на экране сотового телефона, т. е. надо подумать о визуализации документа XML.

Прежде всего документ надо разобрать, проанализировать (parse) его структуру.

Анализ документа XML

Вначале проводится *лексический анализ* (lexical parsing) документа XML. На этом этапе 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 `StringTokenizer` и `StreamTokenizer` из стандартной поставки J2SE SDK.

Затем выполняется *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 тегов (tree-based parsing). Такие анализаторы 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 между ними. При возникновении события вызывается какой-нибудь метод его обработки: `startElement()`, `endElement()`, `characters()`. 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.

В стандартную поставку Java 2 и Standard Edition или 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* (Simple API for XML Parsing), можно создавать парсеры второго типа, реагирующие на события. Сейчас применяется второе поколение этого набора, называемое *SAX2*. 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 API

Основу SAX2 составляет интерфейс `ContentHandler`, описывающий методы обработки событий: начала документа; появления открывающего тега, тела элемента, закрывающего тега; окончания 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: "адрес" пространства имен `uri`, имя тега без префикса `name` и полное имя с префиксом имени `qname`, а также атрибуты элемента `attrs`, если они есть. Если пространство имен не определено, то значения первого и второго аргумента `null`. Если атрибутов нет, 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 `ContentHandler` реализован классом `DefaultHandler`. В нем сделана пустая реализация всех методов. Разработчику остается реализовать только те методы, которые ему нужны.

В листинге 25.3 приведен пример класса-обработчика для документа XML из листинга 25.1, методы которого собирают информацию об учебных курсах в объекты класса `Course`, описанного в *главе 2*.

Листинг 25.3. Класс-обработчик документа XML

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import javax.xml.parsers.*;
import java.util.*;
import java.io.*;
import sdo.*;

public class CoursesHandler extends DefaultHandler{

   Course course;
   static Vector crs = new Vector(), topic;
   boolean inCourseName, inTopicName;

  public void startElement(String uri, String name,
        String qname, Attributes attrs)
                          throws SAXException{

    if (qname.equals("course")){

       course = new Course();
       topic = new Vector();

    }else if (qname.equals("course-name"))
       inCourseName = true;

     else if (qname.equals("topic-name"))
       inTopicName = true;
  }
```

```
public void characters(char[] buf, int offset, int len)
     throws SAXException{

    String s = new String(buf, offset, len);

    if (inCourseName){

       course.setName(s);
       inCourseName = false;

    }else if (inTopicName){

       inTopicName = false;
       topic.add(s);
    }
}

public void endElement(String uri, String name,
     String qname) throws SAXException{

    if (qname.equals("course")){
       course.setTopics(topic);
       crs.add(course);
     }
 }

public static void main(String[] args){

  try{
    CoursesHandler handler = new CoursesHandler();

    SAXParserFactory fact = SAXParserFactory.newInstance();

    SAXParser saxParser = fact.newSAXParser();

    File f = new File("courses.xml");

    saxParser.parse(f, handler);
```

```
      for (int k = 0; k < crs.size(); k++)
       System.out.println(((Course)crs.get(k)).getName());

    }catch(Exception e){
      System.err.println(e);
    }
  }
}
```

После того как класс-обработчик написан, проанализировать документ очень легко. Стандартные действия приведены в методе `main()` программы листинга 25.3.

Поскольку реализация парсера сильно зависит от его программного окружения, SAX-парсер — объект класса `SAXParser` — создается не конструктором, а фабричным методом `newSAXParser()`. После того как объект создан, остается только применить метод `parse()`, передав ему имя анализируемого файла и экземпляр класса — обработчика событий.

В классе `SAXParser` есть несколько методов `parse()`, позволяющих извлечь документ из входного потока класса `InputStream`, объекта класса `File`, адреса URI или из специально созданного источника класса `InputSource`.

Анализ документов XML с помощью DOM API

Консорциум W3C разработал набор интерфейсов DOM (Document Object Model), методами которых 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. Однако все это требует большого объема оперативной памяти для загрузки большого дерева.

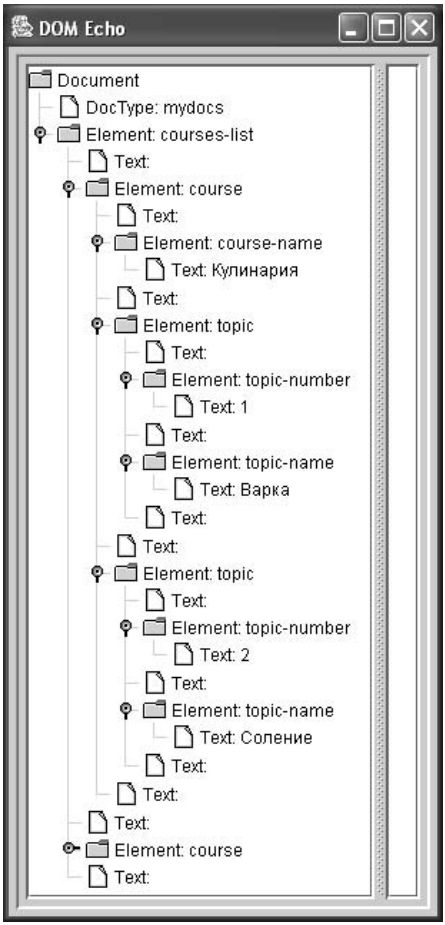
Технология Java реализовала интерфейсы DOM в пакетах javax.xml.parsers и org.w3c.dom. Воспользоваться этой реализацией очень легко:

