ТИМУР МАШНИН

bhv®

СОВРЕМЕННЫЕ Java ТЕХНОЛОГИИ НА ПРАКТИКЕ

- ОБЗОР JAVA-ТЕХНОЛОГИЙ И ОБЛАСТЕЙ ИХ ПРАКТИЧЕСКОГО ПРИМЕНЕНИЯ
- БИБЛИОТЕКИ, СПЕЦИФИКАЦИИ, СЕРВИСЫ
- АРХИТЕКТУРА ПЛАТФОРМ JAVA SE, JAVA ME И JAVA EE
- РАЗРАБОТКА АППЛЕТОВ, НАСТОЛЬНЫХ, МОДУЛЬНЫХ И РАСПРЕДЕЛЕННЫХ JAVA-ПРИЛОЖЕНИЙ

PRO
ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

+CD

Тимур Машнин

Современные Java технологии на практике

Санкт-Петербург
«БХВ-Петербург»
2010

УДК 681.3.06
ББК 32.973.26-018.2
М38

Машнин Т. С.

М38 Современные Java-технологии на практике. — СПб.: БХВ-Петербург, 2010. — 560 с.: ил. + CD-ROM — (Профессиональное программирование)

ISBN 978-5-9775-0561-1

Рассмотрено создание широкого круга Java-приложений с помощью современных Java-технологий и среды разработки NetBeans. Подробно рассмотрена архитектура платформ Java SE, Java ME и Java EE. Показано создание апплетов с использованием графических библиотек AWT и Swing, настольных приложений на основе платформы Swing Application Framework, а также расширяемых Java-приложений с использованием библиотек ServiceLoader API, Lookup и др. для платформы Java SE. Рассмотрено создание мобильных приложений на основе конфигурации CLDC и профиля MIDP для платформы Java ME. Показано применение технологий Java Servlet, JavaServer Pages, JavaServer Faces, Web-сервисов, Enterprise JavaBeans и др. при программировании для платформы Java EE. Материал книги сопровождается большим количеством примеров с подробным анализом исходных кодов. На компакт-диске находятся проекты примеров приложений.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 31.05.10.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 45,15.
Тираж 1500 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

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

ISBN 978-5-9775-0561-1

© Машнин Т. С., 2010
© Оформление, издательство "БХВ-Петербург", 2010

Оглавление

Введение 1

Что такое технология Java? 1
Архитектура технологии Java 4
Как разрабатываются приложения Java? 5
Обзор сред разработки Eclipse и NetBeans 6
Установка необходимого программного обеспечения 12

ЧАСТЬ I. ПЛАТФОРМА JAVA SE 15

Глава 1. Создание апплетов с использованием графической библиотеки AWT (Abstract Window Toolkit) 17

Обзор графической библиотеки AWT 17
Применение AWT и сравнение с другими графическими Java-библиотеками 19
Использование AWT на примере создания апплета-игры "Звездные войны" 20

Глава 2. Создание апплетов с использованием графической библиотеки Swing 37

Графическая библиотека Swing и ее применение 37
Использование Swing на примере создания апплета с графическим интерфейсом пользователя 39
Контроль работы апплетов системой безопасности платформы Java SE 64

Глава 3. Создание настольных приложений на базе платформы Swing Application Framework 69

Платформа приложений Swing Application Framework (SAF) и ее применение 69
Пример разработки настольного приложения для поиска в Интернете 71
Структура приложения 73
Код класса *SearchInternetApp* 77
Код класса *SearchInternetView* 81

Код класса *SearchInternetAboutBox* для диалогового окна 94
Разработка графического интерфейса приложения 98
Программирование работы с сетью 105
Сборка и запуск проекта 112
Технология Java Web Start (JWS) 114
Внешний вид и поведение графических компонентов *LookAndFeel* 121
Инструмент *javadoc* 127
Тестирование Java-приложений 134

Глава 4. Создание расширяемых Java-приложений 143

Понятие расширяемых приложений и их назначение 143
Практика применения принципов расширяемости и модульности 144
Способы создания расширяемых Java-приложений с помощью библиотек ServiceLoader API и Lookup, платформы NetBeans и технологии OSGi 145
Пример создания расширяемого приложения с использованием библиотеки ServiceLoader API платформы Java SE 6 151
Разработка сервиса 153
Разработка графического интерфейса приложения 160
Пример создания расширяемого приложения с использованием библиотеки Lookup платформы NetBeans 166
Пример создания модуля NetBeans и модульного приложения на базе платформы NetBeans 175
Пример создания OSGi-приложения 196

ЧАСТЬ II. ПЛАТФОРМА JAVA ME 219

Глава 5. Технологии платформы Java ME 222

Технология CLDC 222
Технология MIDP 224
Дополнительные пакеты технологии Java ME 239
Технология CDC 241
Технологии Foundation, Personal Basis Profile и Personal Profile 242
Графическая библиотека Light Weight User Interface Toolkit (LWUIT) 244

Глава 6. Создание Java-приложений на основе платформы Java ME 246

Пример создания приложения для чтения TXT-файлов с использованием высокоуровневой графической библиотеки 248
Пример создания приложения для чтения TXT-файлов с использованием низкоуровневой графической библиотеки 262

ЧАСТЬ III. ПЛАТФОРМА JAVA EE 273

Глава 7. Клиент-серверная архитектура платформы Java EE 277

Структура приложения Java EE 277
Система безопасности платформы Java EE 278

Web-модули приложения Java EE ... 280
EJB-модули приложения Java EE ... 289
Клиент приложения Java EE ... 302
Дескрипторы развертывания сервера приложений Java EE ... 305

Глава 8. Технологии платформы Java EE ... 330

Технологии Web-приложений ... 330
Технология Java Servlet ... 330
Технология JavaServer Pages ... 331
Технология JavaServer Faces ... 332
Технологии Web-сервисов ... 333
Технология RESTful ... 333
Технология XML-Based RPC (JAX-RPC) ... 335
Технология XML-Based Web Services (JAX-WS) ... 340
Технологии Java EE Enterprise Application ... 345
Технология Enterprise JavaBeans ... 345
Технология JavaMail ... 360
Технологии Hibernate, JDO, Struts, Echo, Spring Framework, Portlet, Google Web Toolkit ... 361
Технология Hibernate ... 361
Технология Java Data Objects ... 362
Технология Struts ... 362
Технология Echo ... 364
Технология Spring Framework ... 364
Платформа Core Container ... 365
Платформа Data Access/Integration ... 365
Платформа Web ... 366
Платформы Aspect Oriented Programming (AOP) и Instrumentation ... 366
Платформа Test ... 366
Технология Portlet ... 366
Технология Google Web Toolkit ... 368

Глава 9. Пример приложения Java EE ... 369

Создание основы приложения ... 369
Создание "тонкого" клиента ... 387

ПРИЛОЖЕНИЯ ... 411

Приложение 1. Структура JRE и JDK ... 412

Структура файловой системы среды выполнения Java Runtime Environment (JRE) ... 412
Структура файловой системы комплекта разработки Java Development Kit (JDK) ... 414

Приложение 2. Структура интерфейса программирования платформы Java SE ... 417

Приложение 3. Проекты Eclipse ... 419

Приложение 4. Основные библиотеки NetBeans API ... 422

Приложение 5. Спецификации платформы Java SE ... 424

Приложение 6. Структура графической библиотеки Swing ... 428

Приложение 7. Коллекция классов пакета *java.io* ... 430

Приложение 8. Справочная система JavaHelp ... 436

Приложение 9. Архитектура технологии OSGi ... 442

Приложение 10. Библиотеки спецификации CLDC 1.0 ... 448

Приложение 11. Синтаксис JSP ... 452

Директивы ... 452
Стандартные действия ... 456
Комментарии ... 463
Скриптовые элементы ... 463
Скриптлеты ... 463
Объявления ... 464
Выражения ... 464
EL-выражения ... 464
Стандартные теги библиотеки JavaServer Pages Standard Tag Library (JSTL) ... 465
Теги библиотеки JSTL ... 466
Тег <*c*> ... 466
Тег <*fmt*> ... 471
Тег <*sql*> ... 478
Тег <*x*> ... 481
Функции библиотеки JSTL ... 485
Пользовательские теги ... 487

Приложение 12. Библиотеки технологии JavaServer Faces ... 491

Библиотека JavaServer Faces API ... 491
Пакет *javax.faces* ... 491
Пакет *javax.faces.application* ... 492
Пакет *javax.faces.component* ... 493
Пакет *javax.faces.component.behavior* ... 495
Пакет *javax.faces.component.html* ... 496
Пакет *javax.faces.component.visit* ... 496
Пакет *javax.faces.context* ... 496
Пакет *javax.faces.convert* ... 497
Пакет *javax.faces.event* ... 497
Пакет *javax.faces.lifecycle* ... 499
Пакет *javax.faces.model* ... 500
Пакет *javax.faces.render* ... 500

Пакет *javax.faces.validator* 501
Пакет *javax.faces.view* 501
Пакет *javax.faces.view.facelets* 502
Пакет *javax.faces.webapp* 502
Библиотеки тегов технологии JavaServer Faces 502
Библиотека тегов Standard HTML RenderKit Tag Library 502
Библиотека тегов JSF Core Tags 534
Библиотека тегов Composite 539
Библиотека тегов Facelets UI 541
Конфигурационный файл faces-config.xml 542

Приложение 13. Описание компакт-диска 549

Предметный указатель 550

Введение

Что такое технология Java?

Начнем с самого понятия технологии программирования. Можно сказать, что *технология программирования* — это совокупность методов и инструментов, позволяющих создавать программное обеспечение. Технологии программирования могут иметь различный уровень применения. В процессе разработки программного обеспечения могут применяться технологии, решающие как конкретные задачи, так и технологии, являющиеся платформой для создания частей приложения или всего приложения. Поэтому, как правило, для создания программного обеспечения применяется целый набор различных технологий.

Применительно к Java, *технология Java* — это язык программирования Java и платформа Java, разрабатываемые компанией Sun Microsystems, Inc. Помимо компании Sun разработкой продуктов Java занимается сообщество Java, объединяющее как компании, так и независимых разработчиков.

Язык программирования Java представляет собой объектно-ориентированный язык программирования, имеющий синтаксис, близкий к синтаксису языка C++. Отличия языка Java от языка C++ обусловлены самим происхождением этих языков программирования. Язык C++ является расширением языка C, который создавался как язык системного программирования. Язык Java, в свою очередь, создавался для решения задач сетевого программирования и является самостоятельным языком программирования. Главные отличия языка Java от языка C++ — это более строгая типизация, ограничения работы с памятью, автоматическая сборка мусора.

Понятно, что для создания программного обеспечения наличие одного языка программирования недостаточно. Для компилируемых языков нужны инструменты, компилирующие исходный код в машинный, исполняемый опера-

ционной системой компьютера. Для интерпретируемых языков программирования необходимы интерпретаторы, выполняющие исходный код в операционной системе. В случае языка Java, реализация платформы Java как раз и обеспечивает выполнение Java-кода в операционной системе компьютера. Таким образом, для того чтобы Java-приложение могло быть запущено, необходима реализация платформы Java.

Я упомянул реализацию платформы Java. Что это такое? *Платформа Java* состоит из виртуальной машины Java Virtual Machine (JVM) и библиотек интерфейса программирования Java Application Programming Interface (API). Для всех распространенных операционных систем существуют свои виртуальные машины JVM, тем самым реализуется принцип "Write Once, Run Anywhere" ("Написанное однажды, работает везде"). *Реализация платформы Java* — это конкретная реализация JVM для конкретной операционной системы плюс библиотеки Java API.

На самом деле, компанией Sun для выполнения Java-приложений предоставляется набор сред выполнения *Java Runtime Environment (JRE)*, охватывающий все распространенные операционные системы. Виртуальная машина JVM составляет основную часть среды выполнения JRE. Помимо JVM, JRE содержит базовые библиотеки API, необходимые для выполнения Java-приложений, а также дополнительные инструменты, включая Java Plug-in для запуска апплетов в браузере и Java Web Start для развертывания Java-приложений через Интернет.

Компанией Sun Microsystems, Inc. также предоставляется минимальный комплект разработки Java-приложений *Java Development Kit (JDK)*, состоящий из набора инструментов, включая компилятор в байт-код `javac`, документации, примеров и среды выполнения JRE.

Язык программирования Java является одновременно и интерпретируемым, и компилируемым. Причина этого кроется в устройстве виртуальной машины JVM.

Виртуальная машина JVM — это набор специальных программ, созданных для конкретной операционной системы. Точкой входа в виртуальную машину JVM является программа `java`, запускающая Java-приложение. Приложения, написанные на языке Java, представляют собой текстовые файлы с расширением java. Чтобы JVM выполнила Java-приложение, приложение должно быть откомпилировано в специальный двоичный формат — *байт-код*. Откомпилированное Java-приложение состоит из файлов с расширением class, которые могут быть упакованы в архивный исполняемый файл с расширением jar. При запуске Java-приложения на вход JVM подается байт-код Java-приложения, а также байт-код используемых приложением библиотек Java API.

Виртуальная машина JVM может выполнять приложения, написанные и на других языках программирования — Ruby, PHP, JavaScript, Python и др., при этом приложения также должны быть откомпилированы в байт-код.

В процессе обработки байт-кода виртуальная машина JVM производит его интерпретацию, т. е. выполняет команды, содержащиеся в байт-коде, или использует компилятор Just-in-time compilation (JIT), который транслирует байт-код в машинный код непосредственно во время выполнения Java-приложения, и тем самым увеличивает скорость обработки байт-кода.

Таким образом, язык Java является компилируемым, потому что необходима компиляция исходного кода в промежуточный по отношению к машинному байт-код, и интерпретируемым, потому что байт-код не может быть исполнен самой операционной системой компьютера, а должен интерпретироваться.

Платформа Java содержит два типа JVM:

- *Java HotSpot Client VM (Client VM).* Вызывается опцией `-client` инструмента `java` и обеспечивает быстрый запуск и потребление небольшого объема оперативной памяти;
- *Java HotSpot Server VM (Server VM).* Вызывается опцией `-server` инструмента `java` и обеспечивает максимальную скорость выполнения приложения.

Для обеих JVM технология Java HotSpot оптимизирует обработку байт-кода, распределение памяти, сборку мусора и управление потоками.

Технология Java — это общее понятие, на самом деле обозначающее широкий спектр Java-технологий. Для технологий Java существует определенный стандарт. Каждая Java-технология состоит из спецификации, стандартной реализации *Reference Implementation (RI)*, набора автоматических тестов *Technology Compatibility Kit (TCK)* на соответствие реализаций спецификации и, как правило, оптимизированной реализации.

> ***Примечание***
>
> Для того чтобы упорядочить процесс разработки спецификаций технологии Java, в 1998 г. был организован формальный процесс разработки *Java Community Process (JCP)*, состоящий из запроса на разработку спецификации, который должен быть одобрен Исполнительным Комитетом, варианта, разрабатываемого группой экспертов и обсуждаемого сообществом, варианта для создания эталонной реализации Reference Implementation (RI) и набора автоматических тестов Technology Compatibility Kit (TCK), финальной реализации и поддержки спецификации.

Архитектура технологии Java

Среда выполнения JRE и комплект разработки JDK являются основными продуктами платформы Java Platform, Standard Edition (Java SE). Подробно структуру JRE и JDK можно посмотреть в *приложении 1*.

Как уже было сказано, платформа Java содержит библиотеки интерфейса программирования Java API. Для чего они предназначены и какую роль они играют?

Библиотеки Java API — это готовые классы и интерфейсы, обеспечивающие для создаваемых Java-приложений общую функциональность. С библиотеками Java API программисту не нужно самому реализовывать ввод/вывод, сетевое соединение, создавать стандартные графические компоненты для интерфейса пользователя и многое другое. Все это уже предоставлено технологией Java.

Интерфейс программирования платформы Java SE представляет собой следующий набор библиотек:

- ❒ базовые библиотеки платформы;
- ❒ интегрированные библиотеки платформы;
- ❒ библиотеки интерфейса пользователя.

Подробная структура Java API платформы Java SE представлена в *приложении 2*.

Платформа Java SE является основой для всех остальных платформ технологии Java. Все вместе Java-платформы обеспечивают применение технологии Java к широкому диапазону устройств — от смарт-карт, встроенных и мобильных устройств до серверов и суперкомпьютеров.

Технология Java представлена следующими платформами:

- ❒ *Java Platform, Standard Edition (Java SE)* — предоставляет среду выполнения и набор технологий и библиотек API для создания и запуска серверных и настольных приложений, апплетов и является основой для остальных платформ. Кроссплатформенность обеспечивается наличием сред выполнения для различных операционных систем. Платформа Java SE включает в себя следующие компоненты: среду выполнения Java Runtime Environment (JRE) и комплект разработчика приложений Java Development Kit (JDK);
- ❒ *Java SE for Embedded* — предназначена для встроенных систем, таких как интеллектуальные маршрутизаторы и коммутаторы, профессиональные принтеры и др. Платформа Java SE for Embedded обеспечивает ту же функциональность, что и платформа Java SE, дополнительно добавляя поддержку платформ, специфических для встроенных систем, оптимиза-

цию использования памяти, а также предоставляя уменьшенную среду выполнения Small Footprint JRE и опцию Headless для устройств, не имеющих дисплея, мыши или клавиатуры;

- *Sun Java Real-Time System (Java RTS)* — коммерческое расширение платформы Java SE, позволяющее увеличить временной контроль над выполнением Java-приложений, используемых в таких областях, как авиакосмическая промышленность, телекоммуникации, научные исследования и др. Платформа Java RTS предоставляет дополнительно новые потоки реального времени, диспетчеризацию и синхронизацию, схему управления памятью, механизмы асинхронной коммуникации, установку времени с высоким разрешением, прямой доступ к физической памяти;
- *Java Platform, Micro Edition (Java ME)* — содержит набор сред выполнения и библиотек API, предназначенный для встроенных и мобильных устройств, таких как мобильные телефоны и карманные персональные компьютеры;
- *Java Card* — позволяет создавать и запускать небольшие приложения (аплеты Java Card) в смарт-картах и других устройствах с очень ограниченными ресурсами, таких как SIM-карты мобильных телефонов, банковские карточки, карты доступа цифрового телевидения и др.;
- *Java Platform, Enterprise Edition (Java EE)* — является расширением платформы Java SE и добавляет библиотеки, позволяющие создавать распределенные, многоуровневые серверные Java-приложения.

Как разрабатываются приложения Java?

Применение грамотной методологии при разработке приложения необходимо для сокращения времени на устранение ошибок и добавление новой функциональности. В случае непродуманного проекта необходимость в видоизменении или устранении недостатков приложения может привести к закрытию проекта и созданию нового. Поэтому важно придерживаться основных этапов разработки приложений, которые мы рассмотрим далее.

Основные этапы разработки приложений:

1. **Определение задач.** В самом начале необходимо определить перечень задач и функций, которые должно выполнять приложение. Список функций должен разделяться на две части — основные функции приложения и функции, которые могут быть добавлены в будущем.
2. **Составление последовательности выполнения действий.** Следующим этапом является разработка алгоритма работы приложения. На этом этапе задачи логически группируются и располагаются в порядке их выполнения.

3. **Определение необходимых ресурсов.** Для каждой задачи необходимо проанализировать данные, которые могут понадобиться для ее выполнения. Данные могут быть внешними или вычисляемыми, изменяемыми или не изменяемыми. Кроме того, важно определить область видимости данных.
4. **Разработка макета пользовательского интерфейса.** Наиболее важной частью приложения является пользовательский интерфейс, с помощью которого пользователь взаимодействует с программой, поэтому очень важно правильно разработать функциональность интерфейса, т. к. именно он определяет привлекательность приложения для конечного потребителя.
5. **Выбор технологий.** Технология Java в свою очередь состоит из огромного набора различных технологий. Выполнение каждой задачи приложения может быть реализовано с помощью одной или нескольких Java-технологий. На данном этапе нужно определить перечень библиотек API, которые будут использоваться при разработке приложения.
6. **Выбор среды разработки.** Существует огромное количество инструментов разработки Java-приложений — как бесплатных, так и коммерческих. На этом этапе необходимо выбрать оптимальную для данного проекта среду разработки приложений.
7. **Создание приложения.** Опираясь на все предыдущие этапы, разрабатывается приложение, решающее все требуемые задачи, обладающее дружественным интерфейсом пользователя и возможностью будущего расширения. Данный этап включает в себя отладку приложения и устранение ошибок.
8. **Тестирование.** Для обеспечения надежности и корректности работы приложения необходимо его тестирование при различных условиях и вводимых данных. В процессе тестирования происходит видоизменение приложения и устранение его недостатков. Тестирование приложений — это отдельная технология со своей методологией и инструментами.
9. **Определение способов распространения приложения.** На этом этапе определяется, как и с помощью каких технологий созданное приложение будет поставляться конечному пользователю и разворачиваться на клиентском компьютере.

Обзор сред разработки Eclipse и NetBeans

Для написания исходного кода приложения хватит простейшего редактора типа Блокнот. Для отладки и компиляции достаточно командной строки. Однако интегрированная среда разработки (IDE) предоставляет массу преимуществ разработчику программного обеспечения, представляя собой полный

набор инструментов для создания приложений. Как правило, данный набор включает в себя текстовый редактор, средства автоматизации сборки и отладки, компилятор/интерпретатор, интегрированные средства контроля версий, конструктор графического интерфейса пользователя, браузер классов, инспектор объектов, средства управления проектами, настраиваемый интерфейс пользователя, средства навигации, шаблоны, средства тестирования и многое другое.

В настоящее время из бесплатных IDE наиболее популярными являются Eclipse и NetBeans.

Примечание

Проект Eclipse изначально создавался как технология, запатентованная фирмой Object Technology International (OTI), которая являлась дочерней компанией корпорации IBM. Целью проекта было создание интегрированной среды для разработки Java-приложений. В 2001 г. IBM открыла исходный код и сформировала консорциум Eclipse, задачей которого было дальнейшее развитие проекта. В 2004 г. консорциум Eclipse был преобразован в некоммерческую организацию Eclipse Foundation, обслуживающую сообщество разработчиков открытого исходного кода и являющуюся координатором разработок Eclipse.

Проект Eclipse представляет собой набор расширяемых за счет встраиваемых плагинов (модулей) проектов с открытым исходным кодом, построенных на основе технологии Equinox OSGi (Open Source Gateway Initiative). Equinox — это название одной из реализаций стандарта OSGi, который определяет связывание удаленных узлов, состыковывая различные платформы и слои архитектуры. Equinox OSGi определяет модель динамически подключаемых модулей Eclipse и является модульной технологией, позволяющей создавать приложения, реализующиеся в наборе компонентов *(bundles)*, которые используют общие сервисы и инфраструктуру, при этом модули Java-классов загружаются по требованию, т. е. динамически. Каждый модуль имеет свои файлы манифеста *(manifest)*, содержащие информацию о его соединениях с другими модулями. При этом модуль может объявить любое количество именованных точек расширения *(extension point)* и любое количество расширений *(extension)* к одной или более точкам других модулей.

Подробно структуру проекта Eclipse можно посмотреть в *приложении 3*.

Примечание

Проект NetBeans начался в 1996 г. как студенческий в Карловом университете г. Праги. Целью проекта было создание 100% Java Delphi-подобной среды Java-разработки. В 1997 г. командой разработчиков была основана компания NetBeans для продвижения коммерческих версий среды разработки. В 2000 г. компания Sun Microsystems приобрела NetBeans и открыла исходные коды проекта, организовав сообщество NetBeans. В настоящее время свыше 500 тыс. разработчиков более чем из 130 стран и большое количество фирм составляют сообщество NetBeans.

Проект NetBeans состоит из:

- платформы NetBeans;
- среды разработки NetBeans IDE.

Платформа NetBeans — модульная и расширяемая за счет интеграции дополнительных модулей база, на основе которой возможно создание насыщенных клиентских приложений, использующих графическую библиотеку Swing. Платформа NetBeans является фундаментом для самой среды разработки NetBeans IDE и предоставляет общие для всех настольных приложений возможности, обеспеченные соответствующими API.

Возможности, предоставляемые платформой NetBeans:

1. **Модульная архитектура разрабатываемого приложения.** Платформа NetBeans предоставляет возможность организации динамической загрузки модулей в созданных на ее основе приложениях.
2. **Управление графическим интерфейсом пользователя.** Платформа обеспечивает оконную систему, которая упрощает управление составными компонентами внутри главного окна, такими как окна, меню, панели инструментов и др., общими для всех настольных приложений, а также систему управления действиями, связанными с этими компонентами. Платформа позволяет добавлять, удалять или изменять меню, панели инструментов, "горячие" клавиши, используя файл layer.xml, являющийся неотъемлемой частью каждого модуля. При изменении позиций, размеров, состояния открытия или закрытия окон приложения платформа сохраняет информацию до следующего запуска приложения.
3. **Управление данными.** Window System API, Visual Library API и Explorer & Property Sheet API обеспечивают богатый набор графических элементов пользовательского интерфейса для представления и манипулирования данными.
4. **Управление установками.** С помощью Options Settings API и NbPreferences API возможно управление сохранением и восстановлением установок.
5. **Графический редактор.** Применяя NetBeans IDE как комплект средств разработки (SDK) для платформы NetBeans, возможно использование GUI Builder для графического представления данных. Кроме того, Visual Library API позволяет использовать вместе комплексное моделирование и графическое представление.
6. **Редактор.** Приложения, созданные на основе платформы NetBeans, могут использовать NetBeans Editor независимо от NetBeans Platform.
7. **Мастер.** Платформа обеспечивает генерацию общего кода согласно шаблонам и установку структуры ресурсов проекта.

8. **Управление хранением данных.** Платформа обеспечивает для модулей доступ к файлам данных, расположенным как локально, так и удаленно на FTP-сервере, в CVS-хранилище, в виде XML-файла или в базе данных, с помощью класса `FileObjects`.
9. **Механизм автообновления**, обеспечивающий динамическое обновление установленного приложения. Возможность использования приложений, созданных на основе платформы NetBeans, как технологии Java Web Start для распространения и обновления, так и Plugin Manager для обновления или добавления новой функциональности приложению через Интернет.

Перечень основных библиотек NetBeans API можно посмотреть в *приложении 4*.

Для создания приложения на базе платформы NetBeans достаточно написать модуль, реализующий специфические задачи приложения, и связать его с модулями платформы. В результате получится кроссплатформенное приложение с гибкой структурой.

Основное отличие платформы NetBeans от платформы Eclipse состоит в том, что NetBeans использует платформно-независимую графическую библиотеку Swing, созданную на языке программирования Java, в то время как Eclipse использует графическую библиотеку SWT, тесно интегрированную с конкретной операционной системой. Модульная архитектура Eclipse основана на технологии OSGi, в то время как NetBeans для построения модульной системы использует NetBeans API.

NetBeans IDE — это модульная, кроссплатформенная, с открытым исходным кодом, интегрированная среда разработки, предназначенная для создания, компилирования, развертывания и отладки настольных, корпоративных, Web- и мобильных приложений.

Среда разработки NetBeans IDE является насыщенным клиентским приложением, созданным на основе платформы NetBeans с использованием языка программирования Java. Поддерживает разработку не только на языке Java, но и на C/C++, Ruby и PHP, имеет встроенную поддержку платформ Java SE, Java EE, Java ME и большой выбор дополнительных модулей.

Базовая среда NetBeans IDE включает в себя следующие средства.

- *Основные компоненты среды разработки*, такие как: Apache Ant — инструмент автоматизации сборки приложения; окна проектов, файлов, сервисов, навигатора; настраиваемый интерфейс пользователя; настраиваемое форматирование кода; настраиваемая оконная система; меню навигации; группировка проектов; встроенные шаблоны программирования и примеры приложений; сборка и запуск проекта; окна стандартного вывода и списка задач; совместное использование проектов; возможность импорта проектов и др.

- *Редактор исходного кода* с поддержкой Java, XML, DTD, CSS, HTML, ERB, RHTML, JSP, Javadoc, JavaScript, PHP, Groovy, Ruby, C/C++, а также таких функций, как: интеллектуальное дополнение кода; выделение кода с учетом синтаксического и семантического анализа; выделение событий, ошибок, предупреждений, подсказок, вариантов быстрого ввода; автоматическое компилирование при сохранении и развертывание при изменении в целом для проекта; автоматическая генерация кода; свертывание исходного кода; настраиваемое форматирование кода; средства улучшения (refactoring) кода; всплывающие окна документации; возможность расширения редактора с поддержкой любого языка и др.
- *Инструменты контроля версий и сотрудничества разработчиков* включают в себя: интегрированные средства, которые обеспечивают автоматическое распознавание существующих каталогов под контролем версий, а также обновление, просмотр, сравнение и переименование файлов, фиксацию изменений; возможность выбора между системами контроля версий: CVS, Subversion и Mercurial; модуль программного средства Rational ClearCase фирмы IBM — системы конфигурационного управления программным проектом; возможность обзора состояния и истории файлов каталога с помощью окна **Versioning**; средство просмотра различий на основе строк с выделением строк условными цветами; модуль Developer Collaboration, который обеспечивает совместное использование проектов в режиме реального времени по сети.
- *Поддержку платформы Java SE* (Java Standard Edition), обеспечивающую: разработку на языке программирования Java с поддержкой JDK; модульное тестирование с использованием библиотеки JUnit; менеджер библиотек и шаблонов, доступный из элемента меню **Tools** среды разработки; окно иерархии вызовов — щелкнув правой кнопкой мыши на строке исходного кода и выбрав в появившемся меню пункт **Call Hierarchy**, можно увидеть появившееся окно, в котором будут отображаться отношения между методами классов; инструменты Javadoc Auto Comment — для генерации комментариев, Javadoc Analyzer — для определения существующих комментариев; автоматическое определение корневого каталога документации и исходного кода; поддержку технологии Java Beans; визуальный конструктор графического пользовательского интерфейса Visual Swing GUI Builder; интегрированный профайлер — инструмент для анализа производительности центрального процессора (CPU), памяти и потоков выполнения с целью оптимизации быстродействия приложения и использования памяти; интегрированный отладчик с поддержкой нескольких языков программирования с возможностью установки точек останова в исходном коде, осуществлением пошагового выполнения кода, многосеансовой и многопоточной отладкой, вычислением выражений.

- *Поддержку платформы Java EE* (Java Enterprise Edition), включая:
  - разработку Web-приложений с использованием технологий и языков программирования Ajax, CSS, JavaScript, PHP, апплетов, Java Web Start (JNLP), JavaServer Faces (JSF) и JavaServer Pages (JSP);
  - интеграцию с базами данных, которая обеспечивает пользовательский интерфейс приложений баз данных;
  - разработку серверных Java-приложений с использованием стандартов J2EE 1.3, J2EE 1.4 и EE Java 5 (включая аннотации), интерфейсов Java Persistence API, Java Servlet API, технологий JavaServer Pages, Enterprise JavaBeans, платформ Hibernate, Spring, серверов приложений Sun Java System Application Server, GlassFish, Apache Tomcat, JBoss, BEA WebLogic, IBM WebSphere, Sailfin и др.;
  - разработку Web-служб на базе SOAP и RESTful с использованием стандартов Web-служб JAX-WS 2.1, JAX-RS (JSR 311), JAX-RPC (JSR 101), а также разработку мобильных Web-служб Java ME (JSR 172).
- *Поддержку платформы Mobility* (Java Micro Edition), которая обеспечивает разработку приложений для мобильных телефонов, карманных компьютеров, телеприставок, для встраиваемых систем, включая: поддержку профилей Mobile Information Device Profile (MIDP) 1.0, 2.0, 2.1 (MSA), Connected Limited Device Configuration (CLDC) 1.0 и 1.1, Connected Device Configuration (CDC); визуальную разработку графических интерфейсов с помощью Visual Mobile Designer (VMD); визуальный редактор мобильных игр Mobile Game Builder для интерфейса MIDP 2.0 Game API; поддержку графики SVG (JSR 226); использование сценариев компоновки Apache Ant и тестирования JMUnit и др.
- *Разработку на языках программирования* Ruby, C/C++, PHP, Python, Groovy, языке UML, языке сценариев JavaFX.
- *Разработку приложений на базе сервис-ориентированной архитектуры* (SOA) с использованием Web-служб, проектировщиков BPEL и XSLT, редактора сборки служб для составных приложений, средства и редактора схем XML, графического редактора для языка описания Web-служб (Web Services Description Language, WSDL).
- *Разработку насыщенных клиентских приложений* на базе платформы NetBeans.
- *Инструмент Plugin Manager*, который объединяет Update Center и Module Manager и позволяет подключать дополнительные модули, доступные на портале Plugin Portal официального сайта NetBeans (**http://plugins.netbeans.org/PluginPortal/**).

И Eclipse и NetBeans представляют собой увлекательные миры, погрузившись в которые каждый найдет для себя много полезного и интересного.

В данной книге мы рассмотрим использование среды разработки NetBeans для создания Java-приложений.

Установка необходимого программного обеспечения

Мы будем рассматривать использование среды NetBeans в операционной системе Windows XP/Vista.

Для начала работы нам потребуется установить комплект разработки JDK (Java™ Platform, Standard Edition Development Kit), а также интегрированную среду разработки NetBeans.

Для установки JDK:

1. Скачайте бесплатно дистрибутив JDK (**http://java.sun.com/javase/downloads/**).
2. Загруженный файл будет представлять собой приложение вида jdk-6ux-windows-i586-p.exe. Запустите это приложение.
3. Откроется окно установщика JDK (рис. B1).
4. Дальше следуйте инструкциям. В результате получится два каталога: jre1.6.0_0x и jdk1.6.0_0x, содержащие Standard Edition Runtime Environment (JRE) и Standard Edition Development Kit (JDK).

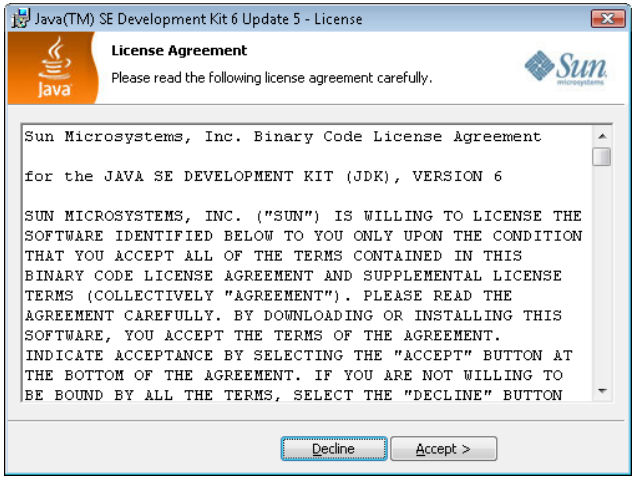

Рис. B1. Окно установщика JDK

Для установки среды разработки NetBeans:

1. Загрузите с сайта по адресу **http://www.netbeans.org** интегрированную среду разработки NetBeans.
2. Загруженный файл будет представлять собой приложение вида netbeans-x.x-ml-windows.exe. Запустите это приложение.
3. Откроется окно инсталлятора среды NetBeans (рис. В2).
4. Следуйте инструкциям. В результате появятся каталоги NetBeans и glassfish-vx, содержащие сервер приложений GlassFish.

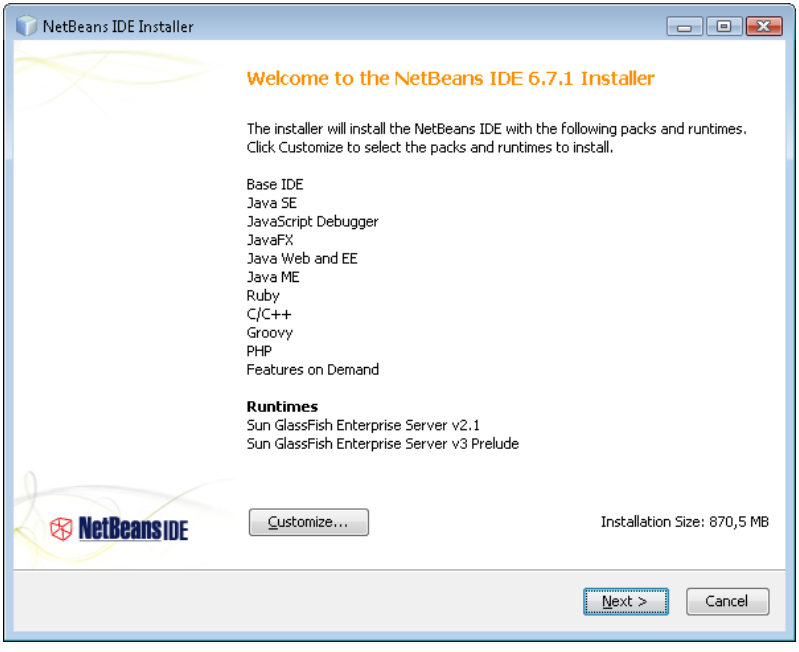

Рис. В2. Окно установщика NetBeans

ЧАСТЬ I

Платформа Java SE

Глава 1	Создание апплетов с использованием графической библиотеки AWT (Abstract Window Toolkit)
Глава 2	Создание апплетов с использованием графической библиотеки Swing
Глава 3	Создание настольных приложений на базе платформы Swing Application Framework
Глава 4	Создание расширяемых Java-приложений

Общая структура платформы Java SE была рассмотрена во *введении*. Еще раз можно повторить, что платформа Java SE является основой для всех остальных платформ технологии Java и предназначена для создания апплетов и настольных приложений. Платформа Java SE содержит набор технологий, которые описываются соответствующими спецификациями. Перечень спецификаций платформы Java SE и их общее применение можно посмотреть в *приложении 5*.

Создаваемые на основе платформы Java SE апплеты представляют собой программы, написанные на языке Java и работающие в среде браузера, который загружает их и запускает виртуальную машину JVM для их выполнения.

В отличие от настольных приложений, апплеты являются управляемыми программными компонентами. Это означает, что апплет не может быть запущен, как настольное приложение, одной только виртуальной машиной JVM. Для его работы необходим браузер, который распознает теги `<APPLET>` или `<OBJECT>` и `<EMBED>`, включающие апплет в HTML-страницу. Главный класс апплета должен быть подклассом класса `java.applet.Applet`, при этом класс `Applet` служит интерфейсом между апплетом и браузером. Жизненным циклом апплета управляет компонент Java Plug-in среды выполнения JRE.

Апплеты могут быть двух типов: имеющие электронную подпись и сертификат и не имеющие сертификата. Не имеющие сертификата апплеты, или апплеты, сертификат которых не принял конечный пользователь, работают в "песочнице" — данное понятие обозначает специальный механизм системы безопасности, ограничивающий доступ работающего приложения к компьютеру пользователя. Апплеты, сертификат которых принят пользователем, работают вне "песочницы", в частности, они способны осуществлять доступ к файловой системе пользователя.

Область применения апплетов достаточно широка. Апплеты используются в качестве элементов, украшающих Web-страничку, создавая продвинутую анимацию, служат интерактивной рекламой, представляют интерактивные игры. Однако назначение апплетов гораздо шире, чем использование в Web-дизайне. При создании приложений уровня предприятия с помощью апплетов создается графический интерфейс пользователя *"тонкого"* клиента.

Настольные приложения платформы Java SE — это независимые Java-приложения, которые выполняются виртуальной машиной JVM, при этом точкой входа в приложение является главный класс приложения, содержащий статический метод `main`. С помощью технологии Java можно создать практически любое настольное приложение с областью применения, ограниченной только фантазией разработчика. Это различные проигрыватели, навигаторы, читалки, переводчики, почтовые клиенты и многое другое. В приложениях уровня предприятия настольное приложение представляет *"толстого"* клиента.

ГЛАВА 1

Создание апплетов с использованием графической библиотеки AWT (Abstract Window Toolkit)

Обзор графической библиотеки AWT

Любое приложение, требующее взаимодействия с пользователем, должно иметь интерфейс пользователя. От интерфейса пользователя зависит привлекательность и удобство работы с программой. Интерфейс пользователя может быть как низкоуровневым, в виде командной строки, так и иметь разнообразные графические компоненты — кнопки, переключатели, меню, поля ввода и другие элементы. Технология Java предлагает набор библиотек классов и интерфейсов, на основе которых можно построить эффективный графический интерфейс пользователя, имеющий необходимый внешний вид и поведение для создания комфортной среды конечному потребителю.

Самой первой графической Java-библиотекой была создана библиотека AWT. Она была включена в первую версию JDK 1.0. Затем библиотека AWT была дополнена библиотекой Java 2D API, расширяющей возможности работы с двухмерной графикой и изображениями. Так как технология Java является платформно-независимой, то соответственно и графическая Java-библиотека должна быть платформно-независимой. Сам по себе язык Java не обладает возможностями низкоуровневого взаимодействия с конкретной операционной системой, обеспечивающими передачу информации от мыши или клавиатуры приложению и вывод пикселов на экран. Поэтому библиотека AWT была создана так, что каждый AWT-компонент имеет своего двойника `peer` — интерфейс, обеспечивающий взаимодействие с конкретной операционной системой. Таким образом, переносимость графической библиотеки AWT обусловлена наличием реализации пакета `java.awt.peer` для конкретной

операционной системы. Вследствие этого, AWT-компоненты называют *тяжеловесными*.

Все AWT-компоненты, кроме элементов меню, представлены подклассами класса `java.awt.Component`. Для элементов меню суперклассом является класс `java.awt.MenuComponent`. Архитектура AWT устроена таким образом, что компоненты размещаются в контейнерах (суперкласс `java.awt.Container`) с помощью менеджеров компоновки — классов, реализующих интерфейс `java.awt.LayoutManager`.

Для настольных приложений корневое окно графического интерфейса пользователя представляет контейнер `java.awt.Window`, который в свою очередь должен содержать окно `java.awt.Frame` с заголовком и границами или диалоговое окно `java.awt.Dialog`, также имеющее заголовок и границы. AWT-компоненты добавляются в панель `java.awt.Panel` — контейнер, который может содержать как компоненты, так и другие панели.

Для апплетов класс `java.applet.Applet`, расширяющий класс `java.awt.Panel`, является корневым контейнером для всех графических компонентов.

Помимо графических компонентов, библиотека AWT содержит классы и интерфейсы, позволяющие обрабатывать различные типы событий, генерируемые AWT-компонентами. Суперклассом, представляющим все AWT-события, является класс `java.awt.AWTEvent`. Для обработки событий компонента необходимо создать класс-слушатель, реализующий интерфейс `java.awt.event.ActionListener`, и присоединить его к данному компоненту.

Кроме пакетов `java.awt` и `java.awt.event` библиотека AWT включает в себя пакеты:

- `java.awt.color` используется для создания цвета;
- `java.awt.datatransfer` применяется для передачи данных внутри приложения и между приложениями;
- `java.awt.dnd` реализует технологию drag-and-drop;
- `java.awt.font` обеспечивает поддержку шрифтов;
- `java.awt.geom` реализует двухмерную геометрию;
- `java.awt.im` обеспечивает поддержку нестандартных методов ввода текста;
- `java.awt.image` используется для создания и редактирования графических изображений;
- `java.awt.print` обеспечивает поддержку печати.

Так как AWT-компоненты основываются на peer-объектах, то использование библиотеки AWT является потоково-безопасной (thread safe), поэтому не нужно беспокоиться о том, в каком потоке обновляется состояние графиче-

ского интерфейса. Однако беспорядочное использование потоков может замедлять работу AWT-интерфейса.

Обобщая вышесказанное, можно сказать, что графическая библиотека AWT представляет собой промежуточный уровень между операционной системой и Java-кодом приложения, скрывая все низкоуровневые операции, связанные с построением графического интерфейса пользователя. Такое прямое взаимодействие с конкретной операционной системой является и основным недостатком AWT, т. к. графический интерфейс, созданный на основе AWT, в операционной системе Windows выглядит как Windows-подобный, а в операционной системе Mac OS X — как Mac-подобный.

Применение AWT и сравнение с другими графическими Java-библиотеками

Графическая библиотека AWT — это стандарт платформы Java SE, однако набор графических компонентов, предлагаемый AWT, ограничен. В частности, отсутствуют такие компоненты, как таблицы, строка состояния, переключатель (radio button) и др. Кроме того, внешний вид и поведение графического AWT-интерфейса пользователя зависят от операционной системы, в которой он отображается.

Все это послужило причиной создания графической библиотеки Swing. Библиотека Swing расширяет AWT и создана полностью на языке Java, поэтому Swing-компоненты называются *легковесными*, за исключением четырех компонентов верхнего уровня: `JWindow`, `JFrame`, `JDialog` и `JApplet`, которые являются прямыми наследниками тяжеловесных AWT-компонентов. Библиотека Swing предлагает гораздо более широкий набор компонентов и менеджеров компоновки по сравнению с библиотекой AWT. Кроме того, т. к. Swing создана полностью на Java, то решается проблема одинакового отображения и поведения интерфейса пользователя в различных операционных системах.

Помимо всего прочего, библиотека Swing расширяет модель обработки событий AWT-библиотекой соответствующих классов и интерфейсов. Однако при этом использование библиотеки Swing не является потоково-безопасной, поэтому Java-код, изменяющий состояние интерфейса и обрабатывающий события, должен выполняться в специальном потоке *Event Dispatch Thread (EDT)*.

Так же как и AWT, библиотека Swing — стандарт платформы Java SE.

Альтернативой AWT и Swing служит библиотека SWT (Standard Widget Toolkit) и набор расширенных возможностей JFace проекта Eclipse, не являющий-

ся стандартом Java SE. Это означает, что их использование требует включения в CLASSPATH Java-приложения.

Примечание

CLASSPATH — переменная среды выполнения, содержащая информацию о расположении Java-файлов. Данная информация используется инструментами javac и java при компиляции и выполнении Java-кода соответственно.

Концепция SWT близка концепции AWT. Компоненты SWT также взаимодействуют с операционной системой с помощью интерфейсов peer. Отличие заключается в том, как данное взаимодействие происходит. В SWT интерфейсы peer выполняют функцию оболочек для графических библиотек конкретных операционных систем. Поэтому приложение, созданное с использованием библиотеки SWT, становится полностью совместимым с конкретной операционной системой, тем самым нарушается принцип "Write Once, Run Anywhere". Однако использование графических возможностей самой операционной системы, вместо создания собственной графики, делает применение библиотеки SWT более эффективным. По ассортименту графических компонентов, менеджеров компоновки и модели обработки событий SWT сравнима со Swing. Так же как и Swing, библиотека SWT не является потоковобезопасной.

В платформу Java SE также включена графическая библиотека Java 3D, которая, правда, не поставляется вместе с комплектом разработки JDK, а требует отдельной загрузки и инсталляции. Библиотека Java 3D позволяет создавать 3D-апплеты и Java-приложения, использующие трехмерную графику. С Java 3D можно эффективно конструировать виртуальные миры, создавая отдельные графические элементы и затем соединяя их в древовидные структуры — *scene graph*. Библиотека Java 3D — это результат синтеза лучших идей, взятых из таких технологий, как Direct3D, OpenGL, QuickDraw3D и XGL.

Казалось бы, при таком выборе графических Java-библиотек библиотека AWT должна потерять свою актуальность. Однако если нет необходимости в широком ассортименте графических компонентов, если требуется работа в основном с двухмерной графикой и изображениями, использование библиотеки AWT удобно. Кроме того, библиотека AWT является частью платформы Java ME и используется для создания приложений, работающих в устройствах с ограниченными возможностями — КПК и телевизионных приставках.

Использование AWT на примере создания апплета-игры "Звездные войны"

В качестве примера рассмотрим создание апплета, представляющего игру "Звездные войны".

Примечание

Файлы с кодами апплета-игры "Звездные войны" расположены на прилагаемом к книге компакт-диске в папке Часть_I\Глава_1\StarWar.

Описание игры

Космический корабль летит в звездном пространстве и поражает огнем лазера появляющихся врагов. За каждого пораженного врага начисляются очки. У корабля есть определенное количество жизней, которое уменьшается при столкновении с врагом. Игра прекращается при обнулении количества жизней космического корабля.

В самом начале создадим в простейшем графическом редакторе Paint, поставляемом с операционной системой Windows, графические изображения, необходимые для работы апплета:

- космический корабль — ship.gif;
- пламя двигателя корабля — fire.gif;
- огонь лазера пушки корабля — laser.gif;
- враги — enemy.gif;
- взрыв при столкновении или при поражении врага — explosion.gif.

Далее в среде разработки NetBeans создаем новый проект:

1. В главном меню среды разработки (верхняя строка) выберем **File | New Project**. Появится диалоговое окно **New Project** (рис. 1.1), в котором выберем **Java Class Library**.
2. В строке **Project Name** введем название проекта StarWar (рис. 1.2).
3. Нажмем кнопку **Finish**.
4. В окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на названии проекта StarWar. В появившемся контекстном меню выбираем **New | Other**.
5. В появившемся окне **New File** выберем **Java | Applet** (рис. 1.3). В этом же окне есть пункт шаблон **JApplet**, который отличается от шаблона **Applet** тем, что использует класс `JApplet`, являющийся расширением класса `Applet` и использующий графическую библиотеку Swing, в то время как класс `Applet` создан на основе графической библиотеки AWT.
6. Далее нажмем кнопку **Next** и введем имя класса и пакета (рис. 1.4).
7. Нажмем кнопку **Finish** и попадем в окно редактора исходного кода среды NetBeans (рис. 1.5).

При создании проекта среда NetBeans сгенерировала код, являющийся основой нашего апплета. Для общего понимания проанализируем этот код, представленный в окне редактора (листинг 1.1).

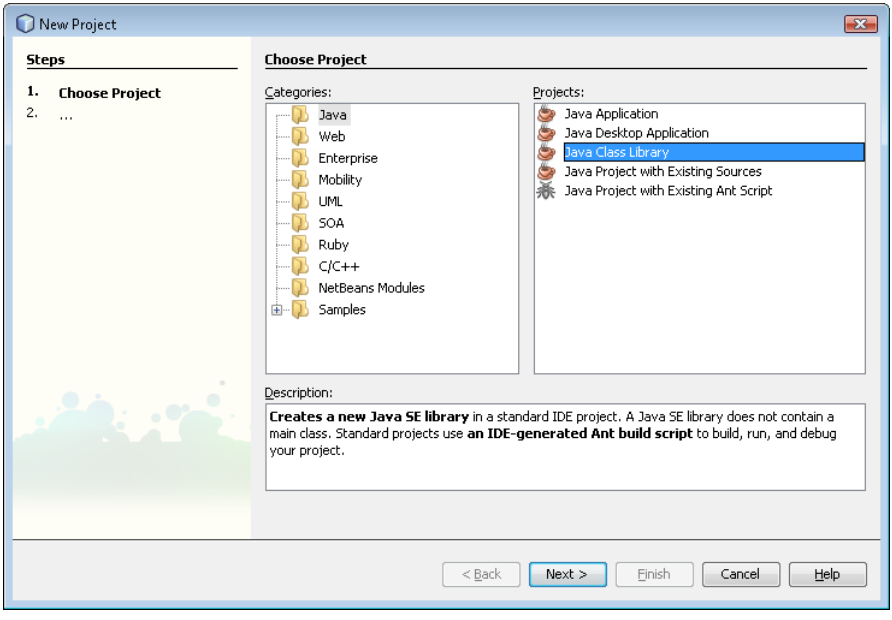

Рис. 1.1. Окно выбора проекта

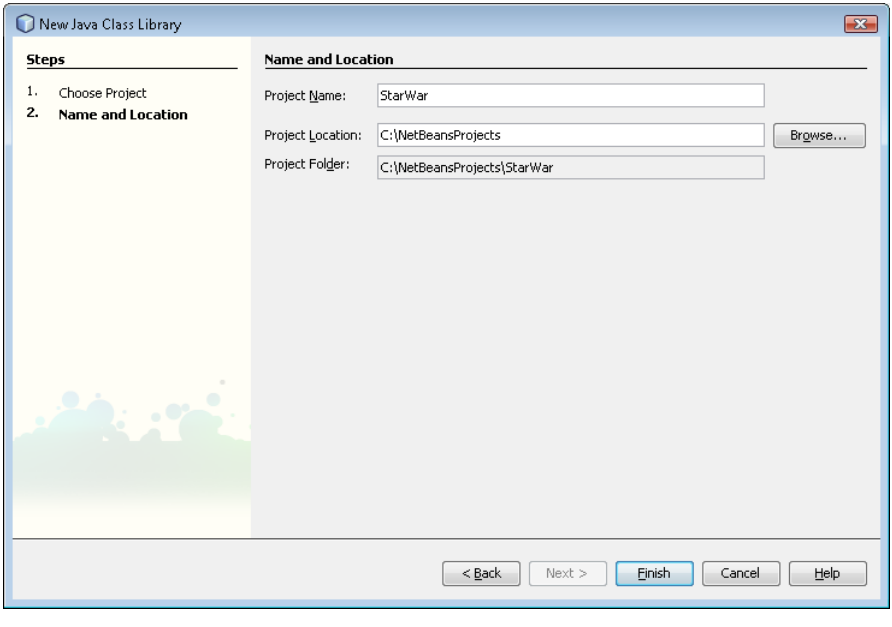

Рис. 1.2. Окно ввода параметров проекта

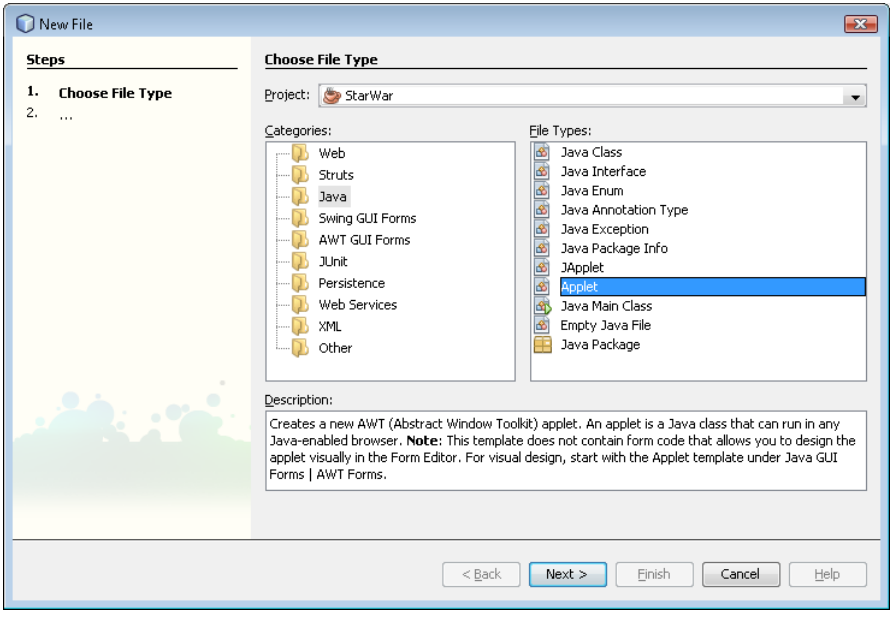

Рис. 1.3. Окно выбора компонентов проекта

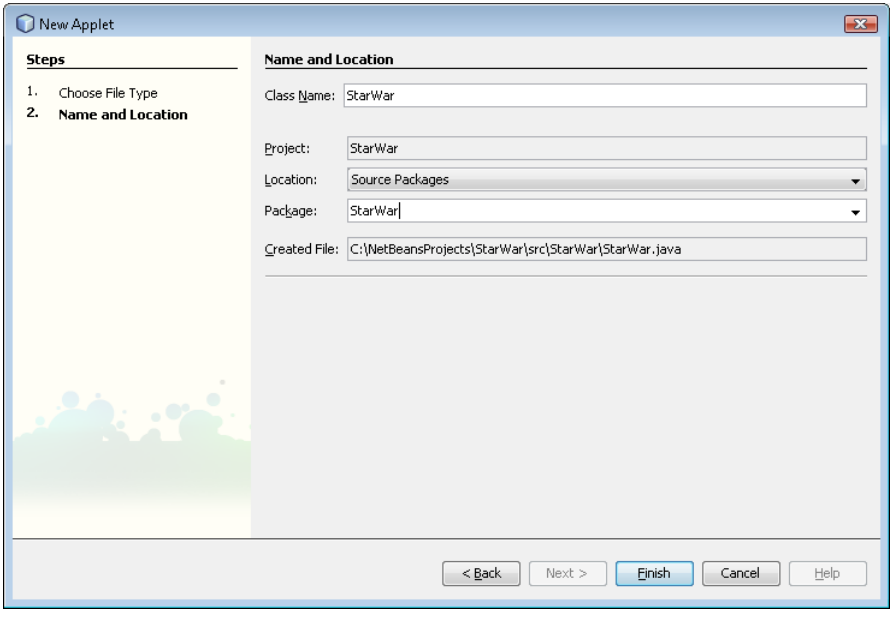

Рис. 1.4. Окно ввода параметров класса и пакета

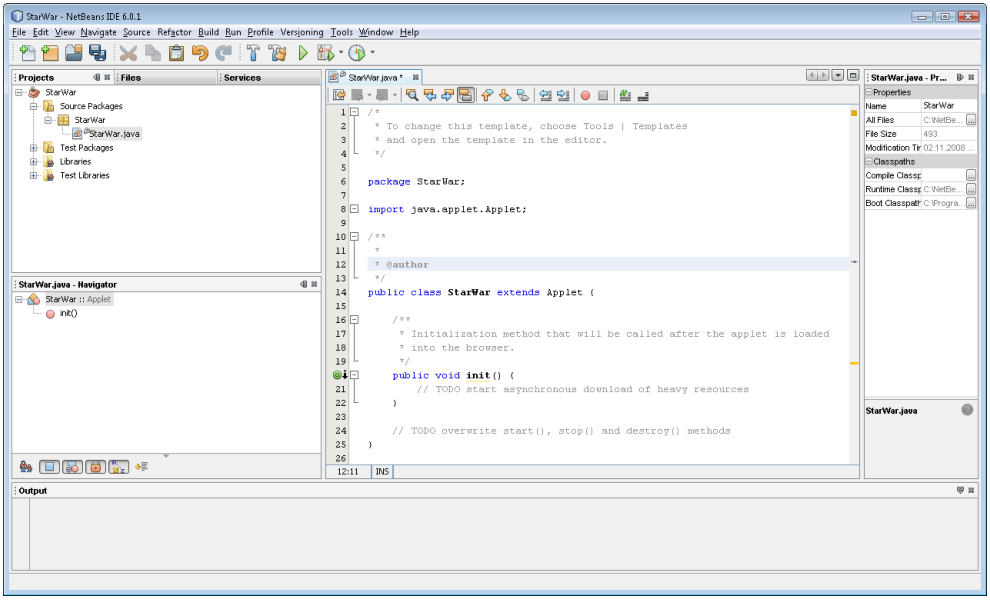

Рис. 1.5. Окно редактора исходного кода

Листинг 1.1. Объявление пакета и класса апплета

```
package StarWar;
import java.applet.Applet;
public class StarWar extends Applet {
    public void init() {
                        }
}
```

В сгенерированном коде строка `import java.applet.Applet;` обеспечивает употребление в коде короткого имени класса `Applet`, принадлежащего пакету `java.applet`. Строка `public class StarWar extends Applet{}` объявляет наш новый класс `StarWar`, расширяющий класс `Applet`. Как уже было сказано, главный класс любого апплета должен быть подклассом класса `Applet`, служащего интерфейсом между апплетом и браузером. Строка `public void init(){}` предназначена для переопределения метода `init()` класса `Applet`.

Роль метода `init()` заключается в том, что он всегда вызывается автоматически браузером для выполнения начальной инициализации апплета.

Теперь необходимо дополнить сгенерированный средой NetBeans код для того, чтобы получить апплет, представляющий создаваемую нами игру.

В качестве первого шага выведем изображение космического корабля в окно апплета (листинг 1.2):

1. Воспользуемся классом `Image` пакета `java.awt`. Абстрактный класс `Image` является суперклассом для всех классов, представляющих графические изображения. Импортируем пакет `java.awt`, тем самым обеспечив употребление коротких имен классов этого пакета.
2. Затем внутри нашего класса `StarWar` объявим объекты класса `Image`, которые будут представлять космический корабль и пламя его двигателя.
3. Вызовем метод `getImage()` класса `Applet`, возвращающий объект `Image`, представляющий изображение, расположенное по адресу, который может быть получен с помощью метода `getCodeBase()` класса `Applet`. Метод `getCodeBase()` возвращает URL-адрес каталога, в котором находится код апплета. Вызовем метод `getImage()` в теле метода `init()`, инициализирующем апплет, при этом воспользуемся ключевым словом `this`, для того чтобы отнести вызываемые методы к описываемому нами классу `StarWar`. В имени графических файлов папка StarWar указывается, потому что был создан одноименный пакет класса, если пакет не создается, папку в имени указывать не надо.
4. Для прорисовки изображений в окне апплета необходимо переопределить метод `paint()`, который наследуется классом `Applet` от класса `Container` пакета `java.awt`. Класс `Container` является суперклассом для всех классов, представляющих компоненты-контейнеры, содержащие другие графические компоненты. В качестве аргумента метода `paint()` выступает объект класса `java.awt.Graphics`. Класс `Graphics` — абстрактный класс для работы с графическим контекстом, представляющим информацию об области изображения, цвете и шрифте. В методе `paint()` зададим черный цвет фона окна апплета при помощи метода `setBackground()`, унаследованного классом `Applet` от класса `java.awt.Component`. В качестве аргумента метода назначим объект класса `java.awt.Color` с определенным полем (`Color.black`). Затем вызовем метод класса `Graphics` — `drawImage()`, аргументами которого выступают: объект `Image`, координаты области, в которой будет нарисовано изображение, и ссылка на объект класса, реализующего интерфейс `ImageObserver`, в нашем случае это сам класс `Applet`, поэтому мы укажем ключевое слово `this`. Интерфейс `ImageObserver` используется для получения информации о процессе конструирования изображения.
5. Чтобы определить при вызове метода `drawImage()` координаты области рисования, привязанные к размерам окна апплета, создадим объект класса `Dimension`, объединяющий такие свойства компонента, как ширина и высота. Затем в методе `init()` присвоим созданному объекту класса `Dimension`

размеры окна апплета при помощи метода `getSize()` класса `Applet`, унаследованного им от класса `Component`. И наконец, определим координаты области рисования с помощью методов класса `Dimension` — `height()` и `width()`, возвращающих, в нашем случае, высоту и ширину окна апплета.

6. В вышеуказанном коде мы переопределяем два метода: `init()` и `paint()`. Проинформируем об этом компилятор с помощью необязательной аннотации `@Override`, которую используют, чтобы избежать ошибок. Если аннотация указана и метод не переопределяет ни одного метода суперкласса, то компилятор выдаст ошибку. Например, ошибка возникнет, если мы вместо `paint()` укажем `Paint()` и, таким образом, создадим новый метод, вместо того чтобы переопределить метод `paint()`.

Листинг 1.2. Вывод изображения космического корабля в окно апплета

```
import java.applet.Applet;
import java.awt.*;
public class StarWar extends Applet {
  Image ship;
  Image fire;
  Dimension d;
  @Override
  public void init() {
    // TODO start asynchronous download of heavy resources
    ship=this.getImage(this.getCodeBase(),"StarWar/ship.gif");
    fire=this.getImage(this.getCodeBase(),"StarWar/fire.gif");
    d=this.getSize();
  }
  // TODO overwrite start(), stop() and destroy() methods
  @Override
  public void paint(Graphics g){
    this.setBackground(Color.Black);
    g.drawImage(ship,d.height/2,d.width-80,this);
    g.drawImage(fire,d.height/2,d.width-40,this);
  }
}
```

Для того чтобы посмотреть результаты нашего труда в окне браузера, соберем наш проект. В окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на названии проекта и выберем пункт **Build**. При сборке среда NetBeans использует инструмент ANT проекта Apache (**http://ant.apache.org/**), который обрабатывает файл build.xml NetBeans-проекта, вызывая компилятор `javac` комплекта разработки JDK. В результате сборки в

каталоге с именем проекта StarWar создается папка build\classes\StarWar, куда переносятся все ресурсные файлы из папки src\StarWar, и создается файл StarWar.class. Кроме того, в каталоге проекта создается папка dist, содержащая специальный исполняемый архивный Java-файл с расширением jar, включающий в себя ресурсные файлы и файлы байт-кода с расширением class.

Далее с помощью простейшего редактора — Блокнота — создадим файл HTML-страницы, содержащей апплет (листинг 1.3). Тег `<applet></applet>` дает браузеру указание на загрузку апплета. Атрибут `CODE` задает имя файла с байт-кодом главного класса апплета. Папка StarWar указана потому, что класс апплета создан в пакете StarWar. Атрибут `ARCHIVE` определяет архивные JAR-файлы, содержащие ресурсные и класс-файлы апплета, в целях предварительной загрузки. Атрибуты `WIDTH` и `HEIGHT` задают размеры окна апплета. Созданный HTML-файл поместим в папку dist проекта StarWar. Применение тега `<applet>` не зависит от того, какой Web-браузер вы используете. Существуют еще два тега — `<OBJECT>` и `<EMBED>`, применяемые для загрузки апплетов в браузерах Internet Explorer и Mozilla соответственно.