```
DocumentBuilderFactory fact = DocumentBuilderFactory.newInstance();

DocumentBuilder builder = fact.newDocumentBuilder();

Document doc = builder.parse("courses.xml");
```

Метод `parse()` строит дерево объектов и возвращает ссылку на него в виде объекта типа `Document`. В классе `DocumentBuilder` есть несколько методов `parse()`, позволяющих загрузить файл, хранящийся по адресу 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. 25.1 показано начало дерева объектов, построенного для документа из файла courses.xml, приведенного в листинге 25.1. Обратите внимание на то, что текст, находящийся в теле тега, хранится в отдельном узле дерева — потомке узла тега. Для каждого атрибута тега тоже создается отдельный узел.

Рис. 25.1. 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` — в узле типа `CDADASection`, хранятся данные типа `CDATA`;

- `COMMENT_NODE` — узел типа `Comment`, содержит комментарий;
- `DOCUMENT_FRAGMENT_NODE` — в узле типа `DocumentFragment` находится фрагмент документа;
- `DOCUMENT_NODE` — обозначает корневой узел типа `Document`;
- `DOCUMENT_TYPE_NODE` — помечает узел типа `Document`;
- `ELEMENT_NODE` — узел является листом дерева типа `Element`;
- `ENTITY_NODE` — в узле типа `Entity` хранится сущность `ENTITY`;
- `ENTITY_REFERENCE_NODE` — узел типа `EntityReference` содержит ссылку 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. Узнать тип узла можно методом

```
public short getNodeType();
```

Имя узла возвращает метод

```
public String getNodeName();
```

Значение, хранящееся в узле, можно получить методом

```
public String getNodeValue();
```

Проверить, есть ли атрибуты у элемента XML, хранящегося в узле в виде объекта типа `NamedNodeMap`, если это узел типа `Element`, можно методом

```
public boolean hasAttributes();
```

Атрибуты возвращает метод

```
public NamedNodeMap getAttributes();
```

Метод `getAttributes()` возвращает `null`, если у элемента нет атрибутов.

Метод

```
public boolean hasChildNodes();
```

проверяет, есть ли у данного узла узлы-потомки. Если они есть, то можно получить их список в виде объекта типа `NodeList` методом

```
public NodeList getChildNodes();
```

Первый и последний узлы в этом списке можно получить методами

```
public Node getFirstChild();
public Node getLastChild();
```

Родительский узел можно получить методом

```
public Node getParentNode();
```

а соседние узлы с тем же предком, что и данный узел, — методами

```
public Node getPreviousSibling();
public Node getNextSibling();
```

Можно получить и ссылку на весь документ методом

```
public Document getOwnerDocument();
```

Следующие методы позволяют изменить дерево объектов.

Добавить новый узел-потомок `newChild` можно методом

```
public Node appendChild(Node newChild);
```

Вставить новый узел-потомок `newChild` перед существующим потомком `refChild` можно методом

```
public Node insertBefore(Node newChild, Node refChild);
```

Заменить один узел-потомок `oldChild` новым узлом `newChild` можно методом

```
public Node replaceChild(Node newChild, Node oldChild);
```

Наконец, удалить узел-потомок можно методом

```
public Node removeChild(Node child);
```

Интерфейс *Document*

Интерфейс `Document` добавляет к методам своего предка `Node` методы работы с документом в целом. Метод

```
public DocumentType getDocType();
```

возвращает общие сведения о документе в виде объекта типа `DocumentType`. Методы `getName()`, `getEntitied()`, `getNotations()` и другие методы интерфейса `DocumentType` возвращают конкретные сведения о документе.

Метод

```
public Element getDocumentElement();
```

возвращает корневой элемент дерева объектов, а методы

```
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.

Создать новый пустой элемент по его имени или по имени с префиксом можно методами

```
public Element createElement(String name);
public Element createElementNS(String uri, String name);
```

Создать узел типа `CDATA_SECTION_NODE` можно методом

```
public CDATASection createCDATASection(String name);
```

Создать узел типа `ENTITY_REFERENCE_NODE` можно методом

```
public EntityReference createEntityReference(String name);
```

Создать узел типа `PROCESSING_INSTRUCTION_NODE` можно методом

```
public ProcessingInstruction createProcessingInstruction(String name);
```

Создать узел типа `TEXT_NODE` можно методом

```
public TextNode createTextNode(String name);
```

Создать узел-атрибут с именем `name` можно методами

```
public Attr createAttribute(String name);
public Attr createAttributeNS(String uri, String name);
```

Узел-комментарий создается методом

```
public Comment createComment(String comment);
```

Наконец, можно создать пустой документ — фрагмент данного документа с целью его дальнейшего заполнения

```
public DocumentFragment createDocumentFragment();
```

Вставить созданный узел, а значит, и все его поддерево, в дерево документа можно методом

```
public Node importNode(Node importedNode, boolean deep);
```

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 элемента XML и методы, позволяющие обратиться к вложенным элементам. Только метод

```
public String getTagName();
```

дает сведения о самом элементе, а именно, его имя.