Листинг 1.3. Код включения апплета в HTML-страницу

```
<applet CODE="StarWar/StarWar.class" ARCHIVE="StarWar.jar"
        WIDTH="500" HEIGHT="500">
</applet>
```

Дважды щелкнув мышью на файле StarWar.html, мы увидим результаты своего труда. Внизу браузера, в строке состояния мы обнаружим, что апплет загрузился.

Второй способ запуска апплета — в окне **Projects** среды NetBeans в папке Source Package проекта StarWar щелкнем правой кнопкой мыши на файле StarWar.java и выберем пункт **Run File**. При этом запустится инструмент Applet Viewer комплекта разработки JDK, который отобразит наш апплет. Правда, в этом случае без сборки проекта не будет создан файл StarWar.jar в папке dist.

В качестве следующего шага сделаем так, чтобы космический корабль можно было перетаскивать по окну апплета с помощью мыши, а пламя корабля ожило. Для этого мы должны организовать два потока. В одном потоке должно непрерывно перерисовываться окно апплета, и, таким образом, мы увидим в нем изменения, а в другом потоке должны обрабатываться щелчки и движения мыши (листинг 1.4).

1. Для организации двух потоков используем интерфейс `Runnable` пакета `java.lang`, который реализуем в нашем классе с помощью ключевого слова

implements в заголовке класса. Интерфейс Runnable используется, если необходимо создать поток, работающий одновременно с основным потоком приложения. Внутри нашего класса переопределим метод run() интерфейса Runnable, при этом код апплета и код метода run() будут работать одновременно как разные потоки.

2. Создание и запуск потока выполним в методе start() класса Applet, переопределенном в теле нашего класса. Метод start() вызывается браузером после метода init(), и каждый раз, когда апплет возобновляется на Web-страничке. Например, если вы перейдете в браузере на новую Web-страничку, а потом вернетесь к страничке с апплетом, то браузером будет вызван метод start(). После создания и запуска потока управление потоком передается методу run(). Поток будем останавливать в переопределенном методе stop() класса Applet. Метод stop() вызывается браузером всякий раз, когда страничка, содержащая апплет, замещается другой Web-страничкой.
3. Для непрерывной перерисовки изображения в окне апплета мы в методе run() организуем бесконечный цикл, в котором будем вызывать метод repaint(), унаследованный классом Applet от класса Component пакета java.awt. Метод repaint() вызывается для принудительной перерисовки изображения или его части в окне апплета, в отличие от метода paint(), который отвечает за целостность изображения в окне апплета. Метод paint() вызывается самим браузером при первоначальной прорисовке изображения в окне апплета и затем в случае, если изображение или его часть были повреждены, например, в результате наложения окон. Тот же самый эффект непрерывной перерисовки изображения в окне апплета в бесконечном цикле, организованном в методе run(), мы могли бы добиться с помощью метода update(), унаследованного классом Applet от класса Container, если бы передавали ему в качестве аргумента объект класса Graphics, представляющий текущий графический контекст. Данный объект Graphics можно получить методом getGraphics(), унаследованным классом Applet от класса Component (листинг 1.5).
4. Для создания эффекта мигания пламени корабля введем флаг, значение которого будет периодически меняться с 0 на 1, и, в зависимости от этого, изображение пламени то будет прорисовываться, то нет. Кроме того, введем задержку в потоке, который управляется методом run() с помощью метода sleep() класса Thread, где в качестве аргумента укажем количество миллисекунд задержки. Так мы добьемся лучшего изображения. Вызов метода sleep() необходимо произвести в блоке перехвата исключений try{} catch(){}, где оператор catch обрабатывает исключительную ситуацию. В случае возникновения исключительной ситуации информация

о ней будет содержаться в объекте класса `InterruptedException` пакета `java.lang`.

5. Для обработки щелчков и движений мыши реализуем в нашем классе интерфейсы `MouseListener` и `MouseMotionListener` пакета `java.awt.event`. Интерфейс `MouseListener` предназначен для обработки таких событий, как нажатие и освобождение кнопки мыши, щелчок мыши, вхождение и уход курсора мыши из компонента. Интерфейс `MouseMotionListener` предназначен для обработки таких событий, как движения курсора мыши с нажатой кнопкой и без нажатой кнопки на компоненте.
6. Так как мы реализуем данные интерфейсы в нашем классе, то должны переопределить все методы этих интерфейсов. Переопределим все методы интерфейсов `MouseListener` и `MouseMotionListener` с пустыми фигурными скобками, т. к. они нам пока не нужны, кроме одного — `mouseDragged()` — метода, обрабатывающего движение курсора мыши с нажатой кнопкой. В качестве аргумента метода выступает объект класса `MouseEvent` — класса для работы с информацией о событиях, инициированных мышью.
7. В методе `mouseDragged` получим координаты мыши с помощью методов `getX()` и `getY()` класса `MouseEvent` и присвоим их предварительно объявленным глобальным переменным `x_ship` и `y_ship` — координатам изображения корабля. Таким образом, мы привяжем координаты изображения корабля к координатам курсора мыши с нажатой кнопкой и сможем перетаскивать изображение корабля по окну апплета с помощью мыши.
8. Однако все это у нас работать не будет, пока мы не объявим в методе `init()`, что наш класс `StarWar` является получателем событий, инициированных мышью. Сделать это необходимо с помощью методов, унаследованных классом `Applet` от класса `Component` — `addMouseListener` и `addMouseMotionListener`.

Листинг 1.4. Перемещение космического корабля мышью и анимация пламени корабля

```
package StarWar;
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class StarWar extends Applet implements Runnable,
              MouseListener, MouseMotionListener{
  Image ship;   // объект для хранения изображения корабля
  Image fire;   // объект для хранения изображения пламени корабля
  Dimension d;  // объект для хранения размеров окна апплета
```

```
Thread this_thread; // объект для управления потоком в методе run()
int FlagFire; // флаг для рисования пламени корабля
int x_ship;   // координаты корабля
int y_ship;
@Override
public void init(){
  ship=this.getImage(this.getCodeBase(), "StarWar/ship.gif");
  fire=this.getImage(this.getCodeBase(), "StarWar/fire.gif");
  d=this.getSize();
  x_ship=d.width/2;
  y_ship=d.height-80;
  this.addMouseListener(this);
  this.addMouseMotionListener(this);
  FlagFire=0;
}
@Override
public void paint(Graphics g){
  this.setBackground(Color.BLACK);
  g.drawImage(ship,x_ship,y_ship,this);
  if (FlagFire==0)
     g.drawImage(fire, x_ship,y_ship+40, this);
}
@Override
public void start(){
  if (this_thread==null){
      this_thread=new Thread(this);
      this_thread.start();
  }
}
@Override
public void run(){
  while (true){
    if (FlagFire==0) FlagFire=1;
    else FlagFire=0;
    repaint();
    try{
      this_thread.sleep(100);
    }
    catch(InterruptedException e){
      System.out.println(e);
    }
  }
}
```

```
  @Override
  public void stop(){
    if (this_thread!=null) {
       this_thread.interrupt();
       this_thread = null;
    }
  }
  @Override
  public void mouseDragged(MouseEvent evt){
    x_ship=evt.getX();
    y_ship=evt.getY();
  }
  @Override
  public void mouseClicked(MouseEvent evt){}
  @Override
  public void mouseMoved(MouseEvent evt){}
  @Override
  public void mouseEntered(MouseEvent evt){}
  @Override
  public void mouseExited(MouseEvent evt){}
  @Override
  public void mousePressed(MouseEvent evt){}
  @Override
  public void mouseReleased(MouseEvent evt){}
}
```

Листинг 1.5. Использование метода `update()` для перерисовки изображения

```
Graphics g;
g=this.getGraphics();
update(g);
```

Таким образом, мы имеем мигающее пламя корабля и корабль, который можно перемещать по окну апплета при нажатой кнопке мыши.

Однако если мы поэкспериментируем с параметром метода `sleep()`, например, сделаем задержку равной 10 миллисекундам, то увидим, что изображение корабля мигает. Происходит это из-за того, что при вызове метода `repaint()` в методе `run()` автоматически происходит обращение к методу `update()`, в котором, в свою очередь, перед вызовом метода `paint()` происходит очистка экрана с помощью метода `clearRect()`. Вот эту очистку мы и видим в виде мерцания.

Для того чтобы устранить мерцание, необходимо создавать графическое изображение окна апплета в оперативной памяти, затем в методе `paint()` цели-

ком выводить его на экран, при этом переопределив метод `update()` таким образом, чтобы устранить очистку экрана (листинг 1.6).

1. С этой целью в нашем классе определим метод `BufferImage()`, в теле которого будем создавать в оперативной памяти буфер — объект класса `Image`, воспользовавшись методом `createImage()`. Данный метод наследуется классом `Applet` от класса `Component`.
2. Затем создадим графический контекст для буферного изображения с помощью метода `getGraphics()`, в котором и будем формировать изображение окна апплета. Методом `setColor()` зададим цвет фона, методом `fillRect()` заполним им окно апплета.
3. Далее загрузим изображения корабля и пламени корабля. Метод `BufferImage()` будем вызывать в методе `run()` перед вызовом метода `repaint()`, который будет обращаться к переопределенному методу `update()`. В методе `update()` мы будем только вызывать метод `paint()`, таким образом, устранив автоматическую очистку экрана.
4. И наконец, в методе `paint()` мы будем целиком выводить на экран буферное изображение методом `drawImage()`.

Листинг 1.6. Устранение мерцания изображения

```
package StarWar;
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class StarWar extends Applet implements Runnable,
           MouseListener, MouseMotionListener{
  Image ship;   // объект для хранения изображения корабля
  Image fire;   // объект для хранения изображения пламени корабля
  Image bufferImage;  // объект для хранения буферного изображения
  Dimension d;        // объект для хранения размеров окна апплета
  Thread this_thread; // объект для управления потоком в методе run()
  int FlagFire; // флаг для рисования пламени корабля
  int x_ship;   // координаты корабля
  int y_ship;
  @Override
  public void init(){
    ship=this.getImage(this.getCodeBase(), "StarWar/ship.gif");
    fire=this.getImage(this.getCodeBase(), "StarWar/fire.gif");
    d=this.getSize();
    x_ship=d.width/2;
    y_ship=d.height-80;
```

```
    this.addMouseListener(this);
    this.addMouseMotionListener(this);
    FlagFire=0;
  }

  public void BufferImage(){
    bufferImage=this.createImage(d.width, d.height);
    Graphics g_buffer=bufferImage.getGraphics();
    g_buffer.setColor(Color.BLACK);
    g_buffer.fillRect(0,0,d.width,d.height);
    g_buffer.drawImage(ship,x_ship,y_ship,this);
    if (FlagFire==0)
       g_buffer.drawImage(fire, x_ship,y_ship+40, this);
  }
  @Override
  public void paint(Graphics g){
    g.drawImage(bufferImage,0,0,this);
  }
  @Override
  public void update(Graphics g){
    paint(g);
  }
  @Override
  public void run(){
    while (true){
      if (FlagFire==0)FlagFire=1;
        else FlagFire=0;
        BufferImage();
        repaint();
        try {
          this_thread.sleep(10);
        }
        catch(InterruptedException e){
          System.out.println(e);
        }
      }
    }
. . . . . .
```

Далее мы введем в нашу игру врагов, которые будут уничтожаться огнем лазера корабля. При уничтожении враг будет взрываться, а при столкновении врага с кораблем взрываться будет корабль. Будем подсчитывать количество уничтоженных врагов и введем количество жизней корабля. Игра будет за-

канчиваться при обнулении количества жизней корабля. Кроме того, разукрасим фон окна апплета звездами.

Примечание

Окончательный код апплета см. в файле StarWar.java папки Часть_I\Глава_1 \StarWar\src\StarWar на прилагаемом к книге компакт-диске.

1. Будем использовать пакет `java.util` для работы с классами `Date` и `Random`. Класс `Date` предназначен для работы со временем. Мы будем использовать из этого класса метод `getTime()`, который возвращает количество миллисекунд, прошедших с 1 января 1970 года и до настоящего времени. Этот метод, совместно с методом `nextInt()` класса `Random`, даст нам возможность организовать случайное появление во времени врагов в окне апплета. Класс `Random` используется для генерации последовательности псевдослучайных чисел. Метод `nextInt(int n)` данного класса возвращает псевдослучайное число в диапазоне от 0 до `n`.
2. Используем интерфейс `AudioClip` пакета `java.applet` и пакет `java.net` для загрузки звукового файла и организации звукового сопровождения в нашей игре. Мы будем использовать метод `play()` интерфейса `AudioClip` для проигрывания звукового файла. Для загрузки звукового файла применим метод `getAudioClip` класса `Applet`, в качестве аргументов которому передается URL-адрес звукового файла, поэтому нам понадобится класс `URL` пакета `java.net`. Объект класса `URL` создадим с помощью конструктора класса `URL(URL context, String spec)`, где `URL context` — адрес каталога, содержащего файл, имя которого представлено объектом `String spec`. Чтобы получить адрес каталога, применим метод `getDocumentBase()` класса `Applet`, который возвращает адрес документа, содержащего апплет. В нашем случае это файл StarWar.html. В случае применения конструктора класса `URL` необходимо обрабатывать возникновение исключительной ситуации блоком `try{} catch(MalformedURLException e){}`, где `MalformedURLException e` — объект класса для индикации возникновения ситуации неверного URL-адреса.
3. В методе `init()` строкой `enemy_array=new Image[5];` определим, что максимальное число врагов в окне апплета не будет превышать 5.
4. Там же зададим случайным образом координаты врагов на границе окна апплета с помощью метода `nextInt`.
5. Далее в теле метода `init()` привяжем координаты луча лазера к координатам корабля и строкой `NumberLifeShip=10;` зададим количество жизней корабля — 10. Количество жизней будет уменьшаться с каждым столкновением корабля с врагом. После этого зададим случайным образом координаты 100 звезд по окну апплета.

6. Определим метод `BufferImageGamePlay()` для формирования изображения сцены игры в оперативной памяти. В теле метода заполним случайным образом фон окна звездами. Методом `setColor()` класса `Graphics` установим текущий цвет — красный и методом `setFont()` того же класса текущий шрифт, после чего методом `drawString()` выведем в буфер количество убитых врагов. Объект класса `Font` пакета `java.awt`, содержащий информацию о шрифте, создается конструктором, который в качестве аргументов принимает название шрифта, его стиль и размер. Случайным образом во времени обеспечим появление изображений врагов в окне апплета. В случае если на пути луча лазера есть враг, то рисуем луч лазера до врага. Проверяем наличие врага на пути лазера с помощью определенного далее метода `CheckRange()`. Луч лазера появляется, когда соответствующий флаг `FlagLaser` устанавливается в значение, равное 1, щелчком мыши. После прорисовки луча лазера флаг снова обнуляется, и координаты лазера привязываются к координатам корабля. При наличии врага на пути лазера вместо изображения врага рисуется изображение взрыва, и счетчик количества убитых врагов увеличивается, в этот момент проигрывается звуковой файл. Пораженный враг убирается из массива `enemy_array`, который содержит число врагов в окне апплета, методом `DeleteEnemy()`. Если лазер не поражает ни одного врага, все враги прорисовываются в буфере. Если лазер не встречает на своем пути врагов, он просто прорисовывается в буфере. В случае если враг сталкивается с кораблем, вместо изображения корабля рисуется изображение взрыва, проигрывается звуковой файл, враг убирается из массива врагов, счетчик количества жизней корабля уменьшается. Попадание врага в корабль проверяется также методом `CheckRange()`. После этого происходит выход из метода оператором `return`. Если же враг не сталкивается с кораблем, то рисуется изображение корабля и пламени корабля.

7. Метод `CheckRange()` проверяет совпадение координат в диапазоне от `down` до `up`. При совпадении возвращает логическое значение `true`.

8. В методе `run()` если количество жизней обнуляется, то выводим в окно апплета буферное изображение — заставку, сформированную методом `BufferImageGameOver()`. При этом проскакиваем дальнейший код в цикле с помощью оператора `continue`. Там же обеспечиваем цикличность изображения пламени корабля флагом `FlagFire`, изменение координат врагов методом `MoveEnemy()` и координат звезд методом `MoveStar()`. После этого формируем буферное изображение сцены игры вызовом метода `BufferImageGamePlay()`.

9. Метод `MoveEnemy()` меняет координаты врагов. Приращением координаты `y` формируем эффект летящего навстречу врага. Как только враг

заходит за границу окна апплета, убираем его из массива врагов в окне методом `DeleteEnemy()`.

10. Метод `DeleteEnemy()` предназначен для удаления текущего врага из массива. Меняем местами последнего врага в массиве с текущим и уменьшаем счетчик количества врагов в окне апплета, таким образом, даем возможность появления нового врага в случайный момент времени. Устанавливаем координаты текущего врага случайным образом на границу окна апплета.
11. Метод `MoveStar()` служит для изменения координат звезд. Приращением координаты `y` создаем эффект летящих навстречу звезд. При достижении звездой границы окна апплета устанавливаем координаты звезды случайным образом вновь на верхней границе окна.
12. Метод `BufferImageGameOver()` формирует изображение заставки в конце игры, в оперативной памяти для дальнейшего вывода в окно апплета. Методом `drawString()` класса `Graphics` рисуем надпись "GAME OVER" и выводим значение счетчика количества убитых врагов.

В итоге у нас есть полная реализация задуманного, и мы создали апплет, расширяющий возможности обычной Web-странички, представляя в ее отдельной части игру "Звездные войны".

ГЛАВА 2

Создание апплетов с использованием графической библиотеки Swing

Графическая библиотека Swing и ее применение

Библиотека Swing — это Java-библиотека классов и интерфейсов, представляющих такие стандартные графические компоненты интерфейса пользователя, как кнопки, таблицы, поля ввода и др., а также методы для работы с ними. Графическая библиотека Swing позволяет создавать для Java-приложений эффективный интерфейс пользователя, отвечающий самым высоким требованиям, таким как полезность, удобство и привлекательность.

Библиотека Swing составляет основную часть библиотеки JFC (Java Foundation Classes), которая объединяет классы и интерфейсы, обеспечивающие интерфейс пользователя Swing-компонентами, выбор внешнего вида и поведения интерфейса, поддержку технологий для людей с ограниченными возможностями, работу с высококачественной двумерной графикой, текстом и изображениями, а также интернационализацию графического интерфейса пользователя.

В настоящее время библиотека Swing входит в набор настольных Java-технологий, предназначенных для создания настольных Java-приложений и апплетов. Помимо Swing данный набор содержит технологии Java Web Start, JNLP, Java Plug-In, Java 2D, Java 3D, Java Sound, Java Bindings for OpenGL (JOGL), Java Advanced Imaging (JAI), AWT (Abstract Window Toolkit), Internationalization, JavaBeans, JavaHelp System, Accessibility и JDesktop Integration Components.

Графическая библиотека Swing была создана на основе библиотеки AWT для решения таких проблем AWT, как недостаточный выбор графических компо-

нентов и зависимость внешнего вида и поведения AWT графического интерфейса пользователя от конкретной операционной системы. Эти проблемы были решены созданием классов и интерфейсов библиотеки Swing с использованием одного только языка Java.

Все классы Swing-компонентов являются наследниками базового класса `javax.swing.JComponent`. При конструировании Swing-интерфейса компоненты помещаются в контейнеры, образующие иерархию, которая берет свое начало от контейнеров верхнего уровня, представленных классами `javax.swing.JApplet`, `javax.swing.JDialog`, `javax.swing.JFrame` и `javax.swing.JWindow`.

Структуру библиотеки Swing можно посмотреть в *приложении 6*.

Основные свойства графической библиотеки Swing можно обобщить как:

- кроссплатформенность;
- механизм сменного внешнего вида и поведения компонентов (pluggable look and feel). Внешний вид и поведение графического интерфейса пользователя может определяться программным образом или выбираться при выполнении приложения;
- расширяемость за счет возможности расширения классов и интерфейсов библиотеки;
- архитектура Swing-компонентов основана на технологии JavaBeans;
- возможность настройки внешнего вида Swing-компонентов за счет изменения свойств таких элементов, как рамки, цвет, фон и др.;
- легковесность. Так как AWT-компоненты непосредственно взаимодействуют с операционной системой и представляют собой каждый маленькое индивидуальное окно, то их называют *тяжеловесными*. Компоненты Swing представляют собой области в окне, характеризующиеся координатами и размером, при этом они не работают напрямую с операционной системой, их отображение основывается на реализации Java 2D API, поэтому их называют *легковесными*. Если легковесные компоненты используются совместно с тяжеловесными, то поскольку тяжеловесные напрямую связаны с операционной системой, то они будут прорисовываться в первую очередь, поэтому в случае размещения тяжеловесных компонентов в легковесных, они будут перекрывать собой легковесные, что приведет к неверному отображению графического интерфейса;
- в библиотеке Swing автоматически заложена двойная буферизация при выводе изображения, которую мы осуществляли вручную в *главе 1*, т. е. изображение сначала полностью формируется в оперативной памяти, а затем целиком выводится на экран, таким образом устраняется эффект мерцания изображения и повышается скорость рисования;

- использование библиотеки Swing не является потоково-безопасной. Доступ к Swing-компонентам должен осуществляться в специальном потоке *Event Dispatch Thread (EDT)*;
- поддержка технологий для людей с ограниченными возможностями;
- снабжение компонентов всплывающими подсказками и механизмом управления клавиатурой;
- возможность настройки текста, отображаемого Swing-компонентами с помощью HTML-тегов;
- возможность отображения значков Swing-компонентами;
- реализация модели MVC (Model — View — Controller, Модель — Вид — Контроллер) — модели отделения данных от внешнего вида и поведения интерфейса. Многие Swing-компоненты имеют связанные интерфейсы (Model), отвечающие за доступ к данным и генерацию событий, связанных с изменением данных.

Архитектура MVC — это объектно-ориентированная архитектура пользовательских интерфейсов, состоящая из трех частей. *Модель* предназначена для хранения, изменения и получения данных графического компонента. *Вид* представляет данные на экране. *Контроллер* обеспечивает реакцию модели и вида в ответ на действия пользователя. Архитектура MVC реализована в библиотеке Swing в виде архитектуры с разделенной моделью, состоящей из двух, а не из трех частей. Вид и контроллер в архитектуре Swing объединены вместе в один элемент — представителя пользовательского интерфейса *(delegate UI)*. Класс графического компонента связывает модель и представителя с помощью менеджера пользовательского интерфейса (`UIManager`), который определяет внешний вид и поведение интерфейса *(Look and Feel)*.

Как видно, графическая библиотека Swing позволяет на своей основе создавать насыщенные или богатые, по сравнению с AWT, интерфейсы пользователей настольных Java-приложений и апплетов, которые могли бы запускаться на любой платформе. При этом внешний вид Swing-интерфейса может легко изменяться и настраиваться для конкретной операционной системы.

Использование Swing на примере создания апплета с графическим интерфейсом пользователя

В качестве примера визуального программирования в среде NetBeans с использованием компонентов Swing рассмотрим создание апплета, представляющего собой программу-собеседника — приложение, в котором пользователь вводит свой вопрос или утверждение, а программа ему отвечает.

Примечание

Файлы с кодами апплета расположены на прилагаемом к книге компакт-диске в папке Часть_I\Глава_2\AppletTalk.

Для начала работы создадим основу нашего проекта:

1. Откроем среду NetBeans и в главном меню среды разработки выберем **File | New Project**. В появившемся окне выбора типа проекта укажем **Java | Java Class Library**, что позволит среде сгенерировать проект без создания класса, содержащего статический метод `main()`, что как раз и требуется для написания апплета.
2. После этого нажмем кнопку **Next**. В появившемся окне ввода параметров проекта введем название проекта — AppletTalk и нажмем кнопку **Finish**.
3. В левом верхнем углу среды NetBeans в окне **Projects** щелкнем правой кнопкой мыши на названии нашего проекта AppletTalk. В появившемся меню выберем **New | Other**. Появится окно, в котором мы выберем **Swing GUI Forms | JApplet Form**, что позволит нам использовать визуальные средства программирования с привлечением компонентов Swing (рис. 2.1). Нажмем кнопку **Next** и введем название класса и пакета — AppletTalk, после этого нажмем кнопку **Finish** (рис. 2.2). Созданный нами класс `AppletTalk` расширяет класс `javax.swing.JApplet`, обеспечивающий поддержку графической библиотеки Swing.

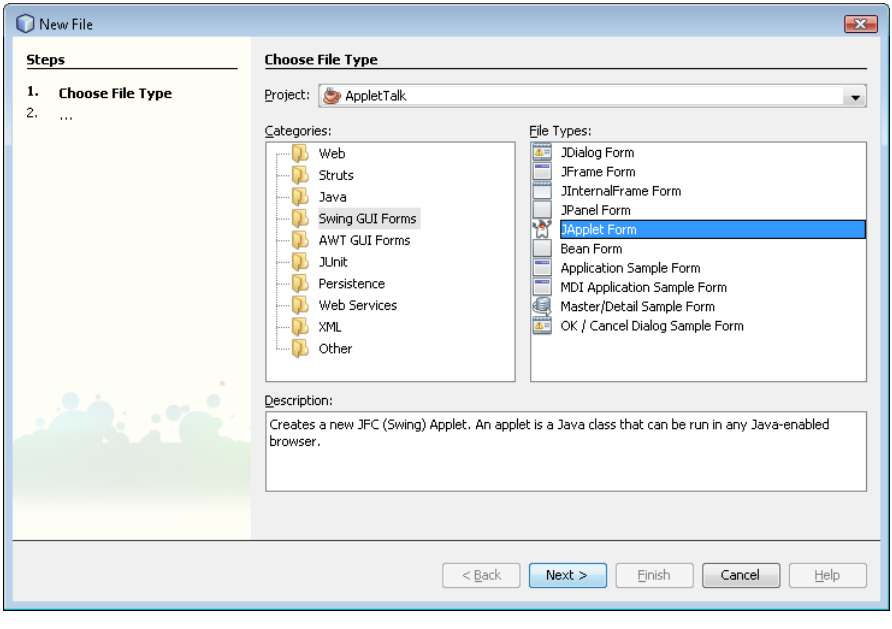

Рис. 2.1. Окно выбора компонентов проекта

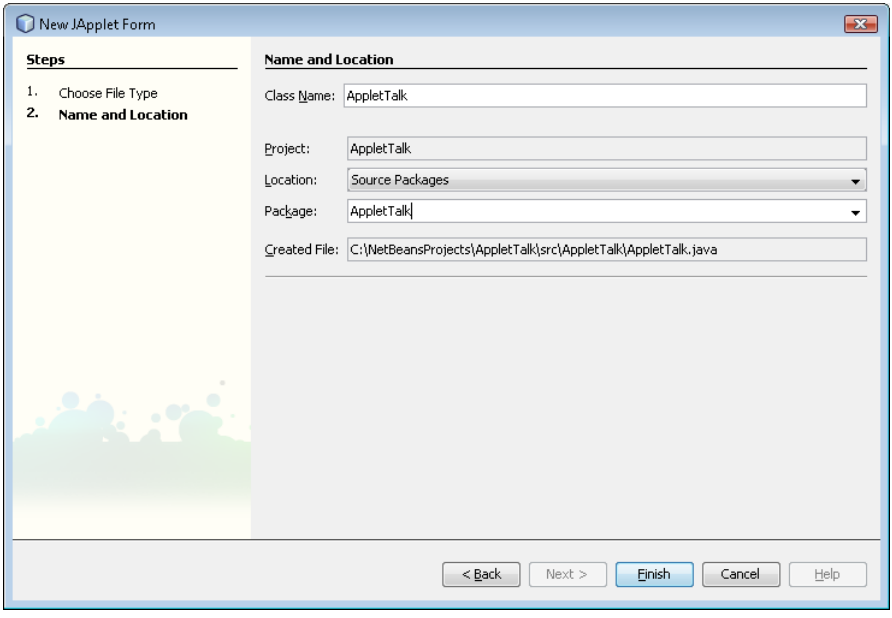

Рис. 2.2. Окно ввода параметров класса и пакета

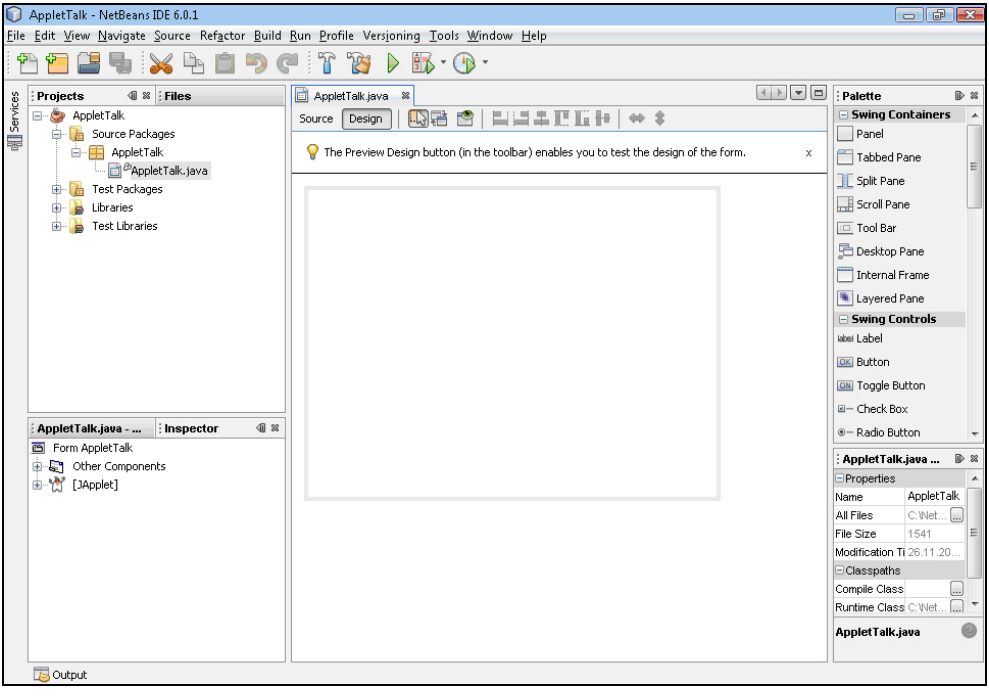

Рис. 2.3. Окно визуального редактора

В результате появится окно визуального редактора среды NetBeans, отображающее форму нашего апплета, в которой мы сможем размещать различные графические компоненты (рис. 2.3).

Теперь нам необходимо произвести насыщение создаваемого нами апплета графическими компонентами интерфейса пользователя:

1. Справа в окне палитры графических компонентов **Palette** найдем элемент кнопки **Button** и, нажав на него левой кнопкой мыши, перенесем его в окно апплета (рис. 2.4). Нажатием этой кнопки пользователь будет заканчивать ввод фразы в создаваемой нами программе-собеседнике.

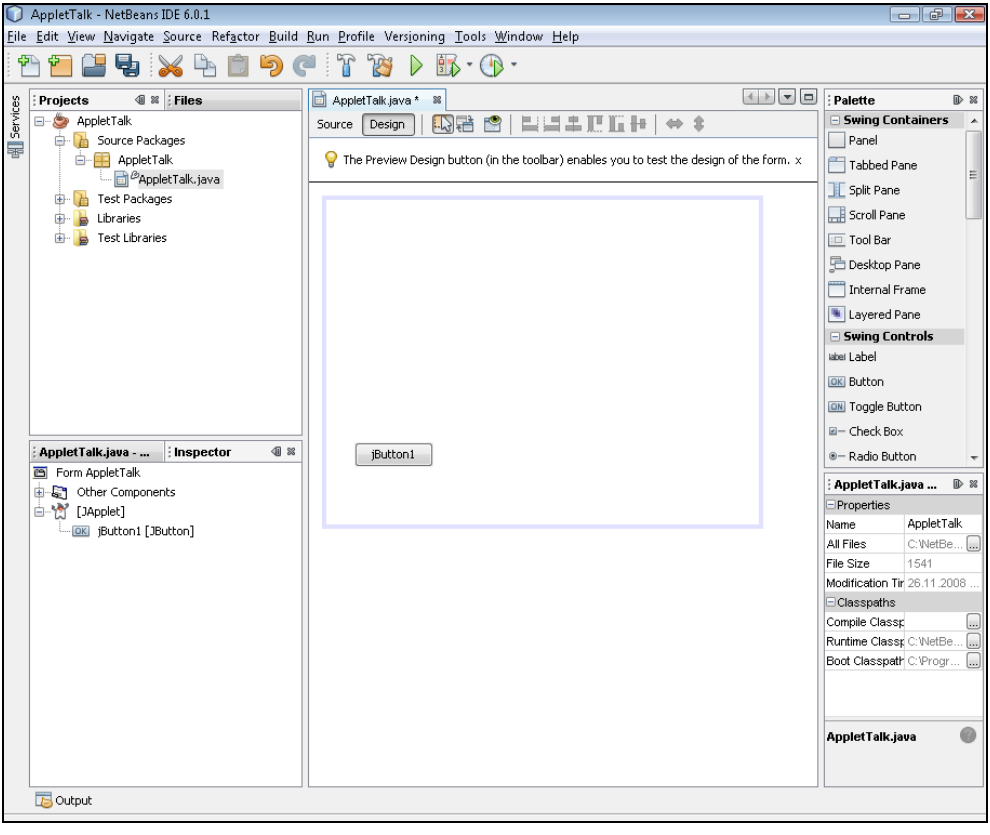

Рис. 2.4. Первый шаг создания кнопки ввода

2. Слева, в окне **Inspector**, где появляются выбранные нами компоненты, щелкнем правой кнопкой мыши на компоненте **jButton1 [jButton]**. В появившемся меню выберем пункт **Properties** и увидим диалоговое окно со свойствами компонента (рис. 2.5). В строке **text** заменим `jButton1` на `ВВОД`. В результате надпись на кнопке будет изменена (рис. 2.6).

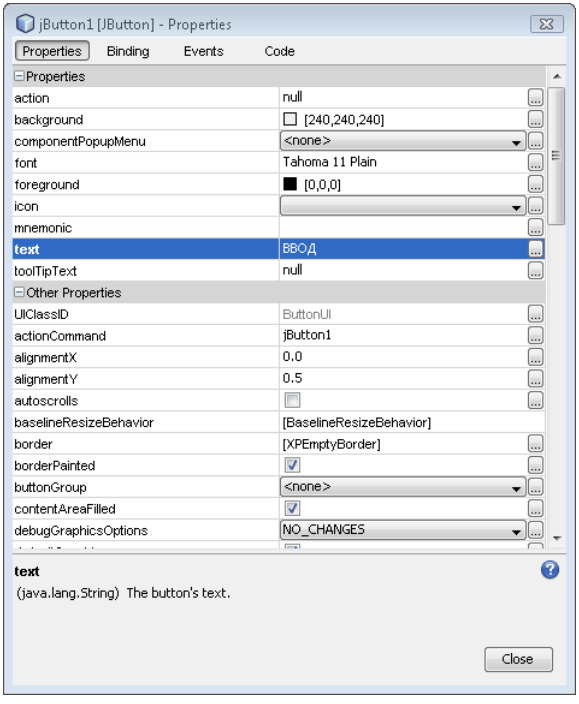

Рис. 2.5. Окно свойств кнопки

3. Еще раз щелкнем правой кнопкой мыши на компоненте **jButton1 [jButton]** в окне **Inspector** и в появившемся меню выберем пункт **Change Variable Name**, далее в диалоговом окне изменим имя кнопки (рис. 2.7).
4. Справа, в окне **Palette**, содержащем графические компоненты, выберем компонент текстового поля **Text Field**, нажав на него левой кнопкой мыши, и перенесем его в окно апплета (рис. 2.8). В это поле пользователь будет вводить свою фразу.
5. Слева, в окне **Inspector** щелкнем правой кнопкой мыши на появившемся компоненте **jTextField1 [jTextField]** и выберем пункт **Properties**. Далее в диалоговом окне свойств очистим свойство **text**, для того чтобы в этом поле не появлялся текст по умолчанию (рис. 2.9).
6. В окне **Inspector** щелкнем правой кнопкой мыши на компоненте **jTextField1 [jTextField]** и выберем пункт **Change Variable Name**, а далее в диалоговом окне изменим имя текстового поля на `jTextFieldInput`.
7. В окне компонентов **Palette** выберем графический компонент отображения текста **Text Area** и, нажав на него левой кнопкой мыши, перенесем его в окно апплета (рис. 2.10). В эту текстовую область программой будет выводиться ответ на введенную пользователем фразу.

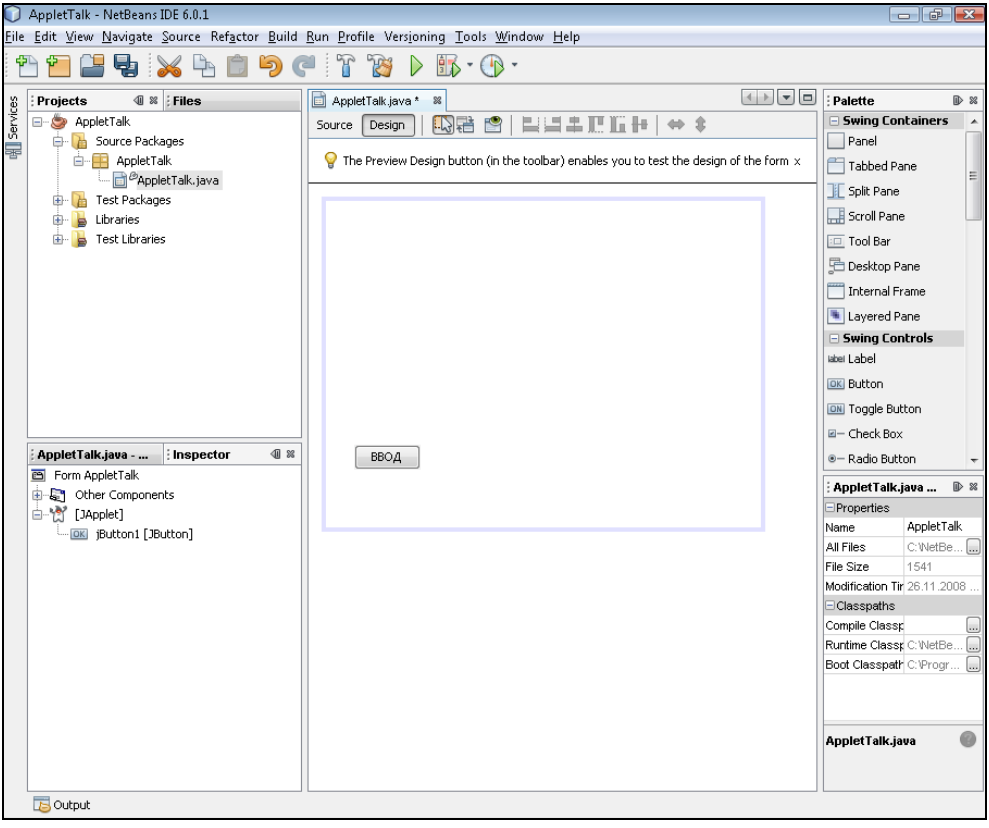

Рис. 2.6. Второй этап создания кнопки ввода

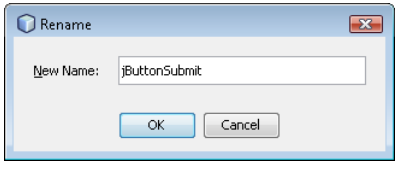

Рис. 2.7. Окно изменения имени объекта

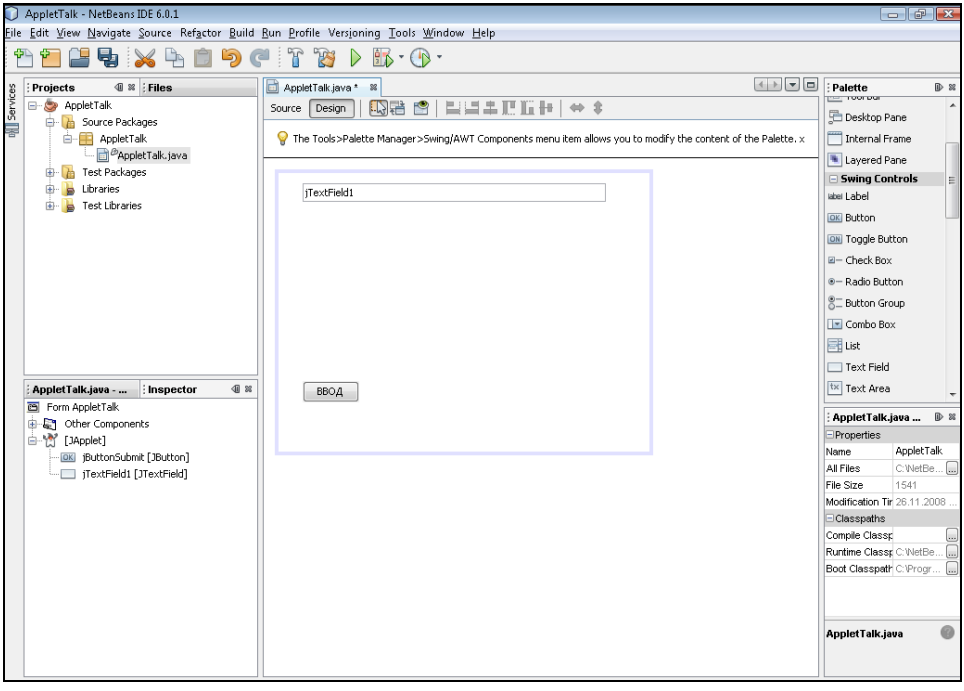

Рис. 2.8. Первый шаг создания поля ввода текста

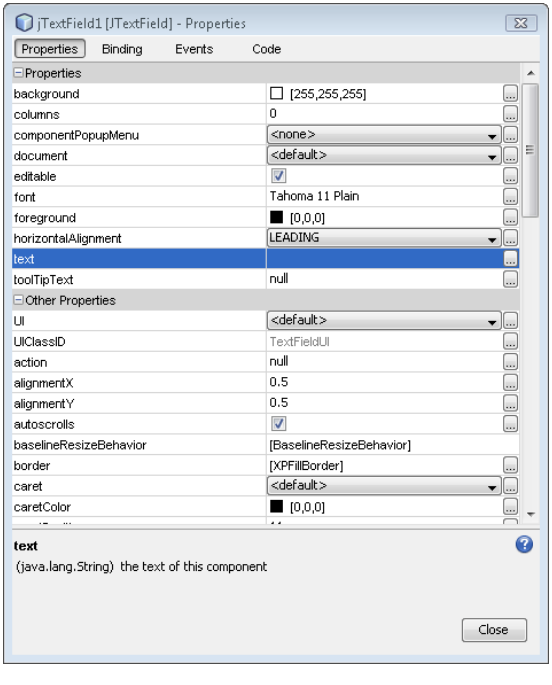

Рис. 2.9. Окно свойств текстового поля

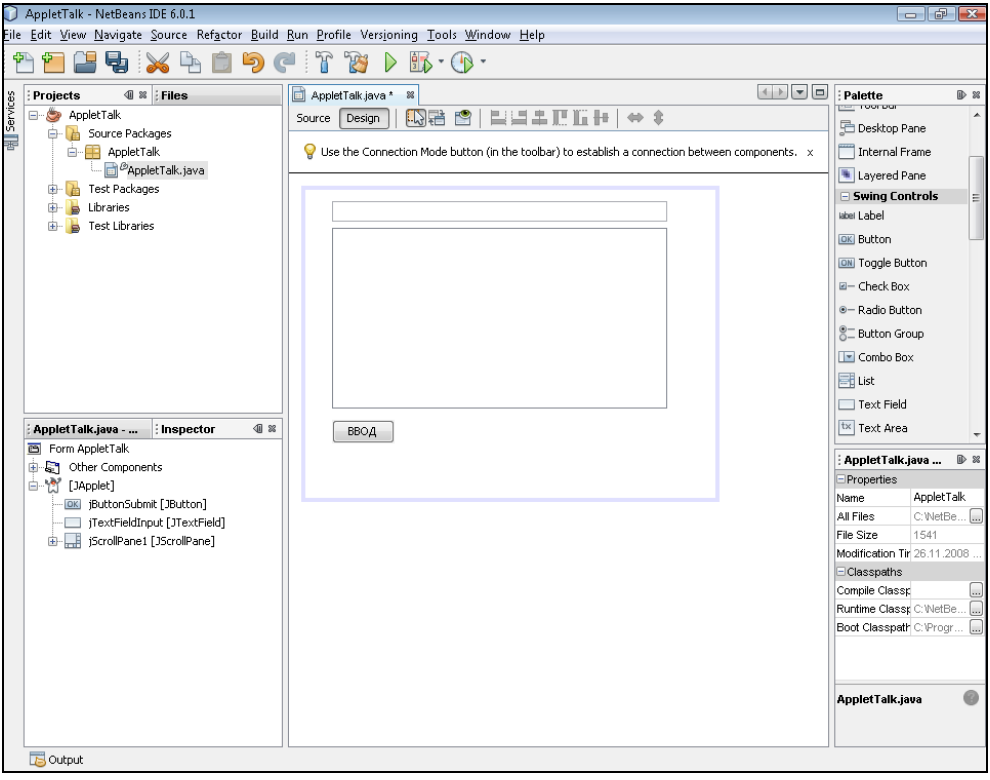

Рис. 2.10. Первый шаг создания области отображения текста

8. Слева, в окне **Inspector**, появится компонент **jScrollPane1 [jScrollPane]**, содержащий в свою очередь компонент **jTextArea1 [jTextArea]**. Компонент **jScrollPane1 [jScrollPane]** — панель с прокруткой, является контейнером, который автоматически осуществляет прокрутку при отображении содержащейся в нем, в нашем случае, области для отображения текста **jTextArea1 [jTextArea]**, причем полосы прокрутки добавляются или исчезают динамически. Щелкая правой кнопкой мыши по этим компонентам, будем выбирать пункт **Change Variable Name** и сменим имена в диалоговых окнах на `jScrollPaneOut` и `jTextAreaOut`.

9. В окне **Inspector** щелкнем правой кнопкой мыши по компоненту **jTextAreaOut** и в появившемся меню выберем пункт **Properties**. После этого увидим диалоговое окно, в котором для свойств **lineWrap** и **wrapStyleWord** поставим флажки для того, чтобы установить перенос слов в строках по границе слова (рис. 2.11).

10. В окне **Palette** выберем компонент переключателя **Radio Button** и перенесем два переключателя, с помощью которых пользователь программы

будет осуществлять выбор языка для разговора, в окно апплета (рис. 2.12).

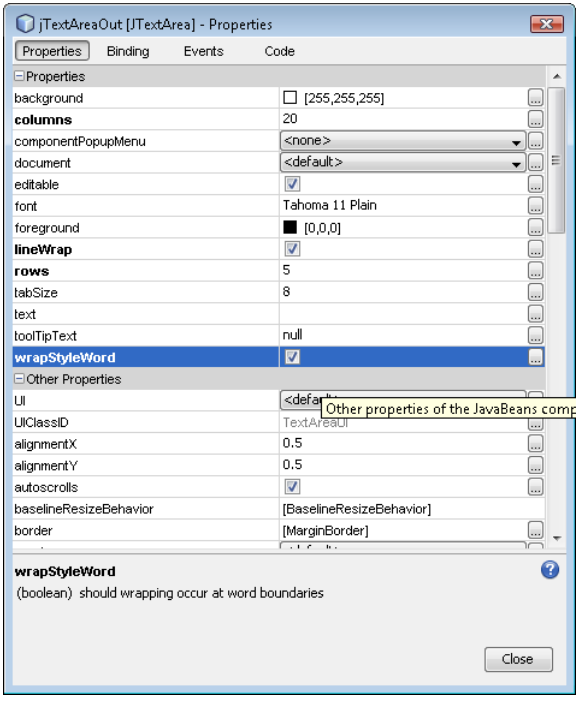

Рис. 2.11. Окно свойств области отображения текста

11. Затем слева в окне **Inspector** с помощью диалоговых окон сменим имена переключателей на `jRadioButtonRus` и `jRadioButtonEn` — выбор русского и английского языков соответственно.
12. Для того чтобы при выборе одного из переключателей обеспечить отмену выбора другого, необходимо создать *группу переключателей*, в которую добавим созданные нами переключатели. Для этого выберем в окне **Palette** компонент **Button Group** и перенесем его в окно апплета, при этом он появится в окне **Inspector** слева. Далее сменим имя группы на `buttonGroupLanguage`.
13. С помощью диалоговых окон ввода свойств компонентов сменим подписи к переключателям и добавим их в созданную нами группу (рис. 2.13 и 2.14). Переключатель с подписью "Русский язык" назначим выбором по умолчанию, для этого для свойства **selected** поставим галочку.

В итоге мы создали требуемый графический интерфейс пользователя (рис. 2.15).

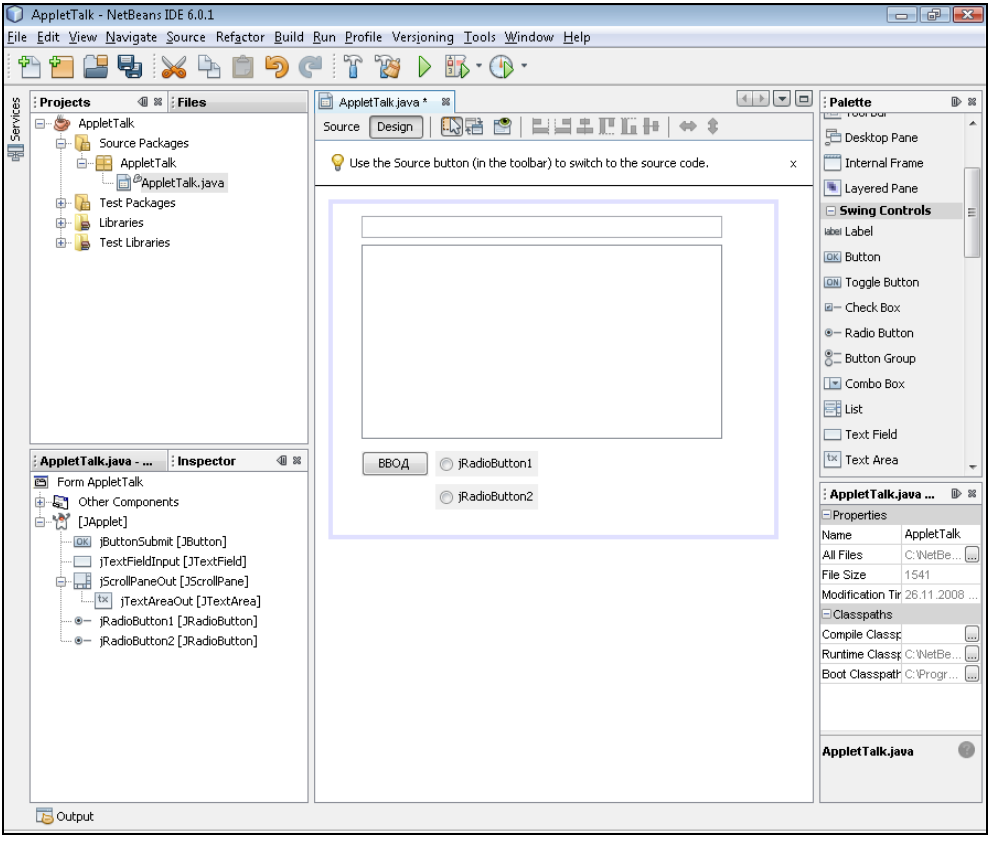

Рис. 2.12. Первый шаг создания переключателей

Теперь можно посмотреть, как будет выглядеть созданный нами графический интерфейс в работе. Для этого в окне **Projects** среды NetBeans в узле **Source Packages** найдем файл AppletTalk.java, щелкнем по нему правой кнопкой мыши и в появившемся контекстном меню выберем пункт **Run File**. В результате средой NetBeans будет запущен инструмент JDK Applet Viewer, который отобразит апплет (рис. 2.16).

Графический интерфейс апплета мы создали, теперь нужно наполнить его содержимым. Я напомню, что мы собрались написать программу-собеседника, т. е. на введенную пользователем в верхнем текстовом поле фразу программа должна что-то отвечать. Сделаем так, чтобы программа отвечала нам афоризмами Козьмы Пруткова при выборе переключателя "Русский язык". Для этого возьмем текстовый файл с афоризмами Козьмы Пруткова и поместим его в папку C:\NetBeansProjects\AppletTalk\src\AppletTalk \КозьмаПрутков.txt, содержащую исходные файлы нашего проекта.

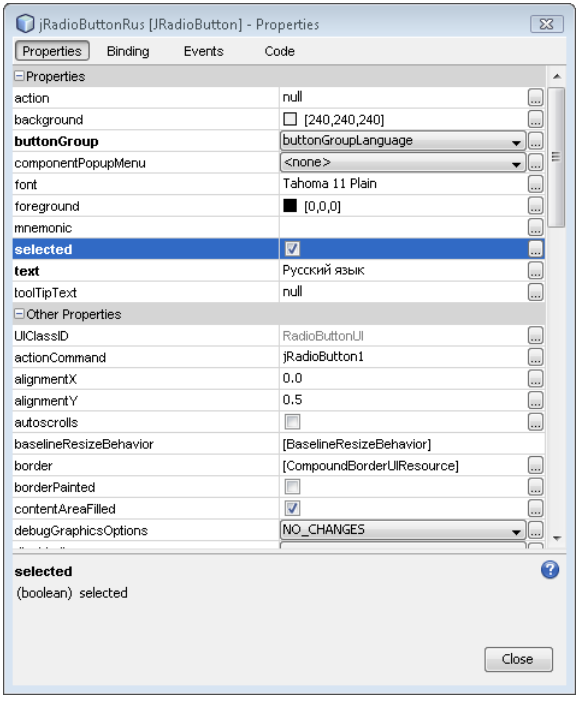

Рис. 2.13. Окно свойств переключателя `jRadioButtonRus`

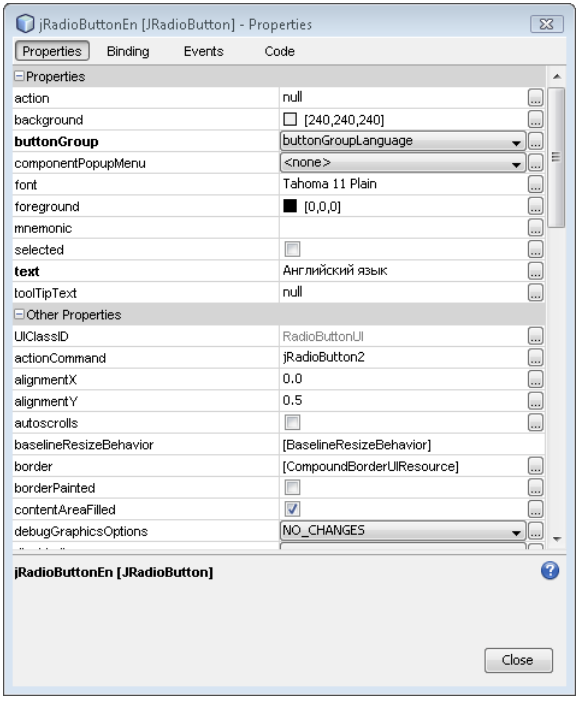

Рис. 2.14. Окно ввода свойств переключателя `jRadioButtonEn`

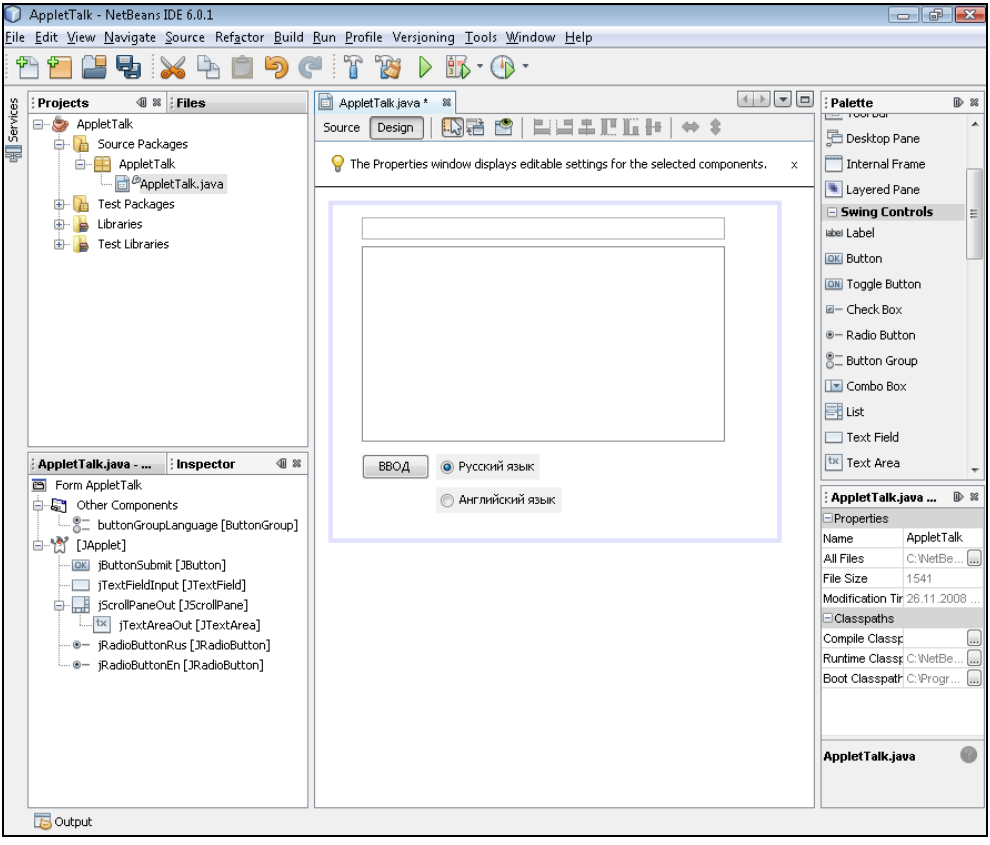

Рис. 2.15. Окончательный графический интерфейс пользователя в визуальном редакторе среды NetBeans

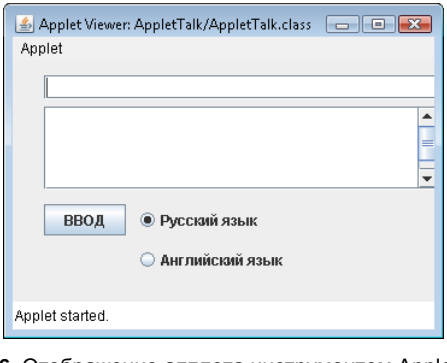

Рис. 2.16. Отображение апплета инструментом Applet Viewer

При выборе переключателя **Английский язык** программа будет отвечать нам английскими поговорками. Поэтому возьмем тексты английских поговорок и сохраним их в файле EnglishText.txt в той же папке, что и афоризмы Козьмы Пруткова.

Далее нам необходимо организовать чтение из этих файлов. Сначала кратко рассмотрим классы, предназначенные в языке Java для организации ввода/вывода.

Необходимые нам классы содержатся в пакете `java.io`. Коллекция классов пакета `java.io` представлена в табл. П7.1 *(см. приложение 7)*.

Нам необходимо выбрать из этого многообразия именно тот класс, который будет полезен для решения нашей задачи. Для начала определимся, какие потоки нас интересуют — символьные или байтовые. Байтовые потоки, как правило, используются для работы с такими двоичными данными, как звук и изображения. Символьные потоки обычно служат для чтения и записи текстовой информации, поэтому мы будем работать с этими потоками. Так как нам необходимо считывать информацию из файла, будем использовать класс `FileReader`. Для повышения скорости будем читать, используя буфер в оперативной памяти, поэтому дополнительно используем класс `BufferedReader`.

Вернемся к среде NetBeans и созданному нами графическому интерфейсу.

Так как все события, происходящие в нашей программе, будут связаны с кнопкой **ВВОД**, то мы должны добавить в программу метод, обрабатывающий нажатие этой кнопки. Для этого щелкнем правой кнопкой мыши на изображении кнопки **ВВОД** и в появившемся контекстном меню выберем **Events | Action | actionPerformed**. В результате в редакторе среды NetBeans мы автоматически попадем в созданный средой метод, в который введем свой код (листинг 2.1):

1. Сделаем так, чтобы программа отвечала нам только в том случае, если мы введем свою фразу в текстовое поле графического интерфейса. Для этого воспользуемся методом `getText()`, который класс `JTextField` наследует от класса `JTextComponent` пакета `javax.swing`. Этот метод возвратит нам строку, введенную в текстовое поле. Далее будем сравнивать ее с пустой строкой с помощью метода `compareTo()` класса `String` пакета `java.lang`.
2. Очистим текстовую область и текстовое поле при нажатии кнопки **ВВОД**, воспользовавшись методом `setText()` класса `JTextComponent`, который является базовым для используемых нами классов `JTextArea` и `JTextField`.
3. Так как и афоризмы Козьмы Пруткова и английские поговорки размечены номерами, то организуем случайный выбор афоризма или поговорки с помощью метода `nextInt()` класса `Random` пакета `java.util`. Здесь `NumberSentence` — общее число афоризмов или поговорок, увеличенное на

единицу, т. к. метод `nexInt()` включает ноль, но не включает максимальное число в выбор.

4. Исключим нулевой номер афоризма или поговорки, т. к. нумерация начинается с 1.
5. Текст афоризмов или поговорок будем хранить в объекте `bufferText` класса `String`. После случайного выбора номера афоризма или поговорки организуем поиск этого номера в тексте с помощью метода `indexOf()` класса `String`, который возвратит нам позицию номера афоризма или поговорки в тексте. Найдем позиции начала искомого афоризма или поговорки и его конец, ассоциированные соответственно с данным номером и со следующим номером, и сохраним их в переменных `beginIndex` и `endIndex`.
6. Установим шрифт вывода текста в текстовую область графического интерфейса при помощи метода `setFont()` класса `Font` пакета `java.awt`.
7. Выведем афоризм или поговорку в текстовую область, воспользовавшись методом `substring()` класса `String`, который возвращает подстроку строки с начальной позиции, в нашем случае — `beginIndex+4` (увеличиваем на 4 для того, чтобы не выводить сам номер афоризма или поговорки), и до конечной позиции, в нашем случае — `endIndex` (позиция номера следующего афоризма или поговорки). Вывод подстроки в текстовую область осуществим с помощью метода `append()` класса `JTextArea` пакета `javax.swing`, который добавляет строку в текстовую область.

Листинг 2.1. Код метода обработки нажатия кнопки *ВВОД*

```
private void jButtonSubmitActionPerformed(java.awt.event.ActionEvent evt)
{
  // TODO add your handling code here:
  if (jTextFieldInput.getText().compareTo("")!=0){
    jTextAreaOut.setText("");
    jTextFieldInput.setText("");
    Random d=new Random();
    int pos=d.nextInt(NumberSentence);
    if (pos==0) pos++;
    String beginPos=""+pos;
    String endPos=""+(pos+1);
    int beginIndex=bufferText.indexOf(beginPos);
    int endIndex=bufferText.indexOf(endPos);
    Font font=new Font("Serif", Font.BOLD, 20);
    jTextAreaOut.setFont(font);
    jTextAreaOut.append(bufferText.substring(beginIndex+4,endIndex));
} }
```

Для того чтобы сохранить текст афоризмов или поговорок в объекте `bufferText` класса `String`, необходимо организовать чтение из соответствующих файлов. Для этого в коде найдем созданный средой NetBeans метод `init()`, который, как мы уже знаем из *главы 1*, всегда вызывается автоматически браузером для выполнения начальной инициализации апплета, и вставим свой код в этот метод (листинг 2.2):

1. Используя класс `FileReader`, мы должны обрабатывать блоком `try{}catch(){}` возникновение исключительной ситуации, описанной в классе `FileNotFoundException`, когда невозможно открыть файл по указанному пути.
2. Определим путь к файлу с афоризмами Козьмы Пруткова, сохранив его в объекте `path` класса `String`. Путь к апплету определим, используя метод `getCodeBase()`, который класс `JApplet` наследует от класса `Applet`. Метод `getCodeBase()` возвращает объект класса `URL`. Нам же нужен объект класса `String`, поэтому применим метод `toString()` класса `URL`, который преобразует объект класса `URL` в объект класса `String`. Далее нам нужно исключить из полученной строки начало `File:`. Для этого применим метод `substring(5)` класса `String`, который выделит из строки подстроку, начиная с 5-й позиции. Документ HTML, вызывающий апплет, будет расположен в одном каталоге с папкой AppletTalk, содержащей байт-код апплета и текстовые файлы, поэтому добавим к полученному абсолютному пути относительный путь к текстовому файлу КозьмаПрутков.txt.
3. Создадим потоки для чтения файла.
4. Текст афоризмов Козьмы Пруткова мы будем хранить в буферном объекте `bufferTextRus` класса `String`. Инициализируем буфер пустой строкой.
5. Организуем чтение из файла построчно, используя метод `readLine()` класса `BufferedReader`, сохраняя полученную таким образом строку в буферном объекте `bufferLine` класса `String`. Полученные из файла строки будем накапливать в буфере `bufferTextRus`. Читать будем до тех пор, пока метод `readLine()` не возвратит нам пустую строку, т. е. до конца файла. Используя метод `readLine()`, необходимо обрабатывать возникновение исключительной ситуации, описанной в классе `IOException` (ошибка в потоке), с помощью блока `try{}catch(){}`.
6. Так как у нас по умолчанию выбран русский язык соответствующим переключателем, то запишем считанный из файла текст афоризмов в буфер `bufferText`, с которым будет работать метод, обрабатывающий нажатие кнопки **ВВОД**. Количество афоризмов запишем в целочисленную переменную `NumberSentence`.