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 `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 к учебному курсу "Кулинария" (см. листинг 25.1) новую тему "Стряпня" со следующим номером темы. Это действие записано в листинге 25.4.

Листинг 25.4. Анализ документа XML с помощью DOM API

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
```

```
public class TreeProcessDOM{

  public static void main(String[] args) throws Exception{

    DocumentBuilderFactory fact =
       DocumentBuilderFactory.newInstance();

    DocumentBuilder builder = fact.newDocumentBuilder();

    Document doc = builder.parse(args[0]);

    NodeList list = doc.getElementsByTagName("course-name");

    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 data = null;

      if (thisNode.getFirstChild() instanceof Text)
         data = ((Text)thisNode.getFirstChild()).getData();

      if (data.equals("Кулинария")) break;
    }

    Node parent = thisNode.getParentNode();

    NodeList topics = ((Element)parent)
             .getElementsByTagName("topic");
```

```
    Node lastTopic = topics.item(topics.getLength() - 1);

    NodeList nl = ((Element)lastTopic)
              .getElementsByTagName("topic-number");

    lastTopic = nl.item(0);

    int lastNumb = Integer.parseInt(
       ((Text)lastTopic.getFirstChild()).getData());

    Node newNode = doc.createElement("topic");

    Node numbNode = doc.createElement("topic-number");
    Text tpNode = doc.createTextNode("" + (lastNumb + 1));
    numbNode.appendChild(tpNode);

    Node nameNode = doc.createElement("topic-name");
    Text textNode = doc.createTextNode("Стряпня");
    nameNode.appendChild(textNode);

    newNode.appendChild(numbNode);
    newNode.appendChild(nameNode);
    parent.appendChild(newNode);
  }
}
```

Дерево объектов можно вывести на экран дисплея, например, как дерево `JTree` — компонент графической библиотеки Swing. Это показано на рис. 25.1. 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**.

Преобразование дерева объектов в XML

Для того чтобы преобразовать дерево объектов DOM обратно в документ XML или в объект другого типа, надо воспользоваться интерфейсами и классами пакетов javax.xml.transform, javax.xml.transform.dom, javax.xml.transform.sax, javax.xml.transform.stream, которые составляют третью часть набора JAXP — реализацию *XSLT* (eXtensible Stylesheet Language for Transformations) средствами Java.

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 объект системы SAX, класс `StreamSource` — просто поток данных. В конструкторы этих классов заносится исходный объект — узел дерева, файл, входной поток. Методы 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, и тоже имеет три реализации — классы `DOMResult`, `SAXResult` и `StreamResult`. В конструкторы этих классов заносится ссылка на выходной объект — узел дерева, объект типа `ContentHandler`, файл, — который будет заполнен результатом преобразования.

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("newCourses.xml");
FileOutputStream fos = new FileOutputStream(newXMLFile);

StreamResult result = new StreamResult(fos);

transformer.transform(source, result);
```

Вначале методом `newInstance()` создается экземпляр `transFactory` фабрики 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 создании этого объекта можно занести в него 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);
```

можно определить свойства преобразованного объекта. Имена свойств задаются следующими константами класса `OutputKeys`:

- `CDATA_SECTION_ELEMENTS` — список имен секций `CDATA`;
- `DOCTYPE_PUBLIC` — открытый идентификатор преобразованного документа;
- `DOCTYPE_SYSTEM` — системный идентификатор преобразованного документа;
- `ENCODING` — кодировка символов преобразованного документа;
- `INDENT` — делать ли отступы: `"yes"` или `"no"`;
- `MEDIA_TYPE` — MIME-тип содержимого преобразованного документа;
- `METHOD` — одно из значений `"xml"`, `"html"` или `"text"`;
- `OMIT_XML_DECLARATION` — не включать объявление XML: `"yes"` или `"no"`;
- `STANDALONE` — отдельный или вложенный документ: `"yes"` или `"no"`;
- `VERSION` — номер версии.

Например, можно задать кодировку символов преобразованного документа следующим методом:

```
transformer.setOutputProperty(OutputKeys.ENCODING, "Windows-1251");
```

Затем для дерева объектов `document` типа `Node` создается объект класса `DOMSource` — производится упаковка дерева объектов для последующего преобразования. Аргумент конструктора этого класса должен иметь тип `Node`, откуда видно, что можно преобразовать не все дерево, а его поддерево, записав в конструкторе класса `DOMSource` корневой узел поддерева.

Наконец, определяется результирующий объект `result`, связанный с файлом `newCourses.xml`, и выполняется 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 или единый стиль оформления документов HTML: цвет, шрифт, заголовки. 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 разработанном подмножестве языка XML, названном *XSL* (eXtensible Stylesheet Language). Все теги документов XSL содержат префикс `xsl`. Простейшая таблица стилей выглядит так, как записано в листинге 25.5.

Листинг 25.5. 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>
```

В приведенной таблице только определяется пространство имен `xsl` и правила вывода, а именно, выводится плоский текст `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 25.5, выводит тела тегов так, как они записаны в документе XML, т. е. просто удаляет теги, оставляя их содержимое.

Эту таблицу стилей записываем в файл, например, simple.xsl, и используем при создании объекта-преобразователя так, как показано в листинге 25.6.