7. Все то же самое проделаем с файлом, содержащим английские поговорки, за исключением записи их в буфер `bufferText`, при этом текст поговорок будет храниться в буфере `bufferTextEn`.

Листинг 2.2. Код метода `init()`, обеспечивающий чтение из файлов

```
try {
  String path=this.getCodeBase().toString().substring(5)+
              "/AppletTalk/КозьмаПрутков.txt";
  File infile=new File(path);
  FileReader rfile=new FileReader(infile);
  BufferedReader bfile=new BufferedReader(rfile);
  bufferTextRus="";
  String bufferLine;
  try {
    while ((bufferLine=bfile.readLine())!=null){
      bufferTextRus=bufferTextRus+bufferLine;
    }
  } catch(IOException e){System.out.println(e);}
  bufferText=bufferTextRus;
  NumberSentence=161;
}
catch(FileNotFoundException e){
System.out.println(e);
}
try {
  String path=this.getCodeBase().toString().substring(5)+
              "/AppletTalk/EnglishText.txt";
  File infile=new File(path);
  FileReader rfile=new FileReader(infile);
  BufferedReader bfile=new BufferedReader(rfile);
  bufferTextEn="";
  String bufferLine;
  try {
    while ((bufferLine=bfile.readLine())!=null){
      bufferTextEn=bufferTextEn+bufferLine;
    }
  } catch(IOException e){System.out.println(e);}
}
catch(FileNotFoundException e){
  System.out.println(e);
}
```

Организуем переключение вывода афоризмов и поговорок с помощью переключателей, расположенных в нашем графическом интерфейсе. Для этого мы должны создать методы, обрабатывающие выбор переключателей:

1. В строке меню визуального редактора среды NetBeans, содержащей кнопки **Source** и **Design**, нажмем кнопку **Design**. Так мы снова увидим созданный нами графический интерфейс. Щелкнем правой кнопкой мыши на переключателе **Русский язык** и в появившемся контекстном меню выберем **Events | Action | actionPerformed**. В результате мы автоматически попадем в созданный средой метод, в который введем свой код (листинг 2.3). В коде запишем текст афоризмов в буфер `bufferText`, с которым будет работать метод, обрабатывающий нажатие кнопки **ВВОД**. Количество афоризмов запишем в целочисленную переменную `NumberSentence`.
2. С помощью кнопки **Design** вернемся к графическому интерфейсу и проделаем все то же самое с переключателем **Английский язык**. Запишем текст поговорок в буфер `bufferText`, с которым будет работать метод, обрабатывающий нажатие кнопки **ВВОД**. Количество поговорок запишем в целочисленную переменную `NumberSentence`.
3. Ну и наконец, в блоке `Variables declaration` зарегистрируем объекты и переменные `bufferText`, `bufferTextRus`, `bufferTextEn` и `NumberSentence`, используемые в различных методах нашего класса `AppletTalk` (листинг 2.4).

Листинг 2.3. Код методов, обрабатывающих выбор переключателей

```
private void jRadioButtonRusActionPerformed(java.awt.event.ActionEvent
evt) {
  // TODO add your handling code here:
  bufferText=bufferTextRus;
  NumberSentence=161;
}
private void jRadioButtonEnActionPerformed(java.awt.event.ActionEvent
evt) {
  // TODO add your handling code here:
  bufferText=bufferTextEn;
  NumberSentence=1001;
}
```

Листинг 2.4. Объявление объектов

```
private String bufferText;
private String bufferTextRus;
private String bufferTextEn;
private int NumberSentence;
```

Теперь мы получили апплет, представляющий собой программу-собеседника. При желании можно усложнить алгоритм, связывающий введенную пользователем фразу с ответом программы. Здесь мы реализовали лишь общий принцип.

Для запуска созданного апплета в окне браузера осуществим сборку проекта, щелкнув в окне **Projects** среды NetBeans правой кнопкой мыши на названии проекта и в появившемся контекстном меню выбрав пункт **Build**. При этом в папке dist каталога проекта будет создан файл AppletTalk.jar. Для загрузки апплета в браузере создадим HTML-страничку AppletTalk.html с соответствующим кодом (листинг 2.5).

Листинг 2.5. Код HTML-страницы, содержащей апплет

```
<applet CODE="AppletTalk/AppletTalk.class"
        ARCHIVE="AppletTalk.jar" WIDTH="500" HEIGHT="500"></APPLET>
```

Созданный файл HTML-страницы поместим в папку dist проекта AppletTalk. Так как относительный путь к файлам КозьмаПрутков.txt и EnglishText.txt в коде указан как /AppletTalk/Name.txt, то в папке dist создадим папку AppletTalk с текстовыми файлами. Открыв страничку AppletTalk.html, мы увидим результаты своего труда (рис. 2.17).

Дополнительно разберем код, который был сгенерирован средой NetBeans при создании нами графического интерфейса (листинг 2.6).

Листинг 2.6. Код инициализации апплета

```
public class AppletTalk extends javax.swing.JApplet {
  /** Initializes the applet AppletTalk */
  public void init() {
    try {
      java.awt.EventQueue.invokeAndWait(new Runnable() {
        public void run() {
          initComponents();
        }
      });
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Сейчас мы увидим, как много сказано этими несколькими строками кода, сгенерированными средой NetBeans.

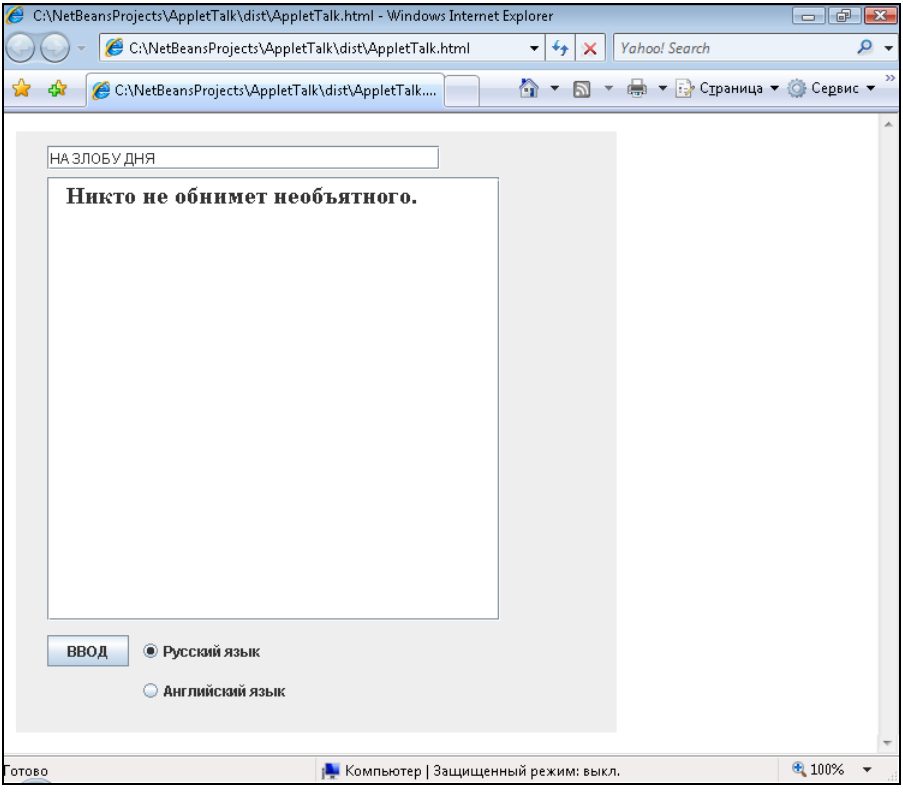

Рис. 2.17. Отображение апплета в окне браузера

EventQueue — платформно-независимый класс, представляющий собой очередь событий, получаемых как из классов-помощников `peer`, которые организуют взаимодействие классов AWT с операционной системой, так и из классов приложения. Данный класс обеспечивает диспетчеризацию событий, т. е. извлечение событий из очереди и отправки их вызовом внутреннего метода `dispatchEvent(AWTEvent event)`, который в качестве параметра принимает объект класса `AWTEvent`, представляющий собой AWT-события. Таким образом, класс `EventQueue` обеспечивает последовательную обработку событий в порядке очереди.

Метод `dispatchEvent()` класса `EventQueue` определяет, к какому графическому компоненту относится данное событие и производит вызов метода `dispatchEvent()` соответствующего компонента. Метод `dispatchEvent()` наследуется каждым компонентом от базового класса `java.awt.Component`. Далее событие передается методом `dispatchEvent()` методу `processEvent(AWTEvent e)`, который в свою очередь передает событие методу `process<event type>Event()`, определенному для каждого класса событий. По-

сле этого метод process<event type>Event() передает событие объекту интерфейса <event type>Listener, зарегистрированному соответствующим слушателем add<event type>Listener(), где оно и обрабатывается методом, определенным в интерфейсе.

Объект класса EventQueue автоматически создается виртуальной машиной Java, когда приложением создается объект — наследник класса java.awt.Component. При этом автоматически формируется также объект класса java.awt.EventDispatchThread, который представляет собой поток, работающий параллельно основному потоку программы. Данный поток собственно и осуществляет диспетчеризацию событий, хранящихся в очереди. Методом pumpEvents() класса EventDispatchThread события извлекаются из очереди и передаются методу dispatchEvent() класса EventQueue.

Метод invokeAndWait(Runnable runnable) класса EventQueue в качестве параметра принимает объект класса (здесь анонимного), реализующего интерфейс Runnable. Объект класса, реализующего интерфейс Runnable пакета java.lang, используется для переопределения метода run(), который вызывается потоком EventDispatchThread. Вызов метода invokeAndWait() является синхронным, т. е. не осуществляет возврат до тех пор, пока поток диспетчеризации событий не исполнит код, определенный в методе run().

Теперь уместно задать вопрос — зачем все это было нужно? Ведь вызов метода initComponents(), который инициализирует наш графический интерфейс, можно было просто осуществить внутри метода init() класса AppletTalk. Дело в том, что обращение к потоку диспетчеризации событий, который отвечает не только за реакцию на события, инициированные пользователем, но и за прорисовку всех компонентов, из другого потока, отличного от потока диспетчеризации событий, с помощью метода invokeAndWait() является правильным, т. к. предотвращает некорректное выполнение программы. При вызове метода init(), который всегда вызывается автоматически браузером для выполнения начальной инициализации апплета, создается основной поток, а мы хотим работать в потоке диспетчеризации событий, прорисовывая компоненты Swing. Метод invokeAndWait() блокирует вызвавший его поток до тех пор, пока не выполнятся все события, находящиеся в очереди, и не выполнится метод run() интерфейса Runnable в потоке диспетчеризации событий. При этом обращение к потоку диспетчеризации событий обрабатывается блоком обработки исключительных ситуаций try{}catch(){} с использованием класса Exception пакета java.lang, который является базовым для всех классов, представляющих исключения. Метод printStackTrace() класса Exception выводит текущий стек выполнения в текстовом виде в стандартный поток ошибок.

Теперь рассмотрим код метода, инициализирующего графический интерфейс (листинг 2.7).

Листинг 2.7. Код метода, отвечающего за инициализацию графического интерфейса, и блока объявления переменных

```
/** This method is called from within the init() method to
  * initialize the form.
  * WARNING: Do NOT modify this code. The content of this method is
  * always regenerated by the Form Editor.
  */
 // <editor-fold defaultstate="collapsed" desc="Generated Code">

private void initComponents() {
  buttonGroupLanguage = new javax.swing.ButtonGroup();
  jButtonSubmit = new javax.swing.JButton();
  jTextFieldInput = new javax.swing.JTextField();
  jScrollPaneOut = new javax.swing.JScrollPane();
  jTextAreaOut = new javax.swing.JTextArea();
  jRadioButtonRus = new javax.swing.JRadioButton();
  jRadioButtonEn = new javax.swing.JRadioButton();
  jButtonSubmit.setText("ВВОД");
  jTextAreaOut.setColumns(20);
  jTextAreaOut.setLineWrap(true);
  jTextAreaOut.setWrapStyleWord(true);
  jTextAreaOut.setRows(5);
  jScrollPaneOut.setViewportView(jTextAreaOut);
  buttonGroupLanguage.add(jRadioButtonRus);
  jRadioButtonRus.setSelected(true);
  jRadioButtonRus.setText("Русский язык");

  buttonGroupLanguage.add(jRadioButtonEn);
  jRadioButtonEn.setText("Английский язык");
  javax.swing.GroupLayout layout =
    new javax.swing.GroupLayout(getContentPane());
  getContentPane().setLayout(layout);
  layout.setHorizontalGroup(
    layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
    .addGroup(layout.createSequentialGroup()
    .addGap(26, 26, 26)
    .addGroup(layout.createParallelGroup
              (javax.swing.GroupLayout.Alignment.TRAILING)
      .addComponent(jScrollPaneOut,
                    javax.swing.GroupLayout.Alignment.LEADING,
                    javax.swing.GroupLayout.DEFAULT_SIZE, 327,
                    Short.MAX_VALUE)
```

```
      .addGroup(javax.swing.GroupLayout.Alignment.LEADING,
                layout.createParallelGroup(
                  javax.swing.GroupLayout.Alignment.LEADING, false)
      .addComponent(jTextFieldInput,
                    javax.swing.GroupLayout.DEFAULT_SIZE,
                    327, Short.MAX_VALUE)
      .addGroup(layout.createSequentialGroup()
      .addComponent(jButtonSubmit)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
      .addGroup(layout.createParallelGroup(
                  javax.swing.GroupLayout.Alignment.LEADING)
      .addComponent(jRadioButtonEn)
      .addComponent(jRadioButtonRus)))))
      .addContainerGap(47, Short.MAX_VALUE))
      );
  layout.setVerticalGroup(
    layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
    .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
              layout.createSequentialGroup()
    .addContainerGap()
    .addComponent(jTextFieldInput,
                  javax.swing.GroupLayout.PREFERRED_SIZE,
                  javax.swing.GroupLayout.DEFAULT_SIZE,
                  javax.swing.GroupLayout.PREFERRED_SIZE)
    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
    .addComponent(jScrollPaneOut,
                  javax.swing.GroupLayout.DEFAULT_SIZE,
                  176, Short.MAX_VALUE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
    .addGroup(layout.createParallelGroup(
                javax.swing.GroupLayout.Alignment.BASELINE)
    .addComponent(jButtonSubmit)
    .addComponent(jRadioButtonRus))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
    .addComponent(jRadioButtonEn)
    .addGap(23, 23, 23))
  );
  jButtonSubmit.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
      jButtonSubmitActionPerformed(evt);
    }
  });
```

```
    jRadioButtonRus.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(java.awt.event.ActionEvent evt) {
        jRadioButtonRusActionPerformed(evt);
      }
    });
    jRadioButtonEn.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(java.awt.event.ActionEvent evt) {
        jRadioButtonEnActionPerformed(evt);
      }
    });

}// </editor-fold>
// Variables declaration - do not modify
private javax.swing.ButtonGroup buttonGroupLanguage;
private javax.swing.JButton jButtonSubmit;
private javax.swing.JRadioButton jRadioButtonEn;
private javax.swing.JRadioButton jRadioButtonRus;
private javax.swing.JScrollPane jScrollPaneOut;
private javax.swing.JTextArea jTextAreaOut;
private javax.swing.JTextField jTextFieldInput;
// End of variables declaration
}
```

Код метода `initComponents()` (см. листинг 2.7) определяет такую последовательность действий:

1. В самом начале создаются объекты классов компонентов Swing.
2. Далее устанавливается надпись на кнопке.
3. После этого определяется размер текстовой области с помощью установки количества строк и столбцов методами `setColumns()` и `setRows()` класса `jTextArea`. Методами `setLineWrap(true)` и `setWrapStyleWord(true)` этого же класса определяется перенос строк по границе слова при отображении текста в текстовой области. Методом `setViewportView()` класса `javax.swing.JScrollPane` текстовая область включается в панель с прокруткой. Теперь, после включения текстовой области в панель с прокруткой, размеры текстовой области будут определяться размерами панели с прокруткой — контейнера, содержащего текстовую область. Установленные числа строк и столбцов учитываться не будут.
4. Созданные переключатели добавляются в группу с помощью метода `add()` класса `javax.swing.ButtonGroup`. Методами `setSelected()` и `setText()`, которые класс `javax.swing.JRadioButton` наследует от класса `javax.swing.AbstractButton`, устанавливается выбор переключателя `jRadioButtonRus` по умолчанию и подписи к переключателям.

5. При создании графического интерфейса располагать графические компоненты друг относительно друга можно двумя способами. Первый способ — это задание абсолютных координат каждого компонента в окне с помощью методов `setLocation()` и `setBounds()`, наследуемых от класса `java.awt.Component`. Этот способ неудобен тем, что при изменении размеров самого окна или изменении размеров какого-либо компонента необходимо корректировать координаты всех расположенных в окне компонентов. Для корректного расположения графических компонентов в окне друг относительно друга и относительно самого окна существуют специальные классы компоновок. Второй способ — это как раз использование таких классов компоновок, как:

   - `java.awt.CardLayout` размещает компоненты на вкладках так, что одновременно отображается только один компонент;
   - `java.awt.BorderLayout` размещает пять компонентов в пяти областях окна — по краям (вверху, внизу, слева, справа) и в центре;
   - `java.awt.FlowLayout` обеспечивает последовательное размещение компонентов в строку справа налево или слева направо;
   - `java.awt.GridBagLayout` обеспечивает табличное размещение компонентов, при этом ячейки таблицы могут быть различных размеров;
   - `java.awt.GridLayout` обеспечивает табличное размещение компонентов в ячейках одинакового размера;
   - `javax.swing.BoxLayout` обеспечивает блочное расположение компонентов;
   - `javax.swing.OverlayLayout` размещает компоненты один над другим;
   - `javax.swing.ScrollPaneLayout` используется классом `javax.swing.JScrollPane`, областью с прокруткой, обеспечивая размещение девяти компонентов — области просмотра, двух полос прокрутки, столбца и строки заголовка, четырех угловых компонентов;
   - `javax.swing.ViewportLayout` используется классом `javax.swing.JViewport`, областью просмотра;
   - `javax.swing.SpringLayout` обеспечивает взаимное расположение объектов класса `javax.swing.Spring`, с которыми ассоциируются компоненты заданием расстояния между парой границ различных компонентов;
   - и наконец, класс `javax.swing.GroupLayout`, который используется в нашем случае средой NetBeans.

   Класс `javax.swing.GroupLayout` является менеджером компоновки, который группирует компоненты в соответствии с их позицией в контейнере,

соблюдая иерархию. Группы компонентов при этом могут быть параллельными или последовательными. В последовательной группе позиции компонентов следуют одна за другой вдоль оси, в параллельной группе компоненты имеют по оси одну и ту же координату.

В коде, сгенерированном средой NetBeans, создается объект класса `javax.swing.GroupLayout` с помощью конструктора `GroupLayout(getContentPane())`, который в качестве параметра получает объект класса `java.awt.Container`, представляющего собой здесь "панель содержимого" нашего окна апплета, т. к. возвращается методом `getContentPane()` класса `javax.swing.JApplet`. Когда создается окно апплета, конструктором класса `JApplet` автоматически создается контейнер "корневая панель", занимающий все пространство окна, — объект класса `javax.swing.JRootPane`, в который и добавляются все остальные компоненты. При создании объекта класса `JRootPane` создается структура, которая состоит из: объекта класса `javax.swing.JLayeredPane` — "многослойная панель", занимающая все пространство корневой панели; объекта класса `javax.swing.JMenuBar` — строка меню, занимающая верхнюю часть окна; объекта класса `Container` — "панель содержимого" (`contentPane`), занимающая всю остальную часть окна, в которую добавляются создаваемые графические компоненты; и объекта класса `java.awt.Component` — "прозрачная панель" (`glassPane`), занимающая все пространство окна и расположенная поверх строки меню и панели содержимого. Прозрачная панель может использоваться для тестирования, анимации и перехвата событий.

6. Для панели содержимого окна апплета устанавливается менеджер компоновки — созданный объект `layout` класса `GroupLayout` с помощью метода `setLayout()` класса `Container`.

7. Методом `setHorizontalGroup()` класса `GroupLayout` компоненты размещаются вдоль горизонтальной оси. В качестве параметра метод принимает объект класса `GroupLayout.Group`, в который добавляются компоненты с помощью метода `addComponent()`. При этом компоненты группируются в последовательные и параллельные группы, вложенные друг в друга. Параллельная группа создается методом `createParallelGroup()` класса `GroupLayout`, при этом параметром служит способ выравнивания компонентов. В случае горизонтального размещения компонентов `LEADING` — выравнивание по левому краю, `TRAILING` — выравнивание по правому краю. Методом `addGroup()` класса `GroupLayout.Group` добавляется группа к уже созданной группе. Методом `addGap()` класса `GroupLayout.Group` добавляется элемент, представляющий дистанцию между компонентами в группе с тремя параметрами — минимальная дистанция, предпочтительная дистанция и максимальная дистанция.

8. Методом `setVerticalGroup()` класса `GroupLayout` компоненты размещаются вдоль вертикальной оси. В случае вертикального размещения компонентов `LEADING` — выравнивание по верхнему краю, `TRAILING` — выравнивание по нижнему краю. Методом `addContainerGap()` класса `javax.swing.GroupLayout.SequentialGroup` добавляется элемент, представляющий дистанцию между краем контейнера и компонентом. Методом `addPreferredGap()` того же класса добавляется элемент, представляющий предпочтительную дистанцию между ближайшими компонентами в группе, при этом параметром служит индикатор связанности компонентов. Параметрами метода `addComponent()` класса `GroupLayout.Group` здесь служат: имя компонента; минимальный размер; предпочтительный и максимальный размеры компонента.
9. Так как мы создали в программе методы, обрабатывающие нажатие кнопки **ВВОД** и выбор переключателей, то среда NetBeans сгенерировала в методе `initComponents()` соответствующий код. Методом `addActionListener()`, который наследуется классами кнопки и переключателей от класса `javax.swing.AbstractButton`, переопределяя метод `actionPerformed()` интерфейса `ActionListener` (данный метод вызывается при возникновении события), соответствующие методы регистрируются как обработчики событий компонентов. Так как `ActionListener` является интерфейсом, то создавать напрямую его экземпляры нельзя. Можно лишь создавать экземпляры класса, реализующего данный интерфейс. Поэтому в данном коде демонстрируется создание экземпляра анонимного класса, реализующего интерфейс `ActionListener` с переопределением метода интерфейса `actionPerformed()`.

В блоке объявления переменных среда NetBeans автоматически генерирует код, отвечающий за объявление объектов классов компонентов Swing (см. листинг 2.7).

Контроль работы апплетов системой безопасности платформы Java SE

Так как виртуальная машина Java при выполнении кода апплета использует *песочницу* — механизм для безопасного исполнения программ, то наш апплет будет запускаться не везде в файловой системе компьютера, где мы разместим папку с документом HTML, обеспечивающим загрузку апплета, файлами байт-кода апплета и ресурсными файлами. Связано это с ограничениями, которые виртуальная Java-машина может наложить на чтение текстовых файлов, содержащих афоризмы и поговорки. В частности, апплет не запустится, если мы поместим папку с кодом и текстовыми файлами в пользовательский

каталог. Самим можно поэкспериментировать с размещением папки для того, чтобы понаблюдать результаты действия песочницы.

Безопасную среду для выполнения апплетов и Java-приложений обеспечивает система безопасности платформы Java SE.

Система безопасности платформы Java SE состоит из библиотек API, различных инструментов и механизмов, обеспечивающих аутентификацию/авторизацию, использование публичных ключей и сертификатов, безопасное интернет-соединение и др. Для конечных пользователей среда выполнения Java ограничивает доступ к файловой системе компьютера, препятствуя работе вредоносных программ. Для разработчиков платформа Java SE содержит библиотеки API для встраивания функций, обеспечивающих безопасность, в Java-приложения. Для системных администраторов, разработчиков и пользователей платформа Java SE предоставляет Java-инструменты для генерации электронных подписей JAR-файлов и проверки подлинности электронных подписей, а также создания и модификации файлов, определяющих политику безопасности для Java-приложений.

Для апплетов поддержку системы безопасности платформы Java SE обеспечивает инструмент Java Plug-in, который осуществляет безопасное интернет-соединение, использует менеджера безопасности Security Manager для ограничения работы апплета, поддерживает механизм публичных ключей и сертификатов.

Инструмент Java Plug-in использует класс Security Manager для предотвращения доступа вирусов к компьютеру через запущенный апплет. Для неподписанных апплетов доступ к ресурсам компьютера запрещен, пока Security Manager не найдет соответствующие разрешения в файле политики безопасности. При попытке доступа к ресурсам компьютера без разрешения возникает исключительная ситуация.

Файл политики безопасности представляет собой текстовый файл, который может быть создан с помощью инструмента `policytool`, содержащегося в папке bin каталога JDK.

Далее рассмотрим создание файла политики безопасности для нашего апплета с использованием инструмента `policytool`:

1. Для начала работы откроем папку bin каталога JDK и найдем инструмент `policytool`.
2. Запустив в исполняемом файле `policytool`, мы увидим окно графического интерфейса инструмента (рис. 2.18).
3. Для создания файла политики безопасности и записи в нем нажмем кнопку **Add Policy Entry**. В результате появится диалоговое окно (рис. 2.19).

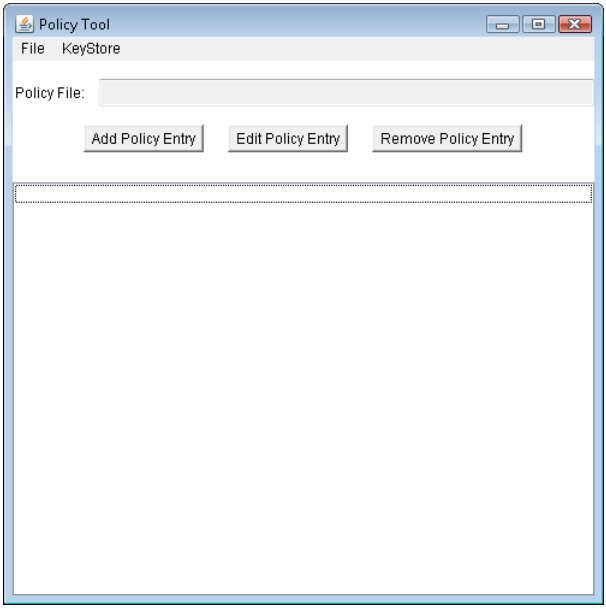

Рис. 2.18. Окно интерфейса инструмента `policytool`

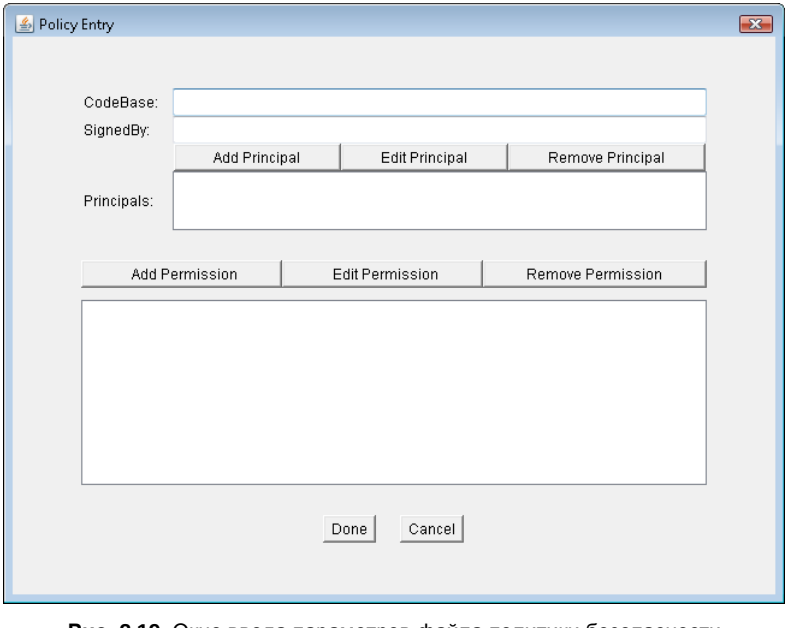

Рис. 2.19. Окно ввода параметров файла политики безопасности

4. В окне ввода параметров файла политики безопасности строка **CodeBase** предназначена для ввода URL-адреса Java-кода, для которого дается разрешение. Если строка пустая, то разрешение дается для любого расположения кода. Строка **SignedBy** предназначена для указания владельца сертификата публичного ключа в случае, если Java-код имеет электронную подпись. Введем адрес каталога, содержащего файлы апплета, в поле **CodeBase** (это должен быть URL-адрес вида "file:C:\...", в конце поставим "\-" для распространения разрешения на вложенные папки), оставив поле **SignedBy** пустым, т. к. наш апплет не подписан, и нажмем кнопку **Add Permission**. Появится диалоговое окно определения разрешений (рис. 2.20).

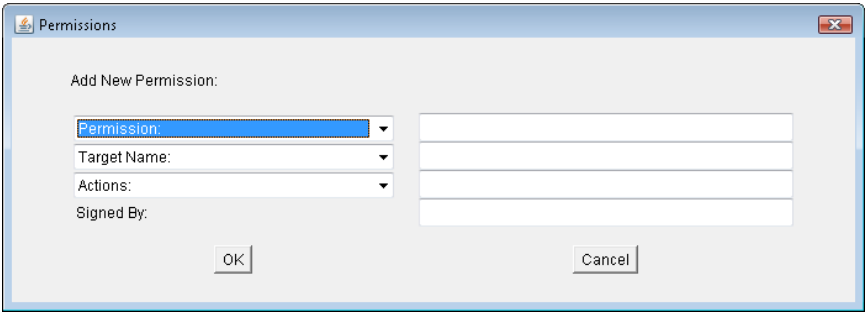

Рис. 2.20. Окно определения разрешений для Java-кода

5. В раскрывающемся списке **Permission** выберем вариант **AllPermission** и нажмем кнопку **OK**, а затем кнопку **Done**. Таким образом, мы назначим полные права апплету.
6. В окне **Policy Tool** в меню **File** выберем пункт **Save As** и в пользовательском каталоге создадим файл политики безопасности с именем java.policy, содержащий разрешения для апплета и автоматически загружаемый Java Plug-in по умолчанию. Теперь наш апплет будет без проблем запускаться в пользовательском каталоге.

В случае запуска настольных Java-приложений виртуальная машина Java не использует автоматически Security Manager, в отличие от Java Plug-in, который применяет менеджер безопасности автоматически.

Для того чтобы запускать Java-приложение вместе с Security Manager, необходимо в командной строке указать опцию `-Djava.security.manager` инструмента `java`. При этом так же, как и в случае с апплетом, будут проверены все необходимые разрешения, содержащиеся в файле политики безопасности.

Для исключения запуска Java-приложений или апплетов с внедренными в процессе передачи пользователю вредоносными функциями используется

технология электронной подписи исходного кода. Данная технология подразумевает подпись кода с помощью персонального ключа и передачи пользователю публичного ключа, который должен соответствовать персональному. Если код оказался модифицированным, то ключи не будут соответствовать друг другу. Для подтверждения того, что публичный ключ также не был модифицирован, используется сертификат, который подтверждает подлинность публичного ключа.

Для реализации технологии электронной подписи необходимо упаковать Java-приложение или апплет в JAR-файл, затем подписать JAR-файл электронной подписью. Для этого первоначально необходимо создать персональный ключ с помощью инструмента `keytool` папки bin каталога JDK, при этом будет создано хранилище, содержащее пару "персональный ключ/публичный ключ", а также сертификат с публичным ключом. Подписать JAR-файл теперь можно, используя инструмент `jarsigner` папки bin каталога JDK. Создать файл с сертификатом для пересылки пользователю можно с помощью опции `-export` инструмента `keytool`. Опция `-import` инструмента `keytool` добавляет сертификат в список сертификатов, доверие к которым подтверждено пользователем.

При запуске подписанного апплета, если пользователь подтвердит доверие к сертификату публичного ключа, Java-код наделяется такими же полными правами, как если бы для него был создан файл политики безопасности с разрешением AllPermission.

ГЛАВА 3

Создание настольных приложений на базе платформы Swing Application Framework

Платформа приложений Swing Application Framework (SAF) и ее применение

Платформа приложений Swing Application Framework (JSR 296) — это небольшой набор классов и интерфейсов, предназначенных упростить создание настольных Java-приложений. Упрощение заключается в предоставлении разработчику классов и интерфейсов, представляющих готовые модели управления жизненным циклом приложения, обработки событий, управления ресурсами и создания графического интерфейса пользователя.

Платформа Swing Application Framework (SAF) основывается на графической библиотеке Swing и библиотеках Java SE API. SAF-классы, представляющие окна графического интерфейса пользователя, созданы на основе классов библиотеки Swing. Модель обработки событий и выполнения задач платформы SAF расширяет функциональность Java SE, добавляя свойства и методы для расширенного управления задачами и событиями приложения. Модель управления ресурсами платформы SAF основывается на библиотеке java.util платформы Java SE, обеспечивая интернационализацию/локализацию приложения.

Платформа SAF обеспечивает для настольных приложений следующие сервисы:

- управление жизненным циклом приложения;
- загрузку и управление ресурсами приложения;
- управление действиями приложения, включая выполнение задач в фоновом потоке;

- сохранение/восстановление состояния графического интерфейса между запусками приложения.

Платформа SAF представлена одним пакетом `org.jdesktop.application`, в котором класс `Application` является базовым классом для создания приложения. Класс `Application` предоставляет методы управления жизненным циклом SAF-приложения — его инициализацией, запуском, готовностью к работе и завершением работы.

Запуск SAF-приложения подразумевает создание и вывод на экран графического интерфейса пользователя. Перед запуском приложения производится инициализация его конфигурации и приведение интерфейса в состояние готовности. При завершении работы приложения класс `Application` обеспечивает закрытие графического интерфейса и освобождение всех задействованных ресурсов операционной системы.

С объектом `Application` приложения связан один-единственный объект класса `ApplicationContext` платформы SAF, обеспечивающий сервисы управления действиями, задачами, ресурсами, сохранением/восстановлением состояния и другими объектами приложения.

Класс `ResourceMap` библиотеки SAF позволяет осуществлять поиск и ввод ресурсов, используя ресурсные файлы приложения. С помощью класса `SessionStorage` состояние графического интерфейса пользователя сохраняется между запусками приложения. Аннотация `@Action` используется для маркировки методов — обработчиков событий, определяя при этом свойства действий приложения. Классы `View`, `FrameView` и `SingleFrameApplication` являются основой для создания графического интерфейса приложения. Класс `Task` обеспечивает выполнение задач приложения в фоновом потоке, не блокирующем интерфейс пользователя.

Таким образом, платформа Swing Application Framework дает возможность построения базового каркаса для Swing-приложений, предоставляя инфраструктуру, общую для большинства настольных приложений, и тем самым упрощая создание Swing-приложений.

В настоящее время проект SAF (**https://appframework.dev.java.net/**) заморожен. Альтернативой SAF является проект Better Swing Application Framework (BSAF) (**http://kenai.com/projects/bsaf**), который представляет собой дальнейшее развитие SAF в целях повышения надежности платформы приложений и расширения ее возможностей.

Однако сейчас среда NetBeans поддерживает только SAF, создавая каркас настольного приложения на основе Swing Application Framework. Этот NetBeans-шаблон обеспечивает базовое приложение типовой инфраструктурой, включающей в себя окно приложения с панелью состояния, меню, индикатором процесса и механизм управления событиями и ресурсами.

Пример разработки настольного приложения для поиска в Интернете

В качестве примера настольного Java-приложения на базе платформы Swing Application Framework (SAF) рассмотрим создание приложения, которое по запросу пользователя осуществляет поиск информации в Интернете и сохраняет найденную информацию в создаваемой папке.

> ***Примечание***
>
> Файлы с кодами приложения расположены на прилагаемом к книге компакт-диске в папке Часть_I\Глава_3\SearchInternet.

Для начала работы создадим проект приложения в среде NetBeans:

1. В среде NetBeans в главном меню выберем **File | New Project**. В появившемся диалоговом окне в списке проектов выберем **Java Desktop Application** (рис. 3.1).
2. Нажмем кнопку **Next**, после чего введем имя проекта SearchInternet (рис. 3.2).
3. После нажатия кнопки **Finish** мы увидим созданную средой NetBeans основу нашего приложения (рис. 3.3).

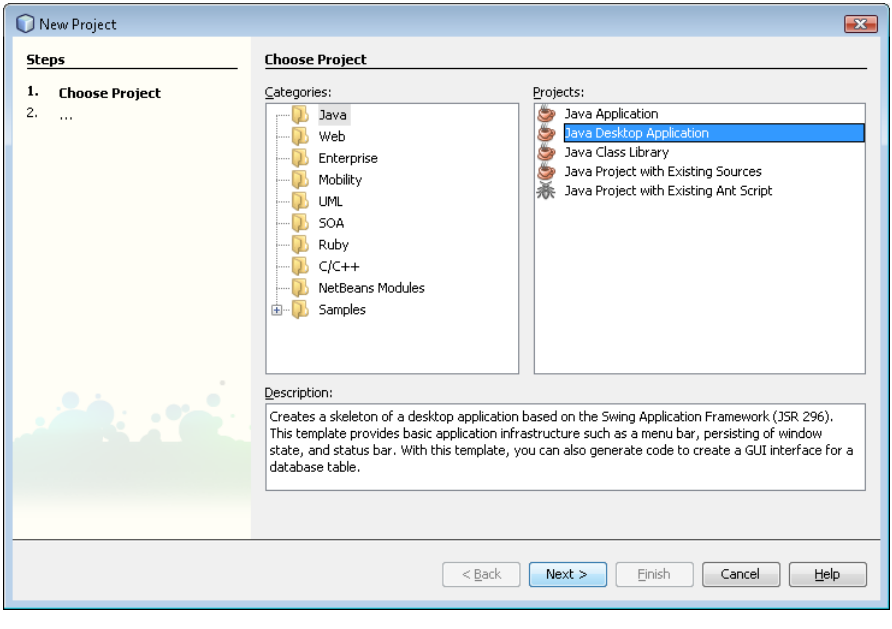

Рис. 3.1. Окно выбора проекта

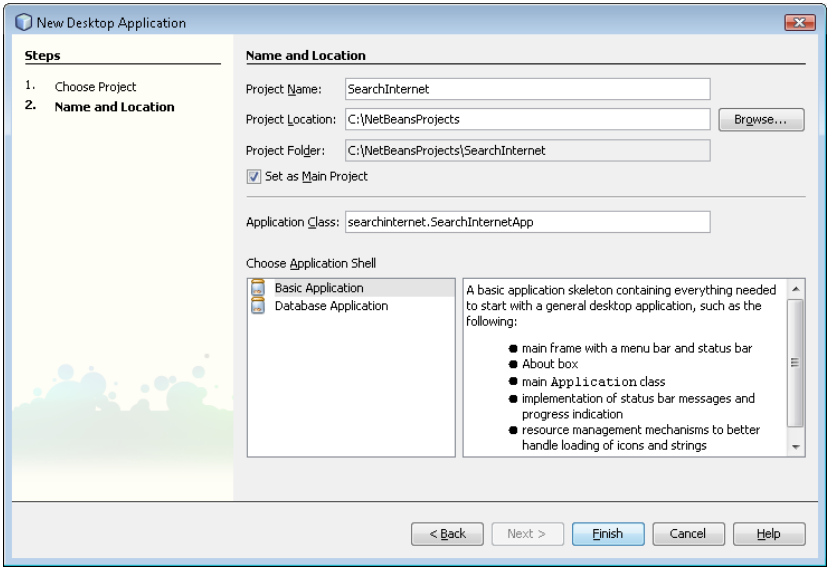

Рис. 3.2. Окно ввода параметров проекта

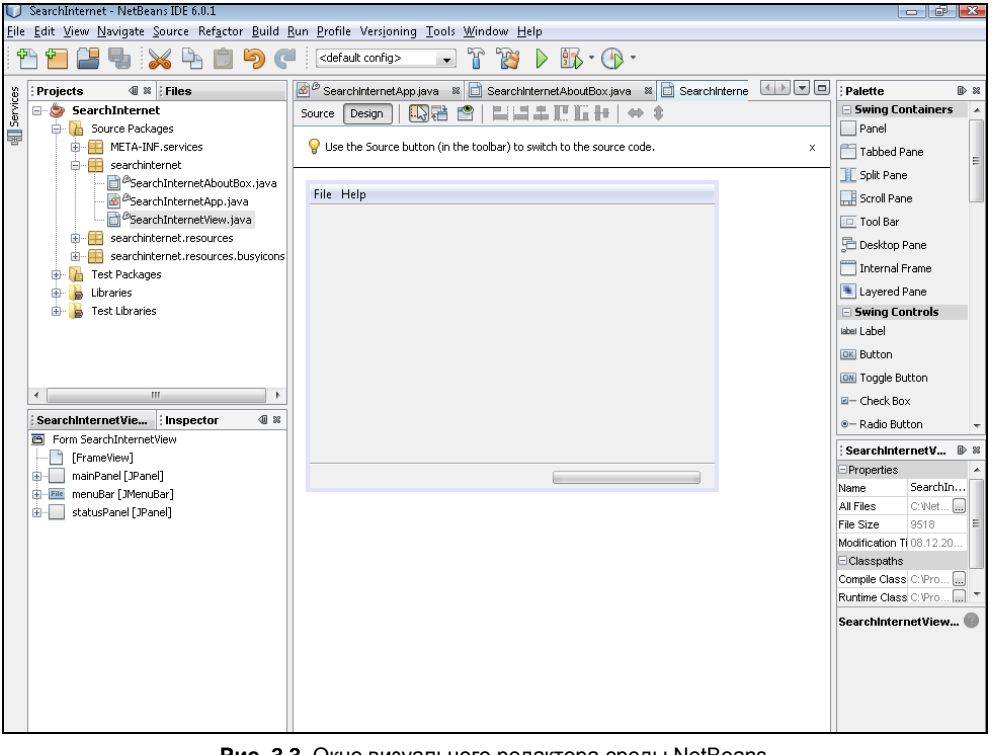

Рис. 3.3. Окно визуального редактора среды NetBeans

Структура приложения

В левом верхнем окне среды NetBeans, на вкладке **Projects**, отображена структура нашего проекта, которую мы рассмотрим далее.

Узел **Source Packages** физически не создается средой NetBeans в файловой системе, а отображается только в структуре проекта и включает в себя узлы **META-INF.services**, **searchinternet**, **searchinternet.resources** и **searchinternet.resources.busyicons**.

Узел **META-INF.services** представляет подпапки, созданные средой NetBeans в папке src, расположенной в каталоге проекта. Папка services содержит текстовый файл с именем, совпадающим с полным именем абстрактного класса `org.jdesktop.application.Application`, являющегося базовым для SAF-приложений. Данный текстовый файл является конфигурационным и содержит в нашем случае одну строку с полным именем класса `searchinternet.SearchInternetApp`, расширяющим абстрактный класс `org.jdesktop.application.SingleFrameApplication`, который в свою очередь является подклассом класса `Application` и базовым классом SAF-приложений с графическим интерфейсом, содержащим одно основное окно.

Папка META-INF\services с конфигурационным файлом, содержащим имя класса, реализующего интерфейс или абстрактный класс, может использоваться для создания расширяемых приложений, рассматриваемых в *главе 4*.

Узел **searchinternet** представляет папку пакета, которая создается в папке src проекта и содержит Java-файлы с исходным кодом и папку resources с ресурсами приложения.

Файл SearchInternetAboutBox.java папки пакета содержит код класса `SearchInternetAboutBox`, расширяющего класс `javax.swing.JDialog`, который предназначен для создания диалогового окна. Класс `SearchInternetAboutBox` отвечает за создание диалогового окна, отображающего информацию о приложении: название приложения, краткая характеристика приложения, версия, производитель, домашняя интернет-страница приложения. Одновременно с файлом SearchInternetAboutBox.java средой NetBeans генерируется файл SearchInternetAboutBox.form, который является XML-файлом и используется визуальным редактором форм среды NetBeans для хранения информации о созданной графической форме. Данный файл не содержится в исполняемом JAR-файле приложения и используется только средой разработки для редактирования графического интерфейса. Файл SearchInternetAboutBox.form находится также в папке пакета и не отображается средой NetBeans в структуре проекта.

Файл SearchInternetApp.java папки пакета содержит код класса `SearchInternetApp`, расширяющего класс `SingleFrameApplication`, который служит базовым классом SAF-приложений с графическим интерфейсом, со-

держащим одно основное окно. Класс `SearchInternetApp` содержит статический метод `main()` — точку входа в приложение и поэтому является главным классом приложения.

Файл SearchInternetView.java папки пакета содержит код класса `SearchInternetView`, расширяющего класс `org.jdesktop.application.FrameView`, который в свою очередь является подклассом класса `org.jdesktop.application.View`. Класс `View` представляет собой класс-обертку для таких стандартных элементов графического интерфейса, как меню, панель инструментов, графические компоненты, панель состояния, и содержит корневую панель, куда добавляются графические компоненты. Класс `SearchInternetView` отвечает за создание главного окна приложения. С файлом SearchInternetView.java ассоциируется файл SearchInternetView.form, который, аналогично файлу SearchInternetAboutBox.form, является XML-файлом и используется визуальным редактором форм среды NetBeans для хранения информации о созданной графической форме. Данный файл располагается в папке пакета и не отображается средой NetBeans в структуре проекта.

Узлы **searchinternet.resources** и **searchinternet.resources.busyicons** представляют папку resources, расположенную в папке src\searchinternet и содержащую ресурсы приложения.

Файлы SearchInternetAboutBox.properties, SearchInternetApp.properties и SearchInternetView.properties папки resources являются ресурсными файлами или файлами наборов ресурсов приложения. Файлы наборов ресурсов используются для интернационализации и локализации приложения.

> ***Примечание***
>
> *Интернационализация* — это способ написания приложения таким образом, что все объекты, зависящие от конкретного языка, отделены от кода приложения. *Локализация* — это процесс сопоставления определенных текстовых строк и изображений локализованным ресурсам, что дает для графического интерфейса пользователя возможность отображения изображений и строк для различных языков.

Кроме того, файлы наборов ресурсов позволяют изменять различные ресурсы без изменения кода приложения и перекомпиляции. Файлы наборов ресурсов состоят из строк, содержащих пары "ключ — значение". Доступ к набору ресурсов, таких как строки текста, значки, определения цвета и шрифта, осуществляется с помощью класса `org.jdesktop.application.ResourceMap`.

Среда NetBeans позволяет задать настройки для управления ресурсами. Для этого необходимо в среде NetBeans, в окне **Projects**, в узле **searchinternet** дважды щелкнуть левой кнопкой мыши на файле класса, отвечающего за создание графической формы. Затем в окне визуального редактора среды

NetBeans, в верхней строке меню с кнопками **Source** и **Design**, нажать кнопку **Design** и переключиться в режим конструктора. После этого в левом нижнем окне **Inspector** щелкнуть правой кнопкой мыши на названии созданной графической формы и в появившемся контекстном меню выбрать пункт **Properties**. В появившемся диалоговом окне строка **Automatic Resource Management** отвечает за требуемые настройки (рис. 3.4).

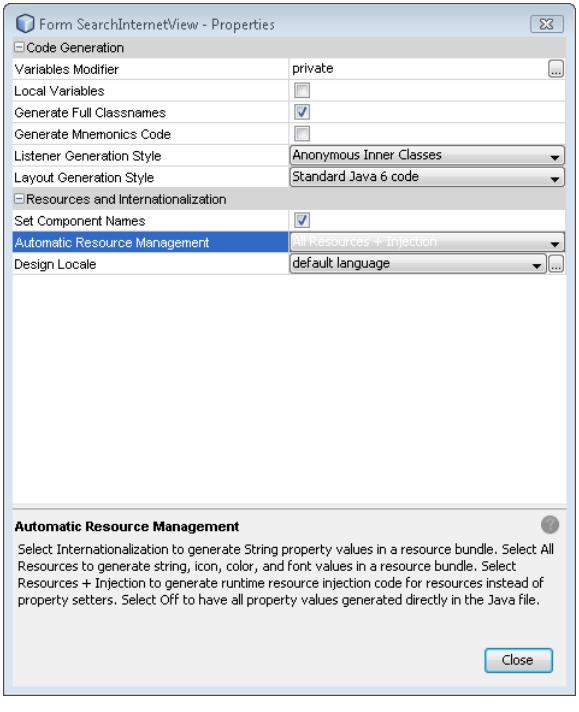

Рис. 3.4. Окно свойств формы

В строке **Automatic Resource Management** в случае выбора варианта **Off** наборы ресурсов не отделяются от кода и properties-файлы не создаются. В случае выбора варианта **Internationalization** файл набора ресурсов создается только для строк. В случае выбора варианта **All Resources** файл набора ресурсов создается для строк, цветов, шрифтов и значков. В случае выбора варианта **Resources + Injection** файл набора ресурсов создается такой же, как и в случае варианта **All Resources**, однако набор ресурсов загружается во время выполнения приложения.

Папка resources помимо ресурсных файлов содержит:

- about.png — изображение в формате PNG (улучшенный по сравнению с GIF формат, имеющий свободную лицензию). Это изображение используется в диалоговом окне, содержащем информацию о приложении;

- splash.png — заставка-рисунок, которая появляется во время загрузки приложения;
- папку busyicons со значками иконки для использования в панели состояния приложения.

Узел **Test Packages** (см. рис. 3.3) отображает пакеты классов, предназначенных для тестирования приложения. Среда NetBeans позволяет создавать каркас для теста на основе платформы JUnit. Библиотека классов JUnit представляет собой платформу для создания автоматических, самоконтролируемых тестов на языке Java, которые в JUnit называются *test cases*. Основной принцип работы тестирующего класса, созданного на основе библиотеки JUnit, — это вызов методов тестируемого класса приложения с заданными параметрами и сравнение возвращаемого методом значения со значением, определенным в тесте. Для создания тестирующего класса в среде NetBeans необходимо на узле **Test Packages** щелкнуть правой кнопкой мыши и в появившемся меню выбрать **New | Other | JUnit | JUnit Test**, затем определить имя и пакет тестирующего класса и выбрать соответствующую библиотеку JUnit. После этого средой NetBeans будет создан каркас тестирующего класса. Запуск теста можно осуществить, щелкнув правой кнопкой мыши в окне **Projects** среды NetBeans на Java-файле, содержащем код тестирующего класса, и выбрав в появившемся меню пункт **Run File**. После этого в среде NetBeans появится окно **JUnit Test Results** с результатами теста. Другой способ — это щелкнуть правой кнопкой мыши на названии проекта в окне **Projects** среды NetBeans и в появившемся меню выбрать пункт **Test**, что приведет к запуску всех тестов приложения.

Узел **Libraries** показывает подключенные библиотеки, необходимые для работы приложения:

- Swing Application Framework — appframework-1.0.3.jar, библиотека классов платформы приложений SAF;
- Swing Application Framework — swing-worker-1.1.jar. В SAF-приложениях существуют три типа потоков. Первый тип — это первоначальный поток, который инициализирует и запускает приложение. Второй тип — это поток распределения событий (Event Dispatch Thread), в котором должны выполняться все действия, связанные с графическим интерфейсом, такие как: рисование и обновление компонентов графического интерфейса; обработка событий интерфейса; любое взаимодействие приложения с платформой Swing. Третий тип — это рабочий или фоновый поток (background thread), в котором должны выполняться все задачи, требующие длительного времени выполнения. Если задача, занимающая ощутимое время, будет выполняться в потоке обработки событий, тогда графический интерфейс будет заблокирован в течение этого промежутка времени. Для решения

этой проблемы платформа Swing Application Framework предлагает класс `org.jdesktop.application.Task<T,V>` библиотеки appframework-1.0.3.jar, который расширяет класс `org.jdesktop.swingworker.SwingWorker<T,V>` библиотеки swing-worker-1.1.jar. Данный класс `SwingWorker` одинаков с классом `javax.swing.SwingWorker<T,V>` библиотеки Java SE 6 API, однако он не содержится в библиотеке классов appframework-1.0.3.jar, поэтому и требуется подключение библиотеки swing-worker-1.1.jar для поддержки класса `Task`;

- библиотека классов JDK 1.6, подключенная по умолчанию.

Узел **Test Libraries** отображает подключенные библиотеки классов платформы JUnit, предназначенные для создания пакетов классов, тестирующих приложение. Среда NetBeans поддерживает как библиотеку JUnit 4.*x*, так и JUnit 3.8.*x*. При создании тестирующего класса в среде NetBeans пользователь должен выбрать ту или другую библиотеку в диалоговом окне. В соответствии с выбором пользователя среда NetBeans создает соответствующий каркас тестирующего класса и удаляет из узла **Test Libraries** ненужную библиотеку. Однако проект может содержать смесь тестов JUnit 4.*x* и JUnit 3.8.*x*. Для этого необходимо подключить соответствующую библиотеку, щелкнув правой кнопкой мыши на узле **Test Libraries** и выбрав в появившемся контекстном меню **Add Library | JUnit**. Это позволяет создать тестирующий класс на основе другой библиотеки JUnit. Библиотека классов JUnit 4.*x* содержит важные улучшения по сравнению с JUnit 3.8.*x*. Основные нововведения в JUnit 4.*x* это: поддержка аннотаций Java 5, что дало возможность создать гибкие фикстуры, которые помогают многократно использовать программный код; новый способ логической группировки тестов; тесты исключений; тесты с ограничением по времени; средства игнорирования тестов и параметризированные тесты.

Код класса *SearchInternetApp*

После рассмотрения структуры проекта разберем код, сгенерированный средой NetBeans при создании основы нашего приложения.

В окне **Projects** среды NetBeans в узле **Source Packages** найдем файл SearchInternetApp.java и дважды щелкнем по нему левой кнопкой мыши. После этого в окне редактора среды NetBeans появится код, содержащийся в этом файле (листинг 3.1).

Листинг 3.1. Код класса `SearchInternetApp`

```
package searchinternet;
import org.jdesktop.application.Application;
import org.jdesktop.application.SingleFrameApplication;
```