Листинг 25.6. 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 courses.xml simple.xsl
```

и получаем на экране дисплея содержимое тегов документа courses.xml.

В более сложных случаях просто изменяем таблицу стилей. Например, если мы хотим получить документ на консоли в том виде, который представлен в начале этой главы, то таблицу стилей для объекта-преобразователя класса `Transformer` надо записать так, как показано в листинге 25.7.

Листинг 25.7. Таблица стилей для показа документа courses.xml

```
<?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="course">
     <xsl:apply-templates />
  </xsl:template>

  <xsl:template match="topic">
     <xsl:value-of select="topic-number" />
     <xsl:value-of select="topic-name" />
  </xsl:template>

</xsl:stylesheet>
```

К сожалению, мы не можем в этой книге изложить язык XSL — только его описание будет толще всей этой книги. Кроме того, язык XSL вместе с правилами преобразования XSLT (XSL Transform) и правилами форматирования XSL FO (XSL Formatting Objects) бурно развивается, и сделать полное его описание сейчас просто невозможно.

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 25.8 показано, как это можно сделать для документа XML листинга 25.1.

Листинг 25.8. 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>Список курсов</title></head>
    <body><h2>Список учебных курсов</h2>
```

```
      <xsl:apply-templates />
    </body></html>
  </xsl:template>

  <xsl:template match="course-name">
    <h3> <xsl:value-of select="." /> </h3>
  </xsl:template>

  <xsl:template match="topic">
     <xsl:value-of select="topic-number" />
     <xsl:value-of select="topic-name" /> <br/>
  </xsl:template>

</xsl:stylesheet>
```

Глава 26

Упаковка и установка компонентов

На протяжении всей этой части книги мы сталкивались с необходимостью установки компонентов разных типов в соответствующие их типам контейнеры. Для облегчения переноса компонентов и всего J2EE-приложения с одного J2EE-сервера на другой и установки их на другом J2EE-сервере компоненты и все J2EE-приложение упаковываются в стандартно организованные JAR-архивы. Архив может быть без труда загружен по сети и установлен на любом J2EE-сервере, удовлетворяющем спецификации "Java 2 Platform Enterprise Edition Specification".

Кроме компонентов, в каждый архив заносится один DD-файл (deployment descriptor) с описанием содержимого архива, написанный на языке XML. Архив делится на один или несколько модулей, каждый из которых содержит один или несколько J2EE-компонентов и один DD-файл модуля.

Компонентами J2EE-приложения служат сервлеты, страницы JSP, компоненты EJB, клиенты J2EE-приложения и адаптеры ресурсов (Resource Adapter), которые не рассматриваются в этой книге. Сервлеты и страницы JSP вместе называются Web-компонентами.

J2EE-модуль — это основная единица упаковки, состоящая из J2EE-компонентов одного типа, т. е. Web-компонентов, компонентов EJB или клиентов и DD-файла. Вначале создаются компоненты, затем их описание заносится в DD-файл и все это упаковывается в J2EE-модуль. Модуль может переноситься, распространяться и устанавливаться на J2EE-сервер как отдельная единица. Его, вместе с другими модулями, можно упаковать в J2EE-приложение, описав в DD-файле приложения.

Описание DTD конфигурационного DD-файла различает четыре типа J2EE-модулей:

- `web` — модуль, содержащий Web-компоненты;
- `ejb` — модуль, содержащий компоненты EJB;
- `java` — модуль с клиентами J2EE-приложения;
- `connector` — модуль с адаптерами ресурсов.

В листинге 26.1 приведен пример простого DD-файла J2EE-приложения, состоящего из трех модулей: модуля клиента, EJB-модуля и Web-модуля.

Листинг 26.1. DD-файл J2EE-приложения

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE application PUBLIC
  '-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN'
  'http://java.sun.com/dtd/application_1_3.dtd'>

<application>

  <display-name>SomeApp</display-name>

  <description>Описание приложения</description>

  <module>
    <java>app-client-ic.jar</java>
  </module>

  <module>
    <ejb>ejb-jar-ic.jar</ejb>
  </module>

  <module>
    <web>
      <web-uri>SimpleWeb.war</web-uri>
      <context-root>web</context-root>
    </web>
  </module>