```
/**
 * The main class of the application.
 */
public class SearchInternetApp extends SingleFrameApplication {
/**
  * At startup create and show the main frame of the application.
  */
  @Override protected void startup() {
    show(new SearchInternetView(this));
  }
/**
  * This method is to initialize the specified window by injecting
resources.
  * Windows shown in our application come fully initialized from the GUI
  * builder, so this additional configuration is not needed.
  */
  @Override protected void configureWindow(java.awt.Window root) {
  }
/**
  * A convenient static getter for the application instance.
  * @return the instance of SearchInternetApp
  */
  public static SearchInternetApp getApplication() {
    return Application.getInstance(SearchInternetApp.class);
  }
/**
  * Main method launching the application.
  */
  public static void main(String[] args) {
    launch(SearchInternetApp.class, args);
  }
}
```

Обсудим код основы класса `SearchInternetApp`.

Как уже было сказано, класс `org.jdesktop.application.Application` является базовым классом для SAF-приложений. Этот класс определяет жизненный цикл приложения: инициализацию, запуск, готовность и завершение с помощью соответствующих методов. Метод `startup()` отвечает за создание и вывод на экран исходного графического интерфейса и выполняется в потоке обработки событий. Данный метод вызывается статическим методом `launch()` и должен переопределяться любым подклассом класса `Application`. Метод `launch()` является статическим, т. е. вызывается без создания экземпляра класса `Application`. Этот метод создает экземпляр подкласса

`Application`, при этом создается экземпляр класса `ApplicationContext`, предназначенный для поиска ресурсов, совместно используемых приложением.

Приложение содержит ресурсы как на уровне классов, так и на уровне приложения. Здесь идет речь о наборе ресурсов на уровне приложения, которые хранятся, в нашем случае, в файле SearchInternetApp.properties. В методе `launch()` с помощью методов `getContext()` и `setApplicationClass()` классов `Application` и `ApplicationContext` в созданный экземпляр класса `ApplicationContext` записывается контекст экземпляра класса `Application`. Каждому экземпляру класса `Application` соответствует единственный экземпляр класса `ApplicationContext`. С помощью метода `getResourceMap()` класса `ApplicationContext` загружается карта ресурсов приложения в экземпляр класса `ResourceMap`, который инкапсулирует наборы ресурсов. С помощью метода `getString()` класса `ResourceMap` и строки `Application.lookAndFeel` в файле SearchInternetApp.properties определяется внешний вид приложения, который затем устанавливается методом `setLookAndFeel()` класса `javax.swing.UIManager`. Далее в методе `launch()` вызываются методы `initialize()` — инициализация графического интерфейса и `startup()` — создание и вывод на экран исходного графического интерфейса. Метод `ready` класса `Application` вызывается после метода `startup()`, после этого приложение готово к использованию. Все подклассы класса `Application` должны переопределять метод `startup()` и вызывать метод `exit()` для корректного завершения приложения. Можно также переопределить метод `shutdown()`, который вызывается при завершении приложения и используется для корректного завершения каких-либо специфических задач.

Классы `Application` и `ApplicationContext` являются расширениями класса `AbstractBean`, который инкапсулирует методы, базирующиеся на методах класса `java.beans.PropertyChangeSupport`. Класс `PropertyChangeSupport` является сервисным классом технологии JavaBeans — технологии компонентного программирования для создания и использования многократно используемых программных компонентов Beans. Если класс `Bean` содержит контролируемый другими компонентами атрибут, он должен иметь список контролирующих его объектов и методы для добавления или удаления из списка. Класс `PropertyChangeSupport` обеспечивает вышеупомянутый список и методы для работы с ним. Для того чтобы реализовать возможности класса `PropertyChangeSupport`, класс `Bean` должен содержать в качестве атрибута объект класса `PropertyChangeSupport` или быть его расширением. В нашем случае классы `Application` и `ApplicationContext` являются его расширениями.

Класс `org.jdesktop.application.SingleFrameApplication` является расширением класса `Application` и служит базовым классом для создания графического интерфейса с одним главным окном. Класс `SingleFrameApplication` обеспечи-

вает ввод свойств компонентов, указанных в файле [Name]App.properties, обработку выхода из приложения и сохранение/восстановление состояния окна приложения при выходе/загрузки приложения. Главное окно приложения — компонент `javax.swing.JFrame` — создается автоматически с обеспечением обработки события, связанного с закрытием окна. Обработчик событий, добавленный с помощью метода `addWindowListener()` класса `java.awt.Window`, в случае закрытия окна вызывает функцию `exit()` класса `Application`. В методе `exit()` происходит вызов метода `shutdown()` с вызовом метода `saveSession()`, в котором и происходит сохранение состояния окна в файл с именем mainFrame.session.xml, расположенный, как правило, в каталоге ${userHome}\Application Data\${vendorId}\${applicationId}\.

Созданный нами класс приложения `SearchInternetApp` расширяет класс `SingleFrameApplication` и является главным классом приложения, т. к. содержит метод `main()` — точку входа в приложение.

В коде класса `SearchInternetApp` метод `startup()` класса `Application` переопределяется с вызовом метода `show(View view)` класса `SingleFrameApplication`, при этом создается и выводится главное окно приложения со следующим порядком действий:

1. Производится ввод свойств компонентов, указанных в файле [Name]App.properties, с помощью метода `configureWindow()` класса `SingleFrameApplication`.
2. Добавляется обработчик событий, получаемых от окна.
3. Восстанавливаются параметры окна, сохраненные в файле mainFrame.session.xml.
4. Окно становится видимым с помощью метода `setVisible(true)` класса `java.awt.Window`.

В случае если свойства компонентов, указанные в файле набора ресурсов, должны быть введены каким-либо специфическим образом, метод `configureWindow()` класса `SingleFrameApplication` должен быть переопределен.

Используемый в коде статический метод `getApplication()`, который возвращает объект `Application` запущенного приложения, может также быть использован при тестировании для получения объекта `Application` без запуска приложения.

Статический метод `main()` класса `SearchInternetApp`, который является точкой входа в приложение и с выполнения которого всегда начинает работу Java-интерпретатор, вызывает метод `launch()` класса `Application`, описанный ранее.

Код класса *SearchInternetView*

Теперь рассмотрим сгенерированный код класса `SearchInternetView`.

В окне **Projects** среды NetBeans в узле **Source Packages** найдем файл SearchInternetView.java и, после того как дважды щелкнем по нему левой кнопкой мыши, увидим в редакторе среды визуальную основу нашего приложения. Над визуальной основой приложения найдем строку меню с кнопками **Source** и **Design** и нажатием кнопки **Source** переключимся в окно редактора, в котором мы увидим сгенерированный средой код.

> ***ПРИМЕЧАНИЕ***
>
> Код класса `SearchInternetView` см. в файле SearchInternetView.java папки Часть_I\Глава_3\SearchInternet\src\searchinternet на прилагаемом к книге компакт-диске.

Обсудим сгенерированный код класса `SearchInternetView`.

В коде класса `SearchInternetView` используются: аннотация `org.jdesktop.application.Action`, классы `org.jdesktop.application.ResourceMap`, `org.jdesktop.application.FrameView`, `org.jdesktop.application.TaskMonitor`, `org.jdesktop.application.Task`, `java.awt.event.ActionEvent`, `javax.swing.Timer`, `javax.swing.JDialog` и `javax.swing.JFrame`, интерфейсы `java.awt.event.ActionListener` и `javax.swing.Icon`, роль и назначение которых мы рассмотрим далее.

Аннотация `org.jdesktop.application.Action`, используемая в коде класса `SearchInternetView`, помечает метод, который будет применяться как метод `actionPerformed()` (обработчик событий) для объекта типа `javax.swing.Action`.

Интерфейс `javax.swing.Action`, расширяющий интерфейс `java.awt.event.ActionListener`, нужен в случаях, когда одно и то же действие связывается с разными элементами управления графического интерфейса. Например, когда диалоговое окно открывается с помощью элемента меню и кнопки. Интерфейс `Action`, помимо обработчика событий — метода `actionPerformed()`, наследуемого от интерфейса `ActionListener`, с помощью своих методов позволяет активировать или дезактивировать действие для связанных с ним компонентов управления, а также определить текстовые строки, описывающие действие (могут использоваться как всплывающий текст кнопки или как текст элемента меню), значки, изображающие действие (могут применяться как изображения в меню) и другие свойства. Реализовать интерфейс `Action` можно, либо создав реализующий его класс, либо создав подкласс класса `javax.swing.AbstractAction`, реализующий данный интерфейс по умолчанию.

Использование интерфейсов Action и ActionListener позволяет организовать обработку событий двумя способами. Первый способ — это создать класс, реализующий интерфейс ActionListener, переопределив в нем метод интерфейса actionPerformed. Затем методом addActionListener класса javax.swing.AbstractButton присоединить созданный объект класса к компоненту управления. Второй способ — это создать класс, расширяющий абстрактный класс AbstractAction. В этом классе определить в конструкторе такие свойства компонента, как текстовые строки, которые могут быть использованы для показа всплывающего текста или для установки текста в меню, значков, состояния активации/дезактивации действия для элементов управления графического интерфейса, "горячих" клавиш и некоторых других свойств. Затем переопределить метод actionPerformed(), задав обработку событий. После этого присоединить созданный объект класса к компоненту управления с помощью метода setAction() класса AbstractButton.

Аннотация @Action платформы Swing Application Framework реализует второй способ обработки событий. Аннотация @Action, помимо маркировки метода — обработчика событий, позволяет определить ресурсы для инициализации свойств действия — всплывающий текст, значки и другие свойства. С помощью параметра аннотации enabledProperty можно связать состояние активации/дезактивации действия со значением логического свойства. С помощью параметра аннотации selectedProperty можно связать состояние выбора действия для таких компонентов управления, как JRadioButton или JToggleButton, со значением логического свойства. С помощью параметра аннотации block можно заблокировать весь графический интерфейс или его часть во время выполнения действия, определенного в качестве фоновой задачи.

Механизм использования аннотации @Action следующий:

1. Для каждой аннотации @Action, в созданном классе приложения, с помощью класса org.jdesktop.application.ApplicationActionMap создается объект класса javax.swing.ActionMap, содержащий один-единственный объект класса org.jdesktop.application.ApplicationAction (реализующий интерфейс javax.swing.Action и расширяющий класс javax.swing.AbstractAction).
2. Свойства объекта класса ApplicationAction инициализируются с помощью ресурсного properties-файла созданного класса приложения в конструкторе класса ApplicationAction.
3. С помощью метода setAction() класса javax.swing.AbstractButton действие связывается с компонентом управления, при этом в качестве аргумента методу передается объект javax.swing.Action, определенный с по-

мощью метода `get` вышеупомянутого объекта класса `ActionMap` с аргументом — именем промаркированного аннотацией метода.

Таким вот образом, с помощью аннотации `@Action` реализуется второй способ организации обработки событий.

Класс `org.jdesktop.application.ResourceMap` инкапсулирует наборы ресурсов приложения, хранящиеся в ресурсных properties-файлах, и обеспечивает доступ к ним, позволяя вводить значения ресурсов в свойства графических компонентов. Кроме того, можно использовать аннотацию `@Resource` для маркировки полей, значения которых можно ввести методом `injectFields()` класса `ResourceMap`, используя ресурсы с тем же именем, что и соответствующее поле.

Класс `org.jdesktop.application.FrameView` расширяет класс `org.jdesktop.application.View`, который содержит контейнер `javax.swing.JRootPane`. В контейнер `JRootPane` добавляются все остальные графические компоненты (меню, панель инструментов, панель состояния и др.), и ему делегируют свои операции контейнеры верхнего уровня графического интерфейса Swing — `JFrame`, `JDialog`, `JWindow` и `JApplet`. Также, класс `View` обеспечивает доступ к наборам ресурсов с помощью метода `getResourceMap()`, возвращающего объект `ResourceMap` для объекта `View`. Класс `FrameView`, расширяя класс `View`, является реализацией окна `JFrame`, содержащего методы для создания окна и инициализации его свойств.

Класс `org.jdesktop.application.TaskMonitor` служит моделью в архитектуре MVC для графического компонента — панели состояния, которая отображает состояние фоновых задач приложения (Task). `TaskMonitor` обеспечивает представление как всех задач, так и состояние единственной приоритетной (foreground) задачи.

Фоновые задачи приложений на основе платформы Swing Application Framework представляют абстрактный класс `org.jdesktop.application.Task<T,V>`, расширяющий класс `org.jdesktop.swingworker.SwingWorker<T,V>` (тождественен классу `javax.swing.SwingWorker<T,V>` библиотеки Java SE 6 API). Абстрактный класс `SwingWorker` обеспечивает выполнение задач, требующих значительного времени в отдельном, фоновом потоке. Таким образом, не блокируется графический интерфейс, активность компонентов которого обеспечивается потоком распределения событий Event Dispatch Thread (EDT).

Класс `SwingWorker` является настраиваемым классом, использующим параметризованные типы `T` и `V`, где `T` — тип результата вычислений в фоновом потоке, возвращаемый методами `doInBackground()` и `get()` класса `SwingWorker`. Любой подкласс класса `SwingWorker` должен переопределить метод `doInBackground()` для организации вычислений в фоновом режиме. Метод

get() возвращает результат вычислений после того, как они будут завершены. V — тип промежуточных результатов, используемый методами publish() и process() класса SwingWorker. Метод publish() применяется в случае, когда необходимо передать промежуточные результаты вычислений, полученные в методе doInBackground(), для обработки в другом потоке — потоке обработки событий EDT, в методе process(). С помощью методов publish() и process() организуется коммуникация между фоновым потоком и потоком обработки событий EDT в процессе фоновых вычислений. Метод done() класса SwingWorker также исполняется в потоке обработки событий EDT после того, как выполнение метода doInBackground() завершено. С помощью метода done() организуется коммуникация между фоновым потоком и потоком обработки событий EDT после завершения фоновых вычислений для обработки результатов вычислений уже в потоке обработки событий. Методом execute() объект подкласса SwingWorker ставится в очередь на выполнение в фоновом потоке. Очередь на выполнение реально формируется в случае, когда все фоновые потоки заняты. В классе SwingWorker заложено максимальное число фоновых (рабочих) потоков — 10.

Класс SwingWorker создан на основе технологии JavaBeans. Он имеет контролируемые извне атрибуты — progress (значение от 0 до 100, определяющее текущее выполнение фоновых вычислений) и state (переменная перечисляемого типа, принимающая значения PENDING, STARTED, DONE — первоначальное состояние, состояние перед вызовом метода doInBackground() и состояние после завершения метода doInBackground() соответственно). Методы addPropertyChangeListener() и removePropertyChangeListener() класса SwingWorker, соответственно, добавляют в список и удаляют из списка объекты, контролирующие данные атрибуты. С помощью метода firePropertyChange() производится уведомление контролирующих объектов об изменении атрибутов.

Класс org.jdesktop.application.Task<T,V> является расширением класса SwingWorker. В класс Task добавлены контролируемые атрибуты.

- Атрибут taskService является объектом класса org.jdesktop.application.TaskService. Атрибут принимает ненулевое значение, если объект класса Task обрабатывается объектом класса TaskService, т. е. фоновая задача запускается с помощью класса TaskService, который инкапсулирует интерфейс java.util.concurrent.ExecutorService. Класс TaskService предназначен для управления фоновыми задачами, которые, например, могут запускаться последовательно или сериями с помощью этого класса.
- Атрибуты title, description, message. Значения атрибутов инициализируются при создании объекта класса Task в конструкторе, используя ресурс-

ный properties-файл, при этом имена атрибутов совпадают с соответствующими ресурсами. Атрибут `title` служит для краткого описания задачи. Атрибут `description` — для более длинного описания задачи, состоящего из нескольких предложений. Атрибут `message` — короткое сообщение пользователю о текущем состоянии задачи.

- Атрибут `userCanCancel` используется для того, чтобы обозначить возможность прекращения пользователем выполнения данной задачи; по умолчанию устанавливается значение `true`.
- Атрибут `inputBlocker` является объектом класса `org.jdesktop.application.Task.InputBlocker`, который определяет, какая часть графического интерфейса блокируется во время выполнения задачи с помощью класса `TaskService`, при этом используется переменная перечисляемого типа `BlockingScope`. Переменная `BlockingScope` может принимать значения: `NONE` (интерфейс не блокируется), `ACTION` (блокируется действие, связанное с компонентом), `COMPONENT` (блокируется компонент), `WINDOW` (блокируется окно верхнего уровня), `APPLICATION` (блокируются все окна приложения). Значение `BlockingScope` устанавливается параметром `block` аннотации `@Action`.

Методы `succeeded()`, `failed()`, `cancelled()`, `interrupted()`, `finished()` добавлены в класс `Task` для расширенной обработки завершения задачи. Класс `Task` использует интерфейс `org.jdesktop.application.TaskListener<T,V>`, который предназначен для мониторинга процесса выполнения фоновой задачи, обеспечивая вызов методов интерфейса `cancelled()`, `doInBackground()`, `failed()`, `finished()`, `interrupted()`, `process()`, `succeeded()` в потоке обработки событий EDT и, таким образом, реализуя механизм уведомления о состоянии выполнения фоновой задачи. Методом `addTaskListener()` класса `Task` регистрируются объекты `TaskListener`, которые уведомляются об изменении состояния выполнения задачи.

С помощью класса `TaskMonitor` осуществляется связь между панелью состояния графического интерфейса и состоянием выполняемых фоновых задач. При создании объекта класса `TaskMonitor` конструктору класса в качестве аргумента передается экземпляр класса `ApplicationContext`, который содержит список объектов класса `TaskService`, управляющих выполнением фоновых задач приложения. Метод `getTaskService()` класса `ApplicationContext` позволяет получить список объектов `TaskService`, управляющих выполнением фоновых задач приложения. Классы `TaskMonitor`, `TaskService` и `Task` построены на основе технологии JavaBeans. Объект `TaskService` является слушателем изменений граничных свойств (контролируемых атрибутов) объекта `Task`, соответственно он же уведомляет об их изменениях внешних слушателей. Объект класса `TaskMonitor`, получив при создании список объектов

TaskService, автоматически при своем создании, в конструкторе, становится слушателем и уведомителем изменений их свойств. Так осуществляется связь между панелью состояния графического интерфейса и состоянием выполняемых фоновых задач через объект TaskMonitor. Кроме того, объект TaskMonitor содержит очередь фоновых задач на выполнение. Свойства первой в очереди задачи (foreground Task) являются свойствами объекта TaskMonitor. После выполнения данной задачи свойства foreground Task сбрасываются и принимают свойства следующей в очереди задачи, которая становится соответственно foreground Task. С помощью метода setAutoUpdateForegroundTask(false) класса TaskMonitor можно предотвратить сброс свойств foreground Task после завершения первой в очереди задачи. Таким образом, обеспечивается контроль над состоянием одной-единственной фоновой задачи.

Класс java.awt.event.ActionEvent предназначен для представления события, указывающего на то, что действие, связанное с компонентом, произошло. Когда происходит действие, сгенерированный объект ActionEvent передается объекту класса, реализующего интерфейс java.awt.event.ActionListener и зарегистрированного для получения таких событий от соответствующего компонента управления с помощью метода addActionListener() класса javax.swing.AbstractButton. При этом вызывается метод actionPerformed() интерфейса ActionListener — обработчик данного события.

Класс javax.swing.Timer отвечает за генерацию одного или более объектов ActionEvent через определенные интервалы времени. При создании объекта класса Timer конструктору передаются в качестве параметров значение временной задержки, которое определяет как первоначальную задержку генерации события, так и интервал между генерациями в миллисекундах, а также объект класса, реализующий интерфейс ActionListener с переопределенным методом actionPerformed(), который собственно и срабатывает через определенные промежутки времени. После создания объекта класса Timer значения первоначальной задержки и интервала между генерациями могут быть изменены независимо, кроме того, могут быть добавлены другие объекты классов, реализующих интерфейс ActionListener. Таймер запускается с помощью методов start() и restart(), останавливается методом stop(). Методом setRepeats(false) устанавливается режим таймера, когда он генерирует только одно событие, а затем останавливается.

Интерфейс javax.swing.Icon отвечает за отображение небольших, фиксированного размера изображений, как правило, используемых для декорирования графических компонентов.

Класс javax.swing.JDialog является главным классом для создания диалоговых окон.

Класс `javax.swing.JFrame` представляет собой Swing-контейнер верхнего уровня, реализующий графический компонент — окно.

После обсуждения предназначения классов и интерфейсов, импортируемых классом `SearchInternetView` нашего приложения, рассмотрим код тела создаваемого класса.

Класс приложения `SearchInternetView` является расширением класса `org.jdesktop.application.FrameView`, базового класса платформы Swing Application Framework для создания форм — окон пользовательского графического интерфейса.

В конструкторе создаваемого класса `SearchInternetView` с помощью ключевого слова `super` вызывается конструктор родительского класса `FrameView` с созданием пустого экземпляра класса `org.jdesktop.application.View` для экземпляра класса `Application`. Далее вызывается метод `initComponents`, который отвечает за инициализацию формы приложения.

После инициализации формы приложения создается объект класса `ResourceMap`, в который с помощью метода `getResourceMap` класса `View` записывается содержимое объекта `ResourceMap`, связанного с объектом класса приложения `SearchInternetView`.

Далее в целочисленную переменную `messageTimeout` методом `getInteger` класса `ResourceMap` записывается значение ресурса с именем `StatusBar.messageTimeout` из файла `SearchInternetView.properties`. Данное значение определяет для объекта класса `Timer` первоначальную задержку и временной интервал между генерациями.

После задания значения переменной `messageTimeout` создается объект `messageTimer` класса `Timer`, который содержит обработчик событий, предназначенный для установки подписи к панели состояния в виде пустой строки.

Метод `setRepeats(false)` устанавливает такой режим таймера, когда он генерирует только одно событие, а затем останавливается.

В целочисленную переменную `busyAnimationRate` методом `getInteger` класса `ResourceMap` записывается значение ресурса с именем `StatusBar.busyAnimationRate` из файла SearchInternetView.properties.

После этого в массив `busyIcons` объектов анонимного класса, реализующего интерфейс `javax.swing.Icon`, с помощью метода `getIcon` класса `ResourceMap` записываются объекты `Icon`, определенные ресурсами с именами `StatusBar.busyIcons[x]`, представляющими собой относительные пути к файлам изображений busy-icon[x].png.

Далее создается объект класса `Timer busyIconTimer`, который содержит обработчик событий, предназначенный для смены изображения, декорирующего

панель состояния. При этом значение переменной `busyAnimationRate` определяет первоначальную задержку и временной интервал между сменой изображений.

Объект `idleIcon`, инициализируемый с помощью ресурса с именем `StatusBar.idleIcon`, декорирует панель состояния между запусками таймера.

Состояние объекта `progressBar` класса `JProgressBar` устанавливается в состояние невидимости с помощью метода `setVisible(false)` класса `javax.swing.JComponent`, наследуемого классом `javax.swing.JProgressBar`. Класс `JProgressBar` представляет графический компонент — индикатор прогресса выполнения задачи.

После первоначального задания состояния объекта `progressBar` создается объект `taskMonitor` класса `TaskMonitor`. С помощью метода `addPropertyChangeListener`, унаследованного классом `TaskMonitor` от класса `AbstractBean`, регистрируется экземпляр анонимного класса, реализующий интерфейс `java.beans.PropertyChangeListener` и являющийся слушателем изменений граничных свойств объекта `TaskMonitor`. При этом переопределяется метод `propertyChange` интерфейса `PropertyChangeListener`. В нашем случае данный метод автоматически вызывается при изменении свойств объекта `TaskMonitor`. В качестве аргумента метод получает объект класса `java.beans.PropertyChangeEvent`, который генерируется при изменении граничных свойств, и содержит имя и значения измененного свойства. Имя измененного свойства может быть получено методом `getPropertyName` класса `PropertyChangeEvent`.

Если фоновая задача запускается, то соответственно изменяется свойство `started` объекта `Task`. В этом случае, в переопределенном методе `propertyChange` организуется смена изображения, декорирующего панель состояния. При этом делается видимым графический компонент — индикатор прогресса выполнения задачи. С помощью метода `setIndeterminate(true)` класса `javax.swing.JProgressBar` устанавливается режим анимации прогресса выполнения фоновой задачи с неизвестными начальным и конечным значениями.

Если фоновая задача завершается, то изменяется свойство `done` объекта `Task`. В этом случае, в переопределенном методе `propertyChange()` останавливается смена изображений. Конечное изображение, декорирующее панель состояния, устанавливается как значок, определенный при инициализации. Индикатор прогресса делаем невидимым, и его состояние устанавливается в первоначальное положение.

При создании объекта `Task` происходит инициализация атрибутов `title`, `description` и `message` в конструкторе класса с использованием ресурсного

properties-файла на уровне приложения, при этом имена атрибутов совпадают с соответствующими ресурсами. Атрибут `message` служит в качестве короткого сообщения пользователю о текущем состоянии задачи.

В момент создания объекта `Task` происходит изменение свойства `message`. В этом случае в переопределенном методе `propertyChange` устанавливается текст подписи к панели состояния в виде пустой строки, т. к. ресурсный файл SearchInternetApp.properties не содержит значения данного атрибута, и метод `getNewValue` класса `PropertyChangeEvent` возвращает в нашем случае `null`. После этого перезапускается таймер `messageTimer`, который с задержкой `messageTimeout` вызовет метод `actionPerformed`. В методе `actionPerformed` текст подписи к панели состояния снова будет установлен в виде пустой строки. При желании можно переустановить тексты подписи к панели состояния для обеспечения большей информативности графического интерфейса. Если во время выполнения фоновой задачи атрибут `message` будет изменяться, то соответственно будет изменяться и текст подписи к панели состояния. Закомментируем перезапуск таймера `messageTimer` для того, чтобы текст панели состояния, в случае появления, не переустанавливался в виде пустой строки.

При выполнении фоновых вычислений в методе `doInBackground()`, унаследованном классом `Task` от класса `SwingWorker`, происходит изменение атрибута `progress`, принимающего значение от 0 до 100 и указывающего текущее выполнение фоновых вычислений. В случае организации фоновых вычислений в методе `doInBackground()`, в переопределенном методе `propertyChange()` устанавливается режим анимации прогресса выполнения фоновой задачи в виде показа числовых значений атрибута `progress` с помощью получения текущего значения атрибута методом `getNewValue()` класса `PropertyChangeEvent`. При этом посредством метода `setVisible(true)` класса `JProgressBar` устанавливается видимым индикатор прогресса выполнения задачи.

Метод `showAboutBox()` маркируется аннотацией `@Action`, при этом он становится методом-обработчиком событий для одноименного действия, методом, в котором с помощью статического метода `getApplication()` класса `SearchInternetApp` создается экземпляр класса, затем методом `getMainFrame()` класса `SingleFrameApplication` получается экземпляр класса `JFrame`, представляющий главное окно приложения. Далее создается объект класса приложения `SearchInternetAboutBox`, представляющий диалоговое окно. Методом `setLocationRelativeTo()`, наследуемым классом `javax.swing.JDialog` от класса `java.awt.Window`, устанавливается расположение диалогового окна относительно главного окна приложения таким образом, что центр диалогового окна совпадает с центром главного окна. С помощью метода `show(javax.swing.JDialog c)` класса `SingleFrameApplication` диалоговое окно выводится на экран.

Аннотация `java.lang.Annotation` Type SuppressWarnings в данном случае запрещает предупреждение "unchecked" компилятора при компиляции метода `initComponents()`, т. к. код данного метода генерируется средой NetBeans, гарантированно отлажен и не может быть изменен в редакторе. Это предупреждение компилятора означает, что компилятор не имеет достаточной информации о типах данных, чтобы выполнить полную проверку типов данных и гарантировать типовую безопасность. Такая ситуация может возникнуть при использовании методов класса с настраиваемыми типами данных. Пример такого класса — `org.jdesktop.application.Task<T,V>`, где `T` и `V` — настраиваемые типы данных.

Далее обсудим код метода `initComponents()` класса `SearchInternetView`, вызываемого из конструктора и с помощью которого производится инициализация формы приложения (окна пользовательского графического интерфейса, включая компоненты).

В методе `initComponents()` создаются объекты классов, представляющие: главную панель окна (контейнер, в который добавляются все остальные графические компоненты); панель меню и элементы меню; панель состояния; разделительную линию, отделяющую панель состояния от остальных графических компонентов; подпись к панели состояния; иконку, декорирующую панель состояния; индикатор прогресса.

С помощью метода `setName()`, наследуемого классом `javax.swing.JPanel` от класса `java.awt.Container`, строка `"mainPanel"` связывается с компонентом `mainPanel`.

Далее создается объект класса `javax.swing.GroupLayout`, являющийся менеджером компоновки, группирующим компоненты в соответствии с их позицией в контейнере с соблюдением иерархии. Группы компонентов при этом могут быть параллельными или последовательными. В последовательной группе позиции компонентов следуют одна за другой вдоль оси, в параллельной группе компоненты имеют по оси одну и ту же координату. Конструктору класса в качестве аргумента передается контейнер `mainPanel` и тем самым определяется как корневой контейнер, в котором и будут располагаться остальные графические компоненты, формируемые менеджером компоновки в группы. Методом `setLayout()`, наследуемым классом `JPanel` от класса `Container`, для главной панели окна `mainPanel` устанавливаем менеджер компоновки — созданный объект `mainPanelLayout` класса `GroupLayout`. Если этого не сделать, то компоненты будут добавлены в панель не в порядке, определенном менеджером, а последовательно в одну строку. Метод `setLayout()` позволяет установить для контейнера один из нескольких созданных различных менеджеров компоновки, что может быть полезно при отладке и тестировании приложения.

С помощью методов `setHorizontalGroup()` и `setVerticalGroup()` класса `GroupLayout` созданные методом `createParallelGroup()` параллельные группы размещаются вдоль горизонтальной и вертикальной осей. В нашем случае созданные группы пустые, т. к. мы не добавляли в главную панель окна приложения какие-либо графические компоненты. Параметр `LEADING` определяет выравнивание компонентов в группе по левому краю.

Методом `setName()`, наследуемым классом `javax.swing.JMenuBar` от класса `java.awt.Container`, строка `"menuBar"` связывается с компонентом `menuBar`.

Далее создается объект класса `org.jdesktop.application.ResourceMap`, обеспечивающий доступ к наборам ресурсов в файле SearchInternetView.properties, в такой последовательности:

1. Метод `getInstance()` класса `Application` возвращает объект главного класса приложения `SearchInternetApp`.
2. Затем метод `getContext()` возвращает соответствующий объекту `SearchInternetApp` объект класса `ApplicationContext`.
3. С помощью метода `getResourceMap()` класса `ApplicationContext` обеспечивается доступ к набору ресурсов класса `SearchInternetView`.

С помощью методов `setText()` и `setName()`, наследуемых классом `javax.swing.JMenu` от классов `javax.swing.AbstractButton` и `java.awt.Component` соответственно, устанавливается надпись на элементе меню `fileMenu`, определенная ресурсом `"fileMenu.text = File"`, и строка `"fileMenu"` связывается с компонентом `fileMenu`.

После этого создается объект класса `ActionMap`, который обеспечивает связь между именем действия и самим действием, и записывается в этот объект экземпляр класса `org.jdesktop.application.ApplicationActionMap`, являющийся расширением класса `ActionMap`, в такой последовательности:

1. Метод `getInstance()` класса `Application` возвращает объект класса `SearchInternetApp`.
2. Метод `getContext()` класса `Application` возвращает объект класса `ApplicationContext`, связанный с нашим объектом класса `SearchInternetApp` и управляющий его общими действиями, ресурсами и задачами.
3. Метод `getActionMap()` класса `ApplicationContext` возвращает объект класса `ApplicationActionMap`, в конструкторе которого для каждого метода, промаркированного аннотацией `@Action`:
   - считывается имя метода, параметры аннотации `enabledProperty`, `selectedProperty` и `block`;

- создается объект класса `org.jdesktop.application.ApplicationAction`, конструктору которого, в качестве аргументов, передается метод, имя метода, параметры аннотации, объект класса `ResourceMap`, который обеспечивает доступ к ресурсам приложения, при этом происходит инициализация свойств действия с помощью наборов ресурсов (имя ресурса должно совпадать с именем метода), а также реализуются параметры аннотации;
- имя метода связывается с самим действием;
- отслеживается изменение значений параметров аннотации `enabledProperty` и `selectedProperty`.

При создании объекта `actionMap` были обработаны не только методы самого класса `SearchInternetView`, промаркированные аннотацией `@Action`, но и методы класса `Application`, родительского для класса `SingleFrameApplication`, для объекта которого создается объект класса `SearchInternetView`. В классе `Application` аннотацией `@Action` промаркирован метод `quit()` — обработчик событий для действия, предназначенного для выхода из приложения. Это действие по умолчанию имеет свойства — подпись "Exit", "горячие" клавиши <Ctrl>+<Q>, всплывающую подсказку "Exit the application".

Методом `setAction()` класса `javax.swing.AbstractButton` действие `quit()` связывается с элементом меню `exitMenuItem`, при этом в качестве аргумента методу передается объект `javax.swing.Action`, определенный с помощью метода `get()` созданного объекта класса `ActionMap` с аргументом — именем промаркированного аннотацией метода `quit()`.

С помощью метода `setName()`, который класс `javax.swing.JMenuItem` наследует от класса `java.awt.Component`, строка `"exitMenuItem"` связывается с компонентом `exitMenuItem`, чтобы затем служить в качестве аргумента метода `add()`, наследуемого классом `javax.swing.JMenuItem` от класса `java.awt.Container`. Методом `add()` компонент `exitMenuItem` добавляется в компонент меню `fileMenu`.

Методом `add()`, наследуемым классом `javax.swing.JMenuBar` от класса `java.awt.Container`, компонент меню `fileMenu` добавляется в панель меню `menuBar`.

С помощью методов `setText()` и `setName()`, наследуемых классом `javax.swing.JMenu` от классов `javax.swing.AbstractButton` и `java.awt.Component` соответственно, устанавливается надпись на элементе меню `helpMenu`, определенная ресурсом `"helpMenu.text = Help"`, и строка `"helpMenu"` связывается с компонентом `helpMenu`.

Методом `setAction()` класса `AbstractButton` действие `showAboutBox` связывается с компонентом управления — элементом меню. При этом в качестве аргу-

мента методу передается объект `Action`, определенный с помощью метода `get()` созданного объекта класса `ActionMap` с аргументом — именем промаркированного аннотацией метода. Так как при создании объекта `actionMap` произошла инициализация свойств действий с помощью наборов ресурсов класса, то с элементом меню `aboutMenuItem` связываются подпись, определенная ресурсом `"showAboutBox.Action.text = &About..."`, и всплывающая подсказка, определенная ресурсом `"showAboutBox.Action.shortDescription = Show the application's information dialog"`. С помощью метода `setName()`, который класс `javax.swing.JMenuItem` наследует от класса `java.awt.Component`, строка `"aboutMenuItem"` связывается с компонентом `aboutMenuItem`, чтобы затем служить в качестве аргумента метода `add()`. С помощью метода `add()` компонент `aboutMenuItem` добавляется в компонент меню `helpMenu`.

С помощью метода `add()` компонент меню `helpMenu` добавляется в панель меню `menuBar`.

Методом `setName()` соответствующие строки связываются с компонентами — панелью состояния `statusPanel`, разделительной линией панели состояния `statusPanelSeparator`, подписью панели состояния `statusMessageLabel`, изображением панели состояния `statusAnimationLabel`, индикатором прогресса `progressBar`. Кроме того, методом `setHorizontalAlignment()` класса `javax.swing.JLabel` устанавливается выравнивание изображения панели состояния по левой стороне.

Для контейнера `statusPanel` создается менеджер компоновки `statusPanelLayout` — объект класса `javax.swing.GroupLayout`. С помощью методов `setHorizontalGroup()`, `setVerticalGroup()` и `createParallelGroup()` создаются параллельные группы вдоль горизонтальной и вертикальной осей, в которые помещаются графические компоненты, используя методы:

- метод `addComponent()` класса `javax.swing.GroupLayout.ParallelGroup` добавляет компоненты в параллельную группу, позволяя устанавливать выравнивание, минимальный, предпочтительный и максимальный размеры компонента;
- метод `addGroup()` класса `ParallelGroup` позволяет добавить группу в уже существующую группу, в нашем случае добавить в параллельную группу последовательную группу, созданную методом `createSequentialGroup()` класса `GroupLayout`;
- метод `addContainerGap()` класса `javax.swing.GroupLayout.SequentialGroup` добавляет элемент, представляющий предпочтительный промежуток между границей контейнера и компонентом, который касается границы контейнера;
- метод `addPreferredGap()` класса `SequentialGroup` добавляет элемент, представляющий предпочтительный промежуток между двумя ближайшими

компонентами, устанавливая тип промежутка (относительный или абсолютный), предпочтительный и максимальный размеры промежутка.

С помощью методов `setComponent()`, `setMenuBar()`, `setStatusBar()`, которые класс `SearchInternetView` наследует от класса `org.jdesktop.application.View` через класс `org.jdesktop.application.FrameView`, устанавливаются атрибуты класса:

- объект `mainPanel` — единственный главный компонент объекта `View`, устанавливается в центр корневой панели;
- объект `menuBar` — панель меню объекта `View`;
- объект `statusPanel` — панель состояния объекта `View`.

Класс `View` является расширением класса `org.jdesktop.application.AbstractBean` и инкапсулирует, в нашем случае, окно верхнего уровня `JFrame` и основные компоненты пользовательского графического интерфейса: главную панель, панель меню, панель инструментов и панель состояния, которые являются граничными свойствами класса. Методами `setComponent()`, `setMenuBar()` и `setStatusBar()` мы устанавливаем созданные объекты `mainPanel`, `menuBar` и `statusPanel` как компоненты формы приложения — атрибуты класса `SearchInternetView`, и делаем их видимыми.

После кода метода `initComponents()`, в блоке `Variables declaration` и ниже, производится объявление полей класса `SearchInternetView`.

Код класса *SearchInternetAboutBox* для диалогового окна

Далее разберем код файла SearchInternetAboutBox.java, сгенерированный средой NetBeans при создании основы нашего приложения и представляющий диалоговое окно, открывающееся из меню главного окна приложения.

> ***Примечание***
>
> Код класса `SearchInternetAboutBox` см. в файле SearchInternetAboutBox.java папки Часть_I\Глава_3\SearchInternet\src\searchinternet на прилагаемом к книге компакт-диске.

Класс `SearchInternetAboutBox`, код которого мы обсудим далее, представляет диалоговое окно, отображающее пользователю информацию о самом приложении. Данное диалоговое окно открывается из основного окна приложения элементом **About** меню **Help**.

Класс приложения `SearchInternetAboutBox` расширяет класс `javax.swing.JDialog`, который является базовым классом для создания диалоговых окон.

В конструкторе класса приложения `SearchInternetAboutBox`, с помощью ключевого слова `super`, вызывается конструктор родительского класса `JDialog`, в котором создается новое, изначально невидимое окно, являющееся дочерним по отношению к окну `parent`. Также создается рамка окна и устанавливается параметр, определяющий, является ли диалоговое окно модальным, т. е. блокирующим окно верхнего уровня, или нет.

По умолчанию диалоговое окно является немодальным. Параметр модальности можно изменить методом `setModal(true)` класса `java.awt.Dialog`, наследуемым классом `JDialog`.

В конструкторе класса `SearchInternetAboutBox` также вызывается метод `initComponents()`, который отвечает за инициализацию диалогового окна.

После вызова метода `initComponents()` методом `setDefaultButton()` класса `JRootPane` кнопка `closeButton` устанавливается кнопкой по умолчанию для корневой панели. Таким образом, кнопка `closeButton` активируется при нажатии клавиши <Enter>. Объект класса `javax.swing.JRootPane` возвращается методом `getRootPane()` класса `JDialog`. Здесь объект `JRootPane` представляет корневую панель-контейнер для нашего диалогового окна, содержащую панель меню и панель `contentPane`, в которую добавляются все остальные графические компоненты.

Метод `closeAboutBox()` класса `SearchInternetAboutBox` маркируется аннотацией `@Action`, при этом он становится обработчиком событий для одноименного действия. В методе `closeAboutBox()` вызывается метод `dispose()`, который класс `JDialog` наследует от класса `java.awt.Window`. Метод `dispose()` закрывает окно, освобождая все занятые ресурсы операционной системы.

Далее обсудим код метода `initComponents()` класса `SearchInternetAboutBox`, который вызывается из конструктора и с помощью которого производится инициализация диалогового окна.

В методе `initComponents()` создаются следующие объекты классов `javax.swing.JButton` и `javax.swing.JLabel`:

- `closeButton` — кнопка по умолчанию для корневой панели;
- `appTitleLabel` — название приложения;
- `versionLabel` и `appVersionLabel` — версия приложения;
- `vendorLabel` и `appVendorLabel` — производитель приложения;
- `homepageLabel` и `appHomepageLabel` — адрес домашней странички приложения;
- `appDescLabel` — краткое описание приложения;
- `imageLabel` — изображение, декорирующее диалоговое окно.

Метод `setDefaultCloseOperation()` класса `JDialog` устанавливает операцию, которая производится, если пользователь закрывает окно, воспользовавшись кнопкой **Закрыть** меню корневой панели. В данном случае происходит вызов метода `dispose()`, описанного ранее.

После применения метода `setDefaultCloseOperation()` создается объект `resourceMap` класса `org.jdesktop.application.ResourceMap`, обеспечивающий доступ к наборам ресурсов как на уровне приложения (файл SearchInternetApp.properties), так и на уровне класса (файл SearchInternetAboutBox.properties).

Далее метод `setTitle()` класса `java.awt.Dialog`, наследуемый классом `JDialog`, устанавливает заголовок для диалогового окна, определяя его с помощью метода `getString()` класса `ResourceMap` ресурсом `"title=About: ${Application.title} ${Application.version}"` файла SearchInternetAboutBox.properties, где `Application.title` и `Application.version` определяются соответствующими ресурсами файла SearchInternetApp.properties.

С помощью метода `setModal(true)` класса `Dialog`, наследуемого классом `JDialog`, диалоговое окно устанавливается модальным, т. е. блокирующим окно верхнего уровня.

Методом `setName()`, который класс `JDialog` наследует от класса `java.awt.Component`, строка — имя `"aboutBox"` связывается с диалоговым окном.

Методом `setResizable(false)` класса `Dialog` устанавливается, что пользователь не сможет изменять размеры диалогового окна.

Далее создается объект `actionMap` класса `javax.swing.ActionMap`, который обеспечивает связь между именами действий и самими действиями, включая их свойства.

С помощью метода `setAction()`, который класс `javax.swing.JButton` наследует от класса `javax.swing.AbstractButton`, действие `"closeAboutBox"` связывается с кнопкой `closeButton`, при этом в качестве аргумента методу передается объект `Action`, определенный с помощью метода `get()` созданного объекта класса `ActionMap` с аргументом — именем промаркированного аннотацией метода, совпадающего с именем действия. При создании объекта `actionMap` инициализируются свойства действий посредством наборов ресурсов класса, поэтому с кнопкой `closeButton` связывается подпись, определенная ресурсом `"closeAboutBox.Action.text = &Close"` файла SearchInternetAboutBox.properties.

Методом `setName()` имя `"closeButton"` связывается с кнопкой управления **Close**.

Для объектов `JLabel`, представляющих в диалоговом окне название, версию, производителя, домашнюю страничку и краткое описание приложения, а также декорирующее изображение, устанавливается шрифт с помощью метода `setFont()`, наследуемого от класса `javax.swing.JComponent`, в такой последовательности:

1. Метод `getFont()`, наследуемый от класса `java.awt.Component`, возвращает объект `java.awt.Font`, представляющий шрифт, установленный по умолчанию. Как уже упоминалось ранее, библиотека Swing реализует архитектуру MVC (Model-View-Controller), как архитектуру, состоящую из двух частей: представитель пользовательского интерфейса (объединяет вид и контроллер и отвечает за обработку событий и вывод на экран графического компонента) и модель. Каждому графическому компоненту соответствует представитель, с которым он связывается с помощью класса `javax.swing.UIManager`. Класс `UIManager` управляет внешним видом компонентов с помощью класса `javax.swing.LookAndFeel`, объект которого инкапсулирует информацию о внешнем виде и поведении всех графических компонентов. Таким образом, все графические компоненты имеют по умолчанию установленные шрифт, цвет, форму и другие свойства.
2. Методом `deriveFont()` класса `Font` создается новый объект `Font` с помощью копирования старого объекта и изменения некоторых его свойств, таких как стиль и размер.
3. Методом `getStyle()` класса `Font` получается целочисленная константа (`PLAIN`, `BOLD`, `ITALIC`, `BOLD+ITALIC`), определяющая стиль шрифта.
4. Оператором `|` смешивается стиль, установленный по умолчанию, и `java.awt.Font.BOLD`.
5. Методом `getSize()` класса `Font` получается целочисленное значение, определяющее размер шрифта по умолчанию, которое затем увеличивается.

С помощью методов `getString()` и `getIcon()` класса `ResourceMap` получаются значения ресурсов, определенных в файле SearchInternetAboutBox.properties, которые методами `setText()` и `setIcon()` класса `JLabel` связываются с соответствующими графическими компонентами.

Методом `setName()`, который класс `JLabel` наследует от класса `java.awt.Component`, соответствующие имена связываются с компонентами.

Для контекстной панели, которая добавлена в корневую панель диалогового окна и которая является контейнером для всех остальных графических компонентов, создается менеджер компоновки `layout` — объект класса `javax.swing.GroupLayout`.

С помощью методов `setHorizontalGroup()`, `setVerticalGroup()`, `createParallelGroup()` и `createSequentialGroup()` класса `GroupLayout` создают-

ся параллельные группы, с вхождением в них последовательных групп, вдоль горизонтальной и вертикальной осей. В созданных группах размещаются графические компоненты с помощью методов `addComponent()` и `addGroup()` класса `javax.swing.GroupLayout.ParallelGroup`, методов `addGroup()`, `addComponent()`, `addGap()`, `addContainerGap()` и `addPreferredGap()` класса `javax.swing.GroupLayout.SequentialGroup`.

Метод `pack()`, который класс `javax.swing.JDialog` наследует от класса `java.awt.Window`, устанавливает предпочтительные размеры и планировку окна.

После кода метода `initComponents()` производится объявление полей класса `SearchInternetAboutBox`.

Разработка графического интерфейса приложения

Получив сгенерированную средой NetBeans основу создаваемого приложения, теперь нам необходимо добавить в графический интерфейс нужные для работы приложения компоненты.

Напомним, что создаваемое приложение по запросу пользователя осуществляет поиск информации в Интернете и сохраняет найденную информацию в специальной папке. Поэтому нам понадобится текстовое поле для ввода запроса, поле для ввода количества страниц, которые необходимо найти в Интернете, и кнопка для отправки запроса.

Для создания окончательного варианта графического интерфейса приложения:

1. Откроем в среде NetBeans файл SearchInternetView.java, переключившись в окно визуализации формы с помощью элемента меню **Design**.
2. Воспользуемся палитрой компонентов **Palette**, с помощью которой разместим в главном окне приложения компоненты управления **Swing Controls**. Нажмем левую кнопку мыши на компоненте **Text Field** и, удерживая кнопку, перенесем компонент на форму (рис. 3.5).
3. Увеличим размеры текстового поля и, щелкнув правой кнопкой мыши на компоненте, выберем в контекстном меню пункт **Change Variable Name**. Далее в появившемся диалоговом окне введем имя компонента `jTextFieldReq` (рис. 3.6).
4. Щелкнув правой кнопкой мыши на компоненте, откроем диалоговое окно **Properties**. Очистим свойство **text** (рис. 3.7). Это даст нам пустое поле при отображении окна интерфейса.
5. Перенесем компонент **Label** на форму (рис. 3.8).

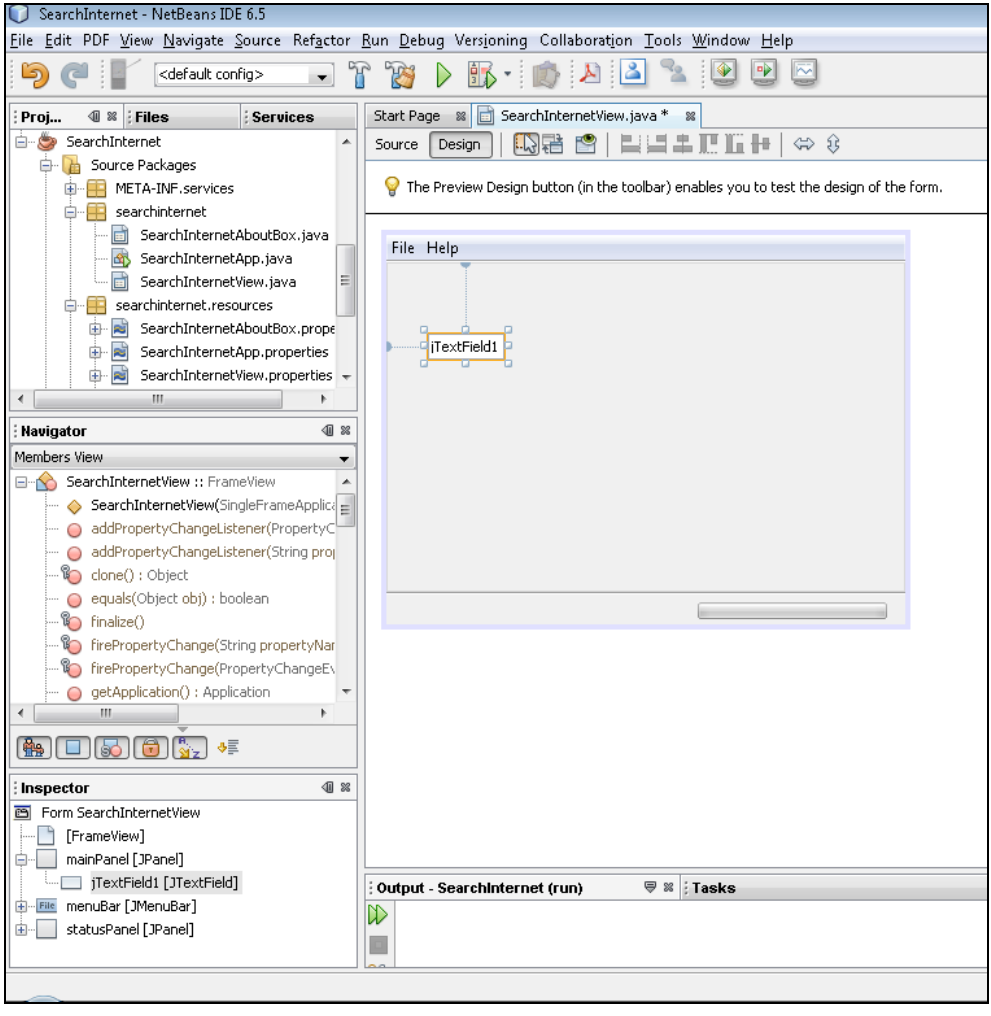

Рис. 3.5. Создание текстового поля

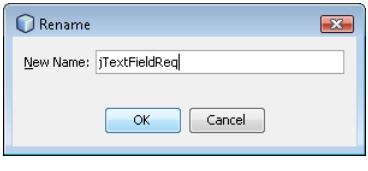

Рис. 3.6. Окно ввода нового имени компонента

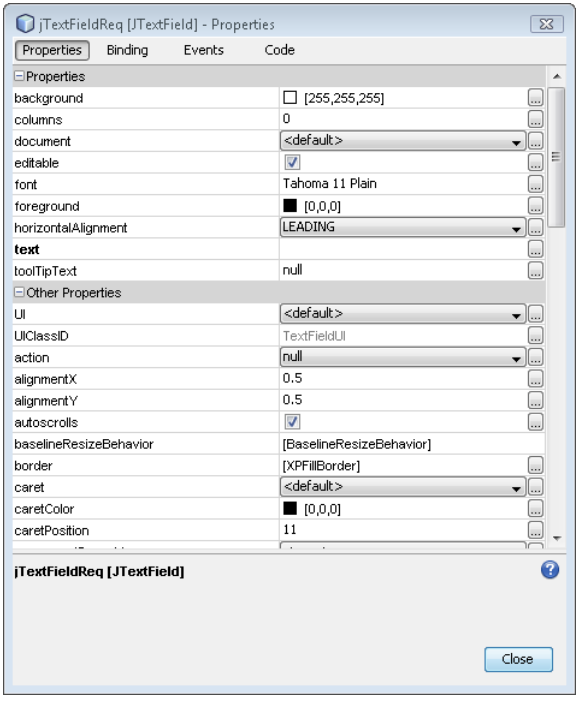

Рис. 3.7. Окно свойств текстового поля

6. Переименуем компонент в `jLabelReq` и в диалоговом окне **Properties**, в строке **text**, введем "Введите тему запроса".

7. Добавим еще одно текстовое поле — `jTextFieldNum`. При этом в диалоговом окне **Properties**, в строке **text**, введем "1" — количество сохраняемых приложением Web-страниц по умолчанию. К текстовому полю добавим подпись `jLabelNum` с текстом "Введите количество страниц".

8. Создадим кнопку `jButtonSubmit` с надписью "Отправить".

Теперь мы имеем графический интерфейс нужного нам вида (рис. 3.9).

В текстовое поле `jTextFieldReq` пользователь будет вводить свой запрос, указывая в поле `jTextFieldNum` количество Web-страниц, которые необходимо сохранить в специальной папке. При нажатии кнопки `jButtonSubmit` в Интернете будет осуществляться поиск информации, соответствующей введенному запросу.

Так как поиск в Интернете требует значительного времени, создадим действие, связанное с кнопкой `jButtonSubmit`, события которого будут обрабатываться в фоновом потоке, не блокирующем графический интерфейс.

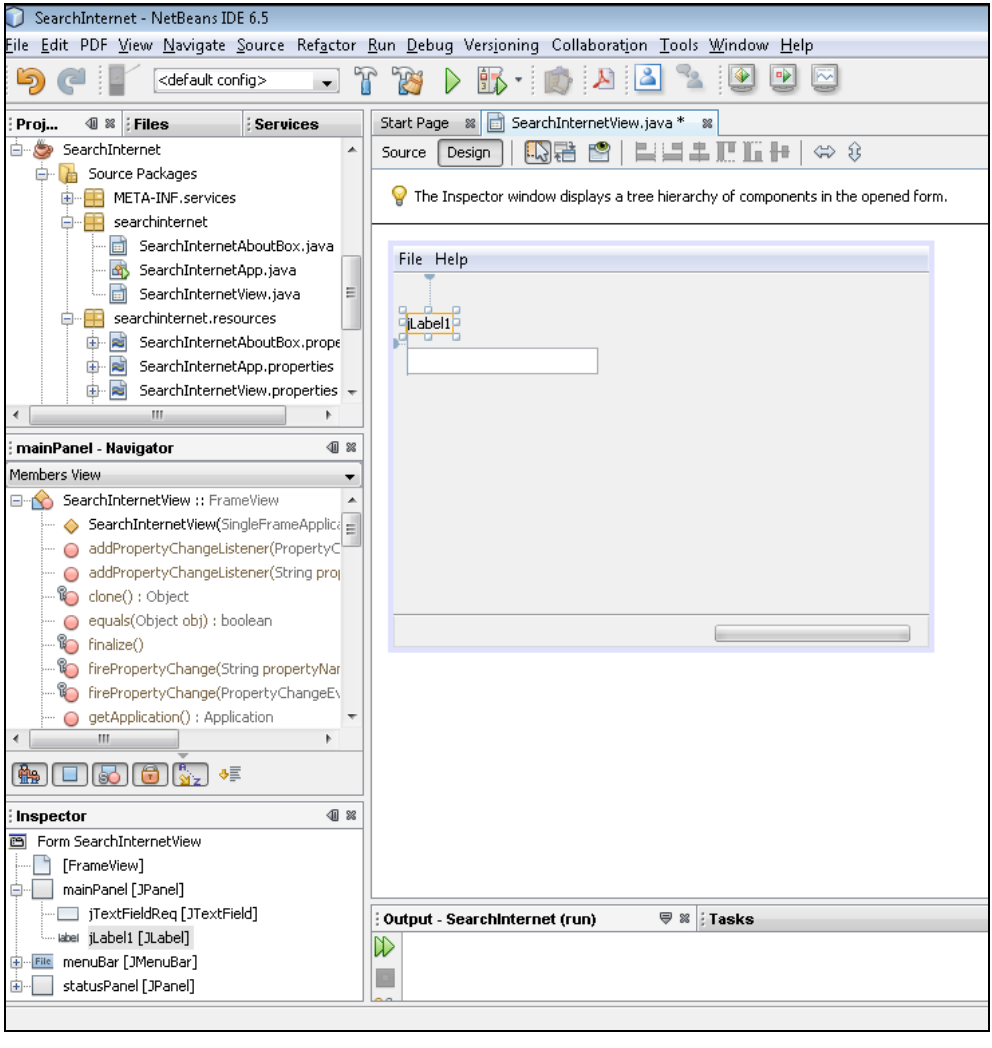

Рис. 3.8. Создание подписи к текстовому полю

Для этого щелкнем правой кнопкой мыши на кнопке **Отправить** и откроем диалоговое окно **Set Action**. В раскрывающемся списке **Action** выберем вариант **Create New Action**. В поле **Action's Method** введем название "Submit" действия и, соответственно, имя метода, обработчика событий действия, который будет промаркирован аннотацией `@Action`. Отметим флажок **Background Task** (рис. 3.10).

После нажатия кнопки **OK** окна **Set Action** среда NetBeans сгенерирует код, которым дополнит код класса `SearchInternetView` (листинги 3.2 и 3.3).

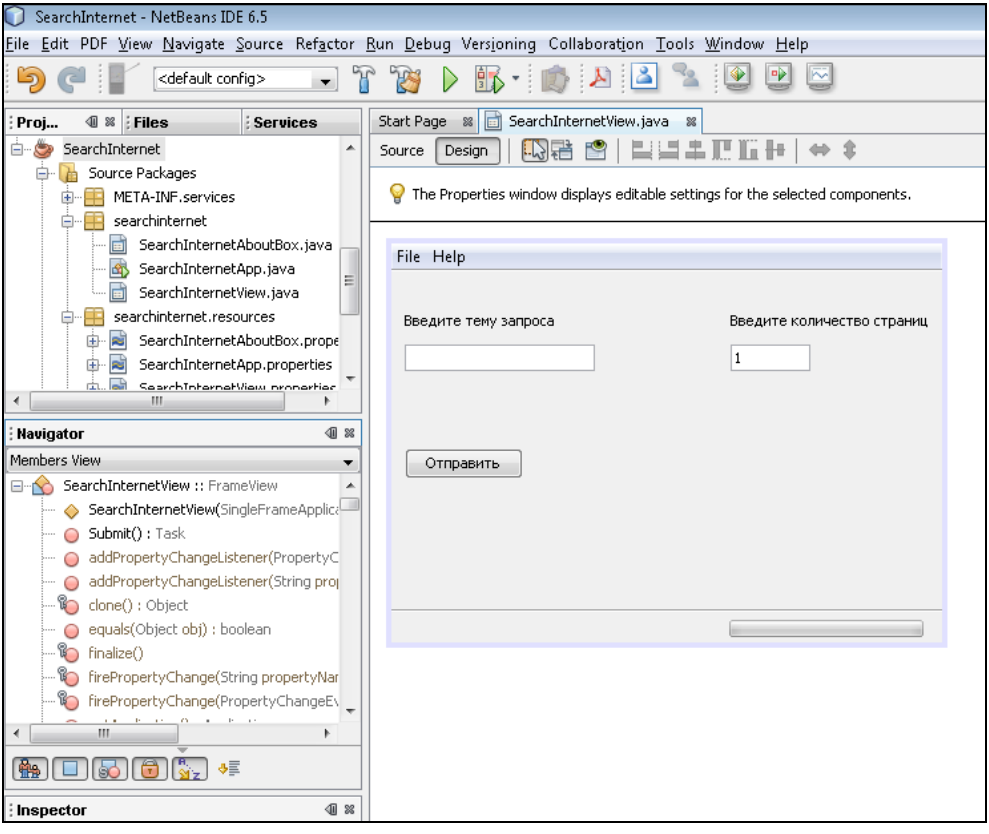

Рис. 3.9. Окончательный вид графического интерфейса

Листинг 3.2. Дополнительный сгенерированный код метода `initComponents()` класса `SearchInternetView`

```
jButtonSubmit.setAction(actionMap.get("Submit")); // NOI18N
jButtonSubmit.setText(resourceMap.getString("jButtonSubmit.text"));
  // NOI18N
jButtonSubmit.setName("jButtonSubmit"); // NOI18N
```

Листинг 3.3. Дополнительный сгенерированный код класса `SearchInternetView`

```
@Action
public Task Submit() {
  return new SubmitTask(getApplication());
}
```

```
private class SubmitTask extends org.jdesktop.application.Task<Object,
    Void> {
  SubmitTask(org.jdesktop.application.Application app) {
    // Runs on the EDT.  Copy GUI state that
    // doInBackground() depends on from parameters
    // to SubmitTask fields, here.
    super(app);
  }
  @Override protected Object doInBackground() {
    // Your Task's code here.  This method runs
    // on a background thread, so don't reference
    // the Swing GUI from here.
    return null;  // return your result
  }
  @Override protected void succeeded(Object result) {
    // Runs on the EDT.  Update the GUI based on
    // the result computed by doInBackground().
  }
}
```

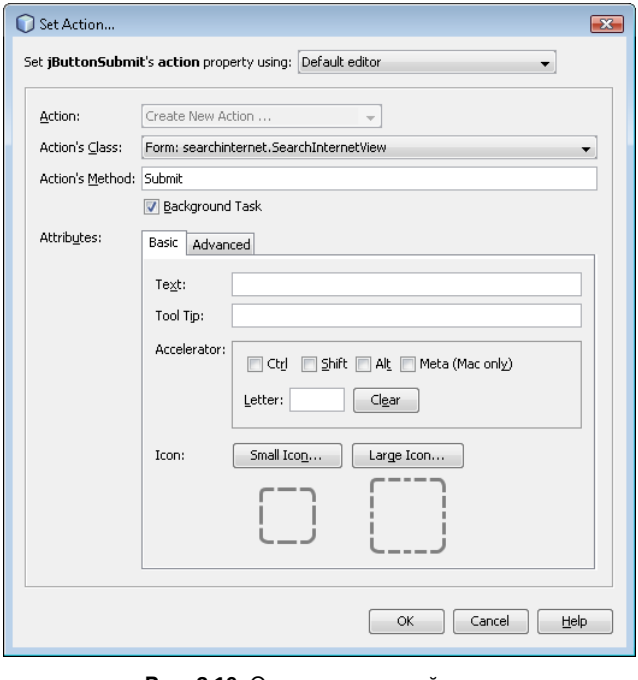

Рис. 3.10. Окно создания действия

В методе `initComponents()` класса `SearchInternetView` среда NetBeans сгенерирует код (см. листинг 3.2), в котором:

1. Методом `setAction()`, который класс `javax.swing.JButton` наследует от класса `javax.swing.AbstractButton`, действие `Submit` связывается с кнопкой `jButtonSubmit`.
2. Методом `setText()` класса `AbstractButton` надпись на кнопке связывается с ресурсом `"jButtonSubmit.text=Отправить"` файла SearchInternetView.properties.
3. Методом `setName()`, наследуемым от класса `java.awt.Component`, устанавливается имя компонента `jButtonSubmit`.

В классе `SearchInternetView` среда NetBeans сгенерирует код (см. листинг 3.3), который мы обсудим далее.

В сгенерированном коде, в обработчике событий действия `Submit` — методе `Submit()` создается объект класса `SubmitTask`, расширяющего класс `org.jdesktop.application.Task`.

В конструкторе сгенерированного внутреннего класса `SubmitTask` с помощью ключевого слова `super` вызывается конструктор родительского класса `Task`, при этом инициализируются атрибуты класса `title`, `description` и `message` с помощью ресурсного файла SearchInternetView.properties. В нашем случае данные атрибуты не определены в наборе ресурсов.

В классе `SubmitTask` переопределенный метод `doInBackground()` предназначен для организации выполнения операций в отдельном, фоновом потоке. Класс `Task` является расширением класса `SwingWorker`. Так как класс `SwingWorker` имеет конструктор по умолчанию, то при вызове конструктора класса `Task` сначала вызывается конструктор родительского класса `SwingWorker`. С помощью ключевого слова `super` необходимо вызывать конструктор родительского класса в случае, если он не имеет конструктор по умолчанию и требуется вызвать конкретный конструктор с аргументами. В конструкторе класса `SwingWorker` происходит вызов метода `doInBackground()`. После завершения метода `doInBackground()` автоматически происходит вызов метода `done()`, который выполняется уже в потоке обработки событий EDT, а не в фоновом потоке. В классе `Task` метод `done()` переопределен так, что в нем происходит вызов метода `succeeded(get())` в случае успешного выполнения метода `doInBackground()`, где `get()` — метод класса `SwingWorker`, возвращающий результат метода `doInBackground()`. В случае насильственного прерывания выполнения фоновой задачи в методе `done()` класса `Task` происходит вызов метода `cancelled()` класса `Task`. В случае возникновения ошибки при выполнении метода `doInBackground()`, т. е. при неудачном завершении метода, в методе `done()` класса `Task` происходит вызов метода `failed()` класса `Task`.

Метод `finished()` класса `Task` автоматически вызывается после выполнения любого из методов `succeeded()`, `failed()`, `cancelled()` или `interrupted()` (ошибочное прерывание при выполнении метода `doInBackground()`). Таким образом, для организации фонового выполнения задачи достаточно создать объект класса, расширяющего класс `Task`, с переопределенными методами `doInBackground()` и `succeeded()`.

Метод `succeeded()` класса `Task` вызывается в случае успешного выполнения фоновой задачи, при получении результата фоновых операций в методе `doInBackground()`. Метод `succeeded()` выполняется уже в потоке обработки событий EDT, и поэтому используется для обновления состояния графического интерфейса, основываясь на результате, возвращенном методом `doInBackground()`.

После создания действия, связанного с кнопкой **Отправить**, нам нужно, чтобы данное действие обеспечивало поиск соответствующей введенному запросу информации в Интернете и сохранение найденной информации в файловой системе компьютера.

Программирование работы с сетью

Так как наше приложение имеет дело с Интернетом, то нам понадобится пакет `java.net`, включающий в себя классы, обеспечивающие работу с сетью. Полученную из Интернета информацию мы собрались сохранять в специально создаваемой для этих целей папке, поэтому нам пригодится пакет `java.io`, предназначенный для организации потоков ввода/вывода. Обеспечим употребление коротких имен классов из этих пакетов с помощью строк кода `import java.net.*;` и `import java.io.*;`.

Далее в классе `SubmitTask` определим методы, которые будут вызываться в методе `doInBackground()`:

- метод `Search()` будет отвечать за поиск, используя поисковую систему Яндекс, ссылок на Web-страницы, содержащие информацию, которая соответствует введенному пользователем запросу;
- метод `Parse()` будет выделять из Web-страниц поисковой системы Яндекс, полученных с помощью метода `Search()`, URL-адреса Web-страниц, содержащих нужную информацию;
- метод `Save()` будет сохранять в специально создаваемой папке, имя которой будет таким же, как и запрос пользователя, Web-странички, находящиеся по адресам, выделенным методом `Parse()`.

Чтобы отслеживать процесс выполнения фоновой задачи, в методе `doInBackground()` будем изменять, после выполнения каждого из методов

Search(), Parse() и Save(), атрибут message класса Task, который связан у нас в конструкторе класса SearchInternetView с подписью к панели состояния statusMessageLabel. Для этого воспользуемся методом message() класса Task (листинг 3.4).

Листинг 3.4. Код использования атрибута message класса Task для отслеживания процесса выполнения фоновой задачи

```
@Override protected Object doInBackground() {
  // Your Task's code here.  This method runs
  // on a background thread, so don't reference
  // the Swing GUI from here.
  message("searchMessage");
  Search();
  message("parseMessage");
  Parse();
  message("saveMessage");
  Save();
  return null;  // return your result
}
```

Метод message() устанавливает свойство message класса Task в соответствии с ресурсом, определенным в аргументе метода. Данный ресурс относится к набору ресурсов на уровне приложения. В нашем случае набор ресурсов на уровне приложения определен в ресурсном файле SearchInternetApp.properties. Поэтому внесем соответствующие ресурсы в файл SearchInternetApp.properties с помощью строк "searchMessage=Поиск в Интернете", "parseMessage=Выделение URL-адресов" и "saveMessage=Загрузка и сохранение".

Информация, вводимая пользователем в текстовые поля, состоит из информационного запроса и количества Web-страниц, которые приложение должно в итоге сохранить. Определим эту информацию в виде атрибутов класса SubmitTask и произведем их инициализацию в конструкторе класса SubmitTask (листинг 3.5).

Листинг 3.5. Объявление полей класса SubmitTask и их инициализация в конструкторе

```
private String Text;
private int Number;

SubmitTask(org.jdesktop.application.Application app) {
```

```
  // Runs on the EDT.  Copy GUI state that
  // doInBackground() depends on from parameters
  // to SubmitTask fields, here.
  super(app);
  Text=jTextFieldReq.getText();
  Number=Integer.parseInt(jTextFieldNum.getText());
}
```

Метод `getText()`, который класс `javax.swing.JTextField` наследует от класса `javax.swing.text.JTextComponent`, возвращает текст, содержащийся в компоненте. Метод `parseInt()` класса `java.lang.Integer` возвращает целочисленное значение, представленное строкой в аргументе.

В переопределенном методе `succeeded()` класса `SubmitTask`, автоматически вызываемом в случае успешного выполнения фоновой задачи, установим подпись к панели состояния, сообщающую об успешном завершении задачи, и приведем текстовые поля, в которые пользователь ввел информацию, в первоначальное состояние (листинг 3.6).

Листинг 3.6. Код метода `succeeded()` класса `SubmitTask`

```
@Override protected void succeeded(Object result) {
  // Runs on the EDT.  Update the GUI based on
  // the result computed by doInBackground().
  statusMessageLabel.setText("Сохранение информации завершено");
  jTextFieldReq.setText("");
  jTextFieldNum.setText("1");
}
```

В методе `Search()` будем передавать запрос пользователя на сервер поисковой системы Яндекс не через форму на Web-странице сайта, а методом `GET()` — в этом случае информация формы передается вместе с URL-адресом (листинг 3.7).

Листинг 3.7. Код метода `Search()` класса `SubmitTask`

```
private void Search() {
  String url="http://yandex.ru/yandsearch?text="+Text+"&clid=22042";
  int n=(int)((9+Number)/10);
  page=new String();
  for (int i=0;i<n;)
  {
    try {
      URL bhv = new URL(url);
```

```
      BufferedReader br = new BufferedReader(
        new InputStreamReader(bhv.openStream()));
      String file = Text;
      File dir = new File(file);
      dir.mkdir();
      String buffer_page;
      while ((buffer_page = br.readLine()) != null) {
        page = page + buffer_page;
      }
      br.close();
      i++;
      url = "http://yandex.ru/yandsearch?" + "p=" + i +
            "&text=" + Text + "&clid=22042";
    } catch (IOException ex) {
      setMessage(ex.toString());
      cancel(true);
    }
  }
}
```

Далее обсудим код метода `Search()` (см. листинг 3.7).

В коде метода `Search()` строка `url` URL-адреса содержит запрос пользователя, представленный строковой переменной `Text`.

Так как страничка поисковой системы Яндекс содержит 10 ссылок по произведенному запросу, а количество сохраняемых страничек сайтов по ссылкам — `Number`, то количество страничек поисковой системы Яндекс равно `(9+Number)/10`, округленное до целого числа.

В методе `Search()` создается объект класса `java.net.URL`, представляющий URL-адрес — указатель на ресурс WWW. Метод `openStream()` класса `URL` открывает соединение с ресурсом и возвращает объект класса `java.io.InputStream`, представляющий входящий байтовый поток. Класс `java.io.InputStreamReader` отвечает за считывание байтов из входного байтового потока и преобразование их в символы. Класс `java.io.BufferedReader` отвечает за считывание символьного потока с организацией буфера в оперативной памяти для увеличения скорости чтения. Класс `java.io.File` предназначен для работы с файлами и каталогами в файловой системе и представляет абстрактное имя файла или каталога вместе с локализацией в файловой системе (pathname). Метод `mkdir()` класса `File` создает каталог с именем, представленным объектом `File`.

В методе `Search()` используется объект `page`, который мы предварительно объявим как поле класса `SubmitTask` с помощью строки кода `private String`

page;. Объект page представляет строку, хранящую текст страничек поисковой системы Яндекс.

После определения и инициализации целочисленной переменной n, представляющей количество страничек поисковой системы Яндекс, создается объект page.

С помощью метода readLine() класса java.io.BufferedReader будем построчно считывать страничку поисковой системы Яндекс, соединение с которой было установлено, используя буферную строку buffer_page и сохраняя считанные строки в строке page.

После того как вся страничка Яндекса будет сохранена в строке page, методом close() класса BufferedReader закроем входящий поток и освободим связанные с ним ресурсы операционной системы.

Если количество сохраняемых приложением Web-страничек Number таково, что необходимо открыть следующую страничку Яндекса, то определим строку URL-адреса следующей странички Яндекса.

Конструктор класса java.net.URL, а также методы openStream() класса URL, readLine() и close() класса BufferedReader требуют использование блока перехвата исключений try{}catch{}. Класс java.io.IOException отвечает за уведомление о возникновении исключительной ситуации, связанной с вводом/выводом. В блоке catch, который будет срабатывать при возникновении исключения, изменим атрибут message класса Task, который связан с текстом панели состояния, используя метод setMessage() класса Task. Применим метод toString(), который класс IOException наследует от класса java.lang.Throwable, для того, чтобы получить короткое описание исключения и передать его методу setMessage(). Методом cancel(true), наследуемым классом Task от класса SwingWorker, прервем выполнение фоновой задачи.

В методе Parse() (листинг 3.8) закрытого класса SubmitTask приложения мы будем извлекать ссылки на Web-странички из строки page, хранящей странички поисковой системы Яндекс, причем количество ссылок не будет превышать заданного пользователем числа Number. Извлеченные ссылки будем сохранять в массиве строк linkSite размером Number. Для этого объявим новый закрытый член класса SubmitTask с помощью строки кода private String linkSite[]; и создадим объект linkSite в конструкторе класса SubmitTask с помощью строки кода linkSite=new String[Number];.

Листинг 3.8. Код метода Parse() класса SubmitTask

```
private void Parse(){
  for (int i=0; i<Number;)
```

```
  {
    int ind=page.indexOf("href=");
    char line_array[]=page.toCharArray();
    char line_buffer[]=new char[line_array.length];

    int j=0;
    int l=0;
    if (ind!=-1) {
      for (l=(ind+6);(line_array[l]!='\"')&&(line_array[l]!='\''); l++)
      {
        line_buffer[j]=line_array[l];
         j++;
      }
      String link=new String(line_buffer);
      if ((link.indexOf("http://")!=-1)&&(link.indexOf("yandex")==-1)
        &&(link.indexOf("moikrug")==-1)
        &&(link.indexOf("ya")==-1)&&(link.indexOf("google")==-1)
        &&(link.indexOf("msn")==-1)
        &&(link.indexOf("yahoo")==-1)&&(link.indexOf("rambler")==-1))
      {
        linkSite[i]=link;
        i++;
      }
      StringBuffer buffer_line=new StringBuffer(page);
      page=buffer_line.delete(0,(l+2)).toString();
    }
  }
  try {
    Thread.sleep(500);
  } catch (InterruptedException ex) {
    setMessage(ex.toString());
    cancel(true);
  }
}
```

Далее обсудим код метода `Parse()` класса `SubmitTask` (см. листинг 3.8).

В коде метода `Parse()` будем извлекать ссылки в цикле от 0 до `Number`, причем следующий шаг цикла будем делать только в том случае, если ссылка была извлечена.

В целочисленной переменной `ind` будем сохранять позицию появления в строке `page` строки атрибута `href` HTML-тега `<A>`, задающего URL-адрес ресурса. Получать данную позицию будем методом `indexOf()` класса `String`.

Методом `toCharArray()` класса `String` конвертируем строку `page` в массив символов `char`. Создадим буферный массив символов `line_buffer[]`, в который посимвольно будем записывать ссылки, начиная с позиции после `"href="`, когда начинается сама ссылка, и заканчивая символом окончания ссылки — `"'"` или `"""`. Полученный массив символов `line_buffer[]` конвертируем в строку `link`. Далее отсекаем не относящиеся к теме ссылки, которые присутствуют в HTML-коде странички поисковой системы Яндекс, сохраняя относящуюся к теме ссылку в массиве строк `linkSite` и делая следующий шаг цикла.

Вырежем из строки `page` найденную ссылку, воспользовавшись классом `java.lang.StringBuffer`, который представляет изменяющуюся последовательность символов. Метод `delete()` класса `StringBuffer` позволяет удалить символы, находящиеся между указанными позициями, из последовательности.

Для того чтобы успеть увидеть сообщение `"parseMessage"` в панели состояния, в блоке `try{}catch{}` сделаем паузу в выполнении фоновой задачи с помощью статического метода `sleep()` класса `java.lang.Thread`, представляющего поток выполнения в программе.

В методе `Save()` (листинг 3.9) класса `SubmitTask`, в цикле от 0 до `Number`, будем открывать соединения с Web-страничками, адреса которых хранятся в массиве строк `linkSite`, и сохранять их в каталоге, уже созданном в методе `Search()`.

Листинг 3.9. Код метода `Save()` класса `SubmitTask`

```
private void Save(){
  try {
    for (int i=0;i<Number;i++) {
      String url=linkSite[i];
      URL bhv = new URL(url);
      BufferedReader br =
        new BufferedReader(new InputStreamReader(bhv.openStream()));
      String file=Text;
      String file_save=file+i+".html";
      FileWriter fw = new FileWriter(file+"/"+file_save);
      String buffer_line;
      while ((buffer_line = br.readLine()) != null)
        { fw.write(buffer_line); }
      br.close();
      fw.close();
    }
```

```
    } catch(IOException ex){
      setMessage(ex.toString());
      cancel(true);
    }
}
```

Далее обсудим код метода `Save()` класса `SubmitTask` (см. листинг 3.9).

По тем же самым причинам, что и в методе `Search()`, все операции метода `Save()` будем производить в блоке перехвата исключений `try{}catch{}`.

В коде метода `Save()` строка `url` содержит строку URL-адреса Web-ресурса, хранящуюся в массиве строк `linkSite`. Создадим объект класса `java.net.URL`, представляющий URL-адрес, содержащийся в строке `url`. Методом `openStream()` класса `URL` установим соединение с ресурсом и откроем входящий байтовый поток. С помощью классов `InputStreamReader` и `BufferedReader` преобразуем его в буферный входящий поток символов.

Класс `java.io.FileWriter` предназначен для записи символов в файл файловой системы. В нашем случае, при создании объекта `fw` класса `FileWriter` будет создан файл с именем `file_save`, в каталоге с именем `file` (текст запроса пользователя). В этот файл будет записана страничка сайта.

Методом `readLine()` класса `java.io.BufferedReader` будем построчно считывать страничку сайта, соединение с которым было установлено, используя буферную строку `buffer_line` и сохраняя считанные строки в файле с помощью метода `write()`, который класс `FileWriter` наследует от класса `java.io.Writer`. После сохранения странички сайта методом `close()` закроем входящий и выходящий потоки, освободив связанные с ними ресурсы операционной системы.

Все! Наше приложение готово!

Сборка и запуск проекта

Осуществим сборку проекта. Для этого в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на имени проекта SearchInternet. В появившемся контекстном меню выберем пункт **Build**. В нижнем окне **Output** среды NetBeans должно появиться сообщение "BUILD SUCCESSFUL". Если этого окна нет, то в главном меню среды NetBeans выберем пункт **Window** и в появившемся списке выберем **Output | Output**. Данное окно должно появиться.

В результате сборки в каталоге проекта будут созданы:

- папка build, которая содержит конфигурационный файл org.jdesktop.application.Application, папку с ресурсами приложения и файлы с расширением class, содержащие *байт-код*, выполняемый виртуальной машиной JVM;

- папка dist, которая содержит папку с Java-библиотеками и исполняемый JAR-файл, который, помимо файлов, находящихся в папке build, включает в себя файл MANIFEST, используемый средой выполнения JRE для нахождения главного класса приложения и Java-библиотек, необходимых для работы приложения.

Запустить наше приложение можно несколькими способами.

Первый способ: в панели инструментов среды NetBeans найдите большую зеленую стрелку с всплывающей подсказкой "Run Projects" и, нажав ее, запустите приложение (рис. 3.11).

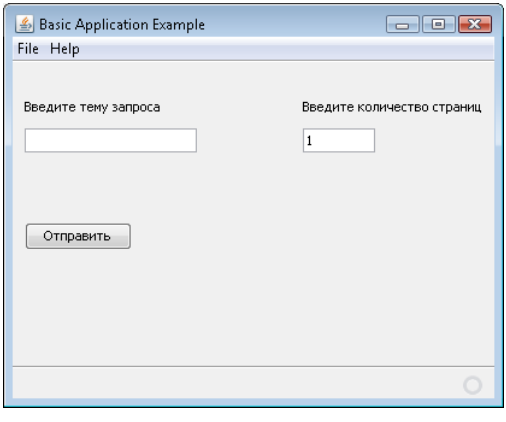

Рис. 3.11. Главное окно графического интерфейса пользователя запущенного приложения

Второй способ: щелкните правой кнопкой мыши в окне **Projects** среды NetBeans на имени проекта и в появившемся контекстном меню выберите пункт **Run**.

Можно запустить приложение из командной строки, набрав строку

```
java -jar "C:\NetBeansProjects\SearchInternet\dist\SearchInternet.jar"
```

Если среда выполнения JRE была правильно установлена в вашей операционной системе, то приложение запустится, если вы просто дважды щелкните мышью на исполняемом JAR-файле приложения.

Для того чтобы во время запуска приложения появлялся рисунок-заставка — splash.png, необходимо, чтобы в файл MANIFEST.MF, находящийся в JAR-файле приложения, была включена строка

```
"SplashScreen-Image: searchinternet/resources/splash.png
```

после строки

```
Main-Class: searchinternet.SearchInternetApp
```

Или в командной строке должно быть указано:

```
java –splash:"C:\NetBeansProjects\SearchInternet\src\searchinternet
\resources\splash.png" –jar
"C:\NetBeansProjects\SearchInternet\dist\SearchInternet.jar"
```

Для запуска приложения можно создать исполняемый exe-файл, воспользовавшись одной из бесплатных программ, например Xenoage JEStart (сайт **www.xenoage.com**). Такая программа генерирует exe-файл для выбранного JAR-файла, позволяя указать значок для исполняемого файла.

Технология Java Web Start (JWS)

Далее рассмотрим загрузку и развертывание Java-приложений с помощью технологии Java Web Start (JWS).

Java Web Start (JWS) — технология, основанная на протоколе Java Network Launching Protocol (JNLP), позволяет загружать и запускать приложения с сайта с помощью браузера. Преимущество данной технологии в том, что пользователю не нужно самому специально инсталлировать приложение на своем компьютере и беспокоиться о его обновлениях. Вся площадка для загрузки и старта приложения находится на сервере сайта.

Web-страничка браузера должна содержать ссылку, указывающую на файл с расширением jnlp, который включает в себя инструкции для загрузки, кэширования и запуска приложения.

После первой загрузки запустить приложение можно также с помощью интерфейса Java Cache Viewer Java-инструмента `javaws`, который находится в папке bin среды JRE. Для этого в командной строке необходимо набрать:

```
javaws –viewer
```

После этого на экране появится интерфейс Java Cache Viewer (рис. 3.12).

При первой загрузке приложения с помощью технологии JWS все необходимые файлы сохраняются в компьютере пользователя в специальной папке cache, поэтому повторно запустить приложение можно без использования браузера и соединения с Интернетом, посредством Java Cache Viewer. Интерфейс Java Cache Viewer позволяет запустить приложение, просмотреть jnlp-файл приложения, инсталлировать ярлык приложения на рабочем столе компьютера, удалить приложение с компьютера и перейти на домашнюю страничку приложения. Интерфейс Java Cache Viewer отражает в качестве названия приложения заголовок главного окна приложения, а не имя JAR-файла.

Открыть интерфейс Java Cache Viewer можно также с помощью контрольной панели Java — JRE-инструмента `javacpl`, во вкладке **General** нажав кнопку **View** (рис. 3.13).

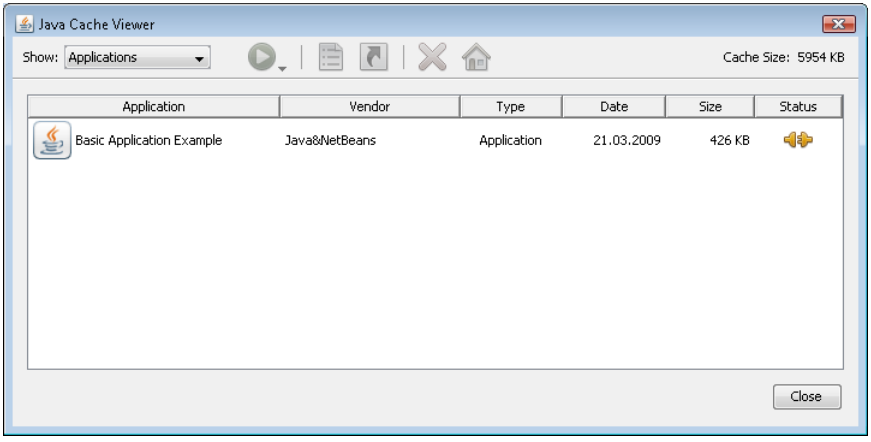

Рис. 3.12. Окно **Java Cache Viewer**

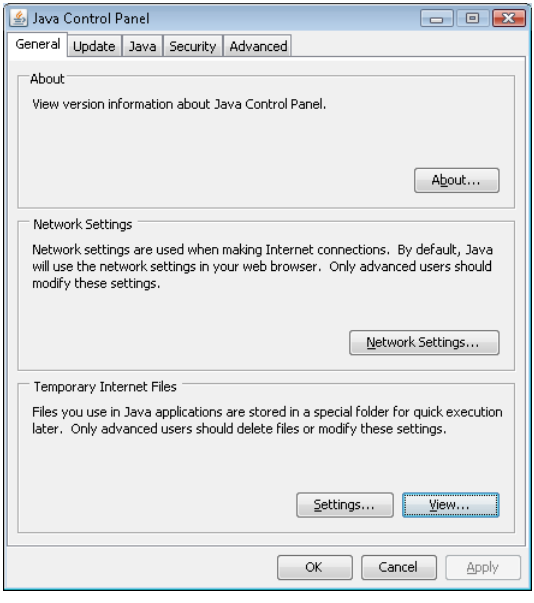

Рис. 3.13. Окно контрольной панели Java

Инсталлировав ярлык приложения с помощью Java Cache Viewer, запускать приложение можно из рабочего стола компьютера пользователя.

Для того чтобы применить технологию Java Web Start к нашему приложению SearchInternet, в среде NetBeans в окне **Projects** щелкнем правой кнопкой мыши на имени проекта SearchInternet. В появившемся контекстном меню выберем пункт **Properties**. В категории **Web Start** установим флажок **Enable Web Start** (рис. 3.14).

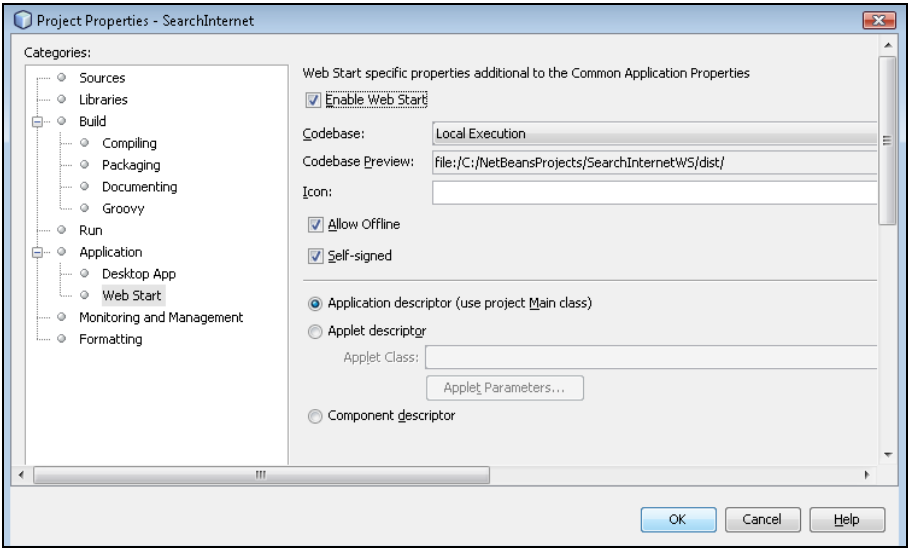

Рис. 3.14. Окно определения свойств проекта

После нажатия кнопки **OK** в окне свойств проекта снова щелкнем правой кнопкой мыши на имени проекта и в появившемся контекстном меню выберем пункт **Build**. В результате сборки проекта в папке dist проекта появится сгенерированная тестовая HTML-страничка (рис. 3.15), содержащая ссылку на сгенерированный jnlp-файл launch, расположенный в этой же папке.

Щелкнув по ссылке, мы загрузим в кэш и запустим наше приложение.

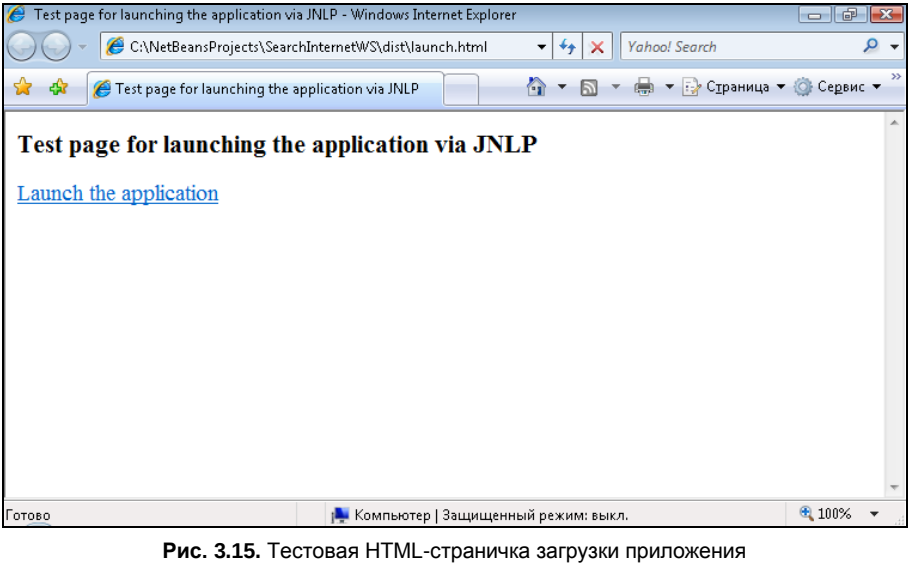

Рис. 3.15. Тестовая HTML-страничка загрузки приложения

Далее рассмотрим HTML-код тестовой странички, содержащей ссылку на jnlp-файл (листинг 3.10).

Листинг 3.10. HTML-код тестовой странички

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
  <HEAD>
    <TITLE>Test page for launching the application via JNLP</TITLE>
  </HEAD>
  <BODY>
    <H3>Test page for launching the application via JNLP</H3>
    <A href="launch.jnlp">Launch the application</A>
    <!-- Or use the following script element to launch -->
    <!-- with the Deployment Toolkit -->
    <!-- Open the deployJava.js script to view its documentation -->
    <!--
    <SCRIPT src="http://java.com/js/deployJava.js"></SCRIPT>
    <SCRIPT>
      var url="http://[fill in your URL]/launch.jnlp"
      deployJava.createWebStartLaunchButton(url, "1.6")
    </SCRIPT>
    -->
  </BODY>
</HTML>
```

В HTML-коде тестовой странички атрибут `href` тега `<A>`, отвечающего за создание гиперссылки, задает URL-адрес jnlp-файла.

HTML-код странички содержит закомментированный JavaScript-код, в котором с помощью атрибута `src` тега `<SCRIPT>`, отвечающего за включение программ-скриптов, загружается скрипт deployJava.js. Скрипт deployJava.js является набором инструментов Deployment Toolkit развертывания апплетов и JWS-приложений. Использование Deployment Toolkit делает загрузку и развертывание Java-приложений через Интернет более удобной и простой, т. к. скрипт обеспечивает проверку установленных сред выполнения JRE и установку необходимых обновлений.

Deployment Toolkit представляет объект `deployJava` со следующими функциями:

- `getJREs()` возвращает список инсталлированных версий JRE;
- `installLatestJRE()` запускает инсталляцию последней версии JRE;
- `installJRE(requestVersion)` запускает инсталляцию указанной версии JRE;

- `versionCheck(version)` возвращает `true`, если указанная версия JRE инсталлирована;
- `writeAppletTag(attributes, parameters)` создает тег `<APPLET>` с указанными атрибутами и параметрами;
- `runApplet(attributes, parameters, minimumVersion)` гарантирует, что подходящая JRE инсталлирована, и запускает апплет;
- `isWebStartInstalled(minimumVersion)` возвращает `true`, если платформа Java Web Start с указанной минимальной версией инсталлирована;
- `createWebStartLaunchButton(jnlp, minimumVersion)` создает кнопку **launch**, при нажатии которой производится проверка, что подходящая JRE инсталлирована, и запускает JNLP-приложение;
- `setInstallerType(type)` устанавливает тип инсталляции **null**, **online**, **kernel**;
- `setAdditionalPackages(packageList)` устанавливает список дополнительных пакетов при системной инсталляции (**kernel**).

В нашем случае используется функция `createWebStartLaunchButton()` для создания кнопки **launch** со ссылкой на jnlp-файл.

Jnlp-файл представляет собой XML-файл, содержащий инструкции для JWS-инструмента `javaws` как загружать и запускать приложение. Именно расширение jnlp-файла при инсталляции JRE ассоциируется с Java-инструментом `javaws`, поэтому при открытии jnlp-файла автоматически запускается загрузчик `javaws`.

Далее рассмотрим код сгенерированного средой NetBeans jnlp-файла launch для нашего приложения (листинг 3.11).

Листинг 3.11. Код jnlp-файла приложения

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<jnlp codebase="file:/C:/NetBeansProjects/SearchInternetWS/dist/"
      href="launch.jnlp" spec="1.0+">
<information>
  <title>Basic Application Example</title>
  <vendor>Java&amp;amp;NetBeans</vendor>
  <homepage href=""/>
  <description>A simple java desktop application based on Swing
Application Framework</description>
  <description kind="short">Basic Application Example</description>
  <offline-allowed/>
</information>
```

```
<security>
  <all-permissions/>
</security>
<resources>
  <j2se version="1.5+"/>
  <jar  href="SearchInternet.jar" main="true"/>
  <jar href="lib/appframework-1.0.3.jar"/>
  <jar href="lib/swing-worker-1.1.jar"/>
</resources>
<application-desc main-class="searchinternet.SearchInternetApp">
</application-desc>
</jnlp>
```

В коде jnlp-файла элемент `<jnlp>` — главный XML-элемент jnlp-файла. Все остальные элементы должны находиться внутри этого тега. Элемент `<jnlp>` имеет набор атрибутов:

- атрибут `codebase` устанавливает базовое местонахождение для всех относительных URL;
- атрибут `href` устанавливает относительный URL-адрес для jnlp-файла;
- атрибут `spec` обозначает минимальную версию JNLP-спецификации для данного jnlp-файла.

Внутри элемента `<information>` содержатся элементы, которые описывают приложение. Элементы `<title>`, `<vendor>`, `<homepage>`, `<description>` определяют соответственно заголовок, производителя, домашнюю страничку и описание приложения. Элемент `<offline-allowed>` указывает на то, что приложение может работать в отсутствие соединения с Интернетом. В нашем случае приложение работает только при установленном соединении с Интернетом, поэтому данный элемент мы исключим.

Элемент `<security>` определяет права доступа. Его вложенный элемент `<all-permissions>` разрешает приложению полный доступ к клиентской машине и локальной сети.

Элемент `<resources>` описывает все ресурсы, необходимые для работы приложения.

Элемент `<j2se>` с атрибутом `version` указывает минимальную версию JRE для данного приложения.

Элемент `<jar>` определяет JAR-файл приложения. Его атрибут `main` указывает на то, что данный JAR-файл содержит главный класс приложения. Атрибут `href` элемента `<jar>` указывает относительный (по отношению к `codebase`) URL-адрес для JAR-файла.

Элемент `<application-desc>` указывает на то, что данный jnlp-файл описывает настольное Java-приложение (jnlp-файл с элементом `<applet-desc>` описывает апплет). Его атрибут `main-class` указывает полное имя главного класса приложения (класса, содержащего метод `main()`).

При наличии элемента `<all-permissions>` в jnlp-файле все JAR-файлы приложения должны быть подписаны электронной подписью. Когда мы в окне **Project Properties** среды NetBeans в категории **Web Start** устанавливали флажок **Enable Web Start**, по умолчанию был отмечен флажок **Self-signed**. При активированном флажке **Self-signed** среда NetBeans в процессе сборки проекта подписывает JAR-файлы и устанавливает элемент `<all-permissions>` в jnlp-файле. При этом используется Java-инструмент `jarsigner`, с помощью которого производится подпись JAR-файлов, находящийся в папке bin каталога JDK. Файл подписывается закрытым ключом, который известен только владельцу ключа. Закрытый ключ проверяется открытым ключом, который удостоверяется в свою очередь сертификатом, содержащим информацию о владельце открытого ключа. Когда JAR-файл подписывается, открытый ключ и сертификат помещаются в файле с расширением dsa в папку META-INF JAR-архива. При этом в файл MANIFEST.MF папки META-INF заносятся хэш-коды содержимого JAR-файлов. В папку META-INF также помещается файл подписи с расширением sf, который содержит хэш-коды содержимого JAR-файлов, совпадающие с хэш-кодами файла MANIFEST.MF. При проверке подписанного JAR-файла вычисляется хэш-код содержимого каждого из JAR-файлов и сравнивается с хэш-кодами файла MANIFEST.MF и sf-файла подписи. Это делается для того, чтобы предотвратить изменение содержимого JAR-файла с момента его подписи. Среда NetBeans при выборе опции **Self-signed** генерирует закрытый, открытый ключи и сертификат, необходимые для подписи JAR-файла.

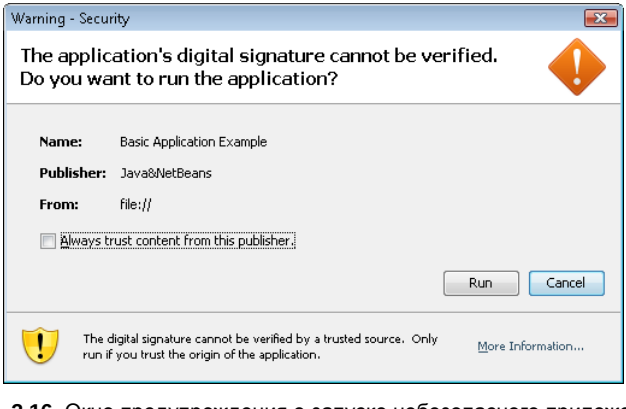

Рис. 3.16. Окно предупреждения о запуске небезопасного приложения

Если запустить jnlp-файл launch, расположенный в папке dist, то при загрузке приложения появится предупреждение, предлагающее подтвердить доверие к сертификату подписанного приложения (рис. 3.16).

Если щелкнуть на ссылке **More Information** окна предупреждения, то можно посмотреть детали сертификата. При установке флажка **Always trust content from this publisher** пользователь подтверждает доверие к данной цифровой подписи, и предупреждение больше появляться не будет.

Внешний вид и поведение графических компонентов *LookAndFeel*

Как уже упоминалось, объект класса `javax.swing.LookAndFeel` инкапсулирует информацию о внешнем виде и поведении всех графических компонентов. В методе `launch()` класса `org.jdesktop.application.Application` при создании экземпляра класса, расширяющего класс `Application`, с помощью ресурса `"Application.lookAndFeel ="` properties-файла главного класса приложения определяется внешний вид приложения, который далее устанавливается с помощью метода `setLookAndFeel()` класса `javax.swing.UIManager`.

Архитектура графической библиотеки Swing делает возможным выбор набора графических компонентов с различным внешним видом и поведением путем разделения каждого компонента на два класса — подкласс `javax.swing.JComponent` и подкласс `javax.swing.plaf.ComponentUI` (UI delegate). Подкласс `JComponent` вызывает методы подкласса `ComponentUI` в случае, если они зависят от конкретного подкласса `LookAndFeel`. Различные подклассы `LookAndFeel` реализуют разные наборы подклассов `ComponentUI`.

Изменить внешний вид приложения можно в среде NetBeans, щелкнув правой кнопкой мыши в окне **Projects** на имени проекта и в появившемся списке выбрав **Properties**. В окне **Project Properties**, в строке **Look & Feel** категории **Desktop App**, можно изменить внешний вид приложения, выбрав **System Default**, **Java Default** или один из следующих классов:

- `com.sun.java.swing.plaf.windows.WindowsClassicLookAndFeel`;
- `com.sun.java.swing.plaf.windows.WindowsLookAndFeel`;
- `com.sun.java.swing.plaf.motif.MotifLookAndFeel`;
- `com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel`;
- `javax.swing.plaf.metal.MetalLookAndFeel`.

Вариант **System Default** определяется во время выполнения приложения, при запросе приложения у системы, которая возвращает имя подходящего компонента `LookAndFeel`. В нашем случае это `WindowsLookAndFeel`.

Вариант **Java Default** совпадает с `javax.swing.plaf.metal.MetalLookAndFeel`.

Пакет `javax.swing.plaf.synth` (библиотека Synth Look and Feel) является частью JDK. Его нет в списке выбора NetBeans, потому что он служит оболочкой `LookAndFeel` (skinnable look and feel) и предназначен для создания новых пользовательских компонентов `LookAndFeel`.

Библиотека Synth позволяет изменять внешний вид приложения с помощью XML-файла. Это дает возможность создать новый внешний вид без написания нового графического Java-пакета, что является очень трудоемким делом, т. к. для создания нового подкласса `LookAndFeel` требуется расширение каждого из классов уже существующих пакетов `LookAndFeel` (базовый пакет `javax.swing.plaf.basic` состоит из 60 классов).

Для того чтобы создать новый внешний вид нашего приложения с помощью пакета `javax.swing.plaf.synth`, включим соответствующий код в метод `startup()` класса приложения SearchInternetApp (листинг 3.12).

Листинг 3.12. Код использования библиотеки Synth

```
@Override protected void startup() {
try {
  javax.swing.plaf.synth.SynthLookAndFeel synth =
    new javax.swing.plaf.synth.SynthLookAndFeel();
  synth.load(SearchInternetApp.class.getResourceAsStream(
      "resources/synth.xml"), SearchInternetApp.class);
  javax.swing.UIManager.setLookAndFeel(synth);
} catch (Exception ex) {
  javax.swing.JOptionPane.showMessageDialog(null, ex.toString());
}
show(new SearchInternetView(this));
}
```

В коде метода `startup()` класса `SearchInternetApp` (см. листинг 3.12) после создания объекта `synth` класса `javax.swing.plaf.synth.SynthLookAndFeel` с помощью метода `load()` загружается набор свойств, определяющих внешний вид графических компонентов. Метод `getResourceAsStream()` класса `java.lang.Class<T>` ищет ресурсы, ассоциированные с указанным классом, и возвращает входящий поток, связанный с ресурсом synth.xml, относительное имя которого указано в аргументе метода `load()`. Второй аргумент метода `load()` определяет локализацию ресурсов, упоминаемых в файле synth.xml, например файлов изображений. Методом `setLookAndFeel()` класса `javax.swing.UIManager` устанавливается внешний вид приложения, определяемый объектом `synth`. При вызове методов `load()` и `setLookAndFeel()` должна обрабатываться исключительная ситуация, связанная с возможным

возникновением ошибок в случае, если не будут найдены ресурсы или классы `LookAndFeel` не будут представлены в системе. Поэтому здесь используется блок перехвата исключений `try{}catch{}`, в котором в случае возникновения такой ошибки на экран будет выведено диалоговое окно с информацией об исключительной ситуации (рис. 3.17).

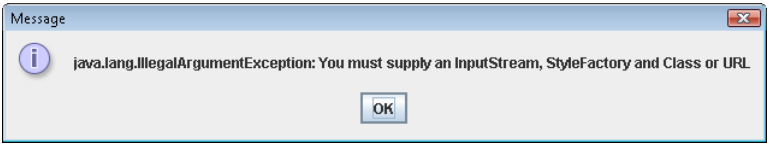

Рис. 3.17. Окно сообщения о возникновении исключительной ситуации

Окно сообщения о возникновении исключительной ситуации создается с помощью класса `javax.swing.JOptionPane`, который предназначен для создания стандартных модальных диалоговых окон, приглашающих пользователя ввести необходимые значения, сделать выбор или выводящих сообщения для пользователя. Метод `showMessageDialog()` класса `JOptionPane` создает модальное диалоговое окно, содержащее сообщение — в нашем случае короткое описание возникшей исключительной ситуации, полученное методом `toString()` класса `java.lang.Throwable`, подклассом которого является класс `java.lang.Exception`, представляющий исключительные ситуации. Параметр `null` в аргументах метода `showMessageDialog()` означает, что создаваемое диалоговое окно не имеет родительского окна, в котором оно будет выведено.

Для изменения внешнего вида интерфейса нашего приложения создадим файл synth.xml в папке resources проекта. Для этого произведем следующие действия:

1. В окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на узле **searchinternet.resources** и в появившемся контекстном меню выберем **New | Other**.
2. Далее в диалоговом окне, в категории **Other**, выберем **Empty File**. Нажав кнопку **Next**, введем имя файла synth.xml и нажмем кнопку **Finish**.
3. В окне редактора среды NetBeans введем код файла synth.xml (листинг 3.13).

Листинг 3.13. Код файла synth.xml

```
<synth>
  <style id="default">
    <font name="Arial Black" style="ITALIC" size="16"/>
    <state>
```

```
    <color value="#FFFFFF" type="FOREGROUND"/>
  </state>
</style>
<bind style="default" type="region" key=".*"/>
<style id="textfield">
  <state>
    <color value="#0000FF" type="FOREGROUND"/>
    <font name="Arial" style="BOLD" size="16"/>
    <insets top="10" left="10" right="10" bottom="10"/>
    <imagePainter method="TextFieldBackground"
                  path="resources/textfield.png"
                  sourceInsets="10 10 10 10"/>
  </state>
</style>
<bind style="textfield" type="region" key="TEXTFIELD"/>
<style id="button">
  <insets top="10" left="20" right="20" bottom="10"/>
  <state>
    <imagePainter method="buttonBackground"
                  path="resources/button.png"
                  sourceInsets="10 10 10 10"/>
  </state>
  <state value="PRESSED">
    <imagePainter method="buttonBackground"
                  path="resources/button_pressed.png"
                  sourceInsets="10 10 10 10"/>
  </state>
  <state value="MOUSE_OVER">
    <imagePainter method="buttonBackground"
                  path="resources/button_over.png"
                  sourceInsets="10 10 10 10"/>
  </state>
</style>
<bind style="button" type="region" key="Button"/>
<style id="menu">
  <state>
    <font name="Calibri" style="BOLD" size="14"/>
  </state>
</style>
<bind style="menu" type="region" key="MENU"/>
<style id="menu_item">
  <state>
    <font name="Calibri" style="BOLD" size="14"/>
  </state>
</style>
```

```
  <bind style="menu_item" type="region" key="MENUITEM"/>
  <style id="MENU_ITEM_ACCELERATOR">
    <state>
      <font name="Calibri" style="BOLD" size="14"/>
    </state>
  </style>
  <bind style="MENU_ITEM_ACCELERATOR" type="region"
        key="MENUITEMACCELERATOR"/>
  <style id="menu_bar">
    <insets top="5" bottom="5"/>
    <state>
      <imagePainter method="MenuBarBackground"
                    path="resources/back_menubar.png"
                    sourceInsets="10 10 10 10"/>
    </state>
  </style>
  <bind style="menu_bar" type="region" key="MENUBAR"/>
  <style id="panel">
    <state>
      <imagePainter method="panelBorder"
                    path="resources/border.png" paintCenter="false"
                    sourceInsets="2 2 2 2"/>
      <imagePainter method="panelBackground"
                    path="resources/back_panel.png"
                    sourceInsets="10 10 10 10"/>
    </state>
  </style>
  <bind style="panel" type="region" key="PANEL"/>
</synth>
```

В коде файла synth.xml (см. листинг 3.13) элемент `<synth>` является корневым и содержит все остальные элементы, определяющие `SynthLookAndFeel`.

Элемент `<style>` содержит элементы, описывающие свойства дизайна, такие как цвет, шрифт, размер и др. Его атрибут `id` определяет уникальный идентификатор для данного стиля.

Элемент `<font>` устанавливает шрифт со следующими основными атрибутами: `id` — идентификатор; `name` — название шрифта; `style` — стиль шрифта; `size` — размер шрифта в пикселах.

Элемент `<state>` устанавливает визуальные свойства компонента, связанные с его указанным состоянием. Элемент имеет следующие основные атрибуты: `id` — идентификатор состояния; `value` — состояние компонента, с которым

связываются визуальные свойства, может принимать значения ENABLED, MOUSE_OVER, PRESSED, DISABLED, FOCUSED, SELECTED или DEFAULT (если значение не указано, визуальные свойства связываются со всеми состояниями компонента).

Элемент <color> устанавливает цвет со следующими основными атрибутами: id — идентификатор цвета; type определяет, в какой части компонента применяется данный цвет, может принимать значения FOREGROUND, BACKGROUND, TEXT_FOREGROUND, TEXT_BACKGROUND или FOCUS; value — значение, определяющее цвет.

Элемент <bind> устанавливает, для какого компонента или региона применяется указанный стиль. Регион — это индивидуальное визуальное пространство Swing-компонента. Регионы компонентов представляют класс javax.swing.plaf.synth.Region. Компонент может поддерживать один или несколько регионов. Например, кнопка поддерживает один регион BUTTON, а панель прокрутки — три: панель прокрутки SCROLL_BAR, ползунок прокрутки SCROLL_BAR_THUMB и полосу прокрутки SCROLL_BAR_TRACK. Элемент <bind> имеет следующие атрибуты: style — идентификатор стиля, который связывается с компонентом или регионом; type принимает два значения: region (стиль связывается с именем региона) или name (стиль связывается с именем компонента); key определяет имя региона или имя компонента.

Элемент <insets> увеличивает размеры компонента или региона, к которому применяется стиль. Атрибуты элемента top, bottom, left и right устанавливают увеличение размеров в пикселах в четырех направлениях.

Элемент <imagePainter> регистрирует объект класса javax.swing.plaf.synth.SynthPainter для данного стиля или состояния, при этом указанный в элементе метод класса SynthPainter визуализирует заданное ресурсное изображение. Атрибут id определяет идентификатор элемента, method задает метод класса SynthPainter для визуализации изображения, direction определяет ориентацию изображения, path указывает относительный адрес изображения. Атрибуты center и sourceInsets устанавливают два различных режима визуализации изображения. Если указан атрибут center="true", то заданное изображение прорисовывается по центру региона или компонента, сохраняя свои исходные размеры. Это может привести к тому, что изображение не будет покрывать всю область пространства компонента или регион. Для подгонки изображения к заданной области пространства используется атрибут sourceInsets, который задает размеры квадратов, покрывающих углы изображения, для их фиксации. При этом все остальные части изображения растягиваются или сжимаются, подгоняясь к области пространства. Атрибут paintCenter определяет, будет ли прорисована центральная часть изображения.

С помощью элемента `<painter>` можно зарегистрировать расширение класса `SynthPainter` для данного стиля или состояния.

В нашем случае, вначале мы задаем стиль по умолчанию для всех компонентов приложения с помощью атрибута `key=".*"`. Этот стиль определяет шрифт и цвет текста всего графического интерфейса. Далее задается стиль для текстовых полей с помощью переопределения шрифта и цвета текста, который будет вводить пользователь, и указывается фоновое изображение, которое будет прорисовано методом `paintTextFieldBackground()` класса `SynthPainter`. С помощью элемента `insets` увеличиваются размеры региона `TEXT_FIELD` — объекта класса `javax.swing.plaf.synth.Region`.

Для кнопок графического интерфейса задаются стили для трех состояний:

- все состояния кнопки, за исключением кнопка "нажата" или на нее навели указатель мыши;
- кнопка "нажата";
- на кнопку навели указатель мыши.

Для элементов меню переопределяется шрифт. Для панели меню указывается фоновое изображение. Стиль региона `PANEL` включает в себя фоновое изображение и изображение рамки.

Таким образом, используя пакет `javax.swing.plaf.synth`, мы получили новый внешний вид нашего приложения (рис. 3.18).

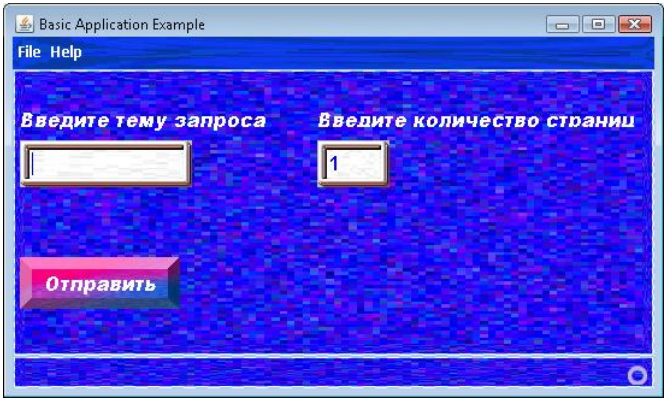

Рис. 3.18. Новый внешний вид интерфейса приложения

Инструмент *javadoc*

Java-инструмент `javadoc`, являющийся неотъемлемой частью JDK, производит разбор комментариев в Java-файлах, содержащих исходный код, и на их

основе создает набор HTML-страниц, описывающих классы, интерфейсы, конструкторы, методы и поля классов. Таким образом, `javadoc` генерирует документацию в HTML-формате из комментариев к исходному коду. JDK также предоставляет Doclet API — пакет `com.sun.javadoc`, содержащий классы и интерфейсы для создания *доклета* (doclet) — программы, которая определяет содержание и формат генерируемой инструментом `javadoc` документации. JDK содержит также стандартный доклет, который используется `javadoc` по умолчанию для генерации документации в HTML-формате. Классы стандартного доклета составляют пакеты `com.sun.tools.doclets` и `com.sun.tools.doclets.standard`, расположенные в JAR-файле tools папки lib JDK. С помощью специально созданного доклета можно генерировать документацию в различных форматах, таких как HTML, SGML, XML, RTF и др. Доклеты могут также использоваться для проверки исходного кода, автоматической генерации кода и других целей, не связанных с генерацией документации. В командной строке `javadoc` созданный доклет включается опцией `-doclet`. Кроме того, JDK содержит Taglet API — интерфейс `com.sun.tools.doclets.Taglet`, предназначенный для создания *теглита* — программы, способной модифицировать и форматировать текст с помощью пользовательских тегов, а также переопределять стандартные теги и создавать выходящий файловый или другой поток для текста. В командную строку `javadoc` теглит включается опцией `-taglet`.

Для того чтобы воспользоваться инструментом `javadoc` и создать документацию, необходимо определенным образом подготовить исходный код. Javadoc-комментарии должны начинаться с двойной звездочки (`/**`), а не с одинарной (`/*`) как обыкновенные комментарии. Внутри блока Javadoc-комментариев каждая строчка начинается со звездочки (`*`). В Javadoc-комментарии могут включаться теги (элементы, обрабатываемые доклетом), которые могут быть блочными и строчными. Блочные теги начинаются с тега и заканчиваются следующим тегом. Блочный тег должен обязательно начинаться с начала строчки, сразу после звездочки, иначе он не будет обработан. Строчные теги ограничиваются фигурными скобками и могут включаться в блочные теги в любом месте комментария.

Стандартные блочные теги:

- `@param` — параметры конструкторов и методов классов;
- `@return` — возвращаемое значение метода класса;
- `@throws` и `@exception` — исключения;
- `@see` — ссылка на дополнительную информацию;
- `@version` — текущая версия;
- `@since` — версия, в которую была включена описываемая информация;

- `@deprecated` отмечает более не используемые конструкторы, методы классов и др.;
- `@author` — автор.

Стандартные строчные теги:

- `{@link}` — ссылка на другой класс, метод и др.;
- `{@docRoot}` заменяется на ссылку на корень документации;
- `{@value}` заменяется на значение поля;
- `{@code}` — с помощью данного тега в Javadoc-комментарии вставляются фрагменты кода.

Можно вставлять в Javadoc-комментарии также HTML-теги, за исключением заголовочных тегов, для дополнительного форматирования документации.

При создании основы приложения среда NetBeans генерирует в том числе и Javadoc-комментарии. Настройки для генерации Javadoc-комментариев можно установить, щелкнув правой кнопкой мыши в окне **Projects** на имени проекта и в появившемся контекстном меню выбрав пункт **Properties**. В диалоговом окне **Project Properties** за настройки генерации Javadoc-комментариев отвечает категория **Documenting**.

Для созданного нами приложения дополним сгенерированные средой NetBeans Javadoc-комментарии и создадим Javadoc-документацию.

В качестве первого шага создадим файл package.html в папке src каталога проекта. При генерации документации Java-инструментом `javadoc` содержимое данного файла будет включено в сгенерированный файл документации package-summary.html, содержащий информацию о пакете классов приложения.

Введем HTML-код в файл package.html (листинг 3.14).

Листинг 3.14. HTML-код файла package.html

```
<body>
  <p>
    Contain all of the classes for Basic Application Example.
  <p>
Desktop application SearchInternet is intended for searching information
in the Internet.
</body>
```

В коде файла package.html с помощью HTML-тега `<p>` (начало параграфа) отделяем первое предложение, которое будет включено в начало странички

package-summary.html как краткая характеристика пакета, от остальной информации, которая вместе с первым предложением будет отражена на страничке package-summary.html под заголовком "Package searchinternet Description".

Далее в файле SearchInternetApp.java дополним Javadoc-комментарии к методу `startup()` (листинг 3.15).

Листинг 3.15. Javadoc-комментарии к методу `startup()` класса `SearchInternetApp`

```
/**
  * At startup create and show the main frame of the application.
  * Package javax.swing.plaf.synth is used for creating a new
  * application's look and feel in this method. File synth.xml
  * contains all of the specified pieces necessary for SynthLookAndFeel.
  * @see SynthLookAndFeel
  */
```

В файле SearchInternetView.java дополним комментарии к классу `SearchInternetView` (листинг 3.16).

Листинг 3.16. Javadoc-комментарии к классу `SearchInternetView`

```
/**
  * The application's main frame encapsulates the Application
  * GUI components:
  * Submit button, Text fields, Menu.
  */
. . .
/* Комментарии к конструктору класса SearchInternetView:
/**
  * Constructs an empty View object for the Application and
  * initialize the form.
  * First of all, invokes parent constructor super(app).
  * After that invokes {@link #initComponents()}
  * and makes status bar initialization — message timeout, idle
  * icon and busy animation, etc.
  * To connect action tasks to status bar via TaskMonitor.
  * @see org.jdesktop.application.Task<T,V>,
  * org.jdesktop.application.TaskMonitor
  * @param app — instance of the main class of the application.
  */
. . .
```

```
/* Комментарии к методу showAboutBox() класса SearchInternetView:
/**
  * Marks the method with annotation @Action so that
  * this method will be used to define the Swing Action object's
  * actionPerformed method.
  * It creates and shows the dialog window About.
  * @see org.jdesktop.application.Action.
  */
. . .
/* Комментарии к методу Submit() класса SearchInternetView:
/**
  * Marks the method with annotation @Action so that
  * this method will be used to define the Swing Action Submit's
  * actionPerformed method.
  * It creates the instance of the class {@link #SubmitTask}.
  * @return — new SubmitTask(getApplication())
  */
. . .
/* Комментарии к классу SubmitTask:
/**
  * Class SubmitTask is used for creating Application's Task.
  * Application's Task is intended for
  * searching and saving information via the Internet.
  * @see org.jdesktop.application.Task<Object, Void>.
 */
. . .
/* Комментарии к полям класса SubmitTask:
/**
  * Text is received from user
  */
/**
  * Number of the Web pages.
  */
/**
  * String for Yandex page's context.
  */
/**
  * String Array for Web site's links.
  */
. . .
/* Комментарии к конструктору класса SubmitTask:
/**
  * Runs on the EDT.  Copy GUI state that
  * doInBackground() depends on from parameters
  * to SubmitTask fields, here.
```

```
  * First of all invokes the parent constructor  super(app).
  * After that it initializes Text, Number, and linkSite fields.
  * @param app — instance of the Application
  */
. . .
/* Комментарии к методу doInBackground() класса SubmitTask:
/**
  * This method runs
  * on a background thread.
  * It invokes {@link #Search()}, {@link #Parse()}, and {@link #Save()}.
  * Task attribute message is changed in the method.
  * @return null
  */
. . .
/* Комментарии к методу succeeded класса SubmitTask:
/**
  * Runs on the EDT.  Update the GUI.
  * Set statusMessageLabel's Text.
  * Update jTextFieldReq and jTextFieldNum in the initial state.
  * @param  result doInBackground()
  */
. . .
/* Комментарии к методу Search() класса SubmitTask:
/**
  * Retrieves information in the Internet via Yandex.
  */
. . .
/* Комментарии к методу Parse() класса SubmitTask:
/**
  * Parses the pages from {@link #Search()} and save sites links to
linkSite.
  */
. . .
/* Комментарии к методу Save()класса SubmitTask:
/**
  * Connects to Web pages using the links from
  * linkSite and save pages in the folder.
 */
. . .
```

В файле SearchInternetAboutBox.java дополним комментарии к классу `SearchInternetAboutBox` (листинг 3.17).

Листинг 3.17. Javadoc-комментарии к классу `SearchInternetAboutBox`

```
/**
  * Creates Application's Dialog Window About.
  */
. . .
/* Комментарии к конструктору класса SearchInternetAboutBox:
/**
  * Creates the dialog About and initializes the form.
  * Creates a modeless dialog without a title with the specified Frame
  * as its owner by invoking parent constructor.
  * Invokes {@link #initComponents()} to initialize the form
  * and specifies that this dialog should be modal.
  * Sets Default Button — closeButton.
  * @param parent instance of the class java.awt.Frame.
  */
. . .
/* Комментарии к методу closeAboutBox()класса SearchInternetAboutBox:
/**
  * Marks the method with annotation @Action so that
  * this method will be used to define the Swing Action object's
  * actionPerformed method.
  * It closes window and releases all of the native screen resources
  * used by this Window.
  * @see org.jdesktop.application.Action.
  */
. . .
```

После создания Javadoc-комментариев к исходному коду нашего приложения теперь можно сгенерировать файлы Javadoc-документации, используя инструмент `javadoc`.

Воспользоваться Java-инструментом `javadoc` в среде NetBeans можно, щелкнув в окне **Projects** правой кнопкой мыши на имени проекта и в появившемся контекстном меню выбрав пункт **Generate Javadoc**. После этого в папке dist каталога проекта будет создана папка javadoc с документацией к нашему приложению, посмотреть которую можно с помощью браузера (рис. 3.19).

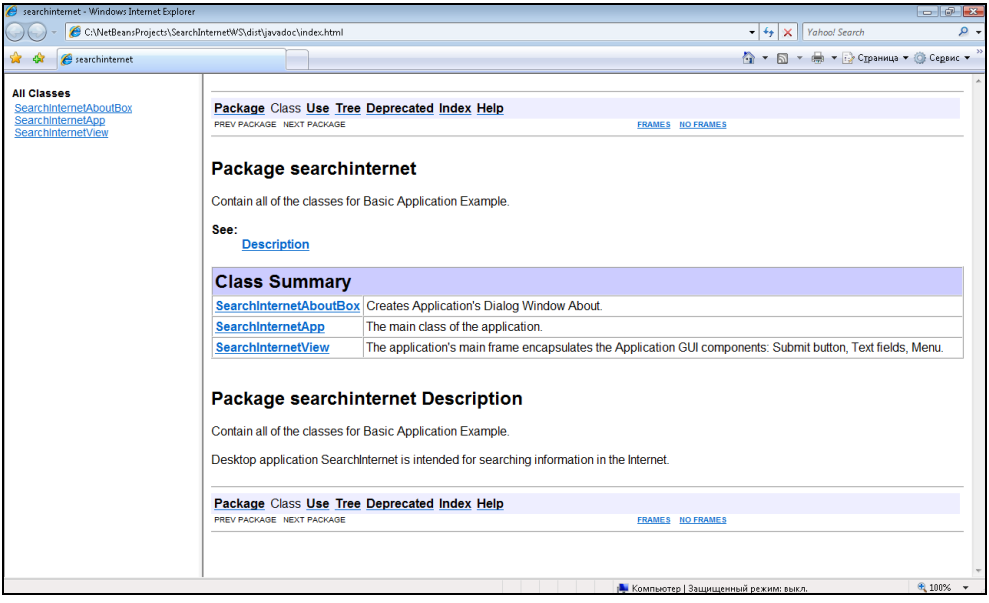

Рис. 3.19. Начальная страничка Javadoc-документации приложения

Тестирование Java-приложений

Среда NetBeans поддерживает платформу тестирования JUnit и на ее основе создает каркасы тестов для Java-классов.

Библиотека JUnit является наиболее популярной платформой для тестирования Java-приложений, позволяя создавать как отдельные тесты для тестирования результатов, возвращаемых отдельными методами в ответ на ввод различных данных, так и наборы тестов, создающие тестирующее окружение для Java-классов. Тестирование с помощью платформы JUnit основано на оперировании утверждениями для проверки ожидаемых результатов, при этом JUnit дает возможность предварительно устанавливать общие для всех тестов данные, выводить информацию о результатах тестирования в различных форматах, запускать одновременно несколько тестов.

Для создания тестов нашего приложения (test cases) в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на узле **Test Packages** и в появившемся контекстном меню выберем **New | Other | Categories: JUnit | Test for Existing Class**. Далее в поле **Class to Test** выберем кнопкой **Browse** searchinternet.SearchInternetApp (рис. 3.20).

После нажатия кнопки **Finish** и выбора библиотеки JUnit 4.*x* в окне редактора среды NetBeans мы увидим сгенерированный код основы теста для класса приложения SearchInternetApp (листинг 3.18).

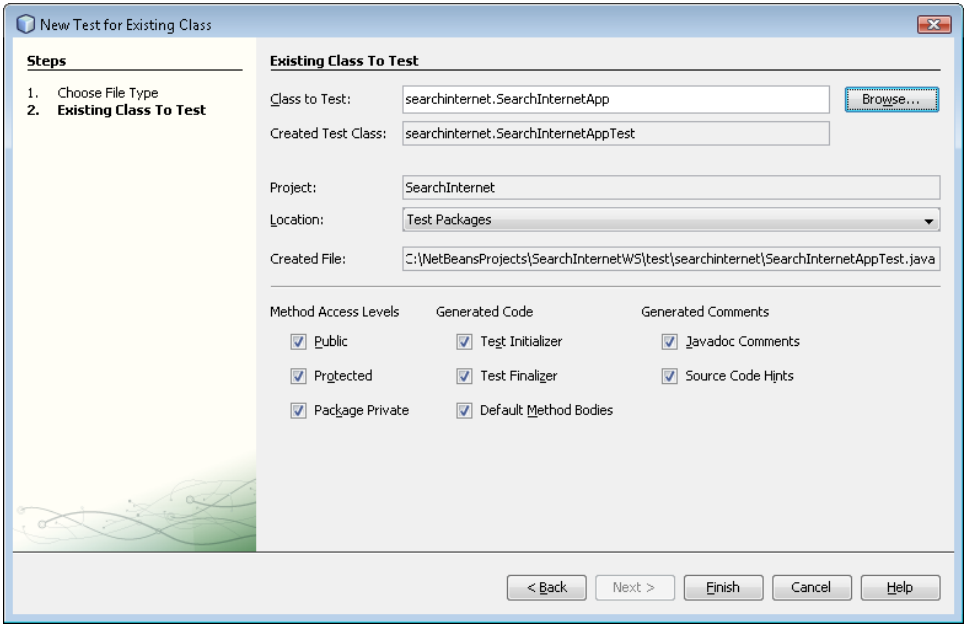

Рис. 3.20. Окно создания каркаса тестов

Листинг 3.18. Код основы теста для класса `SearchInternetApp`

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package searchinternet;
import java.awt.Window;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
/**
 *
*/
public class SearchInternetAppTest {
  public SearchInternetAppTest() {  }
  @BeforeClass
  public static void setUpClass() throws Exception {  }
```

```
  @AfterClass
  public static void tearDownClass() throws Exception {  }
  @Before
  public void setUp() {  }
  @After
  public void tearDown() {  }
/**
  * Test of startup method, of class SearchInternetApp.
  */
  @Test
  public void testStartup() {
    System.out.println("startup");
    SearchInternetApp instance = new SearchInternetApp();
    instance.startup();
    // TODO review the generated test code and remove the default
    // call to fail.
    fail("The test case is a prototype.");
  }
/**
  * Test of configureWindow method, of class SearchInternetApp.
  */
  @Test
  public void testConfigureWindow() {
    System.out.println("configureWindow");
    Window root = null;
    SearchInternetApp instance = new SearchInternetApp();
    instance.configureWindow(root);
    // TODO review the generated test code and remove the default
    // call to fail.
    fail("The test case is a prototype.");
  }
/**
  * Test of getApplication method, of class SearchInternetApp.
  */
  @Test
  public void testGetApplication() {
    System.out.println("getApplication");
    SearchInternetApp expResult = null;
    SearchInternetApp result = SearchInternetApp.getApplication();
    assertEquals(expResult, result);
    // TODO review the generated test code and remove the default
    // call to fail.
    fail("The test case is a prototype.");
  }
```

```
/**
  * Test of main method, of class SearchInternetApp.
  */
  @Test
  public void testMain() {
    System.out.println("main");
    String[] args = null;
    SearchInternetApp.main(args);
    // TODO review the generated test code and remove the default
    // call to fail.
    fail("The test case is a prototype.");
  }
}
```

Обсудим код основы теста для класса `SearchInternetApp` (см. листинг 3.18).

В коде теста для класса `SearchInternetApp` аннотация `org.junit.BeforeClass` позволяет промаркировать в классе теста статический метод, который при запуске теста будет вызван самым первым, перед всеми остальными методами. Этот метод используется для выполнения необходимых первоначальных перед тестированием операций, например, установки соединения с базой данных.

Промаркированный аннотацией `org.junit.AfterClass` статический метод выполняется после всех методов класса теста и используется для освобождения всех ресурсов, задействованных в методе, который был промаркирован аннотацией `@BeforeClass`.

Аннотацией `org.junit.Before` маркируется метод, в котором производится инициализация объектов, общих для всех тестирующих методов. Данный метод вызывается перед всеми методами, которые были промаркированы аннотацией `@Test`.

Аннотацией `org.junit.After` в классе теста маркируется метод, вызываемый после всех тестирующих методов. В этом методе освобождаются все ресурсы, задействованные в методе, который был промаркирован аннотацией `@Before`.

Аннотацией `org.junit.Test` маркируется метод, являющийся тестовым методом. При вызове этого метода сначала создается экземпляр тестируемого класса, а затем вызывается тестируемый метод.

Класс `org.junit.Assert` представляет набор методов, используемых при тестировании. Метод `fail()` класса `Assert` вызывается в случае неудачного завершения теста. Метод `assertEquals()` класса `Assert` сравнивает два объекта,

в случае их неравенства тест завершается неудачно и вызывается метод `fail()`.

Теперь нам нужно дополнить код основы теста кодом, который обеспечит тестирование корректной работы класса приложения SearchInternetApp.

Для этого в JUnit-тесте SearchInternetAppTest для класса `SearchInternetApp` приложения, в тесте `testStartup()` вызовем метод `startup()` класса `SearchInternetApp`, который должен вывести на экран главное окно приложения. Сделаем паузу в 500 миллисекунд, чтобы увидеть главное окно приложения (листинг 3.19).

Листинг 3.19. Код теста метода `startup()` класса `SearchInternetApp`

```
@Test
public void testStartup() {
  System.out.println("startup");
  SearchInternetApp instance = SearchInternetApp.getApplication();
  instance.startup();
  try {
    Thread.sleep(500);
  } catch (Exception ex) {
    javax.swing.JOptionPane.showMessageDialog(null, ex.toString());
  } }
```

Далее исключим тест `testConfigureWindow()` для метода `ConfigureWindow()` класса `SearchInternetApp`, т. к. этот метод пустой и не используется.

В тесте `testGetApplication()` будем проверять, возвращает ли метод `getApplication()` экземпляр класса `SearchInternetApp` (листинг 3.20).

Листинг 3.20. Код теста метода `getApplication()` класса `SearchInternetApp`

```
@Test
public void testGetApplication() {
  System.out.println("getApplication");
  SearchInternetApp expResult = null;
  SearchInternetApp result = SearchInternetApp.getApplication();
  // TODO review the generated test code and remove the default
  // call to fail.
  if (result==expResult)
    fail("application is null");
}
```

И наконец, в тесте `testMain()` будем проверять наличие контекста приложения (листинг 3.21).

Листинг 3.21. Код теста метода `getContext()` класса `SearchInternetApp`

```
@Test
public void testMain() {
  System.out.println("main");
  String[] args = null;
  SearchInternetApp.main(args);
  // TODO review the generated test code and remove the default
  // call to fail.
  if (SearchInternetApp.getApplication().getContext()==null)
    fail("Application's context is null");
}
```

Запустить тесты класса `SearchInternetApp` можно, щелкнув правой кнопкой мыши в окне **Projects** в узле **Test Packages** на файле SearchInternetAppTest.java и в появившемся контекстном меню выбрав пункт **Run File**. После этого в окне **JUnit Test Results** мы увидим, что все три теста прошли успешно.

Аналогичным образом создадим JUnit-тест для класса `SearchInternetAboutBox` приложения (листинг 3.22).