</application>
```

Кроме описания модулей J2EE-приложения в DD-файл application.xml с помощью элемента `<display-name>` заносится имя приложения, предназначенное для утилиты установки. В элементе `<description>` дается краткое описание приложения. Кроме того, в элементе `<icon>`, не показанном в листинге 26.1, можно указать имена GIF- или JPEG-файлов со значками, используемыми утилитами, работающими с J2EE-приложением. Еще один не показанный в листинге элемент — `<security-role>` — описывает правила безопасности J2EE-приложения.

Правила упаковки модуля каждого типа записаны в его спецификации. Рассмотрим подробнее эти особенности.

Упаковка Web-модуля

Архив JAR, содержащий Web-модуль, записывается в файл с расширением war (Web ARchive). Он содержит архивы всех Web-компонентов, входящих в модуль, а также специальный каталог WEB-INF, в котором содержится DD-файл с названием web.xml, подкаталог WEB-INF/classes, содержащий классы сервлетов и подкаталог WEB-INF/lib с библиотекой JAR-архивов сопутствующих классов. Содержимое каталога WEB-INF недоступно клиенту, оно относится к контексту сервлета и может быть получено методами `getResource()` и `getResourceAsStream()` интерфейса `ServletContext`.

Конфигурационный DD-файл сервлета web.xml уже описан в *главе 20* (см. его пример в листинге 20.1). При установке Web-компонента в J2EE-приложение в DD-файл заносится дополнительная информация (assembly information) о связях сервлета с другими компонентами J2EE-приложения. Эти сведения записываются в следующие XML-элементы:

- `<env-entry>` — JNDI-имя и значение параметра окружения, который обычно находится в контексте `java:comp/env/`;
- `<ejb-ref>` — JNDI-имя компонента EJB и его интерфейсов удаленного клиента, обычно размещается в контексте `java:comp/env/ejb/`;
- `<ejb-local-ref>` — JNDI-имя компонента EJB и его интерфейсов локального клиента;
- `<resource-env-ref>` — JNDI-имя и тип другого ресурса, например, очереди сообщений JMS из контекста `java:comp/env/jms/`;
- `<resource-ref>` — JNDI-имя и тип внешнего ресурса, например, базы данных из контекста `java:comp/env/jdbc/`.

Упаковка EJB-модуля

При упаковке компонентов EJB можно для каждого компонента создать свой EJB-модуль со своим DD-файлом. Такой способ обеспечивает наибольшую переносимость и независимость компонента EJB, но требует дополнительных усилий при установке и записи в DD-файл сведений о других компонентах EJB, связанных с данным компонентом.

Другое решение — упаковать все компоненты EJB в один EJB-модуль. Это самый простой подход, не требующий записи связей между компонентами. Он будет наилучшим решением в том случае, когда все компоненты работают как единое целое.

Можно выбрать и компромиссное решение — создать несколько EJB-модулей с несколькими компонентами в каждом модуле, объединив в один EJB-модуль тесно связанные между собой компоненты.

Архив с EJB-модулем записывается в так называемый ear-jar-файл с расширением ear (Enterprise ARchive). Этот файл содержит классы компонентов EJB, их home- и remote-интерфейсы, заглушки, а также класс первичного ключа, если он есть, и сопутствующие классы.

Конфигурационный DD-файл EJB-модуля называется ear-jar.xml и хранится в каталоге META-INF. Он описан в *главе 21* (см. его пример в листинге 21.1). При установке EJB-модуля в J2EE-приложение в него заносится дополнительная информация (application assembly information), например, ссылки на другие компоненты EJB. При установке каждого клиента компонента EJB на J2EE-сервер, в DD-файл EJB-модуля, в элемент `<ejb-client-jar>` записывается информация о клиенте.

Упаковка модуля клиента

Клиентская часть J2EE-приложения упаковывается в обычный для JAR-архива файл с расширением jar. В него входят классы клиента и другие классы, к которым обращается клиент.

Конфигурационный DD-файл клиента называется application-client.xml и хранится в каталоге META-INF архива. В DD-файле клиента хранится информация о связях клиента с другими компонентами J2EE-приложения. Эта информация образует следующие XML-элементы:

- `<env-entry>` — JNDI-имя и значение параметра окружения клиента, обычно хранится в контексте `java:comp/env/`;
- `<ejb-ref>` — JNDI-имя компонента EJB, к которому обращается клиент, и имена его интерфейсов удаленного клиента, как правило, из контекста `java:comp/env/ejb/`;
- `<resource-env-ref>` — JNDI-имя и тип другого ресурса, к которому обращается клиент, например, очереди сообщений JMS из контекста `java:comp/env/jms/`;
- `<resource-ref>` — JNDI-имя и тип внешнего ресурса, например, базы данных из контекста `java:comp/env/jdbc` или почтовой службы JavaMail, чье имя обычно хранится в контексте `java:comp/env/mail/`.

В листинге 26.2 приведен простейший DD-файл клиента session-компонента EJB.

Листинг 26.2. DD-файл клиента

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE application-client PUBLIC

  '-//Sun Microsystems, Inc.//DTD J2EE Application Client 1.3//EN'
  'http://java.sun.com/dtd/application-client_1_3.dtd'>

<application-client>

  <display-name>Client</display-name>

  <ejb-ref>

    <ejb-ref-name>ejb/simple</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>SimpleHome</home>
    <remote>Simple</remote>

  </ejb-ref>

</application-client>
```

Установка E2EE-приложения на J2EE-сервер

Полученный при упаковке EAR-архив копируется на J2EE-сервер и используется для установки приложения на сервере.

Во время установки J2EE-приложения J2EE-сервер для каждого компонента EJB выполняет следующие действия.

1. Создает и компилирует заглушки home- и remote-интерфейсов.
2. Обеспечивает безопасность работы компонента EJB по правилам, описанным в его DD-файле.
3. Включает компонент в транзакции, как описано в DD-файле.
4. Регистрирует все JNDI-имена.
5. Для CMP-компонента создает таблицы в базах данных.

Web-компоненты устанавливаются после компонентов EJB, поскольку они ссылаются на компоненты EJB. Для каждого устанавливаемого Web-компонента J2EE-сервер выполняет следующие действия.

1. Создает виртуальные каталоги для URL-запросов к компоненту.
2. Обеспечивает безопасность работы компонента.
3. Регистрирует все JNDI-имена.
4. Создает контекст сервлета.
5. Компилирует страницы JSP, как описано в их DD-файлах.