Листинг 3.22. Код теста для класса `SearchInternetAboutBox`

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package searchinternet;
import javax.swing.JFrame;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
public class SearchInternetAboutBoxTest {
  private JFrame mainFrame;
  public SearchInternetAboutBoxTest() {  }
  @BeforeClass
```

```
  public static void setUpClass() throws Exception {  }
  @AfterClass
  public static void tearDownClass() throws Exception {  }
  @Before
  public void setUp() {
    mainFrame = SearchInternetApp.getApplication().getMainFrame();
  }
  @After
  public void tearDown() {  }
/**
  * Test of closeAboutBox method, of class SearchInternetAboutBox.
  */
  @Test
  public void testCloseAboutBox() {
    System.out.println("closeAboutBox");
    SearchInternetAboutBox instance =
      new SearchInternetAboutBox(mainFrame);
    instance.closeAboutBox();
    assertEquals(instance.isDisplayable(),false);
  }
}
```

В коде теста для класса `SearchInternetAboutBox` приложения создадим поле класса `SearchInternetAboutBoxTest` — объект `mainFrame` класса `javax.swing.JFrame`, представляющий главное окно приложения — родительское для диалогового окна **About**. Объект `mainFrame` инициализируем в методе `setUp()`. В тесте `testCloseAboutBox()` с помощью метода `isDisplayable()`, который класс `javax.swing.JDialog` наследует от класса `java.awt.Component`, убедимся, что диалоговое окно становится невизуализируемым, т. е. не имеющим доступа к ресурсам операционной системы, после вызова метода `closeAboutBox()` класса `SearchInternetAboutBox` приложения.

И последним создадим JUnit-тест для класса `SearchInternetView` приложения (листинг 3.23).

Листинг 3.23. Код теста для класса `SearchInternetView`

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package searchinternet;
import org.jdesktop.application.Task;
```

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
public class SearchInternetViewTest {
private SearchInternetView instance ;
  public SearchInternetViewTest() {  }
  @BeforeClass
  public static void setUpClass() throws Exception {  }
  @AfterClass
  public static void tearDownClass() throws Exception {  }
  @Before
  public void setUp() {
    instance =
      new SearchInternetView(SearchInternetApp.getApplication());
  }
  @After
  public void tearDown() {  }
/**
  * Test of showAboutBox method, of class SearchInternetView.
  */
  @Test
  public void testShowAboutBox() {
    System.out.println("showAboutBox");
    instance.showAboutBox();
  }
/**
  * Test of Submit method, of class SearchInternetView.
  */
  @Test
  public void testSubmit() {
    System.out.println("Submit");
    Task expResult = null;
    Task result = instance.Submit();
    if (result== expResult)
      fail("The test case is a prototype.");
  }}
```

В коде теста для класса `SearchInternetView` приложения создадим поле — объект `instance` класса `SearchInternetView`, который инициализируем в методе `setUp()`. После этого в тесте `testShowAboutBox()` методом `showAboutBox()`

класса `SearchInternetView` выведем на экран диалоговое окно **About**. В тесте `testSubmit()` убедимся, что метод `Submit()` класса `SearchInternetView` создает объект внутреннего класса `SubmitTask`, представляющего фоновую задачу поиска информации в Интернете.

Запустим все созданные тесты для нашего приложения. Для этого в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на имени проекта и выберем пункт **Test**. После этого в окне **Test Results** мы увидим, что все тесты приложения прошли успешно.

При желании можно увеличить количество тестов для проверки правильности работы методов и конструкторов классов приложения.

ГЛАВА 4

Создание расширяемых Java-приложений

Понятие расширяемых приложений и их назначение

Основная идея расширяемого приложения — это создание приложения, которое позволяет расширить свою функциональность путем добавления программных компонентов без перекомпиляции всего приложения.

Расширяемость приложения — это его свойство, характеризующее расширение функциональных возможностей как приложения в целом, так и его отдельных компонентов. Один из основных способов придать свойство расширяемости приложению — это создание приложения, состоящего из таких дискретных программных компонентов — модулей, что изменение одного из них не требует изменения остальных. Модульная архитектура приложения позволяет расширять его функциональность не только на стадии разработки, но и динамически, во время выполнения приложения, путем динамического подключения модулей, ранее не использовавшихся.

Такая архитектура приложения достигается применением технологии программирования, которая называется *модульным программированием*. В этом случае каждый модуль программируется, отлаживается и компилируется отдельно от остальных модулей. Более того, могут создаваться многократно используемые модули, которые возможно включать в состав различных приложений.

Основной целью модульного программирования является упрощение разработки больших программных систем, т. к. деление системы на отдельные независимые части упрощает разработку и повышает надежность, гибкость и простоту обслуживания конечного продукта. Надежность системы обеспечивается слабым связыванием ее отдельных частей, при котором ошибка выполнения отдельного модуля не приводит к аварийному завершению работы

всей системы. Простота обслуживания и сопровождения конечного продукта обеспечивается его модульной архитектурой, которая позволяет легко устранять ошибки и настраивать систему путем замены и композиции ее отдельных частей. Гибкость программного комплекса означает его способность адаптации к изменяющимся требованиям относительно функциональности системы. Гибкость как раз и обеспечивается свойством расширяемости приложения, которое означает не только возможность добавления или удаления отдельных частей, но и доступность их оперативного обновления.

Модульная архитектура приложений, если еще они имеют и открытый исходный код или специальные библиотеки для подключения модулей, дает возможность участвовать в разработке и расширении возможностей программного продукта не только его авторам, но и большому числу сторонних разработчиков.

Таким образом, расширяемые приложения имеют в своей основе модульную архитектуру, позволяющую создавать большие программные комплексы, обладающие способностью к своему дальнейшему развитию.

Практика применения принципов расширяемости и модульности

Принципы расширяемости и модульности применяются в настоящее время при разработке практически всех программных продуктов, начиная с операционных систем и заканчивая Web-сайтами.

При создании операционной системы Windows компания Microsoft стала реализовывать модульную архитектуру, начиная с Windows Vista и Windows Server 2008. В Windows Vista упрощен переход от одной версии к другой, не подразумевающий полную переустановку операционной системы. В Windows Server 2008 при установке пользователь может выбрать одну из ролей, которые определяют конфигурацию системы. Полная модульная архитектура обещана в Windows 7, в которой пользователь сам сможет добавлять к ядру операционной системы модули, обеспечивающие нужную функциональность.

В отличие от Windows, UNIX-подобные операционные системы изначально создавались с модульной архитектурой, состоящей из небольшого ядра и большого набора дополнительных модулей, представляющих различного рода сервисы.

Все распределенные системы реализуют принципы расширяемости и модульности, т. к. состоят из взаимодействующих между собой самостоятельных компонентов, которые могут свободно добавляться или удаляться из

системы. Примерами распределенных систем служат Всемирная паутина WWW, клиент-серверные приложения, использующие технологию SOA (Service-Oriented Architecture), распределенные информационные системы и т. д.

Множество настольных приложений, таких как графические редакторы, офисные приложения, браузеры и многие другие, используют плагины, расширяющие их функциональность.

Наконец, среды разработки NetBeans и Eclipse имеют в своей основе модульную архитектуру, позволяющую реализовывать их широкие возможности.

Как видно, в настоящее время уметь применять принципы расширяемости и модульности при разработке приложений — жизненная необходимость для профессиональных программистов.

Способы создания расширяемых Java-приложений с помощью библиотек ServiceLoader API и Lookup, платформы NetBeans и технологии OSGi

Технология Java в самой своей основе реализует принцип расширяемости, т. к. содержит библиотеки пакетов классов и интерфейсов, которые созданы с учетом своего расширения. Кроме того, Java предоставляет специальные технологии, позволяющие создавать расширяемые Java-приложения.

Далее рассмотрим несколько способов создания расширяемых Java-приложений. Первый способ — это использование ServiceLoader API платформы Java SE 6. Второй способ — это использование библиотеки Lookup платформы NetBeans. Третий способ — это создание модульного приложения на основе платформы NetBeans. И рассмотрим еще один способ — использование технологии OSGi.

Как выше было классифицировано, первым способом создания расширяемых Java-приложений является применение класса `java.util.ServiceLoader<S>` платформы Java SE 6.

Класс `ServiceLoader` позволяет создавать расширяемые приложения без модификации исходного кода. Использование класса `ServiceLoader` помогает улучшить функциональность приложения с помощью подключения различных реализаций интерфейса или абстрактного класса.

Разберемся с понятиями, которые лежат в основе работы класса `ServiceLoader`.

Класс `ServiceLoader` обеспечивает загрузку *поставщика сервиса*. Здесь поставщик сервиса (service provider) — конкретная реализация *сервиса*, представляющего собой набор интерфейсов и абстрактных классов одного типа. Сервис может быть представлен и одним интерфейсом или абстрактным классом.

Поставщик сервиса — это один класс или набор классов, реализующих интерфейс и/или расширяющих абстрактные классы сервиса. Класс поставщика сервиса не является конечной реализацией сервиса, он представляет лишь конкретную функциональность, заложенную в сервисе. Разнообразие поставщиков сервиса обеспечивает разнообразие функций, которые может дать данный сервис. Конечной реализацией сервиса является класс приложения, который использует различных поставщиков сервиса для реализации своей функциональности.

Поставщик сервиса подключается к классу приложения с помощью *конфигурационного файла поставщика сервиса*, расположенного в каталоге META-INF\services приложения. Чтобы поставщик сервиса имел возможность быть подключенным, его конфигурационный файл должен иметь имя, идентичное полному имени типа сервиса, т. е. носить полное имя интерфейса или абстрактного класса сервиса. Кроме того, конфигурационный файл поставщика сервиса должен содержать список полных имен классов поставщика сервиса.

Таким образом, создание расширяемого приложения с применением класса `ServiceLoader` подразумевает следующие шаги:

1. Создание интерфейса или абстрактного класса, представляющего сервис — набор специфических возможностей, обеспечивающий определенную функциональность приложения.
2. Создание класса — поставщика сервиса, реализующего интерфейс или расширяющего абстрактный класс сервиса.
3. Создание конфигурационного файла в каталоге META-INF\services\, содержащего строку с именем класса поставщика сервиса.
4. Создание класса приложения, использующего все вышеперечисленное.

Вторым и третьим способом создания расширяемых Java-приложений было упомянуто использование платформы NetBeans. Платформа NetBeans является модульной и сама обеспечивает базу для создания модульного, расширяемого приложения, предоставляя возможность включать свои модули в приложение. В частности, среда разработки NetBeans IDE построена на базе платформы NetBeans.

NetBeans-модуль — это JAR-файл, включающий в себя:

- набор Java-классов, снабжающих приложение специфическими возможностями;

- файл манифеста, в котором с помощью свойств `OpenIDE-Module` JAR-файл объявляется модулем. Свойства `OpenIDE-Module` файла манифеста модуля описывают имя, версию модуля, сам модуль, файл layer модуля.

Каждый NetBeans-модуль снабжается XML-файлом layer.xml, содержимое которого определяет файлы или объекты, которые модуль добавляет в файловую систему NetBeans, контролирующую общую конфигурацию приложения.

Файловая система NetBeans является виртуальной файловой системой, управляемой с помощью Filesystems API платформы NetBeans *(см. приложение 4)*. Файловая система NetBeans собирается из конфигурационных файлов layer.xml всех NetBeans-модулей вместе с используемыми каталогами файловой системы компьютера. Библиотека Filesystems API обеспечивает доступ к объектам класса `org.openide.filesystems.FileObject`, который представляет объекты файловой системы NetBeans. Отличие класса `FileObject` платформы NetBeans от класса `java.io.File` платформы Java SE состоит в том, что объекты `FileObject` получают из файловой системы как уже существующие, а не создают программным образом. Кроме того, объекты `FileObject` имеют MIME-типы, которые идентифицируют тип содержимого файловых объектов, атрибуты, состоящие из пар "ключ — значение", представляющие метаданные файловых объектов, и собственные входящие и выходящие потоки. Также имеется возможность отслеживать изменение свойств объектов `FileObject`. Filesystems API может использоваться для представления любых данных, подобных `File`, с иерархической структурой хранения. Объекты `FileObject` могут представлять как пользовательские данные, хранящиеся на диске компьютера, так и данные, хранящиеся в XML-файле layer.xml модуля.

Минимальная конфигурация взаимосвязанных модулей платформы NetBeans (NetBeans Runtime Container), на основе которой возможно создание модульного приложения, состоит из:

- пакета `org.netbeans.core.startup`, представляющего модуль `Startup`;
- пакета `org.netbeans`, представляющего модуль `Bootstrap`;
- модулей File System API, Module System API и Utilities API.

Далее рассмотрим функции и взаимосвязи модулей конфигурации NetBeans Runtime Container.

Модуль `Startup` связан со всеми остальными модулями NetBeans Runtime Container и осуществляет следующие основные функции:

- обеспечивает общий контроль над модульной системой, включая загрузку и инсталляцию модулей;
- обеспечивает регистрацию событий во время запуска модульной системы;

- снабжает модульную систему механизмом поиска поставщиков сервиса по умолчанию;
- обеспечивает запуск модульной системы в графическом режиме;
- производит регистрацию событий во время запуска модульной системы;
- устанавливает зависимости между модулями и производит обработку файлов манифеста модулей;
- осуществляет взаимодействие с окружением модульной системы;
- устанавливает местонахождение инсталлируемых файлов модулей;
- обеспечивает хранение информации о модуле — был ли модуль инсталлирован или требует восстановления, версии модуля и требует ли он обновления;
- создает хранилище модулей на основе файловой системы компьютера;
- осуществляет взаимодействие с файловой системой NetBeans.

Модуль `Bootstrap` связан с модулями Module System API и Utilities API. Основные функции модуля `Bootstrap`:

- обеспечивает запуск модульной системы NetBeans;
- отслеживает события, происходящие в модульной системе;
- обеспечивает загрузку классов из набора JAR-файлов и локальных каталогов;
- отвечает за открытие JAR-файла NetBeans-модуля, чтение файла манифеста, установление зависимостей модуля, подготовку к инсталляции;
- инсталлирует содержимое JAR-файла модуля в NetBeans IDE;
- позволяет реализовывать политику безопасности.

Модуль File System API связан с модулем Utilities API и обеспечивает доступ к файлам виртуальной файловой системы NetBeans. Кроме того, модуль содержит классы, позволяющие разрабатывать пользовательскую виртуальную файловую систему.

Модуль Module System API связан с модулем Utilities API и описывает, как NetBeans-модуль добавляется и управляется в модульной системе.

Модуль Utilities API обеспечивает регистрацию и отчетность о сбоях в модульной системе, управляет жизненным циклом и обеспечивает общую инфраструктуру модульной системы, обработку XML-документов, снабжает некоторыми стандартными действиями, используемыми для взаимодействия с пользователем, обеспечивает передачу данных, в том числе передачу со-

ставных объектов, управление потоками ввода/вывода, представляет механизм расширения.

Модуль Utilities API включает в себя библиотеку Lookup, представляющую механизм расширения, который решает проблему регистрации и взаимодействия компонентов в системе. Библиотека Lookup имеет сходство с библиотекой Jini — платформой для разработки распределенных сетевых сервисов. Отличие библиотеки Lookup от Jini в том, что Lookup решает проблему взаимодействия модулей в модульной системе, а не сетевой интеграции приложений. Библиотека Lookup состоит из Lookup API и Lookup SPI.

Как раз использование библиотеки Lookup платформы NetBeans и было обозначено нами как второй способ создания расширяемых Java-приложений.

Библиотека Lookup API платформы NetBeans представлена основным классом `org.openide.util.Lookup` и его вложенными классами `Lookup.Item<T>`, `Lookup.Provider`, `Lookup.Result<T>`, `Lookup.Template<T>`, классом `org.openide.util.LookupEvent` и интерфейсом `org.openide.util.LookupListener`. Библиотека Lookup API позволяет осуществлять поиск поставщиков сервиса и следить за изменениями в наборе имеющихся поставщиков сервиса.

Библиотека Lookup SPI состоит из классов пакета `org.openide.util.lookup` и упрощает разработку собственного механизма регистрации и поиска поставщиков сервиса.

Отличие библиотеки Lookup платформы NetBeans от ServiceLoader API платформы Java SE 6 заключается в следующем:

- библиотека Lookup является расширяемой в отличие от финального класса `java.util.ServiceLoader<S>`;
- библиотека Lookup позволяет следить за изменениями в наборе доступных поставщиков сервиса и тем самым осуществлять динамическую синхронизацию набора доступных поставщиков сервиса с набором задействованных модулей модульной системы;
- библиотека Lookup может применяться для приложений, создаваемых на основе более ранних версий JDK, чем 1.6;
- с помощью библиотеки Lookup поиск поставщиков сервиса может быть разделен и масштабирован в пределах приложения;
- в конфигурационном файле каталога META-INF\services\ с помощью специального комментария `#-` можно исключить регистрацию поставщика сервиса;
- в случае регистрации нескольких поставщиков сервиса в одном конфигурационном файле каталога META-INF\services\ с помощью комментария `#position=` можно установить порядок загрузки поставщиков сервиса.

Для создания расширяемого приложения с использованием библиотеки Lookup платформы NetBeans в простом случае необходимо проделать те же самые действия, что и при создании расширяемого приложения с применением класса `ServiceLoader`:

1. Создать интерфейс или абстрактный класс, представляющий сервис.
2. Создать поставщиков сервиса.
3. Зарегистрировать поставщиков сервиса.
4. Создать класс приложения, использующий сервис.
5. Дополнительно необходимо добавить файл org-openide-util.jar, содержащий классы библиотеки Lookup, в путь приложения.

В качестве третьего способа создания расширяемых Java-приложений нами было названо использование самой платформы NetBeans как основы модульного приложения. В этом случае Java-приложение использует для обеспечения своей основной инфраструктуры модули платформы NetBeans, дополняя их набор NetBeans-модулями, реализующими специфические функции приложения.

При создании модульного приложения на основе платформы NetBeans достаточно написать модули, реализующие специфические задачи приложения, и связать их с необходимыми для работы приложения модулями платформы. Кроме того, можно создать модуль, предназначенный для интеграции в среду NetBeans, и тем самым способный расширять ее возможности, при этом функциональность созданного NetBeans-плагина может расширяться за счет подключения к нему других модулей.

И, наконец, как последний способ создания расширяемых Java-приложений, нами было упомянуто использование технологии OSGi.

Технология OSGi — это технология динамической модульной Java-системы, представляющей собой платформу компонентной среды выполнения, в которой выполняются Java-приложения, состоящие из взаимодействующих между собой модулей.

Технология OSGi разрабатывается некоммерческой организацией OSGi Alliance (**http://www.osgi.org/**) и описывается набором спецификаций, определяющих модульную архитектуру, состоящую из среды выполнения — OSGi-платформы и OSGi-модулей *(bundles)*, которые инсталлируются и запускаются в среде выполнения OSGi.

На сегодняшний день существуют три реализации технологии OSGi с открытым исходным кодом — это Apache Felix (**http://felix.apache.org/**), Knopflerfish (**http://www.knopflerfish.org/**) и Equinox (**http://www.eclipse.org/equinox/**).

Сама OSGi-платформа, обеспечивающая среду выполнения для OSGi-модулей, имеет многоуровневую архитектуру, в которой на основе стандартной среды выполнения Java, предоставленной Java-платформами, действуют:

- управление взаимодействием между OSGi-модулями;
- управление жизненным циклом OSGi-модулей — их динамической инсталляцией, запуском, остановкой, обновлением и деинсталляцией;
- поддержка реестра сервисов, предназначенного для хранения информации о зарегистрированных сервисах, реализованных OSGi-модулями и посредством которого OSGi-модули находят нужные им сервисы.

В терминологии сервисов и поставщиков сервисов OSGi-модуль может являться поставщиком сервиса, представленным интерфейсом или абстрактным Java-классом. При своем запуске OSGi-модуль, являющийся поставщиком сервиса, регистрирует свой сервис в реестре сервисов. Остальные OSGi-модули, потребители сервиса, следят за изменениями в реестре сервисов и, в случае появления в нем информации о нужном им сервисе, образуют динамическую связь с поставщиком сервиса.

Таким образом, для создания расширяемого приложения с использованием технологии OSGi необходимо:

1. Установить реализацию технологии OSGi — Apache Felix, Knopflerfish или Equinox.
2. Создать OSGi-модули, реализующие сервисы, и OSGi-модули, потребляющие сервисы.
3. Инсталлировать и запустить созданные OSGi-модули, используя платформу OSGi.

Пример создания расширяемого приложения с использованием библиотеки ServiceLoader API платформы Java SE 6

В качестве примера расширяемого Java-приложения рассмотрим программу, которая позволяет генерировать документацию на основе комментариев к исходным Java-файлам, используя различные подключаемые доклеты.

Первым способом создания такого Java-приложения будет использование ServiceLoader API платформы Java SE 6.

Класс `java.util.ServiceLoader<S>` платформы Java SE 6 является финальным, т. е. не позволяющим создать свой подкласс, и предназначен для реализации загрузки поставщиков сервиса. Напомним, что сервисом в данной термино-

логии является интерфейс или абстрактный класс, представляющий какие-либо специфические возможности. Поставщик сервиса (service provider) — это реализация сервиса, классы поставщика сервиса реализуют интерфейс или расширяют абстрактный класс сервиса.

Для того чтобы поставщик сервиса был загружен классом `ServiceLoader`, он должен быть предварительно инсталлирован.

Поставщик сервиса может быть инсталлирован путем добавления в путь приложения (user class path) или помещением своего JAR-файла в каталог \lib\ext среды ваполнения JRE, таким образом, реализуя общий механизм расширения платформы Java SE.

Механизм расширения платформы Java SE предназначен для использования дополнительных пакетов (optional packages), расширяющих платформу Java SE. Этот механизм подразумевает при запуске приложения поиск необходимых class-файлов сначала в архивах JAR папки ext среды выполнения JRE, а затем, в случае если необходимые файлы не будут найдены, использование пути *user class path* — списка каталогов, архивов JAR и ZIP, которые содержат необходимые class-файлы. Значение *user class path* по умолчанию — `"."`. Это означает, что файлы классов расположены в текущем каталоге. Опции командной строки `-cp` или `-classpath` переопределяют значение *user class path* по умолчанию. Если используется опция `-jar` командной строки, то поиск осуществляется в соответствующем архиве. Значение *user class path* может также быть установлено значением переменной окружения `CLASSPATH`, которая определяется либо опцией `Class-Path` файла MANIFEST.MF, либо в командной строке с помощью `set classpath=`. Можно также определить глобальную переменную окружения `CLASSPATH` в качестве переменной среды JRE в свойствах операционной системы, но этого делать не рекомендуется.

Основное требование к классу поставщика сервиса заключается в том, что он должен иметь конструктор без аргументов для того, чтобы использовать его по умолчанию при загрузке классом `ServiceLoader` для создания экземпляра класса поставщика сервиса. Поставщик сервиса должен также быть зарегистрирован с помощью конфигурационного файла каталога META-INF\services\, при этом имя конфигурационного файла должно совпадать с полным именем сервиса и содержать строку с полным именем класса, реализующего интерфейс или расширяющего абстрактный класс сервиса. Если класс поставщика сервиса зарегистрирован более чем в одном конфигурационном файле или упоминается в одном конфигурационном файле несколько раз, то все повторы игнорируются при загрузке. Каталог META-INF\services\, содержащий конфигурационный файл, не обязательно должен находиться в JAR-файле поставщика сервиса, главное, что он должен быть доступен через путь приложения.

Поставщик сервиса, как правило, не содержит главный класс приложения, скорее, классы поставщика сервиса используются приложением как библиотечные классы.

Разработка сервиса

Для создания задуманного нами приложения с использованием класса `ServiceLoader` в качестве первого шага создадим сервис — интерфейс `createJavadocIface`, который предоставляет возможность создания документации из комментариев к исходному Java-коду.

> ***Примечание***
>
> Файлы с кодами приложения расположены на прилагаемом к книге компакт-диске в папке Часть_I\Глава_4\JavadocServiceLoader.

Для создания сервиса выполним следующие действия:

1. В среде NetBeans в меню **File** выберем **New Project**. В появившемся диалоговом окне выберем **Java Class Library** (рис. 4.1).

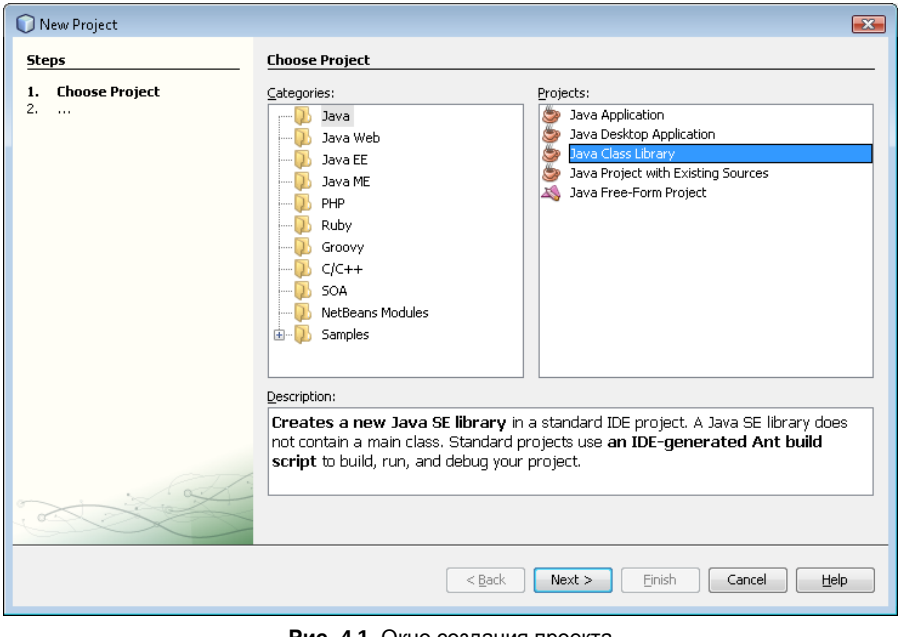

Рис. 4.1. Окно создания проекта

2. Далее, после нажатия кнопки **Next**, введем имя проекта createJavadocIface и нажмем кнопку **Finish**.
3. После этого в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на узле **Source Packages** и выберем **New | Java Package**. В появив-

шемся диалоговом окне введем имя пакета createjavadoc и нажмем кнопку **Finish**. В узле **Source Packages** появится название созданного пакета.

4. Щелкнем правой кнопкой мыши на имени пакета и выберем **New | Java Interface**. В появившемся диалоговом окне введем имя интерфейса createJavadocIface и нажмем кнопку **Finish**.
5. В окне редактора исходного кода среды NetBeans в теле интерфейса определим метод `cmdJavadoc()`, который будет вызывать командную строку с Java-инструментом `javadoc`, включая опцию `-doclet`, определяя указанный доклет (листинг 4.1). При этом аргументами метода будут служить:
   - адрес папки bin каталога JDK, содержащей Java-инструмент `javadoc`;
   - адрес каталога, в котором расположена папка пакета с исходными Java-файлами, содержащими Javadoc-комментарии;
   - адрес каталога, в котором будут создаваться файлы документации;
   - имя пакета Java-классов, для которого будет сгенерирована документация.
6. Осуществим сборку проекта createJavadocIface. Для этого в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на имени проекта и выберем пункт **Build**. В результате в папке dist проекта будет создан JAR-файл интерфейса `createJavadocIface`, который понадобится нам в дальнейшем.

Листинг 4.1. Код интерфейса сервиса

```
package createjavadociface;
  public interface createJavadocIface {
  public void cmdJavadoc(String jdk,String src,String out,String pkg);
}
```

Теперь нам нужно создать поставщиков нашего сервиса.

Поставщики сервиса в нашем случае должны быть представлены классами, реализующими интерфейс `createJavadocIface`.

В качестве примера поставщиков сервиса создадим два модуля, которые будут, реализуя интерфейс `createJavadocIface`, генерировать документацию, используя различные доклеты. Один поставщик сервиса будет генерировать документацию в формате PDF, а другой — в формате RTF.

Для создания поставщика сервиса, генерирующего документацию в формате PDF:

1. В среде NetBeans в меню **File** выберем **New Project | Java Class Library**, введем имя проекта createJavadocPDF, представляющего поставщика сер-

виса, генерирующего документацию в формате PDF, и нажмем кнопку **Finish**.

2. В окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на узле **Source Packages** созданного проекта createJavadocPDF и выберем **New | Java Package**. В появившемся диалоговом окне введем имя пакета createjavadocpdf и нажмем кнопку **Finish**.
3. В узле **Source Packages** щелкнем правой кнопкой мыши на названии созданного пакета createjavadocpdf и выберем **New | Java Class**. В появившемся диалоговом окне введем имя класса `createJavadocPDF` и нажмем кнопку **Finish**.
4. В окне редактора исходного кода среды NetBeans в теле класса переопределим метод `cmdJavadoc()` интерфейса `createJavadocIface`, при этом определив класс `createJavadocPDF` как класс, реализующий данный интерфейс с помощью ключевого слова `implements` (листинг 4.2).
5. Подключим JAR-файл интерфейса `createJavadocIface` в качестве библиотеки проекта. Для этого в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на узле **Libraries** проекта createJavadocPDF и выберем **Add JAR/Folder**, в появившемся диалоговом окне найдем файл createJavadocIface.jar в папке dist проекта createJavadocIface и нажмем кнопку **Open**. В результате в узле **Libraries** появится имя файла — createJavadocIface.jar.

Листинг 4.2. Код класса поставщика сервиса, генерирующего документацию в формате PDF

```
package createjavadocpdf;
import createjavadociface.createJavadocIface;
public class createJavadocPDF implements createJavadocIface{
  @Override
  public void cmdJavadoc(String jdk,String src,String out,String pkg){
    Runtime r =Runtime.getRuntime();
    String pathDoclet=this.getClass().getResource(
       "createJavadocPDF.class").getPath();
    StringBuffer buf=new StringBuffer(pathDoclet);
    buf.delete(pathDoclet.indexOf("createJavadocPDF"),
               pathDoclet.length());
    buf.delete(0,6);
    pathDoclet=buf.toString();
    String cmd=jdk+"/bin/javadoc"+
               " -doclet com.tarsec.javadoc.pdfdoclet.PDFDoclet"+
               " -docletpath "+pathDoclet+"pdfdoclet.jar"+
               " -sourcepath "+src+" "+pkg;
```

```
    try {
      r.exec(cmd, null,new java.io.File(out));
    }
    catch(Exception ex)
     {javax.swing.JOptionPane.showMessageDialog(null, ex.toString());}
  }
}
```

Обсудим код класса `createJavadocPDF` (листинг 4.2).

В коде класса `createJavadocPDF` установим контроль над правильностью переопределения метода `cmdJavadoc()` интерфейса `createJavadocIface` с помощью аннотации `@Override`.

Далее в коде метода `cmdJavadoc()` получим объект класса `java.lang.Runtime`, соответствующий нашему приложению. При запуске каждого Java-приложения создается один-единственный объект класса `Runtime`, позволяющий приложению взаимодействовать со средой выполнения. Статический метод `getRuntime()` класса `Runtime` возвращает этот объект для данного приложения.

В коде метода `cmdJavadoc()` вызывается метод `exec()` класса `Runtime`, который создает процесс операционной системы и возвращает экземпляр класса, расширяющего абстрактный класс `java.lang.Process`. Класс `Process` обеспечивает методы для контроля процесса и получения информации о нем.

Метод `exec()` класса `Runtime` исполняет командную строку, которая определяется первым аргументом метода, в отдельном процессе с переменными окружения, определяемыми вторым аргументом метода, и рабочим каталогом, указанным в третьем аргументе метода. Если переменные окружения наследуются создаваемым процессом от существующего процесса, то второй аргумент метода принимает значение `null`. Если рабочий каталог создаваемого процесса совпадает с рабочим каталогом существующего процесса, то третий аргумент метода принимает значение `null`. Применение метода `exec()` класса `Runtime` требует перехвата исключений, вызванных возможными ошибками в аргументах метода, ошибками ввода/вывода и конфликтами с политикой безопасности.

Модуль `createJavadocPDF` будет создавать документацию в формате PDF с помощью доклета pdfdoclet.jar, который можно скачать, используя страничку по адресу **http://java.sun.com/j2se/javadoc/faq/index.html**. Скачаем JAR-файл доклета и поместим его в каталог, содержащий проект createJavadocPDF. Сформируем командную строку для метода `exec()`, состоящую из:

- строковой переменной `jdk`, определяющей абсолютный путь каталога JDK;

- дополнения к переменной `jdk`, которое определяет путь Java-инструмента `javadoc`;
- опции `-doclet`, подключающей главный класс доклета `com.tarsec.javadoc.pdfdoclet.PDFDoclet`;
- опции `-docletpath`, определяющей путь доклета;
- строковой переменной `pathDoclet`, содержащей абсолютный путь к каталогу доклета;
- названия доклета pdfdoclet.jar, которое вместе с абсолютным путем к каталогу доклета формирует абсолютный путь доклета;
- опции `-sourcepath`, определяющей путь, содержащий папку пакета, для которого создается Javadoc-документация;
- строковой переменной `src`, содержащей абсолютный путь папки пакета;
- строковой переменной `pkg` с именем пакета, для которого будет сгенерирована документация.

В коде метода `cmdJavadoc()` строковую переменную `pathDoclet`, содержащую абсолютный путь к каталогу, в котором расположен JAR-файл доклета, будем определять программным образом. Методом `getClass()` класса `java.lang.Object` получим объект класса `java.lang.Class<T>`, представляющий исполняемый виртуальной Java-машиной класс. С помощью метода `getResource()` класса `Class<T>` получим адрес файла createJavadocPDF.class, который приведем к типу `String` методом `getPath()` класса `java.net.URL`. Используя метод `delete()` класса `java.lang.StringBuffer`, выделим из строки адреса файла createJavadocPDF.class абсолютный путь доклета.

После создания класса поставщика сервиса необходимо обеспечить его регистрацию.

Для того чтобы зарегистрировать поставщика сервиса:

1. Создадим в папке src проекта каталог META-INF\services, содержащий конфигурационный файл с именем createjavadociface.createJavadocIface (полное имя сервиса). Для этого в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на узле **Source Packages** проекта и выберем **New | Other | Other | Folder**. Введем имя каталога — META-INF\services и нажмем кнопку **Finish**.
2. Далее создадим конфигурационный файл. В окне **Projects** щелкнем правой кнопкой мыши на созданном каталоге META-INF\services и выберем **New | Other | Other | Empty File**. Введем имя файла — createjavadociface.createJavadocIface и нажмем кнопку **Finish**.

3. В окне редактора исходного кода среды NetBeans введем в созданном файле одну строку — `createjavadocpdf.createJavadocPDF`.

Таким образом, мы зарегистрируем нашего поставщика сервиса.

В качестве последнего шага создания поставщика сервиса, генерирующего документацию в формате PDF, осуществим сборку проекта createJavadocPDF, щелкнув правой кнопкой мыши на названии проекта в окне **Projects** среды NetBeans и выбрав пункт **Build**. В результате в папке dist проекта появится созданный файл createJavadocPDF.jar.

Создадим второго поставщика сервиса, который будет генерировать Javadoc-документацию в формате RTF. Для этого выполним следующие действия:

1. В среде NetBeans в меню **File** выберем **New Project | Categories: Java | Java Class Library**, введем имя проекта createJavadocRTF и нажмем кнопку **Finish**.
2. В окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на узле **Source Packages** созданного проекта createJavadocRTF и выберем **New | Java Package**. Введем имя пакета createjavadocrtf и нажмем кнопку **Finish**.
3. В узле **Source Packages** проекта щелкнем правой кнопкой мыши на названии созданного пакета createjavadocrtf и выберем **New | Java Class**. В появившемся диалоговом окне введем имя класса createJavadocRTF и нажмем кнопку **Finish**.
4. В окне редактора исходного кода среды NetBeans определим класс `createJavadocRTF` как класс, реализующий интерфейс `createJavadocIface` с помощью ключевого слова `implements`. В теле класса переопределим метод `cmdJavadoc()` интерфейса `createJavadocIface` (листинг 4.3).
5. Подключим JAR-файл интерфейса `createJavadocIface` в качестве библиотеки проекта. Для этого в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на узле **Libraries** проекта createJavadocRTF и выберем **Add JAR/Folder**, в появившемся диалоговом окне найдем файл createJavadocIface.jar в папке dist проекта createJavadocIface и нажмем кнопку **Open**. В результате в узле **Libraries** появится имя файла — createJavadocIface.jar.

Листинг 4.3. Код класса поставщика сервиса, генерирующего документацию в формате RTF

```
package createjavadocrtf;
import createjavadociface.createJavadocIface;
public class createJavadocRTF implements createJavadocIface {
public void cmdJavadoc(String jdk,String src,String out,String pkg){
  Runtime r =Runtime.getRuntime();
```

```
  String pathDoclet=this.getClass().getResource(
    "createJavadocRTF.class").getPath();
  StringBuffer buf=new StringBuffer(pathDoclet);
  buf.delete(pathDoclet.indexOf("createJavadocRTF"),
             pathDoclet.length());
  buf.delete(0,6);
  pathDoclet=buf.toString();
  String cmd=jdk+"/bin/javadoc"+" -doclet RTFDoclet"+
            " -docletpath "+pathDoclet+"RTFDoclet.jar"+
            " -sourcepath "+src+" -filename "+pkg+".rtf "+pkg;
  try {
    r.exec(cmd, null,new java.io.File(out));
  }
  catch(Exception ex)
    {javax.swing.JOptionPane.showMessageDialog(null, ex.toString());}
}
}
```

Обсудим код класса `createJavadocRTF` (см. листинг 4.3).

Код класса `createJavadocRTF` аналогичен коду класса `createJavadocPDF` (см. листинг 4.2) за исключением командной строки.

Точно так же, как и в классе `createJavadocPDF`, в теле метода `cmdJavadoc()` класса `createJavadocRTF` получим объект класса `java.lang.Runtime`, соответствующий нашему приложению, используя статический метод `getRuntime()` класса `Runtime`. Методом `exec()` класса `Runtime` создадим процесс операционной системы, который будет выполнять командную строку.

Модуль `createJavadocRTF` будет создавать документацию в формате RTF с помощью доклета RTFDoclet.jar, который также можно скачать, используя страничку по адресу **http://java.sun.com/j2se/javadoc/faq/index.html**. Аналогичным образом поместим JAR-файл скачанного доклета в каталог, содержащий проект createJavadocRTF.

Сформируем командную строку для метода `exec()`, состоящую из:

- строковой переменной `jdk`, определяющей абсолютный путь каталога JDK;
- дополнения "/bin/javadoc" к переменной `jdk`, которое определяет путь Java-инструмента `javadoc`;
- опции `-doclet`, подключающей главный класс доклета RTFDoclet;
- опции `-docletpath`, определяющей путь доклета;
- строковой переменной `pathDoclet`, содержащей абсолютный путь к каталогу доклета;

- названия доклета RTFDoclet.jar, которое вместе с абсолютным путем к каталогу доклета формирует абсолютный путь доклета;
- опции `-sourcepath`, определяющей путь к каталогу пакета, для которого создается Javadoc-документация;
- строковой переменной `src`, содержащей абсолютный путь к каталогу пакета;
- опции `-filename`, определяющей имя файла с Javadoc-документацией;
- строковой переменной `pkg`, содержащей имя пакета, которая вместе с расширением rtf будет указывать имя создаваемого файла с Javadoc-документацией;
- строковой переменной `pkg` с именем пакета, для которого будет сгенерирована документация.

Значение строковой переменной `pathDoclet`, представляющей абсолютный путь к каталогу, в котором расположен JAR-файл доклета, так же как и в классе `createJavadocPDF`, будем определять программным образом. Методом `getClass()` класса `java.lang.Object` получим объект класса `java.lang.Class<T>`, представляющий исполняемый виртуальной Java-машиной класс. С помощью метода `getResource()` класса `Class<T>` получим адрес файла createJavadocRTF.class, который приведем к типу `String` методом `getPath()` класса `java.net.URL`. Используя метод `delete()` класса `java.lang.StringBuffer`, извлечем из строки адреса файла createJavadocRTF.class абсолютный путь к каталогу доклета.

После создания класса поставщика сервиса обеспечим его регистрацию, создав в папке src проекта createJavadocRTF каталог META-INF\services с конфигурационным файлом createjavadociface.createJavadocIface (полное имя сервиса), который должен содержать строку `"createjavadocrtf.createJavadocRTF"`.

И последним этапом создания поставщика сервиса, генерирующего документацию в формате RTF, осуществим сборку проекта createJavadocRTF, щелкнув правой кнопкой мыши на названии проекта в окне **Projects** среды NetBeans и выбрав пункт **Build**. В результате в папке dist проекта появится созданный файл createJavadocRTF.jar, необходимый нам в дальнейшем.

Разработка графического интерфейса приложения

Теперь мы можем создать графический интерфейс приложения, взаимодействующий с модулями поставщиков сервиса.

Для создания настольного Java-приложения, имеющего графический интерфейс пользователя и расширяемого за счет подключения различных модулей поставщиков сервиса:

1. В среде NetBeans в меню **File** выберем **New Project | Categories: Java | Java Desktop Application**, введем имя проекта ServiceDocDesktop и нажмем кнопку **Finish**. Среда NetBeans создаст каркас настольного приложения на основе платформы Swing Application Framework.
2. Подключим JAR-файлы интерфейса `createJavadocIface`, классов `createJavadocPDF` и `createJavadocRTF`, доклетов pdfdoclet.jar и RTFDoclet.jar как библиотечные файлы проекта ServiceDocDesktop. Для этого в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на узле **Libraries** проекта ServiceDocDesktop и выберем **Add JAR/Folder**, в появившемся диалоговом окне найдем файл createJavadocIface.jar в папке dist проекта createJavadocIface и нажмем кнопку **Open**. В результате в узле **Libraries** появится имя файла — createJavadocIface.jar. Аналогичные действия выполним для остальных JAR-файлов.
3. Модифицируем код класса `ServiceDocDesktopView`, который представляет главное окно настольного приложения, таким образом, чтобы интерфейс пользователя приложения позволял выбрать поставщика сервиса для специфической генерации Javadoc-документации, а также давал возможность пользователю легко указать необходимые данные для запуска генерации документации. Для создания главного окна интерфейса приложения в узле **Source Packages | servicedocdesktop** окна **Projects** среды NetBeans выберем файл ServiceDocDesktopView.java и, открыв его в окне редактора среды, модифицируем код класса `ServiceDocDesktopView`.

> ***Примечание***
>
> Код класса `ServiceDocDesktopView` см. в файле ServiceDocDesktopView.java папки Часть_I\Глава_4\JavadocServiceLoader\ServiceDocDesktop\src\servicedocdesktop на прилагаемом к книге компакт-диске.

Обсудим код класса `ServiceDocDesktopView`.

В коде класса `ServiceDocDesktopView` с помощью ключевого слова `import` обеспечим употребление короткого имени интерфейса `createJavadocIface` и класса `JFileChooser`.

Класс `javax.swing.JFileChooser` нам понадобится для того, чтобы пользователь, при взаимодействии с графическим интерфейсом, смог визуально указать в своей файловой системе компьютера:

- каталог JDK и тем самым определить строковую переменную `jdk` метода `cmdJavadoc()` интерфейса `createJavadocIface`;

- каталог папки пакета, для которого создается Javadoc-документация, и тем самым определить строковую переменную `src` метода `cmdJavadoc()` интерфейса `createJavadocIface`;
- пакет, для которого будет сгенерирована документация, и тем самым определить строковую переменную `pkg` метода `cmdJavadoc()` интерфейса `createJavadocIface`;
- каталог, в котором будут создаваться файлы документации, и тем самым определить строковую переменную `out` метода `cmdJavadoc()` интерфейса `createJavadocIface`.

Класс `javax.swing.JFileChooser` снабжает механизмом поиска файлов или каталогов на локальном компьютере, предоставляя графический интерфейс пользователя в виде модального диалогового окна для навигации файловой системы и возможности выбора или ввода имени файла или каталога.

В конструкторе класса `ServiceDocDesktopView` обеспечим механизм, который позволит автоматически добавить все зарегистрированные классы поставщиков сервиса в меню главного окна графического интерфейса. Для этого создадим объект `loader` класса `ServiceLoader<S>` для сервиса `createJavadocIface` с помощью статического метода `load()`; при этом объект `loader` будет поддерживать кэш, содержащий автоматически созданные экземпляры зарегистрированных классов поставщиков сервиса, реализующих интерфейс `createJavadocIface`. В цикле `for` посчитаем количество зарегистрированных классов поставщиков сервиса. Объект `inst` типа `createJavadocIface` является переменной цикла `for`, которая получает экземпляры, содержащиеся в `loader` последовательно один за другим от начала к концу. Интерфейс `java.lang.Iterable<T>`, который реализует класс `ServiceLoader<S>`, позволяет объектам обрабатываться таким циклом `foreach`.

Далее создадим массив элементов меню — переключателей, объектов `jRadioButtonMenuItem` класса `javax.swing.JRadioButtonMenuItem`. Размер массива — количество зарегистрированных классов поставщиков сервиса, вычисляется в вышеуказанном цикле `for`.

Последовательно, в цикле типа `foreach`, будем создавать объекты — элементы массива `jRadioButtonMenuItem`. Методом `setSelected()`, унаследованным от класса `javax.swing.AbstractButton`, будем устанавливать состояние переключателя по умолчанию в положение *"выбран"*. Методом `setText()` класса `AbstractButton` будем устанавливать текст переключателя как полное имя класса поставщика сервиса, полученное с помощью метода `getClass()` класса `java.lang.Object`, возвращающего исполняемый класс, и метода `getName()` класса `java.lang.Class<T>`, возвращающего имя класса. Используя метод `add()`, унаследованный от класса `java.awt.Container`, добавим элементы ме-

ню — переключатели в контейнер — объект `jMenuDoclets` класса `javax.swing.JMenu`, который представляет пункт меню **Doclets** главного окна графического интерфейса; название пункта меню определено в ресурсном файле ServiceDocDesktopView.properties свойством `"jMenuDoclets.text="`. Меню **Doclets** предварительно добавим в главное окно графического интерфейса с помощью палитры компонентов.

Конкретная реализация интерфейса `createJavadocIface`, т. е. создание объекта класса, реализующего интерфейс, и вызов переопределенного метода интерфейса `cmdJavadoc` требуют определения значений четырех строковых переменных:

- адреса каталога JDK, содержащего папку bin с Java-инструментом `javadoc`;
- адреса каталога, в котором расположена папка пакета с исходными Java-файлами, содержащими Javadoc-комментарии;
- адреса каталога, в котором будут создаваться файлы документации;
- имени пакета Java-классов, для которого будет сгенерирована документация.

В главное окно графического интерфейса, используя палитру компонентов **Palette** среды NetBeans, поместим четыре пустых текстовых поля: `jTextFieldJdk`, `jTextFieldSrc`, `jTextFieldOut` и `jTextFieldPkg`. К ним добавим четыре подписи: `jLabelJdk`, `jLabelSrc`, `jLabelOut` и `jLabelPkg` с соответствующими текстами — **JDK:**, **Source:**, **Out:** и **Package:**, определенными в ресурсном файле ServiceDocDesktopView.properties соответствующими свойствами. К текстовым полям добавим четыре кнопки: `jButtonBrowseJdk`, `jButtonBrowseSrc`, `jButtonBrowseOut` и `jButtonBrowsePkg` с текстом **Browse**, определенном в том же ресурсном файле. Создадим также кнопку `jButtonCreate` с текстом **Create javadoc**, которая будет инициировать генерацию Javadoc-документации.

Создадим методы — обработчики событий, связанных с кнопками **Browse**. Для этого в среде NetBeans в окне **Design** файла ServiceDocDesktopView.java щелкнем правой кнопкой мыши на кнопке **Browse** и в появившемся контекстном меню выберем **Events | Action | actionPerformed**. Выполним эти действия для каждой кнопки.

Для кнопки **Create javadoc** создадим действие и метод — обработчик событий этого действия, промаркированный аннотацией `@Action`, который будет обрабатывать события в фоновом потоке, не блокирующем графический интерфейс. Щелкнем правой кнопкой мыши на кнопке и откроем диалоговое окно **Set Action**. В списке **Action** выберем вариант **Create New Action**. В строке **Action's Method** введем название `createJavadoc` действия и, соответственно, имя метода, обработчика событий действия, который будет про-

маркирован аннотацией `@Action`. Отметим флажок **Background Task** и нажмем кнопку **OK**.

В результате этих действий среда NetBeans сгенерирует дополнительный код в методе `initComponents()`, который вызывается в конструкторе класса `ServiceDocDesktopView` для инициализации формы графического интерфейса. Будут созданы соответствующие объекты классов, представляющие графические компоненты, свойства компонентов будут загружены из ресурсного файла класса, сами компоненты будут сгруппированы менеджером компоновки. Для кнопок **Browse** в качестве обработчиков событий будут зарегистрированы соответствующие методы. Действие `createJavadoc` будет связано с кнопкой **Create javadoc**.

В теле методов `jButtonBrowseJdkActionPerformed()`, `jButtonBrowseOutActionPerformed()`, `jButtonBrowseSrcActionPerformed()` и `jButtonBrowsePkgActionPerformed()`, зарегистрированных в методе `initComponents()` в качестве обработчиков событий для кнопок **Browse**, создадим объект класса `javax.swing.JFileChooser`, обеспечивающего механизм поиска файлов и каталогов. Методом `setFileSelectionMode()` класса `JFileChooser` установим возможность выбора пользователем только директорий. С помощью метода `setDialogTitle()` класса `JFileChooser` определим заголовок навигационного диалогового окна файловой системы. Используя метод `showOpenDialog()` класса `JFileChooser`, выведем на экран навигационное диалоговое окно. Аргумент метода `null` указывает, что навигационное диалоговое окно не привязано к какому-либо родительскому окну. Метод `showOpenDialog()` возвращает целочисленное значение, представляющее три возможных состояния окна в результате действий пользователя:

- значение `JFileChooser.CANCEL_OPTION` указывает, что пользователь нажал кнопку **Cancel** диалогового окна;
- значение `JFileChooser.APPROVE_OPTION` указывает, что файл или каталог были успешно выбраны пользователем;
- значение `JFileChooser.ERROR_OPTION` указывает, что произошла ошибка.

В случае если метод `showOpenDialog()` возвращает значение `JFileChooser.APPROVE_OPTION`, методом `getSelectedFile()` класса `JFileChooser` получим объект класса `java.io.File`, представляющий имя пути выбранного пользователем файла или каталога. С помощью метода `getPath()` класса `File` конвертируем имя пути в строку. Используя метод `setText()`, унаследованный классом `javax.swing.JTextField` от класса `javax.swing.text.JTextComponent`, установим текст соответствующего текстового поля как строку — абсолютный путь выбранного пользователем каталога. В строковую переменную `pathChooser` будем записывать вышеуказанную строку для того, чтобы при создании объ-

екта `JFileChooser` устанавливать текущий каталог как последний выбранный пользователем. В текстовое поле `jTextFieldPkg` будем записывать не абсолютный путь, а имя пакета, для которого создается Javadoc-документация. Имя пакета будем вычленять из абсолютного пути каталога, содержащего исходные файлы, заменяя разделители файловой системы Windows "\" на разделители в имени пакета ".".

В теле метода `doInBackground()` класса `CreateJavadocTask`, код которого был сгенерирован средой NetBeans при создании действия `createJavadoc`, связанного с кнопкой **Create javadoc** интерфейса приложения, создадим объект `loader` класса `ServiceLoader<S>` для сервиса `createJavadocIface` с помощью статического метода `load()`. В цикле типа `foreach` в случае, если переключатель меню **Doclets**, связанный с одноименным поставщиком сервиса, будет выбран, в строковые переменные `jdk`, `src`, `out` и `pkg` запишем тексты соответствующих текстовых полей для того, чтобы передать их в качестве аргументов переопределенному методу `cmdJavadoc()` класса поставщика сервиса.

В методе `succeeded()` класса `CreateJavadocTask`, вызываемом после успешного выполнения метода `doInBackground()`, создадим таймер, который после указанной задержки, определенной для того, чтобы дать время поставщикам сервиса создать файлы Javadoc-документации в файловой системе пользователя, установит подпись панели состояния, сообщающую, что Javadoc-документация была успешно сгенерирована.

Модификацией кода класса `ServiceDocDesktopView` мы завершили создание расширяемого Java-приложения, генерирующего Javadoc-документацию в различных форматах и состоящего из настольного Java-приложения, взаимодействующего с подключаемыми модулями поставщиков сервиса генерации документации.

Осуществим сборку проекта ServiceDocDesktop, щелкнув правой кнопкой мыши на имени проекта в окне **Projects** среды NetBeans и выбрав пункт **Build**. В результате в папке dist проекта будет создан исполняемый JAR-файл приложения ServiceDocDesktop.jar и появится папка lib с библиотечными файлами и файлами доклетов.

Таким образом, применение ServiceLoader API платформы Java SE 6 позволяет подключить нового поставщика сервиса `createJavadocIface` без перекомпиляции исходного кода классов пакета `servicedocdesktop`, представляющего настольное приложение с графическим интерфейсом пользователя.

Для подключения нового поставщика сервиса достаточно поместить JAR-файл поставщика сервиса и файл нового доклета, если такой используется новым поставщиком сервиса, в библиотечную папку lib приложения и указать данные файлы в свойстве `Class-Path:` файла MANIFEST.MF JAR-файла

приложения. После этого, запустив приложение, можно увидеть название нового поставщика сервиса в меню приложения (рис. 4.2).

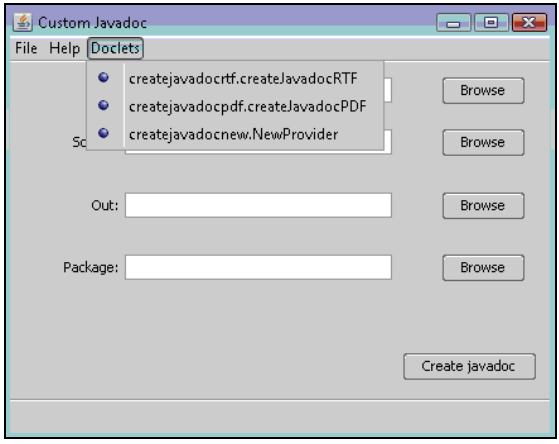

Рис. 4.2. Главное окно интерфейса приложения, содержащее меню выбора поставщиков сервиса

Пример создания расширяемого приложения с использованием библиотеки Lookup платформы NetBeans

Модифицируем предыдущий проект, в котором мы создавали расширяемое Java-приложение с использованием ServiceLoader API платформы Java SE 6, подключением библиотеки Lookup платформы NetBeans и соответствующим образом изменив код класса `ServiceDocDesktopView`, представляющего главное окно интерфейса настольного Java-приложения.

Примечание

Файлы с кодами приложения расположены на прилагаемом к книге компакт-диске в папке Часть_I\Глава_4\JavadocLookupSimple.

Для того чтобы воспользоваться библиотекой Lookup платформы NetBeans, необходимо добавить файл org-openide-util.jar, содержащий классы библиотеки Lookup, как библиотеку в проект. Для этого в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на узле **Libraries** проекта ServiceDocDesktop и выберем **Add JAR/Folder**. В появившемся диалоговом окне найдем файл org-openide-util.jar в папке platform\lib каталога среды NetBeans и нажмем кнопку **Open**.

Теперь модифицируем код класса `ServiceDocDesktopView`.

В качестве первого шага обеспечим употребление короткого имени класса Lookup в коде класса приложения ServiceDocDesktopView с помощью строки

```
import org.openide.util.Lookup;
```

Далее в конструкторе класса приложения ServiceDocDesktopView вместо блока автоматического добавления поставщиков сервиса в меню главного окна приложения с помощью класса `ServiceLoader` вставим блок автоматического добавления поставщиков сервиса в меню с помощью класса `Lookup` (листинг 4.4).

Листинг 4.4. Код автоматического добавления поставщиков сервиса в меню с помощью класса `Lookup`

```
Lookup lookup = Lookup.getDefault();
Lookup.Template template = new Lookup.Template(createJavadocIface.class);
Lookup.Result result = lookup.lookup(template);
collServiceProvider=result.allInstances();
initMenuDoclet();
```

В листинге 4.4 статический метод `getDefault()` класса `org.openide.util.Lookup` возвращает экземпляр класса `Lookup`, который по умолчанию обеспечивает поиск поставщиков сервиса, зарегистрированных с помощью конфигурационного файла каталога META-INF\services и JAR-файлы которых добавлены в `CLASSPATH` приложения.

Вложенный статический финальный, настраиваемый по типу, класс `Lookup.Template<T>` определяет фильтрацию поиска поставщиков сервиса по указанному типу.

Вложенный абстрактный статический класс `Lookup.Result<T>` представляет результаты поиска и обеспечивает доступ к найденным экземплярам классов поставщиков сервиса.

Метод `lookup(Lookup.Template<T> template)` класса `Lookup` возвращает объект типа `Lookup.Result<T>`, содержащий результаты поиска. Получить результаты можно с помощью метода `allInstances()` класса `Lookup.Result<T>`, который возвращает объект типа `java.util.Collection<E>`, содержащий все найденные экземпляры классов поставщиков сервиса. Данный объект является неизменяемой коллекцией, к которой неприменимы такие методы интерфейса `java.util.Collection<E>`, как `clear()`, `remove()` и др., изменяющие содержимое коллекции. В нашем случае мы будем хранить результаты поиска в кол-

лекции `private java.util.Collection collServiceProvider` — поле класса приложения ServiceDocDesktopView.

В классе `Lookup.Result<T>` существует метод `allItems()`, который возвращает неизменяемую коллекцию, содержащую объекты вложенного класса `Lookup.Item<T>`. Класс `Lookup.Item<T>` представляет элемент результата поиска, описывающий класс поставщика сервиса без создания его экземпляра. Поэтому коллекция, получаемая с помощью метода `allItems()`, в отличие от метода `allInstances()`, содержит не экземпляры классов поставщиков сервиса, а элементы, описывающие их.

В листинге 4.4 также указан вызов метода `initMenuDoclet()`, отвечающего за добавление поставщиков сервиса в меню главного окна приложения, с помощью коллекции `collServiceProvider` (листинг 4.5).

Листинг 4.5. Код метода `initMenuDoclet()`

```
private void initMenuDoclet(){
  jRadioButtonMenuItem =
    new javax.swing.JRadioButtonMenuItem[collServiceProvider.size()];
  int k=0;
  for (java.util.Iterator i = collServiceProvider.iterator();
       i.hasNext(); ){
    jRadioButtonMenuItem[k] = new javax.swing.JRadioButtonMenuItem();
    jRadioButtonMenuItem[k].setSelected(true);
    createJavadocIface inst=(createJavadocIface)i.next();
    jRadioButtonMenuItem[k].setText(inst.getClass().getName()); // NOI18N
    jMenuDoclets.add(jRadioButtonMenuItem[k]);
    k++;
  }
}
```

В коде метода `initMenuDoclet()` (см. листинг 4.5) метод `iterator()` интерфейса `Collection<E>` возвращает объект типа `java.util.Iterator<E>`, обеспечивающий доступ к элементам коллекции `collServiceProvider`. Метод `hasNext()` интерфейса `Iterator<E>` возвращает `true`, если существует следующий элемент коллекции. Метод `next()` интерфейса `Iterator<E>` возвращает следующий элемент коллекции.

После модификации блока автоматического добавления поставщиков сервиса в меню главного окна приложения, в методе `doInBackground()` внутреннего класса `CreateJavadocTask` обеспечим вызов метода `cmdJavadoc()` классов поставщиков сервиса с помощью элементов коллекции `collServiceProvider` (листинг 4.6).

Листинг 4.6. Код метода `doInBackground()`

```
@Override protected Object doInBackground() {
  // Your Task's code here.  This method runs on a background
  // thread, so don't reference the Swing GUI from here.
  int k=0;
  for (java.util.Iterator i = collServiceProvider.iterator();
       i.hasNext(); ) {
    createJavadocIface inst=(createJavadocIface)i.next();
    if (jRadioButtonMenuItem[k].isSelected()){
      String jdk=jTextFieldJdk.getText();
      String src=jTextFieldSrc.getText();
      String out=jTextFieldOut.getText();
      String pkg=jTextFieldPkg.getText();
      inst.cmdJavadoc(jdk,src,out,pkg);
    }
    k++;
  }
  return null;  // return your result
}
```

В принципе, расширяемое Java-приложение с использованием библиотеки Lookup платформы NetBeans готово, однако в данной конфигурации использование библиотеки Lookup платформы NetBeans не дает никаких видимых преимуществ по сравнению с использованием библиотеки ServiceLoader API платформы Java SE 6. В частности, нет возможности следить за изменениями в наборе доступных поставщиков сервиса. Поэтому модифицируем код класса приложения ServiceDocDesktopView таким образом, чтобы была возможность динамического добавления нового поставщика сервиса в работающее приложение.

Примечание

Файлы с кодами приложения расположены на прилагаемом к книге компакт-диске в папке Часть_I\Глава_4\JavadocLookupListener.

Для начала изменим в конструкторе класса приложения ServiceDocDesktopView код блока автоматического добавления поставщиков сервиса в меню главного окна приложения (листинг 4.7).

Листинг 4.7. Измененный код автоматического добавления поставщиков сервиса в меню с помощью класса `Lookup`

```
classLoaderForCurrentThread =
  Thread.currentThread().getContextClassLoader();
initLookupResult();
```

```
dynamicContent = new InstanceContent();
dynamicContent.add(classLoaderForCurrentThread);
Lookup dl = new AbstractLookup (dynamicContent);
resultContent =dl.lookup (new Lookup.Template(ClassLoader.class));
resultContent.addLookupListener(new LookupListener() {
  public void resultChanged(LookupEvent e) {
    initLookupResult();
  }
});
```

В измененном коде блока автоматического добавления поставщиков сервиса в меню в поле `private ClassLoader classLoaderForCurrentThread` класса приложения ServiceDocDesktopView будем хранить объект `ClassLoader`, содержащий информацию, необходимую для загрузки классов и ресурсов при выполнении кода класса `ServiceDocDesktopView`. С помощью статического метода `currentThread()` класса `java.lang.Thread` получим ссылку на объект, представляющий поток, выполняемый в данный момент. Метод `getContextClassLoader()` класса `Thread` возвратит нам требуемый контекст для данного потока, содержащий информацию о `CLASSPATH` приложения (контекст `ClassLoader` для данного потока).

Метод `initLookupResult()`, вызываемый в конструкторе класса приложения ServiceDocDesktopView, отвечает за поиск и добавление поставщиков сервиса в меню главного окна приложения (листинг 4.8).

Листинг 4.8. Код методов `initLookupResult()` и `initMenuDoclet()`

```
private void initLookupResult(){
  Lookup lookup = Lookups.metaInfServices(classLoaderForCurrentThread);
  Lookup.Template template=new Lookup.Template(createJavadocIface.class);
  Lookup.Result result= lookup.lookup(template);
  collServiceProvider=result.allInstances();
  initMenuDoclet();
}

private void initMenuDoclet(){
  jMenuDoclets.removeAll();
  jRadioButtonMenuItem =
    new javax.swing.JRadioButtonMenuItem[collServiceProvider.size()];
  int k=0;
  for (java.util.Iterator i = collServiceProvider.iterator();
       i.hasNext(); ){
    jRadioButtonMenuItem[k] = new javax.swing.JRadioButtonMenuItem();
    jRadioButtonMenuItem[k].setSelected(true);
```

```
    createJavadocIface inst=(createJavadocIface)i.next();
    jRadioButtonMenuItem[k].setText(inst.getClass().getName()); // NOI18N
    jMenuDoclets.add(jRadioButtonMenuItem[k]);
    k++;
  }
}
```

Далее обсудим код метода `initLookupResult()`, отвечающего за поиск и добавление поставщиков сервиса в меню главного окна приложения (см. листинг 4.8).

Напомним, что наша задача состоит в организации динамического добавления поставщиков сервиса в приложение. Добиться этого возможно с помощью динамического изменения списка `CLASSPATH` приложения, добавляя в него JAR-файл нового поставщика сервиса. Динамически изменяя `CLASSPATH` приложения, мы соответственно будем динамически изменять объект `classLoaderForCurrentThread`. С помощью объекта `classLoaderForCurrentThread` будет осуществляться поиск поставщиков сервиса, JAR-файлы которых добавлены в список `CLASSPATH` приложения.

Статический метод `getDefault()` класса `org.openide.util.Lookup` не позволит нам учитывать динамическое изменение объекта `classLoaderForCurrentThread`, т. к. при своем вызове он создает объект `Lookup`, использующий контекст `ClassLoader` для данного потока, который существовал при первом вызове метода `getDefault()`.

Поэтому мы обратимся к статическому методу `metaInfServices` класса `org.openide.util.lookup.Lookups`. Класс `Lookups` относится к пакету `org.openide.util.lookup`, представляющему библиотеку Lookup SPI, используемую для разработки собственного механизма регистрации и поиска поставщиков сервиса. Класс `Lookups` содержит статические методы для создания объектов Lookup в различных реализациях. В частности, метод `metaInfServices` возвращает объект `Lookup`, реализующий механизм поиска поставщиков сервиса, основанный на использовании конфигурационных файлов каталога META-INF\services\. В качестве аргумента метод `metaInfServices()` принимает контекст `ClassLoader` для данного потока.

В методе `initLookupResult()` происходит вызов метода `initMenuDoclet()` (см. листинг 4.8), который отвечает за добавление поставщиков сервиса в меню главного окна приложения с помощью коллекции `collServiceProvider`, хранящей результаты поиска. Так как мы собираемся динамически обновлять набор доступных поставщиков сервиса, то перед добавлением поставщиков сервиса в меню **Doclets** очистим его от старого набора методом `removeAll()` класса `javax.swing.JMenu`.

В конструкторе класса приложения ServiceDocDesktopView после вызова метода `initLookupResult()` создадим объект `dynamicContent` класса `org.openide.util.lookup.InstanceContent` (специальный контент), который будет хранить информацию о существующем контексте `ClassLoader`. Объект `dynamicContent` будет передаваться конструктору класса `org.openide.util.lookup.AbstractLookup` в качестве аргумента. Класс `AbstractLookup` обеспечивает хранение пар "класс — объект" для их использования объектом `Lookup`. Конструктор класса `AbstractLookup` создает объект `Lookup`, связанный с указанным контентом. Результаты поиска объектов класса `java.lang.ClassLoader`, хранящиеся в контенте `dynamicContent`, будем сохранять в объекте `resultContent` класса `Lookup.Result`. Это позволит нам следить за изменением контента `dynamicContent` с помощью метода `addLookupListener` класса `Lookup.Result`, который регистрирует переопределенный метод `resultChanged` интерфейса `org.openide.util.LookupListener`, вызываемый при изменении объекта `Lookup.Result`. Таким образом, в случае изменения контекста `ClassLoader`, связанного с контентом `dynamicContent`, будет автоматически вызываться метод `initLookupResult()`, который заново переопределит коллекцию `collServiceProvider` и меню **Doclets**.

Для динамического добавления нового поставщика сервиса создадим кнопку **Add Doclet** интерфейса приложения, воспользовавшись окном **Palette** среды NetBeans (рис. 4.3).

После создания кнопки **Add Doclet** зарегистрируем обработчик событий кнопки.

Для этого щелкнем правой кнопкой мыши на кнопке и в появившемся контекстном меню выберем **Events | Action | actionPerformed**. В результате в коде класса `ServiceDocDesktopView` будет создан метод `jButtonAddDocletActionPerformed()`, который будет вызываться в переопределенном методе `actionPerformed()` интерфейса `java.awt.event.ActionListener`, зарегистрированном для кнопки **Add Doclet** методом `addActionListener()` класса `javax.swing.AbstractButton` в методе `initComponents()` класса приложения ServiceDocDesktopView.

Далее введем код метода `jButtonAddDocletActionPerformed()`, позволяющего при нажатии кнопки **Add Doclet** выбрать в файловой системе компьютера JAR-файл нового поставщика сервиса и добавить его в список `CLASSPATH` приложения (листинг 4.9).

Листинг 4.9. Код обработки нажатия кнопки *Add Doclet*

```
private void jButtonAddDocletActionPerformed(
  java.awt.event.ActionEvent evt) {
```