Затем J2EE-сервер устанавливает клиента, определяя JNDI-имена всех ресурсов, к которым обращается клиент.

Как видно из этого описания, процедуры упаковки и установки требуют точности, и соблюдения общих правил создания архивов для J2EE-приложения. Поэтому каждый производитель J2EE-серверов выпускает утилиты для упаковки и установки J2EE-приложений на свой сервер. Утилиты разных производителей сильно отличаются друг от друга, здесь описываются только стандартные утилиты, входящие в поставку J2EE SDK.

Утилиты упаковки и установки

В стандартную поставку J2EE SDK входят утилиты, облегчающие упаковку и установку компонентов. Они заносят в архив еще один файл — DD-файл времени выполнения (runtime deployment descriptor) с именем sun-j2ee-ri.xml. Этот файл содержит информацию, необходимую J2EE-серверу, входящему в J2EE SDK, для установки J2EE-приложения. Файл sun-j2ee-ri.xml не описан спецификацией J2EE, он не нужен другим J2EE-серверам.

Утилита упаковки

Утилита упаковки packager работает только из командной строки. Ее синтаксис зависит от выполняемой задачи.

Для упаковки EAR-компонентов и создания файла EAR-JAR надо набрать командную строку с флагом `-ejbJar`:

```
packager -ejbJar
  <root-directory> <file-list> <ejb-dd> <ejb-jar>
```

Для упаковки Web-компонента и создания WAR-архива в командной строке применяется флаг `-webArchive`:

```
packager -webArchive
  [-classpath <root-directory> [-classFiles <file-list>]]
  <content-root> [-contentFiles <file-list>] <web-dd> <web-war>
```

Для упаковки клиентского архива следует использовать флаг `-applicationClient`:

```
packager -applicationClient
  <root-directory> <file-list>
  <main-class> <appclient-dd> <appclient-jar>
```

Для упаковки всего J2EE-приложения — флаг `-enterpriseArchive`:

```
packager -enterpriseArchive <file-only-list>
  [-alternativeDescriptorEntries <file-only-list>]
  [-libraryJars <file-list>] <app-name> <app-ear>
```

В приведенных командных строках применены следующие обозначения:

- `<root-directory>` — каталог с классами приложения;
- `<content-root>` —каталог с сопутствующими файлами;
- `<file-list>` — список файлов и/или каталогов через двоеточие;
- `<file-only-list>` — список файлов, но не каталогов;
- `<ejb-dd>` — DD-файл EJB-модуля, обычно ejb-jar.xml;
- `<web-dd>` — DD-файл Web-модуля, обычно web.xml;
- `<appclient-dd>` — DD-файл клиента, обычно application-client.xml;
- `<ejb-jar>`, `<web-war>`, `<appclient-jar>`, `<app-ear>` — создаваемые файлы J2EE-архива;
- `<main-class>` — класс клиента с методом `main()`.

Просмотреть все флаги утилиты можно, набрав командную строку

```
packager -help
```

Примеры:

1. Создать архив EJB-модуля с именем PrepRegEJB.ear, содержащий класс и интерфейсы компонента EJB, а также DD-файл ear-jar.xml:

   ```
   packager -ejbJar /sdo/classes/
   PrepReg.class:PrepRegHome.class:PrepRegBean.class
   ear-jar.xml PrepRegEJB.ear
   ```

2. Создать архив Web-модуля с именем Students.war:

   ```
   packager -webArchive -classpath /sdo/classes/
   -classFiles Students.class:Courses.class
   web.xml Students.war
   ```

3. Создать архив клиента с именем StudClient.jar:

   ```
   packager -applicationClient /sdo/classes/Client.class
   Client application-client.xml StudClient.jar
   ```

4. Создать архив всего J2EE-приложения с именем StudentsApp.ear:

```
packager -enterpriseArchive
PrepRegEJB.ear:Students.war:StudClient.jar
StudentsApp StudentsApp.ear
```

Многие пользователи предпочитают графический режим работы с утилитами. Утилита packager встроена в графический режим работы утилиты deploytool.

Утилита проверки

Утилита verifier проверяет правильность упаковки J2EE-приложения. Она входит в графический режим утилиты deploytool, но имеет свою графическую оболочку, вызываемую из командной строки следующим образом:

```
verifier -u
```

Утилита verifier может работать из командной строки со следующим синтаксисом:

```
verifier [-v] [-o <output-file>] [-a -f -w] <filename>
```

Необязательные флаги имеют такой смысл:

- `-v` — более полный вывод результатов проверки;
- `-o <output-file>` — имя файла с результатами проверки, по умолчанию файл results.txt в текущем каталоге;
- `-a` — вывести все результаты;
- `-f` — вывести только сообщения об ошибках;
- `-w` — вывести только предупреждения.

По умолчанию выводятся сообщения об ошибках и предупреждения.

Утилита установки

Утилита установки deploytool может работать в графическом режиме или запускаться из командной строки. В графическом режиме она выполняет и упаковку, и установку компонентов. В командном режиме утилита только устанавливает компоненты в контейнеры, упаковкой занимается утилита packager.

Утилита deploytool в командном режиме управляется следующими флагами.

- `-deploy <ear-filename> <server-name> [<client-stub-jar>]`

  Устанавливает J2EE-приложение или отдельный модуль, записанный в EAR-архив `<ear-filename>` на J2EE-сервер, работающий на машине с именем `<server-name>`. Здесь же можно установить клиента этого приложения, указав его файл `<client-stub-jar>`.

- `-deployConnector <rar-filename> <server-name>`

  Устанавливает адаптер ресурсов, находящийся в файле с именем `<rar-filename>` на J2EE-сервер `<server-name>`.

- `-listApps <server-name>`

  Показывает на консоли список J2EE-приложений, установленных на J2EE-сервере, работающем на машине с именем `<server-name>`.

- `-listConnectors <server-name>`

  Показывает на консоли список адаптеров ресурсов, установленных на J2EE-сервере, работающем на машине с именем `<server-name>`.

- `-undeployConnector <rar-filename> <server-name>`

  Удаляет адаптер ресурсов, находящийся в файле с именем `<rar-filename>` из J2EE-сервера, работающего на машине с именем `<server-name>`.

- `-uninstall <app-name> <server-name>`

  Удаляет J2EE-приложение с именем `<app-name>` с J2EE-сервера, работающего на машине с именем `<server-name>`.

Просмотреть список флагов можно, набрав командную строку

```
deplotyool -help
```

Запустить утилиту в графическом режиме можно, просто набрав ее имя или применив флаг `-ui`:

```
deploytool -ui
```

Использование утилиты в графическом режиме ничем не отличается от применения других графических инструментов и не требует описания.

Список литературы и Internet-источников информации

1. Арнольд К., Гослинг Дж., Холмс Д. Язык программирования Java. 3-е изд.: Пер. с англ. — М.: Вильямс, 2001. — 624 с., ил.
2. Бадд Т. Объектно-ориентированное программирование в действии. Пер. с англ. — СПб.: Питер, 1997. — 464 с., ил.
3. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на С++. 2-е изд. Пер. с англ. — М.: "Издательство Бином", СПб.: "Невский диалект", 1998. — 560 с., ил.
4. Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя. Пер. с англ. — М.: ДМК, 2000. — 432 с., ил.
5. Гамма Э., Хелм Р., Джонсон Р., Влиссидис Дж. Приемы объектно-ориентированного проектирования. — СПб.: Питер, 2001. — 368 с., ил.
6. Даконта М., Саганич А. XML и Java 2. Библиотека программиста. — СПб.: Питер, 2001. — 384 с., ил.
7. Заккар М. Разработка приложений для электронной коммерции на Oracle8i и Java. Пер. с англ.: Уч. пос. — М.: Вильямс, 2000. — 336 с., ил.
8. Коуд П., Норт Д., Мейфилд М. Объектные модели. Стратегии, шаблоны и приложения. Пер. с англ. — М.: Лори, 1999.
9. Ларман К. Применение UML и шаблонов проектирования. 2-е изд. Пер. с англ.: Уч.пос. — М.: Вильямс, 2002. — 624 с., ил.
10. Морган М. Java 2. Руководство разработчика. Пер. с англ.: Уч. пос. — М.: Вильямс, 2000. — 720 с., ил.
11. Ноутон П., Шилдт Г. Java 2: Пер. с англ. — СПб.: БХВ-Петербург, 2000. — 1072 с., ил.
12. Перроун П. Дж., Чаганти Венканта С. Р. Создание корпоративных систем на основе Java 2 Enterprise Edition. Руководство разработчика. Пер. с англ. — М.: Вильямс, 2001. — 1184 с., ил.
13. Риккарди Г. Системы баз данных. Теория и практика использования в Internet и среде Java.: Пер. с англ. — М.: Вильямс, 2001. — 480 с., ил.
14. Фаулер М., Скотт К. UML. Основы. 2-е изд. Пер. с англ. — М.: Символ-Плюс, 2002. — 192 с., ил.

15. Хабибуллин И. Ш. Самоучитель Java. — СПб.: БХВ-Петербург, 2001. — 464 с., ил.
16. Хейк Б. В. JDBC. Java и базы данных. — М.: Лори, 1999. — 324 с.
17. Холл М. Сервлеты и JavaServer Pages. Библиотека программиста. — СПб.: Питер, 2001. — 496 с., ил.
18. Эккель Б. Философия Java. Библиотека программиста. Пер. с англ. — СПб.: Питер, 2001. — 880 с., ил.
19. Cooper James W., The Design Patterns. Java Companion, Addison-Wesley, 1998, **http://www.patterndepot.com/put/8/DesignJavaPDF.zip**.
20. Eckel Bruce, Thinking in Patterns with Java, **http://www.bruceeckel.com/**.
21. Roman Ed, Mastering EJB, 2nd ed., **http://www.theserverside.com/**.