```
    // TODO add your handling code here:
    JFileChooser chooser = new JFileChooser();
    chooser.setDialogTitle("Choose Doclet to add");
    int returnVal = chooser.showOpenDialog(null);
    if (returnVal == JFileChooser.APPROVE_OPTION) {
      java.io.File file=chooser.getSelectedFile();
      ClassLoader urlCl;
      try {
        urlCl = java.net.URLClassLoader.newInstance(
          new java.net.URL[]{file.toURI().toURL()},
          classLoaderForCurrentThread);
        Thread.currentThread().setContextClassLoader(urlCl);
        classLoaderForCurrentThread =
          Thread.currentThread().getContextClassLoader();
        dynamicContent.add(classLoaderForCurrentThread);
      } catch (Exception ex) {
        System.out.println(ex.toString());
      }
    }
}
```

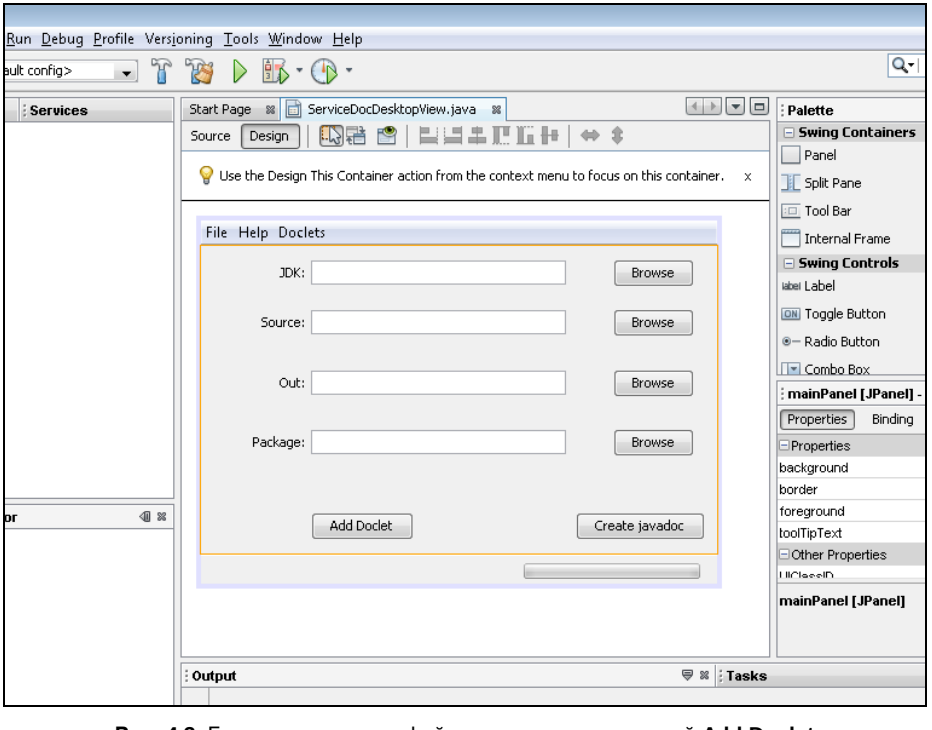

Рис. 4.3. Главное окно интерфейса приложения с кнопкой **Add Doclet**

В коде обработки нажатия кнопки **Add Doclet** (см. листинг 4.9) с помощью класса `javax.swing.JFileChooser` обеспечим возможность выбора в файловой системе JAR-файла нового поставщика сервиса и сохранения имени выбранного файла вместе с его абсолютным путем в объекте `java.io.File`. Используя статический метод `newInstance()` класса `java.net.URLClassLoader`, который является расширением класса `java.lang.ClassLoader`, создадим новый объект `ClassLoader` из текущего контекста `ClassLoader` с учетом URL-адреса выбранного JAR-файла нового поставщика сервиса. Методом `setContextClassLoader()` класса `java.lang.Thread` зарегистрируем вышеуказанный новый объект `ClassLoader` в качестве контекста `ClassLoader` для текущего потока. Добавим новый контекст `ClassLoader` в контент `dynamicContent` методом `add()` класса `org.openide.util.lookup.InstanceContent`, тем самым изменив объект `resultContent`. Изменение объекта `resultContent` спровоцирует вызов метода `initLookupResult()`, который переопределит коллекцию `collServiceProvider` и меню **Doclets**, представляющие набор доступных поставщиков сервиса.

В итоге произведенная модификация кода класса приложения ServiceDocDesktopView позволяет использовать преимущества библиотеки Lookup платформы NetBeans и динамически добавлять нового поставщика сервиса в работающее приложение (рис. 4.4).

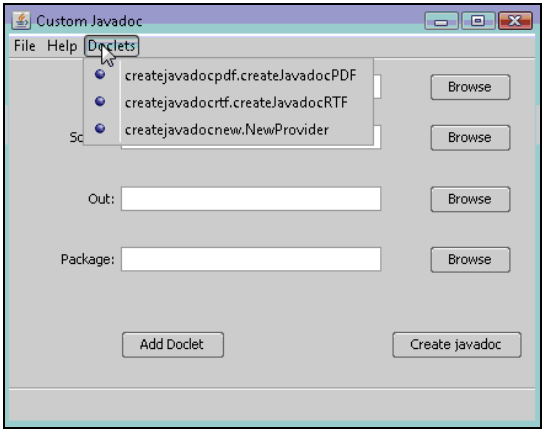

Рис. 4.4. Главное окно интерфейса приложения, позволяющего динамически добавлять поставщиков сервиса для генерации Javadoc-документации

Пример создания модуля NetBeans и модульного приложения на базе платформы NetBeans

Платформа NetBeans позволяет создавать на своей основе как модули для самой среды разработки NetBeans, так и насыщенные клиентские приложения (Rich Client Applications), использующие модули платформы NetBeans, которые обеспечивают общую инфраструктуру приложения, включая управление окнами, действиями, файлами и другие свойства, необходимые для работы приложения.

В качестве дальнейшего развития примера расширяемого Java-приложения, генерирующего Javadoc-документацию в различном формате, рассмотрим создание NetBeans-модуля, который может быть интегрирован в среду NetBeans, и создание модульного приложения на основе платформы NetBeans. При этом рассмотрим реализацию механизма поиска, основанную на регистрации поставщиков сервиса с помощью файла layer.xml модуля NetBeans.

ПРИМЕЧАНИЕ

Файлы с кодами приложения расположены на прилагаемом к книге компакт-диске в папке Часть_I\Глава_4\JavadocNBM.

Для начала работы создадим NetBeans-проект.

Для этого в меню **File** среды NetBeans выберем **New Project**. В категории **NetBeans Modules** выберем тип проекта **Module Suite**, представляющий набор взаимосвязанных модулей. Далее в поле **Project Name** введем имя проекта JavadocNBM и нажмем кнопку **Finish**. В результате в окне **Projects** среды NetBeans появится проект JavadocNBM, содержащий два узла **Modules** и **Important Files**. В узел **Modules** будут добавляться создаваемые нами модули, а в узле **Important Files** отображаются файлы, необходимые для сборки проекта.

Теперь нам нужно создать NetBeans-модули поставщиков сервиса.

Для создания первого NetBeans-модуля поставщика сервиса, генерирующего Javadoc-документацию в формате PDF, в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на узле **Modules** и выберем **Add New**. В строке **Project Name** введем имя модуля `createJavadocPDF`. Далее в строке **Code Name Base** — идентификаторе модуля введем имя пакета `createjavadocpdf`. Отметим флажок **Generate XML Layer** и нажмем кнопку **Finish**. В результате в окне **Projects** среды NetBeans появится сгенерированная основа модуля `createJavadocPDF`.

Модуль `createJavadocPDF` будет использовать интерфейс `createJavadocIface`, поэтому мы должны установить между ними связь. В окне **Projects** щелкнем правой кнопкой мыши на узле **Libraries** проекта модуля и выберем пункт **Add New Library**. В появившемся диалоговом окне нажмем кнопку **Browse**, связанную со строкой **Library**, и найдем ранее созданный нами JAR-файл интерфейса в папке dist каталога createJavadocIface. Далее в поле **Code Name Base** введем имя пакета интерфейса createjavadociface и нажмем кнопку **Finish**. В результате будет сгенерирована оболочка модуля, связывающая NetBeans-модули с используемой библиотекой.

В модуле `createJavadocPDF` создадим Java-класс поставщика сервиса. Для этого в окне **Projects** щелкнем правой кнопкой мыши на названии пакета `createjavadocpdf` и выберем **New | Java | Java Class**. В поле **Class Name** введем имя класса createJavadocPDF и нажмем кнопку **Finish.**

В окне редактора NetBeans введем разработанный нами ранее код класса `createJavadocPDF` (см. листинг 4.2).

И последним шагом создания первого NetBeans-модуля обеспечим регистрацию данного поставщика сервиса. Для этого в узле **Source Packages** модуля `createJavadocPDF` создадим папку META-INF\services, содержащую конфигурационный файл createjavadociface.createJavadocIface с именем поставщика сервиса createjavadocpdf.createJavadocPDF, и тем самым его зарегистрируем.

Таким же образом создадим NetBeans-модуль `createJavadocRTF`, отвечающий за генерацию Javadoc-документации в формате RTF. Модуль `createJavadocRTF` также будет использовать интерфейс `createJavadocIface` и содержать зарегистрированный с помощью конфигурационного файла createjavadociface.createJavadocIface каталога META-INF\services класс поставщика сервиса createjavadocrtf.createJavadocRTF. Отличие будет заключаться в способе установки связи модуля `createJavadocRTF` с интерфейсом `createJavadocIface`. Так как модуль `createJavadocIface` уже был создан, то мы установим зависимость между этими двумя модулями, щелкнув правой кнопкой мыши на узле **Libraries** проекта модуля `createJavadocRTF` и выбрав пункт **Add Module Dependency**. В появившемся диалоговом окне найдем модуль `createJavadocIface` и нажмем кнопку **OK**. Код класса поставщика сервиса createJavadocRTF ранее был уже нами разработан (см. листинг 4.3).

Классы поставщиков сервиса `createJavadocPDF` и `createJavadocRTF` используют JAR-файлы доклетов pdfdoclet.jar и RTFDoclet.jar, поэтому мы должны установить связь файлов доклетов с соответствующими модулями, используя опцию **Add New Library**. Таким образом, в нашем наборе появятся еще два модуля: `pdfdoclet` и `RTFDoclet`.

Теперь мы можем создать основной NetBeans-модуль созданного нами набора модулей JavadocNBM. Данный модуль будет использовать остальные мо-

дули набора, и, будучи интегрирован в среду NetBeans, добавит элемент **Custom Javadoc** в меню **File** среды NetBeans.

Для создания основного модуля щелкнем правой кнопкой мыши на узле **Modules** набора JavadocNBM в окне **Projects** среды NetBeans и выберем пункт **Add New**. Введем имя модуля `ServiceDoc`. Далее введем имя пакета servicedoc и нажмем кнопку **Finish**. В окне **Projects** щелкнем правой кнопкой мыши на названии пакета `servicedoc` и выберем **New | Other | Module Development | Action**. Выберем переключатель **Always Enabled** и после нажатия кнопки **Next** в появившемся окне **GUI Registration** в строке **Category** выберем **Other**. В переключателе **Global Menu Item** в строке **Menu** выберем **File** и в строке **Position** выберем **HERE-Exit**. После этого введем имя класса модуля `ServiceDoc`, а в строке **Display Name** укажем отображаемое имя элемента меню **File** среды NetBeans — **Custom Javadoc** и нажмем кнопку **Finish**.

В результате среда NetBeans сгенерирует содержимое файла layer.xml модуля `ServiceDoc`, позволяющее добавить в меню **File** среды NetBeans элемент **Custom Javadoc**, расположенный в конце списка перед элементом **Exit** и связанный с действием, которое представлено сгенерированным кодом класса `ServiceDoc`.

Файлы layer.xml NetBeans-модулей при запуске модульной системы участвуют в формировании виртуальной файловой системы NetBeans (System Filesystem). Виртуальная файловая система NetBeans формируется из трех частей, представляющих файлы на диске компьютера, файлы внутри JAR-файлов и конфигурационные данные. Файлы layer.xml NetBeans-модулей как раз формируют ту часть виртуальной файловой системы NetBeans, которая обеспечивает доступ к конфигурационным данным. Для того чтобы файл layer.xml NetBeans-модуля участвовал в формировании виртуальной файловой системы NetBeans, он должен быть зарегистрирован в файле MANIFEST.MF модуля с помощью свойства `OpenIDE-Module-Layer:`. Файл layer.xml NetBeans-модуля позволяет добавлять данные в уже существующие папки System Filesystem, а также добавлять новые папки с данными. В частности, файл layer.xml модуля `ServiceDoc` добавляет действие, представленное созданным классом `ServiceDoc`, в специальную папку Actions, которая содержит все зарегистрированные модулями действия. Кроме того, файл layer.xml модуля `ServiceDoc` добавляет в подпапку File папки Menu, формирующую меню графического интерфейса модульной системы NetBeans, ссылку на экземпляр класса `ServiceDoc`, который, используя ресурсный файл Bundle.properties модуля, определяет отображаемое имя элемента меню **Custom Javadoc**, связанного с действием, представленным этим же классом (листинг 4.10).

Листинг 4.10. Код файла layer.xml модуля `ServiceDoc`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE filesystem PUBLIC "-//NetBeans//DTD Filesystem 1.1//EN"
"http://www.netbeans.org/dtds/filesystem-1_1.dtd">
<filesystem>
  <folder name="Actions">
    <folder name="Other">
      <file name="servicedoc-ServiceDoc.instance"/>
    </folder>
  </folder>
  <folder name="Menu">
    <folder name="File">
      <file name="servicedoc-ServiceDoc.shadow">
        <attr name="originalFile"
          stringvalue="Actions/Other/servicedoc-ServiceDoc.instance"/>
        <attr name="position" intvalue="2550"/>
      </file>
    </folder>
  </folder>
</filesystem>
```

Обсудим код файла layer.xml модуля `ServiceDoc` (см. листинг 4.10).

Элемент `<filesystem>` файла layer.xml соответствует корневому каталогу System Filesystem, папки которого представлены элементами `<folder>`.

Элементы `<file>` представляют файлы, содержащиеся в папках виртуальной файловой системы System Filesystem. Атрибут `name` элемента `<file>` определяет имя папки или файла.

Дополнительные атрибуты файла указываются в элементе `<attr>`. Атрибут с именем `position` задает порядок расположения файлов в папке с помощью переменной `intvalue`. Атрибут `originalFile` определяет ссылку на файл с помощью переменной `stringvalue`. Расширение файла shadow указывает, что данный файл является ссылкой на другой файл. Расширение instance представляет экземпляр Java-класса. С помощью класса `InstanceDataObject` пакета `org.openide.loaders`, входящего в библиотеку Datasystems API платформы NetBeans, имя файла с расширением instance конвертируется в имя файла с расширением class и создается экземпляр данного класса.

Платформа NetBeans содержит три уровня абстракции для представления файловых объектов в файловой системе.

Первый уровень — это класс `org.openide.filesystems.FileObject`, относящийся к File System API. Класс `FileObject` обеспечивает возможность полу-

чения базовой информации о файловом объекте (имя, MIME-тип, расширение, местонахождение, атрибуты, представляющие дополнительные метаданные и т. д.), возможность следить за изменениями файлового объекта, возможность получения входящего и выходящего потоков, связанных с файловым объектом. Класс `FileObject` является оберткой класса `java.io.File` в случае, когда файловые объекты представляют файлы или папки файловой системы компьютера.

Следующий уровень — это класс `org.openide.loaders.DataObject`, относящийся к Datasystems API. Класс `DataObject` поддерживает механизмы Cookies и Lookup и обеспечивает программный доступ к содержимому файловых объектов.

Объект класса `DataObject` может представлять несколько объектов класса `FileObject`. Например, для редактора форм NetBeans создается объект `DataObject`, представляющий набор одноименных файлов с расширениями java, form и class. Конструктор класса `DataObject` в качестве аргументов принимает объект класса `FileObject`, представляющий основной файловый объект, в предыдущем примере — файл с расширением java, и объект класса `org.openide.loaders.DataLoader`. Класс `DataLoader` отвечает за распознавание типа файлового объекта `FileObject` и создание объекта `DataObject`, представляющего данный объект `FileObject`.

При запуске модульной системы на основе платформы NetBeans создается загрузочный пул — объект класса `org.openide.loaders.DataLoaderPool`, который обеспечивает доступ к набору зарегистрированных объектов `DataLoader`. Набор зарегистрированных объектов `DataLoader` состоит из системных объектов `DataLoader` платформы NetBeans и зарегистрированных объектов `DataLoader` в папке Loaders\mime\type\Factories с помощью файлов layer.xml модулей. Каждый зарегистрированный объект `DataLoader` представляет определенный тип файлового объекта (java, html, instance и т. д.). Для каждого типа файлов существует один `DataLoader` и для каждого файла этого типа существует один `DataObject`. Загрузочный пул сканирует файловые объекты, логически группируя их и определяя какой тип данных каждый из файловых объектов представляет. При сканировании создается запрос к зарегистрированным объектам `DataLoader` — может ли данный файловый объект быть обработан. Первый же объект `DataLoader`, который распознает файловый объект, становится его собственником и создает соответствующий объект `DataObject`, представляющий файл.

Подкласс `MultiFileLoader` класса `DataLoader` предназначен для распознавания наборов файловых объектов (как в случае примера с редактором форм NetBeans) и создания объектов класса `MultiDataObject`, который является подклассом `DataObject` и обеспечивает обработку составных объектов. Каж-

дый файловый объект `FileObject` в наборе представлен объектом класса `MultiDataObject.Entry`, который позволяет производить с файловым объектом определенные операции, такие как копирование и создание из шаблона. Объекты `MultiDataObject.Entry` создаются объектом `MultiFileLoader` с помощью методов `createPrimaryEntry()` для основного файлового объекта набора и `createSecondaryEntry()` для остальных файловых объектов набора.

Подкласс `UniFileLoader` класса `MultiFileLoader` предназначен для распознавания одного файлового объекта, создавая объект `MultiDataObject`.

Класс `DataObject` обеспечивает *cookies*. Cookie — это модель, позволяющая отделить реализацию интерфейса от объекта, с которым данная реализация связана. Таким образом, класс объекта не должен сам реализовывать интерфейс. Вместо этого класс объекта должен реализовывать интерфейс `org.openide.nodes.Node.Cookie`, входящий в Nodes API и являющийся базовым для всех cookies. Интерфейс `Node.Cookie` не имеет методов и, по сути, является маркером.

В данном контексте cookie называют как интерфейс, расширяющий `Node.Cookie`, так и класс, реализующий этот интерфейс. Класс `DataObject` реализует интерфейс `Node.Cookie`, и метод `getCookie(Class<T> c)` класса `DataObject` позволяет определить, поддерживается ли данный cookie объектом `DataObject` или нет. Здесь `Class<T>` — это Java-класс, представляющий cookie. Метод `getCookie()` возвращает cookie или `null`, если cookie не поддерживается. Получив объект cookie методом `getCookie()`, дальше можно вызывать методы cookie, которые добавляют определенную функциональность. Использование метода `getCookieSet()` класса `MultiDataObject` позволяет динамически добавлять и удалять cookies из связи с объектом. Метод `getCookieSet()` возвращает объект класса `org.openide.nodes.CookieSet`, который обеспечивает хранение и поиск cookies, а также обеспечивает уведомление об изменениях в наборе cookies. Метод `add()` класса `CookieSet` позволяет добавить cookie в набор. Метод `getCookie()` класса `CookieSet` позволяет получить cookie из набора.

Пакет `org.openide.cookies` Nodes API содержит стандартные cookies, которые обеспечивают такие операции, как закрытие, открытие, сохранение объекта и т. д.

Класс `DataObject` реализует интерфейс `org.openide.util.Lookup.Provider` библиотеки Utilities API, соответственно метод `getLookup()` класса `DataObject` возвращает объект Lookup, представляющий контент объекта `DataObject`. Поддержка классом `DataObject` механизма Lookup допускает использование `getLookup().lookup(Class<T> c)` вместо `getCookie(Class<T> c)`. Использование метода `getLookup()` позволяет получить объект класса, реализующего указанный интерфейс, даже если этот интерфейс не расширяет `Node.Cookie`,

поэтому метод `getLookup()` является более общей заменой методу `getCookie(Class<T> c)`.

В пакет `org.openide.cookies` Nodes API стандартных cookies входит интерфейс `InstanceCookie`, который предназначен для создания экземпляра класса, используя файловый объект с расширением instance, зарегистрированный в System Filesystem с помощью файла layer.xml модуля. Класс `org.openide.loaders.InstanceDataObject` расширяет класс `MultiDataObject` и реализует интерфейс `InstanceCookie`, тем самым обеспечивая создание экземпляра класса в случае распознавания файла с расширением instance. При этом файловый объект может иметь атрибуты:

- `instanceCreate` используется для определения имени статического метода класса, создающего экземпляр класса. Имя метода указывается переменной `methodvalue` атрибута;
- `instanceOf` используется для указания имени суперкласса или реализованного интерфейса/интерфейсов экземпляра класса. Использование атрибута `instanceOf` позволяет загружать суперкласс или реализованный интерфейс/интерфейсы тогда, когда это необходимо, и таким образом оптимизирует использование памяти. Класс `InstanceDataObject` реализует интерфейс `InstanceCookie.Of`, который обеспечивает поддержку атрибута `instanceOf`.

Класс `InstanceDataObject` представляет собой пример программного доступа к содержимому файловых объектов.

Третий уровень абстракции для представления файловых объектов в файловой системе — это класс `org.openide.nodes.Node` библиотеки Nodes API. Каждый объект `DataObject` файловой системы NetBeans имеет своего представителя — объект `Node` — узел, который обеспечивает его визуальное представление и снабжает пользователя необходимым интерфейсом. Получить `Node`-представителя для объекта `DataObject` можно с помощью метода `getNodeDelegate()` класса `DataObject`. Объекты класса `Node` могут представлять не только объекты `DataObject`, но и другие объекты, организованные в иерархию — например объекты окна **Debugging Breakpoints** среды NetBeans. Для представления объектов `DataObject` в Datasystems API существует класс `org.openide.loaders.DataNode`, базовым классом которого является класс `Node`.

Класс `Node` предназначен для визуального представления объекта с поддержкой иерархии, обеспечения наборов свойств объекта, включая механизм уведомления об их изменениях, а также механизма Lookup, cookies и действий. Объект `Node` имеет, помимо наборов свойств и связанных с ним действий, также отображаемое имя и иконку.

Хотя архитектура Nodes API основана на концепции JavaBeans, имеется ряд отличий:

1. В то время как архитектура JavaBeans обеспечивает иерархию с помощью пакета `java.beans.beancontext`, снабжающего компоненты Beans контейнером, который может быть вложенным, поддержка иерархии в Nodes API основана на классе `org.openide.nodes.Children`, который является фабрикой для дочерних объектов `Node`.
2. Класс `Node` не имеет конструктора по умолчанию в отличие от стандартных компонентов Beans.
3. Cookies и действия, связанные с объектом `Node`, не поддерживают интроспекцию JavaBeans.
4. Механизм сохранения и восстановления JavaBeans основан на интерфейсе `java.io.Serializable` или `java.io.Externalizable`. Nodes API для сериализации предлагает метод `getHandle()` класса `Node` и для десериализации метод `getNode()` вложенного интерфейса `Node.Handle`.
5. Nodes API обеспечивает свойства объектов `Node` на трех уровнях. Первый уровень представляет класс `Node.Property<T>` и расширяющий его класс `Node.IndexedProperty<T,E>`, которые описывают свойства, соответствующие свойствам JavaBeans. Второй уровень представляет класс `Node.PropertySet`, который оперирует одним набором свойств. Третий уровень представляет класс `Sheet`, который отвечает за создание наборов свойств. Таким образом, Nodes API предоставляет более широкую поддержку свойств объектов по сравнению с JavaBeans API. Nodes API позволяет создавать свойства объекта двумя способами. Первый способ — это использование метода `createSheet()` класса `org.openide.nodes.AbstractNode`, являющегося базовой реализацией абстрактного класса `Node`. В этом случае не поддерживается интроспекция JavaBeans, основанная на использовании класса `java.beans.Introspector` и интерфейса `java.beans.BeanInfo`, что позволяет динамически модифицировать наборы свойств объекта. Второй способ — это создание объекта класса `org.openide.nodes.BeanNode<T>`, который представляет JavaBeans-компонент как объект `Node`. В этом случае обеспечивается полное соответствие технологии JavaBeans.

Визуальное отображение объектов `Node` обеспечивает библиотека Explorer API платформы NetBeans, которая состоит из трех пакетов.

- Пакет `org.openide.explorer.view` содержит классы, представляющие графические компоненты (Explorer views), которые визуально отображают объекты `Node` различным образом. Классы `BeanTreeView`, `ContextTreeView`, `NodeTreeModel`, `TreeTableView` и `TreeView` отображают объекты `Node` в виде

дерева. Класс `ChoiceView` отображает объекты `Node` в виде раскрывающегося списка. Класс `IconView` — в виде ряда значков. Класс `MenuView` — в виде меню. Классы `ListView` и `NodeListModel` — в виде списка. Классы `NodeTableModel`, `OutlineView` и `TableView` — в виде таблицы. Классы пакета `org.openide.explorer.view` расширяют и используют классы графической библиотеки Swing.

- Пакет `org.openide.explorer.propertysheet` содержит классы, которые представляют графические компоненты, отвечающие за визуальное отображение и редактирование свойств объектов `Node`.
- Пакет `org.openide.explorer` содержит класс `ExplorerManager`, который управляет выбором объектов `Node`, с тем, чтобы в дальнейшем с выбранными узлами могли быть произведены какие-либо действия. Кроме того, класс `ExplorerManager` управляет установкой корневого для иерархии объекта `Node` и обеспечивает синхронизацию составных отображений объектов `Node` (Explorer views). Для того чтобы объекты `Node` могли быть отображены, Explorer views должны быть добавлены в AWT-/Swing-контейнер верхнего уровня, который реализует интерфейс `ExplorerManager.Provider`. Explorer views могут также добавляться в контейнер, который является дочерним по отношению к контейнеру, реализующему интерфейс `ExplorerManager.Provider`. Вложенный интерфейс `ExplorerManager.Provider` содержит один метод `getExplorerManager`, возвращающий объект `ExplorerManager`. Таким образом, когда Explorer views добавляются в контейнер, они осуществляют поиск контейнера, реализующего интерфейс `ExplorerManager.Provider` и устанавливающего для них объект `ExplorerManager`. Nodes API содержит абстрактный класс `org.openide.nodes.NodeOperation`, снабжающий общими операциями для объектов `Node`. Метод `explore(Node n)` класса `NodeOperation` обеспечивает отображение указанного объекта `Node` и его подиерархии в новом Explorer view. Модуль Explorer & Property Sheet API платформы NetBeans содержит класс `org.netbeans.modules.openide.explorer.NodeOperationImpl`, который расширяет класс `NodeOperation` и обеспечивает реализацию общих операций для объектов `Node` по умолчанию. Это позволяет отображать иерархию узлов `Node` вызовом метода `explore()`, при этом будет создано немодальное диалоговое окно, содержащее панель — объект класса, расширяющего класс `JPanel` и реализующего интерфейс `ExplorerManager.Provider`. В данную панель будет добавлена панель с прокруткой, содержащая объект класса `org.openide.explorer.view.BeanTreeView`, который отображает иерархию узлов в виде дерева.

После обсуждения кода файла layer.xml модуля `ServiceDoc` вернемся к нашему модульному проекту JavadocNBM.

При создании действия, связанного с новым элементом меню **Custom Javadoc**, средой NetBeans был сгенерирован код класса `ServiceDoc` одноименного модуля, входящего в набор JavadocNBM (листинг 4.11).

Листинг 4.11. Сгенерированный код класса `ServiceDoc`

```
package servicedoc;
import org.openide.util.HelpCtx;
import org.openide.util.NbBundle;
import org.openide.util.actions.CallableSystemAction;
public final class ServiceDoc extends CallableSystemAction {
  public void performAction() {
    // TODO implement action body
  }
  public String getName() {
    return NbBundle.getMessage(ServiceDoc.class, "CTL_ServiceDoc");
  }
  @Override
  protected void initialize() {
    super.initialize();
    // see org.openide.util.actions.SystemAction.iconResource()
    // Javadoc for more details
    putValue("noIconInMenu", Boolean.TRUE);
  }
  public HelpCtx getHelpCtx() {
    return HelpCtx.DEFAULT_HELP;
  }
  @Override
  protected boolean asynchronous() {
    return false;
  }
}
```

Обсудим сгенерированный средой NetBeans код класса `ServiceDoc` (см. листинг 4.11).

В коде класса `ServiceDoc` используется пакет `org.openide.util.actions` библиотеки Utilities API платформы NetBeans, который содержит абстрактные классы и интерфейсы, обеспечивающие различные типы стандартных действий, включая их визуальное представление.

Далее для общего понимания рассмотрим структуру пакета `org.openide.util.actions`.

Класс `SystemAction` пакета `org.openide.util.actions` является базовым для всех NetBeans-реализаций действий графического интерфейса пользователя. Класс `SystemAction` расширяет абстрактный класс `org.openide.util.SharedClassObject` библиотеки Utilities API, основной задачей которого является обеспечение доступа к общим данным для различных экземпляров одного класса, расширяющего класс `SharedClassObject`. Общие данные при этом хранятся в виде пар "имя — значение" в хэш-таблице — объекте класса `java.util.HashMap<K,V>`. Абстрактный класс `SystemAction` реализует интерфейс `javax.swing.Action`, тем самым представляя действия. Статический метод `get(Class<T> actionClass)` класса `SystemAction` возвращает синглетон класса, расширяющего класс `SystemAction`, т. е. экземпляр класса, существующий в единственном числе. Так как класс `SystemAction` расширяет класс `SharedClassObject`, то для каждого синглетона обеспечивается хранение свойств, таких как имя, значок, текущее состояние действия (активировано/дезактивировано). Реализация здесь шаблона программирования Singleton предотвращает возможные конфликты, т. к. каждое действие графического интерфейса управляется только одним экземпляром класса, который представляет данное действие. Это позволяет специфицировать действие именем класса, который его представляет.

Абстрактный класс `org.openide.util.actions.BooleanStateAction` расширяет `SystemAction` и представляет тип действия, состояние которого переключается при изменении значения статического поля класса `PROP_BOOLEAN_STATE`.

Абстрактный класс `org.openide.util.actions.CallableSystemAction`, расширяющий `SystemAction`, предоставляет метод `performAction()`, который асинхронно выполняет действие в отдельном потоке.

Абстрактный класс `org.openide.util.actions.CallbackSystemAction` расширяет класс `CallableSystemAction` и обеспечивает выполнение различных действий, выбор которых зависит от того, какой графический компонент находится в фокусе.

Вложенные интерфейсы `Presenter.Menu`, `Presenter.Popup` и `Presenter.Toolbar` пакета `org.openide.util.actions` позволяют действиям обеспечить свое визуальное представление соответственно в меню, всплывающем меню и панели инструментов.

Библиотека Actions API платформы NetBeans содержит различные варианты расширений абстрактных классов и интерфейсов пакета `org.openide.util.actions`.

В коде класса `ServiceDoc` также используются классы `org.openide.util.NbBundle` и `org.openide.util.HelpCtx` библиотеки Utilities API платформы NetBeans.

Класс `NbBundle` библиотеки Utilities API является аналогом класса `java.util.ResourceBundle` платформы Java SE и предназначен для загрузки локализованных ресурсов.

Класс `HelpCtx` библиотеки Utilities API позволяет связать свойства модульной системы NetBeans с файлами HTML-документации, которые представляют справочную систему JavaHelp *(см. приложение 8)*.

Класс `ServiceDoc` расширяет абстрактный класс `CallableSystemAction`, который представляет действия платформы NetBeans, вызываемые программным образом. Это означает, что при выборе созданного элемента меню **Custom Javadoc** будет вызван обработчик событий — переопределенный метод `performAction()` класса `ServiceDoc`. Мы хотим, чтобы при этом появилось окно графического интерфейса, позволяющее создавать Javadoc-документацию. Поэтому создадим в модуле `ServiceDoc` класс `ServiceDocJDialog`, представляющий диалоговое окно графического интерфейса, и вызовем в методе `performAction()` статический метод `main()` класса `ServiceDocJDialog`, создающий и визуализирующий диалоговое окно (листинг 4.12).

Листинг 4.12. Код метода `performAction()` класса `ServiceDoc`

```
public void performAction() {
  // TODO implement action body
  ServiceDocJDialog.main(null);
}
```

Для создания окна графического интерфейса, позволяющего генерировать Javadoc-документацию, в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на названии пакета `servicedoc` и в появившемся контекстном меню выберем **New | Other | Categories: Swing GUI Forms | JDialog Form**. Введем имя класса `ServiceDocJDialog` и нажмем кнопку **Finish**.

В окне редактора форм среды NetBeans щелкнем на кнопке **Design**, переключимся таким образом в режим разработки диалогового окна, представленного классом `ServiceDocJDialog`, и воспользуемся палитрой графических компонентов **Palette** для того, чтобы добавить все необходимые компоненты графического интерфейса (рис. 4.5).

Далее создадим и свяжем соответствующие действия с кнопками **Browse**, **Create Javadoc** и **Add Doclet**. В результате получим законченный код класса `ServiceDocJDialog`, который будет создавать диалоговое окно, позволяющее пользователю:

- указать каталог JDK, содержащую инструмент `javadoc`;
- определить каталог с папкой пакета исходных Java-файлов, содержащих Javadoc-комментарии;

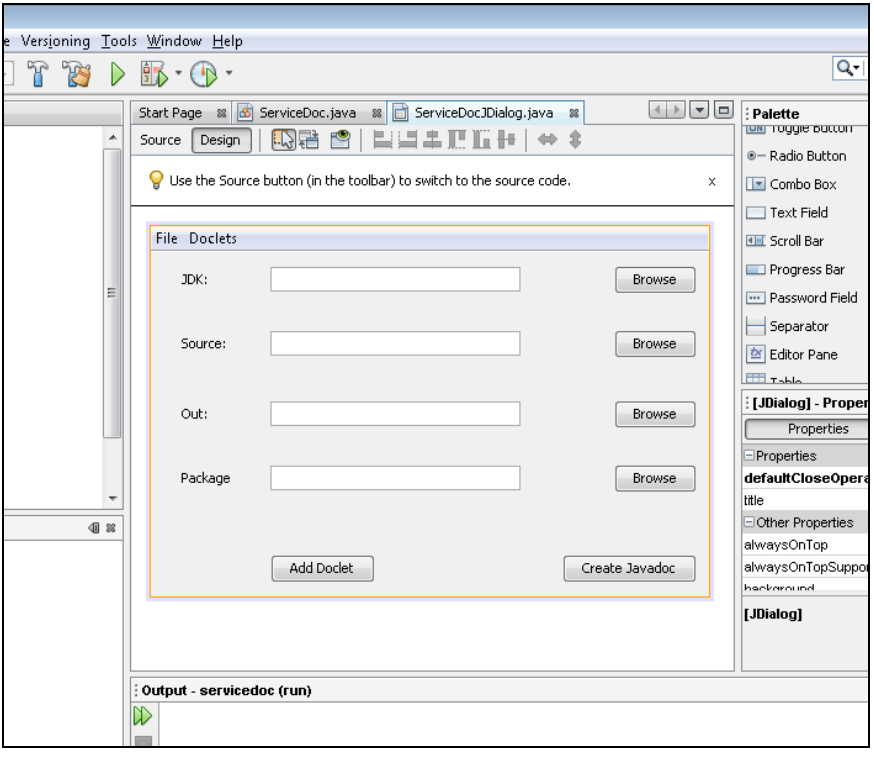

Рис. 4.5. Дизайн окна графического интерфейса, позволяющего генерировать Javadoc-документацию

- указать каталог, в котором будут создаваться файлы документации;
- указать имя пакета Java-классов, для которого будет сгенерирована документация.

> ***Примечание***
>
> Окончательный код класса `ServiceDocJDialog` см. в файле ServiceDocJDialog.java папки Часть_I\Глава_4\JavadocNBM\ServiceDoc\src\servicedoc на прилагаемом к книге компакт-диске.

Меню **Doclets** будет отображать набор доступных поставщиков сервиса, а также давать возможность активировать и деактивировать их. Кнопка **Create Javadoc** будет запускать процесс генерации Javadoc-документации, а кнопка **Add Doclet** позволит динамически добавить поставщика сервиса в набор.

Класс `ServiceDocJDialog` будет использовать сервис `createJavadocIface`. Поэтому необходимо установить связь модуля `ServiceDoc` с модулем сервиса `createJavadocIface`. Для этого щелкнем правой кнопкой мыши на узле **Libraries** модуля `ServiceDoc` и выберем пункт **Add Module Dependency**.

В появившемся диалоговом окне найдем модуль createJavadocIface и нажмем кнопку **OK**.

Разберем код класса `ServiceDocJDialog`.

Класс `JFileChooser` понадобится в коде класса `ServiceDocJDialog` для организации навигации по файловой системе компьютера и выбора необходимых каталогов. Пакеты `org.openide.util` и `org.openide.util.lookup` предоставят возможность воспользоваться механизмом Lookup платформы NetBeans.

В конструкторе класса `ServiceDocJDialog` сначала, используя ключевое слово `super`, вызывается конструктор класса `JDialog`, создающий диалоговое окно без заголовка и с указанной модальностью в родительском окне. Затем вызывается метод `initComponents()`, инициализирующий компоненты диалогового окна. Методом `setLocationRelativeTo`, унаследованным от класса `java.awt.Window`, диалоговое окно устанавливается в центр по отношению к родительскому окну.

В поле `classLoaderForCurrentThread` класса `ServiceDocJDialog` записывается объект `ClassLoader`, обеспечивающий загрузку классов и ресурсов при выполнении кода класса `ServiceDocJDialog`. Далее вызывается метод `initLookupResult()`, в котором коллекция `collServiceProvider` заполняется экземплярами поставщиков сервиса `createJavadocIface`, добавляя их в качестве элементов меню **Doclets** диалогового окна графического интерфейса.

После этого в конструкторе класса `ServiceDocJDialog` создается объект `dynamicContent` класса `org.openide.util.lookup.InstanceContent`, хранящий информацию о существующем контексте `ClassLoader`. Объект `resultContent` класса `Lookup.Result` позволит следить за изменением контекста `ClassLoader` и в случае его изменения вызывать метод `initLookupResult()`, который заново переопределит коллекцию `collServiceProvider` и меню **Doclets**.

Как уже было сказано, код метода `initComponents()` инициализирует графические компоненты диалогового окна, включая их взаимное расположение, и определяет обработчики событий для элементов контроля.

Загрузка ресурсов в методе `initComponents()` производится с помощью статического метода `getMessage()` класса `org.openide.util.NbBundle`, который ищет ресурс с указанным именем в файле Bundle.properties того же пакета, что и указанный класс.

В коде класса `ServiceDocJDialog` после метода `initComponents()` идет определение метода `jMenuItemExitActionPerformed()`, являющегося обработчиком событий, связанных с элементом **Exit** меню **File** диалогового окна. В методе `jMenuItemExitActionPerformed()` происходит вызов метода `dispose()`, унаследованного от класса `java.awt.Window` и освобождающего все задействованные ресурсы оперативной памяти при закрытии диалогового окна.

С кнопкой **Browse** текстового поля **JDK** связан обработчик событий `jButtonJDKActionPerformed`, который открывает окно навигации файловой системы компьютера, приглашая выбрать каталог JDK, и записывает ее путь в поле `jTextFieldJDK`.

С кнопкой **Browse** текстового поля **Source** связан обработчик событий `jButtonSourceActionPerformed`, открывающий окно навигации файловой системы компьютера, приглашая выбрать каталог, содержащий исходные файлы с Javadoc-комментариями, и записывает ее путь в поле `jTextFieldSource`.

С кнопкой **Browse** текстового поля **Out** связан обработчик событий `jButtonOutActionPerformed`, который открывает окно навигации файловой системы компьютера, приглашая выбрать каталог, в котором будут сохраняться созданные файлы Javadoc-документации, и записывает ее путь в поле `jTextFieldOut`.

С кнопкой **Browse** текстового поля **Package** связан обработчик событий `jButtonPackageActionPerformed`, также открывающий окно навигации файловой системы компьютера, приглашая выбрать папку пакета, для которого создается Javadoc-документация, и записывает имя пакета в поле `jTextFieldPackage`.

С кнопкой **Add Doclet** связан обработчик событий `jButtonAddDocletAction-Performed`, который открывает окно навигации файловой системы компьютера, приглашая выбрать JAR-файл нового поставщика сервиса `createJavadocIface`, и с его учетом создает новый контекст `ClassLoader`, изменяя контент `dynamicContent` и тем самым инициируя вызов метода `initLookupResult()` с переопределением коллекции `collServiceProvider` и меню **Doclets**.

С кнопкой **Create Javadoc** связан обработчик событий `jButtonCreateJavadocActionPerformed`, в котором в отдельном потоке последовательно вызывается переопределенный метод `cmdJavadoc()` активированных поставщиков сервиса `createJavadocIface` и, соответственно, создаются различные варианты Javadoc-документации.

Платформа NetBeans предлагает свой класс `org.openide.util.RequestProcessor` для выполнения задачи в отдельном потоке. С помощью статического метода `getDefault()` класса `RequestProcessor` можно получить общий объект `RequestProcessor`, способный выполнять до 50 задач параллельно. Метод `post(Runnable run)` класса `RequestProcessor` без отсрочки запускает переопределенный метод `run()` экземпляра класса, реализующего интерфейс `java.lang.Runnable`. Поэтому в методе `jButtonCreateJavadoc-ActionPerformed()` создадим экземпляр анонимного класса, реализующего интерфейс `Runnable`, и переопределим метод `run()`, последовательно извлекая в нем из коллекции `collServiceProvider` экземпляры классов поставщиков

сервиса `createJavadocIface` и вызывая метод `cmdJavadoc()` активированных поставщиков сервиса для генерации Javadoc-документации. Затем созданный экземпляр анонимного класса передадим в качестве аргумента методу `post` класса `RequestProcessor`.

Изначально среда NetBeans сгенерировала код статического метода `main()` класса `ServiceDocJDialog`, который создает экземпляр класса `ServiceDocJDialog` и делает диалоговое окно видимым.

Здесь используется статический метод `invokeLater()` класса `java.awt.EventQueue`, вызывающий переопределенный метод `run()` класса, реализующего интерфейс `Runnable`, в потоке распределения событий, в котором обеспечивается работа графического интерфейса. Таким образом, гарантируется создание диалогового окна в потоке распределения событий (Event Dispatch Thread). Напомним, что интерфейс `Runnable` должен реализовываться классом, чьи экземпляры предполагают осуществлять свою работу в отдельном потоке.

В коде метода `main()`, сгенерированном средой NetBeans, в переопределенном методе `run()`, с помощью метода `addWindowListener()`, унаследованного от класса `java.awt.Window`, с созданным экземпляром диалогового окна связывается обработчик событий — метод `windowClosing()` класса `java.awt.event.WindowAdapter`, вызываемый в процессе закрытия окна. Переопределением метода `windowClosing()` в нем вызывается статический метод `exit()` класса `java.lang.System`, который завершает текущую работу виртуальной машины Java. Это входит в противоречие с вызовом метода `setDefaultCloseOperation()` класса `JDialog` в методе `initComponents()` класса `ServiceDocJDialog`, т. к. метод `setDefaultCloseOperation(javax.swing.WindowConstants.DISPOSE_ON_CLOSE)` определяет операцию `dispose()` при закрытии диалогового окна, которая освобождает все задействованные ресурсы операционной системы. Поэтому удалим эту часть кода из метода `main()`.

Далее в классе `ServiceDocJDialog` объявляются его поля.

Вернемся к рассмотрению класса `ServiceDoc` одноименного модуля набора JavadocNBM (см. листинг 4.11).

Класс `ServiceDoc` переопределяет метод `getName()`, унаследованный от класса `org.openide.util.actions.SystemAction` и возвращающий отображаемое название действия. Для того чтобы получить название действия из файла Bundle.properties модуля, здесь используется статический метод `getMessage()` класса `org.openide.util.NbBundle`.

Класс `ServiceDoc` также переопределяет методы `initialize()` класса `SystemAction`, `asynchronous()` класса `CallableSystemAction` и `getHelpCtx()` класса `SystemAction`.

Метод `initialize()` класса `SystemAction` переопределяет метод `initialize()` класса `org.openide.util.SharedClassObject`, активируя действие. Переопределенный метод `initialize()` класса `ServiceDoc` дополнительно с помощью метода `putValue()` класса `SystemAction` определяет отсутствие значка для действия.

Переопределенный метод `asynchronous()` класса `ServiceDoc` возвращает `false`. Это означает, что действие будет выполняться синхронно в потоке распределения событий (Event Dispatch Thread).

Переопределенный метод `getHelpCtx()` класса `ServiceDoc` возвращает значение статического поля `DEFAULT_HELP` класса `org.openide.util.HelpCtx` — объект `HelpCtx`, тем самым связывая действие с идентификатором файла HTML-справки JavaHelp-системы NetBeans, представляющего информацию о том, что данное свойство не имеет своей справки.

Хотелось бы, чтобы созданный элемент меню **Custom Javadoc** был оснащен справкой, поэтому создадим контекстную JavaHelp-справку для нашего действия.

Для этого щелкнем правой кнопкой мыши в окне **Projects** среды NetBeans на названии пакета `servicedoc` модуля `ServiceDoc` и выберем **New | Other | Categories: Module Development | JavaHelp Help Set**. В результате среда NetBeans создаст в каталоге проекта ServiceDoc папку javahelp\servicedoc\docs с файлами набора HelpSet, а в папке пакета src\servicedoc — файл servicedoc-helpset.xml, содержащий URL-адрес файла HelpSet. Кроме того, файл layer.xml модуля `ServiceDoc` дополнится регистрацией набора HelpSet в файловой системе NetBeans (листинг 4.13).

Листинг 4.13. Регистрация набора HelpSet в файловой системе NetBeans в файле layer.xml модуля `ServiceDoc`

```
<folder name="Services">
  <folder name="JavaHelp">
    <file name="servicedoc-helpset.xml" url="servicedoc-helpset.xml">
      <attr name="position" intvalue="3675"/>
    </file>
  </folder>
</folder>
```

Далее изменим код метода `getHelpCtx()` класса `ServiceDoc` для того, чтобы связать элемент меню **Custom Javadoc** с идентификатором `servicedoc.about` сгенерированного файла servicedoc-about.html (листинг 4.14).

Листинг 4.14. Измененный код метода `getHelpCtx()` класса `ServiceDoc`

```
@Override
public HelpCtx getHelpCtx() {
  return new HelpCtx("servicedoc.about");
}
```

Теперь, при выборе элемента меню **Custom Javadoc**, если нажать клавишу <F1>, то в появившемся окне JavaHelp-системы в области отображения справки будет показано содержимое файла servicedoc-about.html.

Осуществим сборку всего проекта. Для этого щелкнем правой кнопкой мыши в окне **Projects** на названии набора модулей JavadocNBM и выберем пункт **Build All**. В результате в папке build\cluster\modules каталога проекта JavadocNBM будут созданы JAR-файлы модулей набора, а также папка ext, содержащая JAR-файлы библиотечных модулей, и папка docs, содержащая JAR-файл набора HelpSet JavaHelp-справки модуля `servicedoc`.

Чтобы проверить работоспособность созданного набора модулей, щелкнем правой кнопкой мыши в окне **Projects** на названии набора модулей JavadocNBM и выберем пункт **Run**. В результате будет создан еще один экземпляр среды NetBeans с временно инсталлированным в нее набором модулей JavadocNBM, который добавит в меню **File** элемент **Custom Javadoc** (рис. 4.6).

Для того чтобы инсталлировать созданный набор модулей в среду NetBeans на постоянной основе, необходимо создать набор NBM-файлов модулей и воспользоваться Plugins Manager, который обеспечивает динамическое управление модульной системой NetBeans.

NBM-файл — это ZIP-архив со специальным расширением nbm, которое распознает Plugins Manager. Отличие NBM-файла от JAR-файла заключается в том, что NBM-файл может содержать несколько JAR-файлов и включает в себя дополнительные метаданные, необходимые для Plugins Manager среды NetBeans.

Создадим NBM-файлы модулей.

Для этого щелкнем правой кнопкой мыши в окне **Projects** на названии набора модулей JavadocNBM и выберем пункт **Create NBMs**. В результате в папке build\updates каталога проекта будет создан набор NBM-файлов модулей.

Для инсталляции набора модулей в среду NetBeans в меню **Tools** выберем пункт **Plugins**. В появившемся диалоговом окне **Plugins Manager** нажмем кнопку **Downloaded**, а затем **Add Plugins**. В появившемся окне навигации файловой системы компьютера найдем папку build\updates каталога проекта JavadocNBM и последовательно добавим модули в окно **Plugins Manager**.

После этого нажмем кнопку **Install** и далее будем следовать инструкциям Plugins Manager. В итоге созданный нами набор модулей будет инсталлирован в среду NetBeans и в меню **File** будет добавлен элемент **Custom Javadoc**.

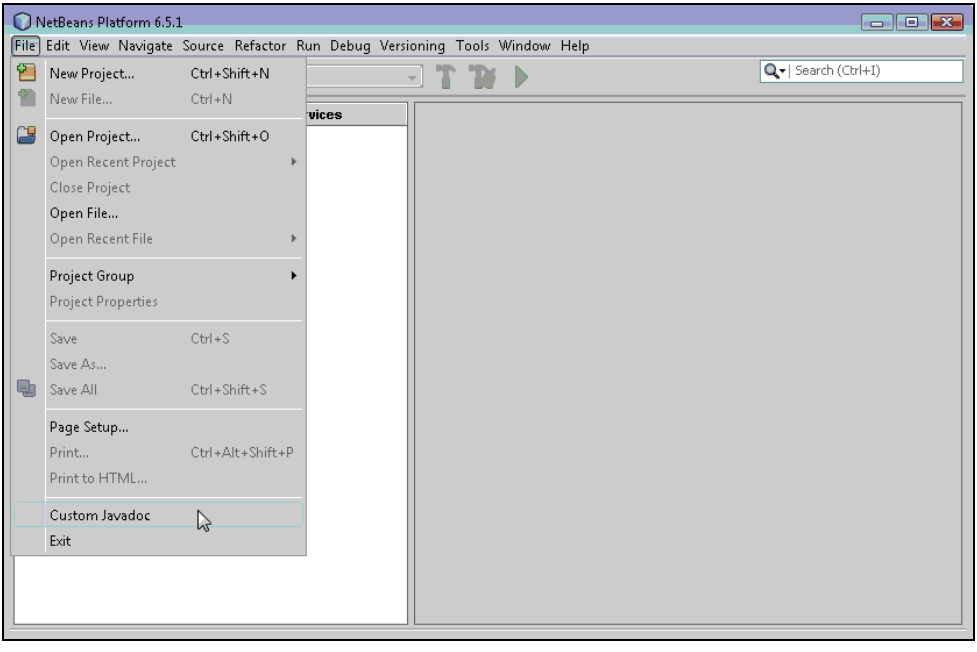

Рис. 4.6. Интерфейс среды NetBeans с новым элементом меню **Custom Javadoc**

Из набора модулей JavadocNBM можно также создать насыщенное клиентское приложение на основе платформы NetBeans.

> ***Примечание***
>
> Файлы с кодами приложения расположены на прилагаемом к книге компакт-диске в папке Часть_I\Глава_4\JavadocNBMLayer.

Для создания насыщенного клиентского приложения щелкнем правой кнопкой мыши в окне **Projects** на названии набора модулей JavadocNBM и выберем **Properties**. В появившемся диалоговом окне определения свойств проекта выберем категорию **Libraries**. В окне **Clusters and Modules** сбросим все флажки **Included** за исключением кластера модулей **platform**. Тем самым мы включим в наше приложение только модули платформы NetBeans. В категории **Build** выберем переключатель **Create Standalone Application** и заполним поля **Branding Name** и **Application Title** (javadocnbm и Custom Javadoc). В категории **Splash Screen** выберем с помощью кнопки **Browse** другую заставку и нажмем кнопку **OK**. После этого снова щелкнем правой кнопкой

мыши в окне **Projects** на названии набора модулей JavadocNBM и выберем пункт **Build ZIP Distribution**. В результате в папке dist каталога проекта JavadocNBM будет создан ZIP-архив с насыщенным клиентским приложением. Извлечем файлы приложения из архива и в папке bin найдем исполняемый файл javadocnbm, запустив который мы увидим окно графического интерфейса созданного приложения.

До сих пор мы рассматривали механизм регистрации поставщиков сервиса, основанный на использовании конфигурационного файла каталога META-INF\services. Библиотека Lookup API платформы NetBeans позволяет реализовать механизм поиска, основанный на регистрации поставщиков сервиса с помощью файла layer.xml модуля NetBeans.

Реализуем механизм регистрации поставщиков сервиса с помощью файлов layer.xml NetBeans-модулей.

Для этого в модуле поставщика сервиса `createJavadocPDF` откроем файл layer.xml и добавим в файловую систему NetBeans папку ServiceProvider, содержащую файл с расширением instance, представляющий экземпляр класса `createJavadocPDF` (листинг 4.15).

Листинг 4.15. Код файла layer.xml модуля `createJavadocPDF`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE filesystem PUBLIC "-//NetBeans//DTD Filesystem 1.1//EN"
    "http://www.netbeans.org/dtds/filesystem-1_1.dtd">
<filesystem>
  <folder name="ServiceProvider">
    <file name="createjavadocpdf-createJavadocPDF.instance"/>
    <attr name="instanceOf"
          stringvalue="createjavadociface.createJavadocIface"/>
  </folder>
</filesystem>
```

То же самое проделаем с поставщиком сервиса `createJavadocRTF` (листинг 4.16).

Листинг 4.16. Код файла layer.xml модуля `createJavadocRTF`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE filesystem PUBLIC "-//NetBeans//DTD Filesystem 1.1//EN"
    "http://www.netbeans.org/dtds/filesystem-1_1.dtd">
<filesystem>
  <folder name="ServiceProvider">
```

```
    <file name="createjavadocrtf-createJavadocRTF.instance"/>
    <attr name="instanceOf"
          stringvalue="createjavadociface.createJavadocIface"/>
  </folder>
</filesystem>
```

Теперь для получения объекта `Lookup` мы можем воспользоваться статическим методом `forPath` класса `org.openide.util.lookup.Lookups`, который создает объект `Lookup`, связанный с указанным путем. Если бы мы зарегистрировали поставщиков сервиса в папке Services файла layer.xml, то метод `metaInfServices()` класса `Lookups`, которым мы пользовались ранее, также успешно бы работал, т. к. это было бы равносильно регистрации поставщиков сервиса с помощью конфигурационного файла из каталога META-INF\services.

Регистрация поставщиков сервиса с помощью файла layer.xml модуля позволяет динамически расширять набор поставщиков сервиса, используя Plugins Manager. Поэтому кнопка **Add Doclet** диалогового окна в данной реализации не нужна. Для расширения возможностей модульного набора достаточно создать модуль нового поставщика сервиса с его регистрацией в файле layer.xml и инсталлировать его с помощью Plugins Manager. В результате набор доступных поставщиков сервиса динамически расширится.

Поэтому уберем кнопку **Add Doclet** и модифицируем конструктор и метод `initLookupResult()` класса `ServiceDocJDialog` (листинг 4.17).

Листинг 4.17. Измененный код конструктора и метода `initLookupResult()` класса `ServiceDocJDialog`

```
public ServiceDocJDialog(java.awt.Frame parent, boolean modal) {
  super(parent, modal);
  initComponents();
  setLocationRelativeTo(parent);
  initLookupResult();
}
private void initLookupResult(){
  Lookup lookup = Lookups.forPath("ServiceProvider");
  Lookup.Template template=new Lookup.Template(createJavadocIface.class);
  Lookup.Result result= lookup.lookup(template);
  collServiceProvider=result.allInstances();
  initMenuDoclet();
}
```

Остальной код останется без изменения. В данной реализации нет никакой необходимости в папках META-INF\services модулей.

Таким образом, мы создали NetBeans-модуль, способный интегрироваться в среду NetBeans и соответственно расширять ее функциональные возможности, и модульное Java-приложение на базе платформы NetBeans, расширяемое за счет подключения новых модулей.

Пример создания OSGi-приложения

В марте 1999 г. была основана организация OSGi (Open Service Gateway initiative) Alliance, членами которой являются более 30 крупных корпораций, таких как IBM, Nokia, Oracle, Intel и др. Основная задача данной организации — создание рынка межплатформенного программного обеспечения. С этой целью OSGi Alliance поддерживает широкое распространение технологии OSGi (OSGi Service Platform) — набора спецификаций, описывающих динамическую модульную Java-систему.

OSGi-спецификации описывают модель, в которой приложения могут динамически формироваться из различных компонентов, взаимодействующих друг с другом посредством сервисов.

Подробно архитектуру технологии OSGi можно посмотреть в *приложении 9*.

Основные три реализации стандарта OSGi — это проекты Apache Felix, Knopflerfish и Equinox. В нашем примере создания модульного приложения мы будем использовать проект Knopflerfish.

Домашняя страничка проекта Knopflerfish находится по адресу **http://www.knopflerfish.org/**.

Для инсталляции платформы Knopflerfish скачаем и запустим исполняемый JAR-файл knopflerfish_osgi_x.jar, используя страничку по адресу **http://www.knopflerfish.org/download.html**.

В результате в файловой системе компьютера будет создан каталог, содержащий файлы платформы Knopflerfish.

Проект Knopflerfish обеспечивает графический интерфейс Knopflerfish OSGi desktop, запустить который можно, щелкнув на исполняемом файле framework.jar, из каталога платформы Knopflerfish (рис. 4.7).

Графический интерфейс Knopflerfish OSGi desktop сам является bundle и позволяет управлять платформой Knopflerfish, включая управление жизненным циклом OSGi-компонентов.

Чтобы создать OSGi-приложение, позволяющее генерировать Javadoc-документацию в различных форматах, нам необходимо модифицировать соз-

данные нами в предыдущих разделах модули в набор OSGi-компонентов, затем инсталлировать и запустить их, используя Knopflerfish OSGi desktop.

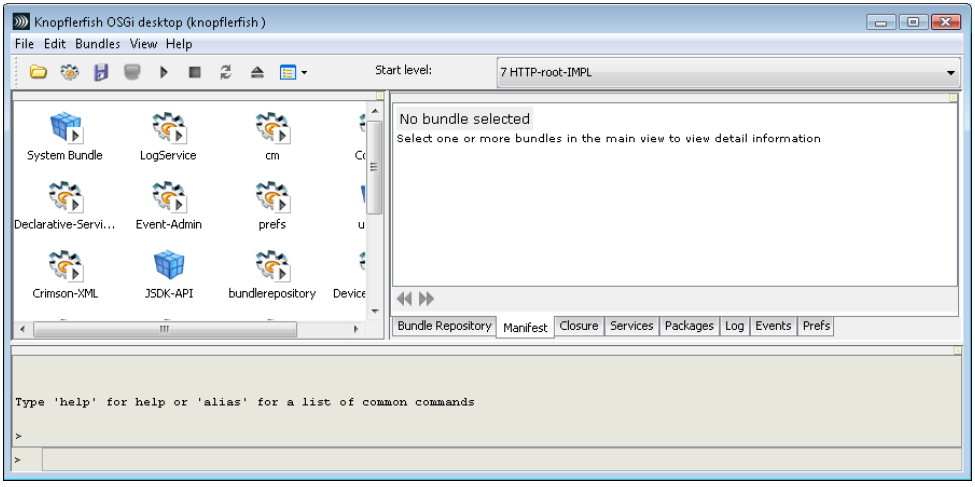

Рис. 4.7. Графический интерфейс Knopflerfish

Примечание

Файлы с кодами приложения расположены на прилагаемом к книге компакт-диске в папке Часть_I\Глава_4\JavadocOSGi.

Из предыдущих разделов у нас есть сервис, представленный интерфейсом `createJavadocIface`, поставщики сервиса — классы `createJavadocRTF`, `createJavadocPDF` и потребитель сервиса `ServiceDocDesktop`.

Для начала работы создадим каталог JavadocOSGi, который будет содержать созданные нами OSGi-компоненты. Начнем с создания bundle — поставщика сервиса `createJavadocRTF`.

Для создания OSGi-компонента, отвечающего за генерацию Javadoc-документации в формате RTF, в каталоге JavadocOSGi создадим папку createJavadocRTF.

Среда NetBeans не поддерживает создание типового проекта OSGi-компонента, поэтому воспользуемся шаблоном **Java Free-Form Project**. Для сборки и запуска проектов среда NetBeans по умолчанию использует Java-инструмент Apache Ant. Инструмент Apache Ant получает инструкции для выполнения из Ant-скрипта — XML-файла build.xml. Для создания Java-проекта свободной формы в среде NetBeans требуется написание собственного Ant-скрипта. Поэтому в папке createJavadocRTF создадим файл build.xml (листинг 4.18).

Листинг 4.18. Код файла build.xml проекта createJavadocRTF

```
<?xml version="1.0"?>
<project name="createJavadocRTF" >
<property name="framework.jar" location="framework/framework.jar"/>
  <target name="all" depends="init,compile,jar"/>
  <target name="init">
    <mkdir dir="build/classes"/>
    <mkdir dir="dist/lib"/>
    <copy todir="dist/lib">
      <fileset dir="lib"/>
    </copy>
  </target>
  <target name="compile">
    <javac destdir = "build/classes"
           debug = "on"
           srcdir = "src">
      <classpath>
        <pathelement location="${framework.jar}"/>
      </classpath>
    </javac>
  </target>
  <target name="jar">
    <jar jarfile  = "dist/${ant.project.name}.jar"
         compress = "true"
         manifest = "manifest.mf">
      <fileset dir="build/classes"/>
      <fileset dir="resources"/>
    </jar>
  </target>
  <target name="clean">
    <delete dir = "build"/>
    <delete dir = "dist"/>
  </target>
</project>
```

Обсудим код файла build.xml (см. листинг 4.18).

Каждый Ant-скрипт должен содержать один проект, ограниченный тегами `<project></project>` и, по крайней мере, одну задачу для исполнения `target`.

С помощью тега `<property>` устанавливаются свойства проекта. Здесь свойство с именем `"framework.jar"` определяет относительный путь JAR-файла платформы Knopflerfish, который мы скопируем и поместим в созданную нами папку framework каталога проекта createJavadocRTF.

Задача с именем "all" связана с задачами "init", "compile" и "jar" с помощью атрибута depends.

Задача "init" создает папки build\classes и dist\lib в каталоге проекта, а также копирует содержимое папки lib в созданную папку dist\lib. В папке lib мы будем хранить файл доклета RTFDoclet.jar.

Задача "compile" компилирует с отладкой исходные файлы проекта, расположенные в папке src каталога проекта, и помещает созданные класс-файлы в папку build\classes. С помощью тега <classpath> в CLASSPATH для компиляции включен файл framework.jar, содержащий классы и интерфейсы платформы OSGi.

Задача "jar" архивирует в папке dist класс-файлы папки build\classes, содержимое папки resources и файл манифеста в JAR-файл с именем проекта.

Задача "clean" удаляет папки build\classes и dist вместе с содержимым.

После создания файла build.xml снабдим проект createJavadocRTF файлом манифеста manifest.mf и папкой src, которая будет содержать исходные файлы поставщика сервиса.

В случае, когда сервисная модель OSGi применяется при создании больших модульных систем, возникает ряд проблем. При запуске модульной системы одни OSGi-компоненты должны зарегистрировать свои сервисы, а другие должны их обнаружить. Все это требует значительного времени для запуска большой модульной системы. При регистрации сервиса в оперативную память загружаются связанные с реализацией сервиса классы. Если данный сервис не используется, то происходит необоснованное заполнение памяти. Для решения этих проблем, а также для упрощения кода, управляющего динамической загрузкой и выгрузкой сервисов, технология OSGi предлагает сервисную компонентную модель для создания декларативных сервисов. В проекте Knopflerfish сервисная компонентная модель реализована в виде bundle, который позволяет OSGi-компонентам декларировать свои связи с сервисами с помощью XML-файла, зарегистрированного в файле манифеста bundle с помощью элемента Service-Component:.

Мы также будем создавать декларативные сервисы, поэтому файл манифеста manifest.mf поставщика сервиса createJavadocRTF будет иметь соответствующее содержание (листинг 4.19).

Листинг 4.19. Файл манифеста manifest.mf поставщика сервиса createJavadocRTF

```
Manifest-Version: 2.0
Bundle-Name: createJavadocRTF
Bundle-SymbolicName: createjavadocrtf
```

```
Bundle-Description: Service Provider for RTF Javadoc
Bundle-Activator: createjavadoc.impl.rtf.Activator
Import-Package: org.osgi.framework, javax.swing
Export-Package: createjavadoc.iface
Service-Component: OSGI-INF/component.xml
```

Элемент файла манифеста поставщика сервиса `createJavadocRTF` (см. листинг 4.19) `Bundle-Activator:` определяет класс `Activator`, реализующий интерфейс `BundleActivator` платформы OSGi и управляющий запуском и остановкой bundle. Так как класс `Activator` будет использовать интерфейсы платформы OSGi, то мы должны указать в элементе `Import-Package:` файла манифеста пакет `org.osgi.framework`. В данной реализации поставщика сервиса мы внесем усовершенствование — при запуске bundle будет появляться диалоговое окно, в котором пользователь будет устанавливать путь к каталогу JDK, содержащему папку bin с инструментом `javadoc`, а также задавать путь к каталогу, содержащему папку lib с файлом доклета. Так как при создании диалогового окна мы будем использовать графическую библиотеку Swing, то в элементе `Import-Package:` мы должны указать пакет `javax.swing`. Поставщик сервиса у нас реализует сервис — интерфейс `createJavadocIface` пакета `createjavadoc.iface`, поэтому в элементе `Export-Package:` мы должны указать пакет `createjavadoc.iface`. В элементе `Service-Component:` мы указываем файл component.xml (листинг 4.20) папки OSGI-INF, который будет описывать декларативный сервис и с помощью которого будет происходить регистрация сервиса в реестре сервисов платформы OSGi.

Папку OSGI-INF мы поместим в папку resources, чтобы она была архивирована в JAR-файл bundle при сборке проекта.