Предметный указатель

A

Accessibility 69
ACID 637
Acknowledgment 392
Actor 49
Application server 29
Attachment 463
Attribute-value-assertion 413
AWT 69

B

Binding 409
BMP-компонент 569
BOA 470
Business logic 55

C

Catalog 287
CGI 27
    программа 10
CMP:
    field 615
    компонент 569, 615
    поле 615
CMR-поле 621
Codebase 357
Commit 637
Compound name 409
Content:
    identification 466
    pane 182
Context 409, 501
Cookie 517
CORBA 22, 469
COS Naming 411
Custom tags 527

D

DAP 440
DD-файл 558
Delegation 76
Deployment 494
    descriptor 495
Design pattern 48
Destination 398
DGC 357
Directory:
    context 410
    service 410
Distinguished names 440
Distributed object 356
DNS 411, 438
DOM API 660

E

EJB (Enterprise JavaBeans) 557
    контейнер 37, 558
    сервер 558
    слой 44
    ссылка 573
Entity-компонент 561
ESC-синтаксис 308
Expung 462

F

Finder method 612

G

GIOP 470
Glass pane 182
GoF 48

H

Home:
    interface 38
    интерфейс 562
    метод 606

I

IDL 469
IIOP 470
IMAP4 454
Inheritance Model 484
Initial context 410
Input Method Framework 69
INS 441

J

J2EE:
    модуль 678
    сервер 29
Java IDL 22, 471
Java L&F 73
Java Look and Feel 73
Java Mail 29
Java Plug-in 69
Java 2D 69
JavaMail 443
JAXP 43, 660
JDBC 43
JFC 69
JMS 29, 37
JNDI 39, 41
JRMP 18, 370
JSP 35, 527
JSP Container 528
JSP Document 529
JSP Page 529
JTA 43, 639
JTS 640

L

L&F 73
Layered pane 180
LDAP 411, 440
LDAPv3 441
Look and Feel 73
Lookup 18

M

Marshalling 19, 355
MDB-компонент 561
Message 387
    queue 387
    service 387
Middleware 44
MIME 444
Model-View-Controller 75
MOM 388
MTA 443
MUA 443
Multipart 447
MVC 75

N

Namespace 410, 659
Naming system 410
NIS 439

O

OOA 47
OOD 47
OOP 47
ORB 469

P

Persistent object 488
PL&F 73, 233
PLAF (Pluggable Look and Feel) 73, 233
POA 470
Point-to-point 387
Policy file 368
POP3 454
Processing instruction 658
PTP 387
Pub/Sub 388
Publish-and-subscribe 388

R

Remote:
    interface 37
    object 356
    интерфейс 562
RMI 18, 356
    registry 356
    реестр 20, 356
    сервер 357, 361
RMI-IIOP 471
Rollback 637
Root pane 181
RPC 355
RUP 48

S

SAX API 660
SAX2 660
Schema 287
Scriptlet 530
Servant 470, 478
Service provider 411
Servlet 13, 28, 493
    container 493
Session-компонент 561
SFSB 568
SLSB 568

SMTP 444, 454
SOAP 489
SPI 411
Stub 355, 356
Subcontext 409
Swing 69

T

Theme 244
Tie 475
  Model 484
TLD 538
Topic 388
Transaction 637

U

UDDI 490
UML 48
Unmarshalling 355
Use case 49
Usenet 453

V

Validating parser 658

W

Web Application 528
Web Components 528
Web Container 528
Web:
  services 489
  компоненты 528
  контейнер 28, 528
  приложение 528
  слой 44
WSDL 489

X

X.500 440
XML View 530
XSL 674
XSLT 671

А

Адресат 398

В

Виртуальная машина Java (JVM) 13
Вычеркивание 462

Г

Графический интерфейс пользователя 69

Д

Дейтаграмма 16
Делегирование 76
Дескриптор установки 32
Диспетчер транзакций 284
Документ 119
  JSP 529
Драйвер:
  JDBC 274
  ODBC 273

З

Заглушка 355, 356
Заказные теги 527

И

Инструкция по обработке 658
Интерфейс:
  JDBC 274
  ODBC 273
Источник:
  данных 279
  события 201

К

Каркас 355
Каталог 287
Класс-адаптер 202
Компонент EJB 557
Контейнер сервлетов 493
Контекст 409, 501
  каталога 410

Конфигурационный файл 495
Корневая панель 182

Л

Локальный клиент 562

М

Метод:
  выбора 623
  поиска 612
Модель:
  наследования 484
  связки 484
Модель—Вид—Контроллер 75

Н

Начальный контекст 410

О

Объект-заглушка (stub) 18
Ограниченное свойство 207
Откат 637
Очередь сообщений 387

П

Панель содержимого 182
Платформенные сведения 232
Подтверждение 392
Пользовательские сведения 232
Посредник запросов объектов 469
Поставщик:
  JMS 388
  услуг 411
Представитель 76
Представление XML 530
Проверяющий анализатор 658
Программа:
  парсер 660
  сканер 660
Прозрачная панель 182
Промежуточный слой 44
Пространство имен 410, 659
Пул физических соединений 282

Р

Распределенный объект 356

С

Сборка 355
Связывание 409
Сеанс связи 394
Семантическое событие 205
Сервер приложений 29
Сервлет 13, 28, 493
Сериализация 17
Система именования 410
Системные сведения 231
Скриптлет 530
Слоеная панель 180
Служба:
  имен 410
  каталогов 410
  сообщений 387
Слушатель события 201
Событие низкого уровня 204
Соединение 285
Сокет 13
Сообщение 387
Составное имя 409
Стандарт CLI 273
Стиль 121
Страница JSP 529
Схема 287, 416

Т

Тема 244
Тип:
  BLOB 334
  CLOB 334
Точка сохранения 285
Транзакция 637

У

Удаленный:
  интерфейс 358
  клиент 562
  объект 356
Установка 494
  сервлета 32

Ф

Фиксация 637