Листинг 4.20. Файл component.xml поставщика сервиса `createJavadocRTF`

```
<?xml version="1.0" encoding="iso-8859-1"?>
<component
  xmlns = "http://www.osgi.org/xmlns/scr/v1.0.0"
  name  = "createJavadocRTF">
  <implementation class="createjavadoc.impl.rtf.createJavadocRTF"/>
  <service>
    <provide interface="createjavadoc.iface.createJavadocIface"/>
  </service>
</component>
```

В файле component.xml поставщика сервиса `createJavadocRTF` (см. листинг 4.20) декларация сервисных зависимостей bundle ограничена тегами `<component></component>`. Атрибут `xmlns` указывает URL-адрес документа, определяющего структуру XML-файла для описания компонента, которое ис-

пользуется средой выполнения Service Component Runtime (SCR). Тег `<service>` содержит имя регистрируемого сервиса. Тег `<implementation>` указывает имя поставщика сервиса.

Теперь, после выполнения всех необходимых подготовительных действий, мы можем создать проект createJavadocRTF в среде NetBeans.

Для создания проекта createJavadocRTF в среде NetBeans в меню **File** выберем **New Project | Categories: Java | Java Free-Form Project**. В диалоговом окне **New Java Free-Form Project** с помощью кнопки **Browse** рядом с полем **Location** найдем папку createJavadocRTF (рис. 4.8).

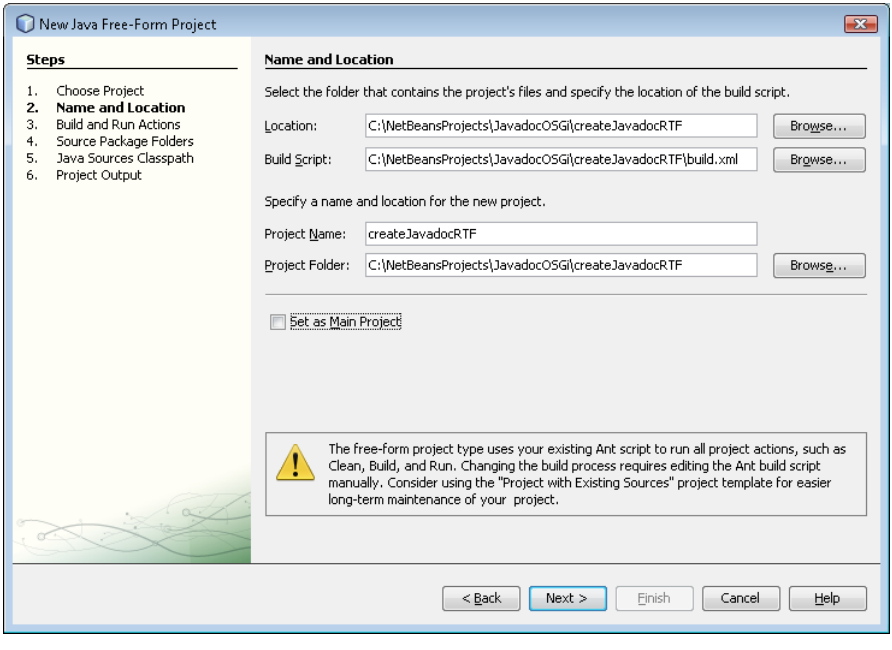

Рис. 4.8. Окно определения параметров проекта createJavadocRTF

Далее, после нажатия кнопки **Next**, в раскрывающихся списках **Build Project** и **Run Project** выберем **all**, тем самым связав действия соответствующих элементов меню среды NetBeans с задачей `"all"` Ant-скрипта (рис. 4.9).

Нажав кнопку **Next**, выберем папку src в качестве папки, содержащей исходные Java-файлы (рис. 4.10). После этого нажмем кнопку **Finish**.

Далее создадим интерфейс, который будет реализовывать поставщик сервиса `createJavadocRTF`. Для этого:

1. В окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на узле **default package** и выберем **New | Other | Categories: Java | Java Package**.

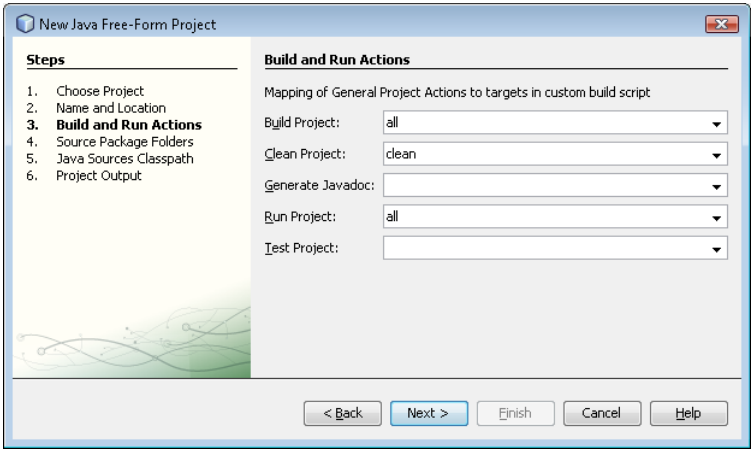

Рис. 4.9. Окно определения действий проекта createJavadocRTF

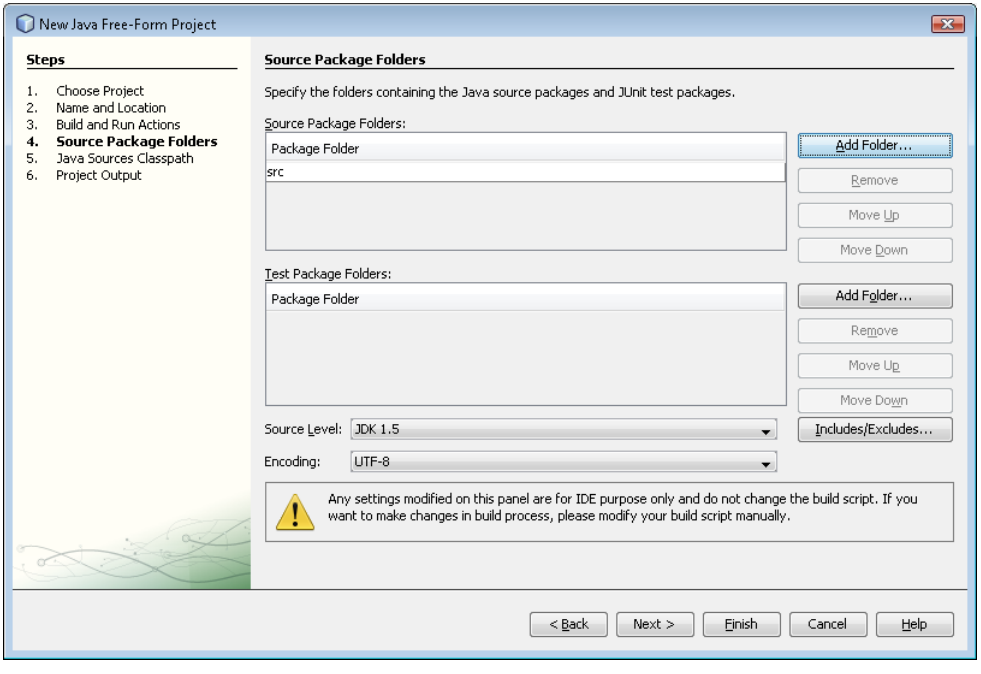

Рис. 4.10. Окно определения папки, содержащей исходные Java-файлы проекта createJavadocRTF

2. В строке **Package Name** введем createjavadoc.iface и нажмем кнопку **Finish**.
3. В окне **Projects** щелкнем правой кнопкой мыши на созданном пакете и выберем **New | Other | Categories: Java | Java Interface**. Введем имя интерфейса createJavadocIface и нажмем кнопку **Finish**.
4. В окне редактора среды NetBeans введем код интерфейса `createJavadocIface` (листинг 4.21).

Листинг 4.21. Код интерфейса `createJavadocIface`

```
package createjavadoc.iface;
  public interface createJavadocIface {
    public void cmdJavadoc(String src, String out, String pkg);
}
```

В коде интерфейса `createJavadocIface` (см. листинг 4.21) отсутствует переменная `jdk`, содержащая путь каталога JDK, т. к. определяться данный путь будет при запуске bundle.

В качестве следующего шага создадим класс `Activator` OSGi-компонента `createJavadocRTF`, управляющий запуском и остановкой bundle. Для этого:

1. В окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на узле **src** и выберем **New | Java Package**.
2. В строке **Package Name** введем имя пакета createjavadoc.impl.rtf и нажмем кнопку **Finish**.
3. В окне **Projects** щелкнем правой кнопкой мыши на созданном пакете и выберем **New | Other | Categories: Java | Java Class**. Введем имя класса `Activator` и нажмем кнопку **Finish**.
4. В окне редактора среды NetBeans введем код класса (листинг 4.22).

Листинг 4.22. Код класса `Activator` OSGi-компонента `createJavadocRTF`

```
package createjavadoc.impl.rtf;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
public class Activator implements BundleActivator{
public static BundleContext bc = null;
  public void start(BundleContext bc) throws Exception {
    Activator.bc = bc;
    JDialog.main(null);
  }
```

```
  public void stop(BundleContext bc) throws Exception {
    Activator.bc = null;
  }
}
```

Класс `Activator` (см. листинг 4.22) реализует интерфейс `BundleActivator` платформы OSGi, позволяющий управлять запуском и остановкой bundle. Интерфейс `BundleActivator` имеет два метода — `start()` и `stop()`, которые необходимо переопределить. В качестве аргумента эти методы получают объект `BundleContext`, который создается платформой OSGi для каждого bundle при его запуске и который обеспечивает взаимодействие bundle с платформой. Здесь мы в методе `start()` создаем диалоговое окно, в котором пользователь определяет пути каталога JDK и папки lib с файлом доклета.

Для создания класса `JDialog`, представляющего диалоговое окно, в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на пакете `createjavadoc.impl.rtf` и выберем **New | Other | Categories: Swing GUI Forms | JDialog Form**. Введем имя класса `JDialog` и нажмем кнопку **Finish**. В окне редактора форм среды NetBeans с помощью палитры графических компонентов **Palette** создадим диалоговое окно, содержащее два текстовых поля с двумя кнопками **Browse** (рис. 4.11).

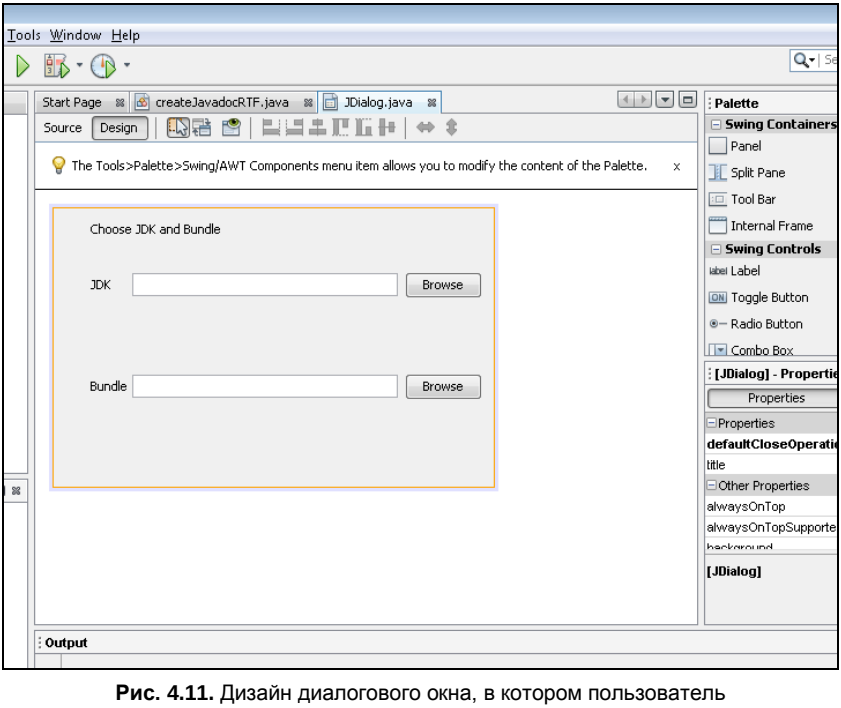

Рис. 4.11. Дизайн диалогового окна, в котором пользователь будет определять пути каталога JDK и папки lib с файлом доклета

Щелкнув правой кнопкой мыши на кнопках **Browse** диалогового окна, свяжем с ними обработчики событий, открывающие навигационные окна файловой системы (листинг 4.23).

Листинг 4.23. Код обработчиков событий кнопок *Browse* диалогового окна

```
private void jButtonJDKActionPerformed(java.awt.event.ActionEvent evt) {
  // TODO add your handling code here:
  JFileChooser chooser = new JFileChooser();
  chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
  chooser.setDialogTitle("Choose JDK Folder");
  int returnVal = chooser.showOpenDialog(null);
  if (returnVal == JFileChooser.APPROVE_OPTION) {
    jTextFieldJDK.setText(chooser.getSelectedFile().getPath());
    JDKPath=jTextFieldJDK.getText();
  }
}

private void jButtonBundleActionPerformed(java.awt.event.ActionEvent evt)
{
  // TODO add your handling code here:
  JFileChooser chooser = new JFileChooser();
  chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
  chooser.setDialogTitle("Choose JDK Folder");
  int returnVal = chooser.showOpenDialog(null);
  if (returnVal == JFileChooser.APPROVE_OPTION) {
    jTextFieldBundle.setText(chooser.getSelectedFile().getPath());
    bundlePath=jTextFieldBundle.getText();
  }
}
```

В коде обработчиков событий кнопок **Browse** диалогового окна (см. листинг 4.23) подразумевается, что папка lib с файлом доклета по умолчанию находится в том же каталоге, что и JAR-файл bundle поставщика сервиса.

Теперь в коде класса `JDialog` проекта createJavadocRTF нам надо создать два статических метода, с помощью которых мы могли бы получать значения статических переменных `JDKPath` и `bundlePath` (листинг 4.24).

Листинг 4.24. Код статических методов, возвращающих статические переменные `JDKPath` и `bundlePath` класса `JDialog`

```
public static String getPathBundle(){
    return bundlePath;
}
```

```
public static String getPathJDK(){
    return JDKPath;
}
// Объявление статических переменных:
    static String bundlePath;
    static String JDKPath;
```

Кроме того, уберем из метода `main()` класса `JDialog` вызов `System.exit(0)` в случае закрытия окна, т. к. в методе `initComponents()` у нас уже определена операция `dispose()` по умолчанию при закрытии окна с помощью метода `setDefaultCloseOperation` (`javax.swing.WindowConstants.DISPOSE_ON_CLOSE`) класса `javax.swing.JDialog` (листинги 4.25 и 4.26).

Листинг 4.25. Вызов `System.exit(0)` в первоначальном коде метода `main()` класса `JDialog`

```
dialog.addWindowListener(new java.awt.event.WindowAdapter() {
  public void windowClosing(java.awt.event.WindowEvent e) {
    System.exit(0);
  }
});
```

Листинг 4.26. Измененный код метода `main()` класса `JDialog`

```
public static void main(String args[]) {
  java.awt.EventQueue.invokeLater(new Runnable() {
    public void run() {
      JDialog dialog = new JDialog(new javax.swing.JFrame(), false);
      dialog.setVisible(true);
    }
  });    }
```

Для создания класса, реализующего интерфейс `createJavadocIface`, в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на пакете `createjavadoc.impl.rtf` и выберем **New | Other | Categories: Java | Java Class**. Введем имя класса `createJavadocRTF` и нажмем кнопку **Finish**. В окне редактора среды NetBeans введем код класса `createJavadocRTF` (листинг 4.27).

Листинг 4.27. Код класса `createJavadocRTF`

```
package createjavadoc.impl.rtf;
public class createJavadocRTF implements
       createjavadoc.iface.createJavadocIface {
```

```
  public void cmdJavadoc(String src,String out,String pkg){
    Runtime r =Runtime.getRuntime();
    String pathdoclet=JDialog.getPathBundle()+"\\lib\\";
    String JDKPath=JDialog.getPathJDK();
    String cmd=JDKPath+"/bin/javadoc"+" -doclet RTFDoclet"+
               " -docletpath "+pathdoclet+"RTFDoclet.jar"+
               " -sourcepath "+src+" -filename "+pkg+".rtf "+pkg;
    System.out.println(cmd);
    try {
      r.exec(cmd, null,new java.io.File(out));
    }
    catch(Exception ex)
      {javax.swing.JOptionPane.showMessageDialog(null, ex.toString());}
  }
}
```

В коде класса `createJavadocRTF` (листинг 4.27) мы воспользовались статическими методами `getPathBundle()` и `getPathJDK()` класса `JDialog` для получения путей к каталогу в папке lib с файлом доклета и каталогу JDK, используемым в командной строке инструмента `javadoc`.

Теперь мы можем создать JAR-файл OSGi-компонента `createJavadocRTF`. Для этого в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на проекте createJavadocRTF и выберем **Build**. В результате в папке dist каталога проекта будет создан JAR-файл bundle поставщика сервиса, отвечающего за генерацию Javadoc-документации в формате RTF.

По аналогии с поставщиком сервиса `createJavadocRTF` создадим bundle поставщика сервиса `createJavadocPDF`.

Для создания потребителя сервиса нам понадобится папка проекта ServiceDocDesktop в каталоге JavadocOSGi.

Создадим папку src, файл манифеста manifest.mf и файл Ant-скрипта build.xml (листинг 4.28) в папке проекта ServiceDocDesktop.

Далее в среде NetBeans в меню **File** выберем **New Project | Categories: Java | Java Free-Form Project**. В диалоговом окне **New Java Free-Form Project** с помощью кнопки **Browse** рядом с полем **Location** найдем папку ServiceDocDesktop. Нажав кнопку **Next**, в раскрывающихся списках **Build Project** и **Run Project** выберем **all**, тем самым связав действия соответствующих элементов меню среды NetBeans с задачей `"all"` Ant-скрипта. После этого выберем папку src в качестве папки, содержащей исходные Java-файлы, и нажмем кнопку **Finish**.

Листинг 4.28. Файл Ant-сценария build.xml проекта ServiceDocDesktop

```
<?xml version="1.0"?>
<project name="ServiceDocDesktop">
<property name="createJavadocIface.jar"
          location="lib/createJavadocIface.jar"/>
  <target name="all" depends="init,compile,jar"/>
  <target name="init">
    <mkdir dir="build/classes"/>
  </target>
  <target name="compile">
    <javac destdir = "build/classes"
           debug = "on"
           srcdir = "src">
      <classpath>
        <pathelement location="${createJavadocIface.jar}"/>
      </classpath>
    </javac>
  </target>
  <target name="jar">
    <mkdir dir="dist"/>
    <jar jarfile  = "dist/${ant.project.name}.jar"
         compress = "true"
         manifest = "manifest.mf">
      <fileset dir="build/classes"/>
      <fileset dir="resources"/>
   </jar>
  </target>
  <target name="clean">
    <delete dir = "build"/>
    <delete dir = "dist"/>
  </target>
</project>
```

В файле Ant-сценария build.xml проекта ServiceDocDesktop (см. листинг 4.28) при компиляции используется JAR-файл интерфейса `createJavadocIface`, который нам необходимо поместить в папку lib каталога проекта ServiceDocDesktop.

Для создания JAR-файла интерфейса `createJavadocIface` нам понадобится папка проекта createJavadocIface в каталоге JavadocOSGi.

Создадим папку src, файл манифеста manifest.mf и файл Ant-скрипта build.xml (листинг 4.29) в папке проекта createJavadocIface. В среде NetBeans

в меню **File** выберем **New Project | Categories: Java | Java Free-Form Project**. В диалоговом окне **New Java Free-Form Project** с помощью кнопки **Browse** рядом с полем **Location** найдем папку createJavadocIface. Нажав кнопку **Next**, в раскрывающихся списках **Build Project** и **Run Project** выберем **all**. Далее выберем папку src в качестве папки, содержащей исходные Java-файлы, и нажмем кнопку **Finish**.

Листинг 4.29. Файл Ant-сценария build.xml проекта createJavadocIface

```
<?xml version="1.0"?>
<project name="createJavadocIface" >
  <target name="all" depends="init,compile,jar"/>
  <target name="init">
    <mkdir dir="build/classes"/>
  </target>
  <target name="compile">
    <javac destdir = "build/classes"
           debug = "on"
           srcdir = "src">
    </javac>
  </target>
  <target name="jar">
    <mkdir dir="dist"/>
    <jar jarfile  = "dist/${ant.project.name}.jar"
         compress = "true"
         manifest = "manifest.mf">
      <fileset dir="build/classes"/>
    </jar>
  </target>
  <target name="clean">
    <delete dir = "build"/>
    <delete dir = "dist"/>
  </target>
</project>
```

Файл манифеста manifest.mf проекта createJavadocIface оставим пустым, т. к. создаваемый нами JAR-файл является просто библиотечным.

Для создания интерфейса `createJavadocIface`:

1. В окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на узле **src** и выберем **New | Java Package**.
2. В поле **Package Name** введем имя пакета createjavadoc.iface и нажмем кнопку **Finish**.

3. В окне **Projects** щелкнем правой кнопкой мыши на созданном пакете и выберем **New | Other | Categories: Java | Java Interface**. Введем имя интерфейса `createJavadocIface` и нажмем кнопку **Finish**.
4. В окне редактора среды NetBeans введем код интерфейса `createJavadocIface` (см. листинг 4.21).
5. Далее в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на проекте createJavadocIface и выберем **Build**. В результате в папке dist проекта createJavadocIface будет создан JAR-файл интерфейса, который мы можем теперь скопировать в папку lib каталога проекта ServiceDocDesktop.

Вернемся к проекту ServiceDocDesktop. Файл манифеста bundle потребителя сервиса будет иметь содержание, указанное в листинге 4.30.

Листинг 4.30. Файл манифеста manifest.mf OSGi-компонента `ServiceDocDesktop`

```
Manifest-Version: 2.0
Bundle-Name: Custom Javadoc
Bundle-SymbolicName: ServiceDocDesktop
Bundle-Version: 1.0.0
Bundle-Description: Desktop for Javadoc
Import-Package: createjavadoc.iface,javax.swing
Service-Component: OSGI-INF/component.xml
```

Так как OSGi-компонент `ServiceDocDesktop` будет содержать графический интерфейс, созданный с использованием графической библиотеки Swing, и будет использовать сервис `createJavadocIface`, то в элементе `Import-Package:` его файла манифеста мы укажем пакеты `createjavadoc.iface` и `javax.swing`.

OSGi-компонент `ServiceDocDesktop` будет использовать декларативные сервисы, поэтому в элементе `Service-Component:` мы укажем файл component.xml (листинг 4.31) папки OSGI-INF, описывающий сервисные зависимости bundle `ServiceDocDesktop`.

Листинг 4.31. Файл component.xml OSGi-компонента `ServiceDocDesktop`

```
<?xml version="1.0" encoding="iso-8859-1"?>
<component
  xmlns = "http://www.osgi.org/xmlns/scr/v1.0.0"
  name  = "ServiceDocDesktop">
  <implementation
     class="servicedocdesktop.Component"/>
```

```
  <reference name = "createJavadocIface"
       interface = "createjavadoc.iface.createJavadocIface"
       bind = "setProvider"
       unbind = "unsetProvider"
       cardinality="0..n"
       policy="dynamic"
  />
</component>
```

В файле component.xml OSGi-компонента `ServiceDocDesktop` (см. листинг 4.31) тег `<reference>` содержит имя сервиса, к которому необходимо получить доступ. Имя сервиса определяется атрибутом `interface`. Атрибуты `bind` и `unbind` указывают методы, вызываемые автоматически при обнаружении необходимого зарегистрированного сервиса в реестре сервисов платформы OSGi и потери связи с сервисом соответственно. Таким образом, обеспечивается динамическая связь `bundle` с сервисами. Атрибут `cardinality` указывает способ связи bundle с сервисом. Значение `"0..n"` означает, что связь может быть необязательна для bundle, и что bundle может быть связан более чем с одним сервисом, зарегистрированным под именем, указанным в атрибуте `interface`. При этом методы, указанные в атрибутах `bind` и `unbind`, вызываются всякий раз при установлении или потери связи с каждым сервисом. Атрибут `policy` определяет методику связи с сервисами. Значение `"dynamic"` означает, что среда выполнения Service Component Runtime (SCR) может изменять набор связанных с bundle сервисов без деактивации конфигурации компонента. Тег `<implementation>` определяет имя класса, содержащего методы, указанные в атрибутах `bind` и `unbind` тега `<reference>`.

Папку OSGI-INF вместе с файлом component.xml мы поместим в папку resources каталога проекта ServiceDocDesktop, тем самым обеспечив ее архивацию при сборке проекта в JAR-файл bundle.

Для создания класса `servicedocdesktop.Component` OSGi-компонента `ServiceDocDesktop`:

1. В окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на узле **src** и выберем **New | Java Package**. В строке **Package Name** введем servicedocdesktop и нажмем кнопку **Finish**.
2. После этого в окне **Projects** щелкнем правой кнопкой мыши на созданном пакете и выберем **New | Other | Categories: Java | Java Class**. Введем имя класса `Component` и нажмем кнопку **Finish**.
3. В окне редактора среды NetBeans введем код класса `Component` (листинг 4.32).

Листинг 4.32. Код класса `Component` OSGi-компонента `ServiceDocDesktop`

```
package servicedocdesktop;
import createjavadoc.iface.createJavadocIface;
import java.util.*;
public class Component {
  static List<createJavadocIface> ListcreateJavadocIface;
  public Component(){
    ServiceDocDesktop.main(null);
    ListcreateJavadocIface=new ArrayList<createJavadocIface>();
  }
  public void setProvider(createJavadocIface i){
    ListcreateJavadocIface.add(i);
  }
  public void unsetProvider(createJavadocIface i){
    ListcreateJavadocIface.remove(i);
  }
  static List<createJavadocIface> getList(){
    return ListcreateJavadocIface;
}}
```

В конструкторе класса `Component` (см. листинг 4.32), который будет вызван автоматически при запуске bundle, создается диалоговое окно графического интерфейса, представленное классом `ServiceDocDesktop`, а также создается объект `ListcreateJavadocIface`, представляющий список зарегистрированных сервисов `createJavadocIface` и являющийся объектом класса `java.util.ArrayList<E>`. Класс `ArrayList<E>` реализует интерфейс `List<E>` и обеспечивает хранение и операции со списком.

В случае если bundle потребитель сервиса использует сервисную компонентную модель для связи с сервисами, при его запуске автоматически создается объект класса, указанного в теге `<implementation>` компонентного файла. Поэтому невозможно одновременно с сервисной компонентной моделью здесь использовать класс, указанный в элементе `Bundle-Activator`: файла манифеста bundle и реализующий интерфейс `BundleActivator` платформы OSGi.

В методе `setProvider()` класса `Component`, автоматически вызываемом при установлении связи с сервисом `createJavadocIface`, найденный зарегистрированный сервис добавляется в список `ListcreateJavadocIface` с помощью метода `add()` класса `ArrayList<E>`. Аналогично в методе `unsetProvider` класса `Component`, автоматически вызываемом при потери связи с сервисом `createJavadocIface`, сервис удаляется из списка методом `remove(Object o)` класса `ArrayList<E>`.

В классе `Component` мы создали статический метод `getList()` для того, чтобы иметь возможность получать список сервисов, с которыми установлена связь.

Теперь мы можем создать класс `ServiceDocDesktop` OSGi-компонента `ServiceDocDesktop`, представляющий диалоговое окно графического интерфейса bundle потребителя сервиса `createJavadocIface`. Для этого в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на пакете `servicedocdesktop` и выберем **New | Other | Categories: Swing GUI Forms | JDialog Form**. Введем имя класса `ServiceDocDesktop` и нажмем кнопку **Finish**. В окне редактора форм среды NetBeans с помощью палитры графических компонентов **Palette** создадим диалоговое окно, содержащее три текстовых поля с тремя кнопками **Browse**, а также кнопки **Create Javadoc** и **Set Doclets** (рис. 4.12).

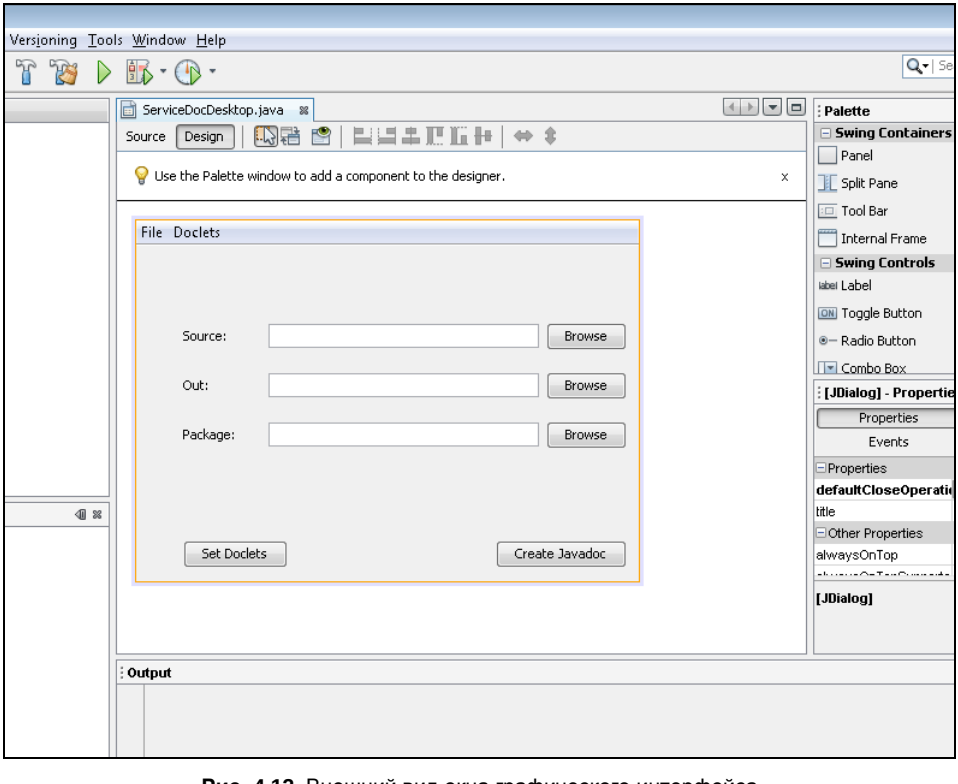

Рис. 4.12. Внешний вид окна графического интерфейса OSGi-компонента `ServiceDocDesktop`

С кнопками **Browse** окна графического интерфейса OSGi-компонента `ServiceDocDesktop` свяжем методы — обработчики событий, открывающие

навигационные окна файловой системы, с помощью которых пользователь будет устанавливать пути к каталогам с исходными Java-файлами, содержащими Javadoc-комментарии, и к каталогу, в котором будут сохранены сгенерированные файлы Javadoc-документации, а также будет указано имя пакета, для которого создается Javadoc-документация (листинг 4.33).

Листинг 4.33. Код обработчиков событий кнопок *Browse* окна графического интерфейса OSGi-компонента `ServiceDocDesktop`

```
private void jButtonSourceActionPerformed(java.awt.event.ActionEvent evt)
{
  // TODO add your handling code here:
  JFileChooser chooser = new JFileChooser(pathChooser);
  chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
  chooser.setDialogTitle("Choose Source Folder");
  int returnVal = chooser.showOpenDialog(null);
  if (returnVal == JFileChooser.APPROVE_OPTION) {
    jTextFieldSource.setText(chooser.getSelectedFile().getPath());
    pathChooser=chooser.getSelectedFile().getPath();
  }
}

private void jButtonOutActionPerformed(java.awt.event.ActionEvent evt) {
  // TODO add your handling code here:
  JFileChooser chooser = new JFileChooser(pathChooser);
  chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
  chooser.setDialogTitle("Choose Out Folder");
  int returnVal = chooser.showOpenDialog(null);
  if (returnVal == JFileChooser.APPROVE_OPTION) {
    jTextFieldOut.setText(chooser.getSelectedFile().getPath());
    pathChooser=chooser.getSelectedFile().getPath();
  }
}

private void jButtonPackageActionPerformed(
        java.awt.event.ActionEvent evt) {
  // TODO add your handling code here:
  JFileChooser chooser = new JFileChooser(pathChooser);
  chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
  chooser.setDialogTitle("Choose Package Folder");
  int returnVal = chooser.showOpenDialog(null);
  if ((jTextFieldSource.getText()).isEmpty()==true)
    javax.swing.JOptionPane.showMessageDialog(null, "Input Source");
```

```
  else {
    if (returnVal == JFileChooser.APPROVE_OPTION) {
      StringBuffer buf=
        new StringBuffer(chooser.getSelectedFile().getPath());
      buf.delete(0,jTextFieldSource.getText().length()+1);
      int flag=0;
      while (flag==0){
        int pos=buf.indexOf("\\");
        if (pos!=-1)
           buf.replace(pos, pos+1, ".");
        else flag=1;
      }
      jTextFieldPackage.setText(buf.toString());
      }}    }
```

В меню **File** окна графического интерфейса OSGi-компонента `ServiceDocDesktop` добавим элемент **Exit**, с которым свяжем обработчик событий, закрывающий окно и освобождающий все занятые диалоговым окном ресурсы операционной системы (листинг 4.34).

Листинг 4.34. Код обработчика событий элемента *Exit* меню *File* окна графического интерфейса OSGi-компонента `ServiceDocDesktop`

```
private void jMenuItemExitActionPerformed(java.awt.event.ActionEvent evt)
{
  // TODO add your handling code here:
  this.dispose();
}
```

С кнопкой **Set Doclets** окна графического интерфейса OSGi-компонента `ServiceDocDesktop` свяжем обработчик событий, который будет получать список зарегистрированных сервисов `createJavadocIface` с помощью статического метода `getList()` класса `Component` и добавлять полученный список в виде переключателей в меню **Doclets** диалогового окна (листинг 4.35).

Листинг 4.35. Код обработчика событий кнопки *Set Doclets* окна графического интерфейса OSGi-компонента `ServiceDocDesktop`

```
private void jButtonSetDocletsActionPerformed(
        java.awt.event.ActionEvent evt) {
  // TODO add your handling code here:
  jRadioButtonMenuItemDoclets=
        new HashMap<String, JRadioButtonMenuItem>();
```

```
  ListcreateJavadocIface=Component.getList();
  for (createJavadocIface i: ListcreateJavadocIface){
    JRadioButtonMenuItem jrb=new JRadioButtonMenuItem();
    jrb.setText(i.getClass().getName());
    jrb.setSelected(true);
    jRadioButtonMenuItemDoclets.put(i.getClass().getName(), jrb);
  }
  setMenuDoclets();
}
```

Переключатели меню **Doclets** окна графического интерфейса OSGi-компонента `ServiceDocDesktop` будут храниться в хэш-таблице `jRadioButtonMenuItemDoclets` в виде пар "имя класса-поставщика сервиса/объект `JRadioButtonMenuItem`". Метод `setMenuDoclets()`, вызываемый в обработчике событий `jButtonSetDocletsActionPerformed` (листинг 4.37), обрабатывает хэш-таблицу, добавляя переключатели из хэш-таблицы в меню **Doclets** (листинг 4.36).

Листинг 4.36. Код метода `setMenuDoclets()` класса `ServiceDocDesktop`

```
private void setMenuDoclets(){
  jMenuDoclets.removeAll();
  Collection <JRadioButtonMenuItem> crb=
     jRadioButtonMenuItemDoclets.values();
  for (Iterator i = crb.iterator(); i.hasNext();){
    JRadioButtonMenuItem jrb=(JRadioButtonMenuItem)i.next();
    jMenuDoclets.add(jrb);
  }
}
```

При этом мы должны добавить соответствующие поля `jRadioButtonMenuItemDoclets` и `ListcreateJavadocIface` в класс `ServiceDocDesktop` с помощью строк:

```
private Map<String, JRadioButtonMenuItem> jRadioButtonMenuItemDoclets;
 private List<createJavadocIface> ListcreateJavadocIface;
```

Объект `ListcreateJavadocIface` класса `ArrayList<E>` мы создадим в конструкторе класса `ServiceDocDesktop` с помощью строки:

```
ListcreateJavadocIface=new ArrayList<createJavadocIface>();
```

С кнопкой **Create Javadoc** окна графического интерфейса OSGi-компонента `ServiceDocDesktop` мы свяжем обработчик событий, который будет извлекать поставщиков сервиса из списка `ListcreateJavadocIface` и вызывать метод

`cmdJavadoc()` в случае, если соответствующий переключатель меню **Doclets** активирован (листинг 4.37).

Листинг 4.37. Код обработчика событий кнопки *Create Javadoc* окна графического интерфейса OSGi-компонента `ServiceDocDesktop`

```
private void
jButtonCreateJavadocActionPerformed(java.awt.event.ActionEvent evt) {
  // TODO add your handling code here:
  for (createJavadocIface i: ListcreateJavadocIface) {
    if (jRadioButtonMenuItemDoclets.get(i.getClass()
          .getName()).isSelected()){
      String src=jTextFieldSource.getText();
      String out=jTextFieldOut.getText();
      String pkg=jTextFieldPackage.getText();
      i.cmdJavadoc(src,out,pkg);
   }    }    }
```

В статическом методе `main()` класса `ServiceDocDesktop` мы уберем вызов `System.exit(0)` в случае закрытия окна (листинг 4.38).

Листинг 4.38. Код метода `main` класса `ServiceDocDesktop`

```
public static void main(String args[]) {
  java.awt.EventQueue.invokeLater(new Runnable() {
    public void run() {
      ServiceDocDesktop dialog =
        new ServiceDocDesktop(new javax.swing.JFrame(), false);
      dialog.setVisible(true);
    }  }); }
```

Теперь мы можем собрать проект и создать JAR-файл OSGi-компонента `ServiceDocDesktop`. Для этого в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на проекте ServiceDocDesktop и выберем **Build**. В результате в папке dist будет создан JAR-файл bundle потребителя сервиса.

Чтобы запустить созданное нами в итоге OSGi-приложение:

1. Откроем графический интерфейс платформы Knopflerfish, щелкнув на файле framework.jar каталога платформы Knopflerfish.
2. Инсталлируем созданные нами OSGi-компоненты, открыв меню **File** и выбрав пункт **Open Bundle File**.
3. С помощью открывающегося навигационного окна файловой системы будем находить JAR-файлы OSGi-компонентов и запускать их, используя элемент **Start** меню **Bundles**.

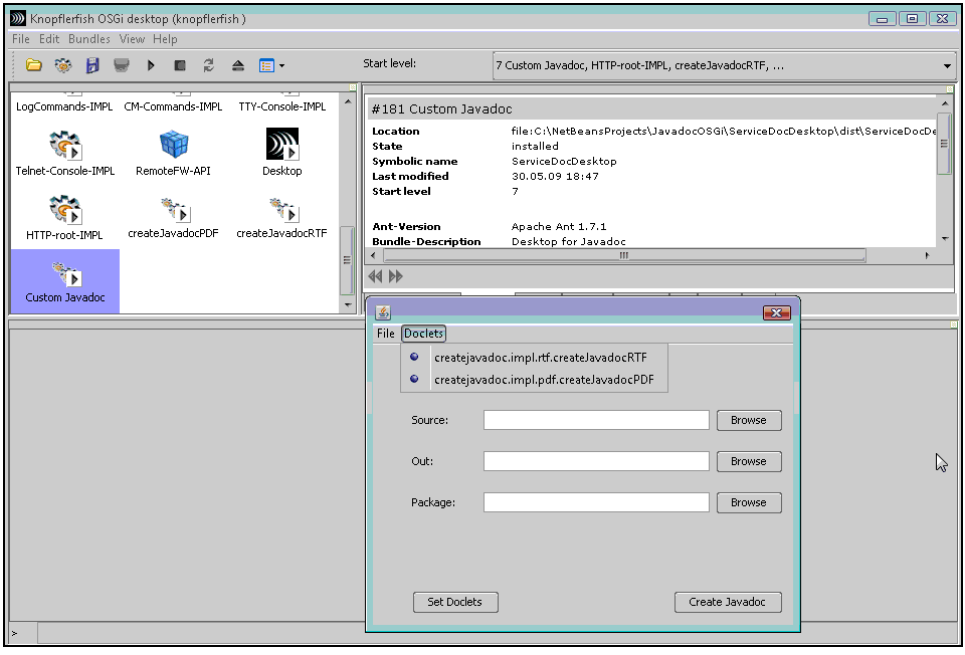

Рис. 4.13. OSGi-приложение, позволяющее генерировать Javadoc-документацию в различных форматах

Таким образом, используя графический интерфейс Knopflerfish OSGi desktop, мы развернули созданное нами модульное OSGi-приложение (рис. 4.13).

ЧАСТЬ II

Платформа Java ME

Глава 5	Технологии платформы Java ME
Глава 6	Создание Java-приложений на основе платформы Java ME

Как уже было сказано во *введении*, технология Java применима к широкому диапазону устройств — от смарт-карт, встроенных и мобильных устройств до серверов и суперкомпьютеров. Для того чтобы обеспечить работу Java-приложений в таком большом наборе устройств, технология Java включает в себя ряд Java-платформ, каждая из которых призвана решать задачи в определенной области.

Платформа Java Platform Micro Edition (Java ME) содержит набор сред выполнения и библиотек API для приложений, выполняемых в мобильных и встроенных устройствах, таких как мобильные телефоны, КПК, телевизионные приставки, принтеры и др.

Платформа Java ME дает возможность создавать приложения для небольших устройств с ограниченной памятью, графическими возможностями и производительностью, снабжая устройства соответствующей средой выполнения для работы приложений.

Реализации платформы Java ME поставляются в виде OEM-версий производителям мобильных и встроенных устройств, которые выводят на рынок свои продукты вместе с программным обеспечением технологии Java ME в виде предустановленной версии.

Примечание

Original Equipment Manufacturer (OEM) — компания, продающая компоненты или программное обеспечение производителю конечного продукта.

Платформа Java ME представляет собой набор технологий, комбинации которых позволяют обеспечить совместимость с широким диапазоном устройств — от промышленных контроллеров до мобильных устройств и телевизионных приставок, формируя среду выполнения Java-приложений для конкретного круга устройств.

Каждая комбинация Java ME технологий основывается на трех элементах:

- конфигурации, предоставляющей базовый набор библиотек и виртуальную Java-машину для широкого диапазона устройств;
- профиля, поддерживающего набор библиотек для более узкого круга устройств и расширяющего конфигурацию до полной платформы, на базе которой может быть создано и развернуто Java-приложение;
- дополнительных пакетов, обеспечивающих специфические возможности и дополнительную функциональность Java-приложениям.

Комбинация конфигурации, профиля и дополнительных пакетов обеспечивает соответствие требованиям конкретного класса устройств, принадлежащих к тому набору, для которого предназначена платформа Java ME.

Платформа Java ME содержит два типа конфигураций — Connected Limited Device Configuration (CLDC) и Connected Device Configuration (CDC), основное отличие которых состоит в том, что конфигурация CDC предназначена для устройств с более широкими возможностями, по сравнению с CLDC-устройствами.

Конфигурация CLDC, как правило, комбинируется с профилем Mobile Information Device Profile (MIDP), вместе обеспечивая среду выполнения для таких широко распространенных устройств, как мобильные телефоны и КПК.

Java-приложения, созданные для среды выполнения, основанной на комбинации CLDC и MIDP, называются MIDlets — *мидлетами*.

Разработчики Java-приложений с помощью мидлетов расширяют возможности мобильных телефонов и КПК, предоставляя огромный выбор приложений, включая игры, различные офисные программы, Web-браузеры и т. д.

Конфигурация CDC комбинируется с тремя профилями — Foundation Profile, Personal Basis Profile и Personal Profile — и предназначена для устройств с расширенными возможностями, таких как высокотехнологичные КПК (например, Zaurus SL), телевизионные приставки, автомобильные навигационные системы и др.

Приложения конфигурации CDC для органайзеров используются в медицине для мобильного сбора информации о пациентах и организации работы персонала, торговле, производстве, финансовой сфере, организуя связь между покупателями и продавцами, осуществляя финансовые операции, реализуя биз-

нес-логику и помогая в управлении технологическими процессами. Для телевизионных компьютерных приставок CDC-приложения представляют различное программное обеспечение, включая игры, приложения интерактивного телевидения и интернет-приложения. Для различного рода встроенных устройств — банкоматов, платежных терминалов и др. — CDC-приложения решают специфические задачи, дополняя их функциональность.

Дополнительные пакеты реализуются в комбинации с профилем и конфигурацией, расширяя среду выполнения Java ME устройств для обеспечения таких дополнительных возможностей, как отправка/получение SMS-сообщений, поддержка Remote Method Invocation (RMI), воспроизведение мультимедийного контента, взаимодействие с базами данных и др.

Таким образом, платформа Java ME предлагает универсальное решение для создания современных приложений, работающих в устройствах с ограниченными возможностями, открытостью своего проекта вовлекая в процесс разработки как крупные корпорации, так и независимых разработчиков.

ГЛАВА 5

Технологии платформы Java ME

Технология CLDC

Конфигурация Connected Limited Device Configuration (CLDC) предназначена для устройств с ограниченными ресурсами (мощность процессора, память, графические возможности), таких как мобильные телефоны и карманные персональные компьютеры. Устройства, в которых реализована конфигурация CLDC, имеют следующие характеристики:

- 16- или 32-битный процессор с частотой 16 МГц и выше;
- как минимум 160 Кбайт энергонезависимой памяти для виртуальной машины и базовых библиотек;
- как минимум 32 Кбайт оперативной памяти;
- низкое энергопотребление;
- нестационарное сетевое соединение с ограниченной пропускной способностью.

Конфигурации и профили платформы Java ME сами по себе являются Java-технологиями, состоящими из API-спецификации, эталонной реализации спецификации (Reference Implementation (RI)) и набора автоматических тестов на соответствие реализации спецификации (Technology Compatibility Kit (TCK)).

Поэтому CLDC-технология содержит:

- спецификации CLDC версии 1.0 (JSR 30) и CLDC версии 1.1 (JSR 139);
- эталонную реализацию спецификации CLDC 1.1 RI, основанную на спецификации CLDC 1.1, и эталонную реализацию CLDC 1.0.4 RI, основанную на спецификации CLDC 1.0. Реализация спецификации CLDC RI базируется на виртуальной машине K Virtual Machine (KVM). KVM — уменьшенная Java-виртуальная машина, требующая до 80 Кбайт памяти.

KVM реализована на языке программирования C и может быть установлена на различных платформах, имеющих C-компилятор;

- оптимизированную реализацию CLDC HotSpot Implementation — оптимизированную виртуальную машину, предназначенную для устройств с увеличенными ресурсами — частотой процессора более 50 МГц, оперативной памятью более 600 Кбайт и доступным дисковым пространством более 1,5 Мбайт. CLDC HI обеспечивает более быстрое выполнение байт-кода и более эффективное использование ресурсов по сравнению с KVM;
- набор автоматических тестов на соответствие реализации спецификации CLDC TCK.

Спецификация CLDC 1.0 описывает компактную виртуальную машину и базовый набор библиотек. Библиотеки CLDC 1.0 делятся на две части: ограниченный набор стандартных библиотек Java SE и библиотеки, специфические для CLDC *(см. приложение 10)*.

Спецификация CLDC 1.0 наследует от Java SE три усеченных пакета: `java.lang`, `java.io` и `java.util`. Все ограничения преследуют одну цель — экономию ресурсов. Конфигурация CLDC 1.0 не поддерживает полную систему безопасности Java SE, состоящую из прав доступа, контроля над доступом и политики безопасности. Система безопасности CLDC 1.0 представляет собой песочницу, т. е. Java-приложение работает в замкнутом окружении, в котором оно может иметь доступ только к библиотекам API, предоставленным набором, состоящим из конфигурации, профиля и дополнительных пакетов и реализованным в данном устройстве. Поэтому вся загрузка и управление Java-приложением осуществляется только виртуальной машиной и отсутствует возможность расширения класса `java.lang.ClassLoader`. Модель песочницы подразумевает также отсутствие библиотеки Java Native Interface (JNI), позволяющей осуществлять доступ к операционной системе устройства. Конфигурация CLDC 1.0 не поддерживает вычисления с плавающей запятой, исключен метод `finalize()` класса `java.lang.Object`, ограничен набор классов `Error`, исключен пакет `java.lang.reflect`, позволяющий получать информацию о классах и объектах, не поддерживается группировка потоков и слабые ссылки `java.lang.ref.WeakReference`, которые не учитываются сборщиком мусора.

Специфический пакет `javax.microedition.io` конфигурации CLDC 1.0 представляет платформу Generic Connection, обеспечивающую поддержку ввода/вывода и работу в сети.

Конфигурация CLDC 1.1 расширяет CLDC 1.0 и добавляет поддержку вычислений с плавающей запятой и слабые ссылки. В конфигурации CLDC 1.1 доработаны классы `Calendar`, `Date`, `TimeZone` и `Thread`, добавлен класс `java.lang.NoClassDefFoundError`, исправлен ряд ошибок. В конфигурации

CLDC 1.1.1 расширена модель безопасности включением классов `Permission`, улучшен класс `Math`, включены классы `java.util.Timer` и `java.util.TimerTask`.

Конфигурация CLDC не охватывает такие вопросы, необходимые для создания и запуска Java-приложений, как управление жизненным циклом приложения, функциональные возможности пользовательского интерфейса, обработка событий и другие дополнительные функции. Обеспечить эти возможности призван профиль, расширяющий набор библиотек CLDC своими библиотеками API.

Для конфигурации CLDC существуют три профиля: Mobile Information Device Profile (MIDP), Information Module Profile (IMP) и Digital Set Top Box Profile.

Профиль Digital Set Top Box Profile предназначен для кабельных телевизионных приставок с ограниченными ресурсами и содержит пакеты, обеспечивающие ввод/вывод, работу в сети, графические возможности, а также включает в себя ограниченный набор библиотек Java TV API.

Профиль Information Module Profile (IMP) предназначен для устройств с очень ограниченными графическими возможностями, таких как парковочные счетчики и беспроводные модули тревожной сигнализации. Профиль IMP дополнительно к библиотекам CLDC содержит пакеты, обеспечивающие управление жизненным циклом Java-приложения (IMlet), а также хранение и доступ к данным. Профиль IMP представляет собой усеченный профиль MIDP 1.0 с исключенными графическими библиотеками.

Профиль Information Module Profile — Next Generation (IMP-NG) представляет собой усеченный профиль MIDP 2.0 с исключенными графическими библиотеками.

Профиль Mobile Information Device Profile (MIDP) наиболее распространен, т. к. предназначен для таких устройств, как мобильные телефоны и карманные персональные компьютеры.

Технология MIDP

Технология Mobile Information Device Profile (MIDP) содержит:

- спецификации MIDP 1.0, MIDP 2.0 и MIDP 3.0;
- эталонные реализации MIDP RI 1.0.3 (основывается на MIDP 1.0 и CLDC RI 1.0.3) и MIDP RI 2.0 (основывается на MIDP 2.0 и CLDC RI 1.0.4);
- оптимизированную реализацию MIDP for Palm OS, основанную на CLDC 1.0 и MIDP 1.0 и оптимизированную для операционной системы Palm OS 3.5;

- набор автоматических тестов MIDP TCK на соответствие реализации спецификации;
- средства разработки Sun Java Wireless Toolkit и NetBeans Mobility Pack.

Устройства, для которых предназначена спецификация MIDP 1.0, имеют следующие минимальные характеристики:

- размер экрана — 96×54, глубина изображения — 1 бит, коэффициент пропорциональности — 1:1;
- один или более механизмов ввода — ввод со стандартной клавиатуры, ввод с помощью клавиатуры сенсорного экрана и т. д.;
- 128 Кбайт доступного дискового пространства для MIDP-компонентов, 8 Кбайт доступного дискового пространства для сохраняемых данных приложения и 32 Кбайт оперативной памяти;
- нестабильное подключение к сети с ограниченной пропускной способностью.

Спецификация MIDP 1.0 расширяет библиотеки CLDC 1.0, добавляя пакеты `javax.microedition.lcdui`, `javax.microedition.rms` и `javax.microedition.midlet`, и обеспечивает базовую функциональность для создания MIDP-приложений, называемых MIDlets — *мидлетами*.

Пакет `javax.microedition.midlet` дает определение MIDP-приложению (MIDlet) и описывает взаимодействие между MIDP-приложением и окружением, в котором оно работает.

Обычное приложение Java SE представляет собой набор Java класс-файлов, содержащий единственный и уникальный статический метод `main()`, вызываемый виртуальной машиной Java при запуске приложения и являющийся точкой входа в приложение. Конфигурация CLDC поддерживает такую модель Java-приложения. Однако профиль MIDP определяет другую модель программирования Java-приложений, похожую на модель апплета. Апплет является встроенным Java-приложением, работающим под контролем Web-браузера, и имеет главный класс, расширяющий класс `java.applet.Applet`. Точно так же мидлет управляется программным обеспечением Application Management Software (AMS), которое является частью программного обеспечения устройства. Мидлет представляет собой набор Java класс-файлов, содержащий главный класс, расширяющий абстрактный класс `javax.microedition.midlet.MIDlet`. При этом главный класс переопределяет три метода жизненного цикла приложения: `startApp()`, `pauseApp()` и `destroyApp()` класса `MIDlet`.

Когда пользователь запускает мидлет, AMS создает новый экземпляр главного класса, вызывая его конструктор по умолчанию (без аргументов), и состояние мидлета становится `Paused`. После этого AMS вызывает переопреде-

ленный метод `startApp()` главного класса, в котором обычно создается и выводится на экран графический интерфейс приложения. Теперь состояние мидлета становится `Active`. Предположим, во время работы мидлета на сотовый телефон приходит SMS-сообщение. AMS деактивирует приложение и вызывает переопределенный метод `pauseApp()`, в котором освобождается часть системных ресурсов, и состояние мидлета становится `Paused`. Если во время работы мидлета по каким-то причинам происходит переполнение памяти, то AMS вызывает переопределенный метод `destroyApp()`, в котором должны освобождаться все системные ресурсы и возможно сохраняться какие-либо данные, и состояние мидлета становится `Destroyed`.

Механизм состояний мидлета создан для обеспечения последовательного и предсказуемого поведения приложения. Состояния `Active`, `Paused` и `Destroyed` могут вызываться программным образом с помощью методов `resumeRequest()`, `notifyPaused()` и `notifyDestroyed()` класса `MIDlet` соответственно.

Спецификация MIDP 1.0 определяет комплект мидлетов (MIDlet Suite), состоящий из:

- среды выполнения, которая обеспечивается AMS;
- JAR-файла комплекта;
- файла дескриптора (Java Application Descriptor, JAD);
- жизненного цикла приложения.

JAR-файл комплекта MIDlet Suite состоит из файла манифеста, ресурсных файлов, класс-файлов одного или нескольких мидлетов и класс-файлов, используемых мидлетами. Комплект MIDlet Suite может содержать несколько MIDP-приложений, которые могут взаимодействовать друг с другом и использовать общие ресурсы в пределах комплекта.

Файл манифеста комплекта MIDlet Suite должен содержать следующие элементы:

- `MIDlet-Name` — имя комплекта (обязательный элемент);
- `MIDlet-Version` — версия комплекта (обязательный элемент);
- `MIDlet-Vendor` — производитель комплекта (обязательный элемент);
- `MIDlet-Icon` — имя файла иконки комплекта в JAR-файле;
- `MIDlet-Description` — описание комплекта;
- `MIDlet-Info-URL` — адрес домашней странички комплекта;
- `MIDlet-<n>` — имя, значок и класс каждого из мидлетов в комплекте (обязательный элемент);
- `MIDlet-Jar-URL` — URL-адрес загрузки JAR-файла комплекта;

- `MIDlet-Jar-Size` — размер в байтах JAR-файла комплекта;
- `MIDlet-Data-Size` — минимальный размер сохраняемых приложением данных;
- `MicroEdition-Profile` — версия профиля (обязательный элемент);
- `MicroEdition-Configuration` — версия конфигурации (обязательный элемент).

JAD-файл комплекта MIDlet Suite, подобно файлу манифеста, содержит информацию о мидлете. Наличие JAD-файла в комплекте не является обязательным, однако JAD-файл обеспечивает AMS информацией о содержимом JAR-файла комплекта. На основе этой информации AMS принимает решение запускать или нет данное приложение в устройстве. Кроме того, JAD-файл может содержать различные именованные свойства для мидлета, доступ к которым обеспечивается методом `getAppProperty()` класса `MIDlet`. Данный метод обеспечивает поиск свойств с использованием JAD-файла и файла манифеста.

Пакет `javax.microedition.lcdui` спецификации MIDP 1.0 обеспечивает создание графических пользовательских интерфейсов для MIDP-приложений. Данный пакет не является усеченной графической библиотекой AWT, потому что использование библиотеки AWT предполагает манипулирование окнами, наличие мыши и других свойств, которые отсутствуют в ресурсно-ограниченных устройствах. Графическая библиотека MIDP UI состоит из двух частей: API верхнего уровня и API нижнего уровня.

Приложения, выполненные с использованием библиотеки API верхнего уровня, не определяют визуальные характеристики графических компонентов (форму, цвет, шрифт и т. д.). Они не управляют такими базовыми свойствами, как навигация, прокрутка и др., а также вводом устройства. Все эти свойства реализованы на уровне устройства и поэтому такие приложения легко переносимы среди устройств данного класса. Реализуют API верхнего уровня графической библиотеки MIDP UI подклассы класса `Screen` пакета `javax.microedition.lcdui`.

Приложения, как правило, игровые, выполненные с использованием библиотеки API нижнего уровня, требуют контроля над расположением и визуальными характеристиками графических компонентов, а также управления вводом данных. Классы `Canvas` и `Graphics` реализуют API нижнего уровня. Для такого приложения не гарантируется переносимость, т. к. оно использует специфические для конкретного устройства свойства.

Пакет `javax.microedition.rms` спецификации MIDP 1.0 обеспечивает механизм постоянного хранения Record Management System (RMS). Класс `javax.microedition.rms.RecordStore` представляет хранилище записей (record

store), которое содержит набор записей. Хранилища записей создаются мидлетами из комплекта MIDlet Suite и имеют имена, уникальные в пределах комплекта. Имена хранилищ являются комбинацией из символьного набора Unicode и чувствительны к регистру. Хранилища записей хранятся устройством до тех пор, пока комплект MIDlet Suite не будет из него удален. Записи, относящиеся к конкретному хранилищу, также имеют идентификаторы, уникальные в пределах хранилища. Идентификаторы записей являются первичным ключом для записей и представляют собой целые значения. Хранилище записей создается и/или открывается статическим методом `openRecordStore()` класса `RecordStore`. Методы `addRecord()` и `getRecord()` класса `RecordStore` добавляют записи в хранилище и обеспечивают доступ к записям соответственно.

Спецификация MIDP 2.0 расширяет MIDP 1.0 и поддерживает свойства конфигурации CLDC 1.1.

Устройства, для которых предназначена спецификация MIDP 2.0, должны иметь большую по сравнению с устройствами, в которых реализован профиль MIDP 1.0, доступное дисковое пространство — 256 Кбайт для MIDP-компонентов и оперативную память — 128 Кбайт. Кроме того, они должны иметь возможность воспроизведения звуков.

Спецификация MIDP 2.0 дополняет систему безопасности MIDP 1.0 (песочницу) механизмом проверки приложения и разрешения допуска к защищенным функциям. Непроверенные приложения выполняются в песочнице, однако проверенные приложения могут иметь доступ к дополнительным защищенным функциям. В MIDP 2.0 функции соединения с сетью определены как защищенные.

Проверенный комплект MIDlet Suite — это подписанный электронной подписью комплект с указанной, в элементах JAD-файла `MIDlet-Jar-RSA-SHA1:` и `MIDlet-Certificate-<n>-<m>:`, электронной подписью и доверенным сертификатом публичного ключа. Производителем в устройстве должны быть реализованы защищенные домены (Protection Domains). Защищенный домен включает в себя набор правил для присоединения к домену и разрешений допуска к защищенным функциям. Например, в устройстве могут быть реализованы два домена — для проверенных приложений и для непроверенных приложений. Непроверенное приложение будет инсталлировано в непроверенный домен и при запуске приложения, если будет произведена попытка подключения к сети, устройство должно вывести на экран приглашение для пользователя подтвердить свое согласие для присоединения к сети. Если пользователь даст разрешение для подсоединения к сети, непроверенный домен позволит доступ к защищенным функциям сетевых протоколов. Проверенное приложение будет инсталлировано в проверенный домен и для соеди-

нения с сетью уже не нужно будет подтверждение согласия от пользователя, и соединение будет происходить автоматически. В JAD-файле могут указываться разрешения, которые обязательны для работы приложения, и дополнительные разрешения, с помощью элементов `MIDlet-Permissions:` и `MIDlet-Permissions-opt:` соответственно. Если при инсталляции приложения защищенный домен не будет содержать обязательных для приложения разрешений, инсталляция будет прервана.

Спецификация MIDP 2.0 расширяет пакет `javax.microedition.io` интерфейсами `CommConnection`, `HttpConnection`, `HttpsConnection`, `SecureConnection`, `SecurityInfo`, `ServerSocketConnection`, `SocketConnection`, `UDPDatagramConnection`, определяющими необходимые методы и константы для различного рода соединений, а также представляет класс `PushRegistry`. Класс `PushRegistry` необходим для реализации технологии Push, которая позволяет запускать мидлет в качестве реакции на входящее сетевое соединение. Например, входящее SMS-сообщение может провоцировать запуск мидлета, управляющего соединением. Как уже было сказано, за запуск мидлета отвечает программное обеспечение Application Management Software (AMS), реализованное производителем в устройстве. Для того чтобы AMS произвело запуск мидлета в ответ на входящее соединение, это соединение необходимо зарезервировать для данного мидлета регистрацией.

Соединение может быть зарегистрировано двумя способами. Первый способ — статически с помощью элемента `MIDlet-Push-<n>:` JAD-файла. Второй способ — динамически с помощью метода `registerConnection()` класса `PushRegistry`. После того как данное соединение зарегистрировано для определенного мидлета, невозможна другая регистрация этого же соединения для другого мидлета. Регистрация соединения будет действовать до тех пор, пока устройство не будет перезагружено или не будет удалена динамически методом `unregisterConnection()` класса `PushRegistry`. Класс `PushRegistry` отвечает за создание списка входящих соединений, который может быть получен методом `listConnections()`. Конкретные типы входящих соединений, которые могут быть использованы в технологии Push, зависят от конкретной реализации устройства.

В MIDP 2.0 пакет `javax.microedition.lcdui` расширен интерфейсом `ItemCommandListener`, для обработки команд, связанных с компонентами экрана `Form`, и классом `Spacer` для разделения компонентов экрана `Form`.

В спецификации MIDP 2.0 добавлены пакеты `javax.microedition.pki`, `javax.microedition.media`, `javax.microedition.media.control` и `javax.microedition.lcdui.game`.

Интерфейс `Certificate` пакета `javax.microedition.pki` обеспечивает приложение информацией о сертификате сервера в случае безопасного соединения

с использованием протокола HTTPS и используется для подтверждения информации для безопасного соединения.

Пакеты `javax.microedition.media` и `javax.microedition.media.control` обеспечивают проигрывание простых звуков и аудиофайлов различных форматов, включая контроль над воспроизведением и громкостью.

Пакет `javax.microedition.lcdui.game` содержит классы, упрощающие создание игр. Класс `GameCanvas` создает основу для графического пользовательского интерфейса игры. Классы `Layer`, `LayerManager`, `Sprite` и `TiledLayer` представляют визуальные игровые элементы и обеспечивают их управление.

Спецификация MIDP 2.0 также описывает распространение MIDP-приложений с помощью технологии Over-The-Air (OTA), которая позволяет удаленное управление абонентскими устройствами, находящимися в зоне покрытия сети сотовой связи, и развертывание MIDP-приложений с помощью WAP-шлюза.

Спецификация MIDP 3.0 расширяет MIDP 2.0 и поддерживает свойства как конфигурации CLDC, так и CDC. Устройства, для которых предназначена спецификация MIDP 3.0, имеют следующие минимальные характеристики:

- размер экрана — 176×220, глубина изображения — 16 битов, коэффициент пропорциональности — 1:1;
- один или более механизмов ввода — ввод со стандартной клавиатуры, ввод с помощью клавиатуры сенсорного экрана и т. д.;
- 1 Мбайт жесткой памяти для MIDP-компонентов, 512 Кбайт жесткой памяти для сохраняемых данных приложения и 1 Мбайт оперативной памяти;
- наличие одного или более логического сетевого интерфейса, а также нестабильное подключение к сети с ограниченной пропускной способностью;
- возможность воспроизведения звуков.

Спецификация MIDP 3.0 вводит понятие LIBlet-компонента — используемого совместно MIDP-приложениями программного компонента. LIBlet-компонент представляет собой набор библиотечных классов и связанных с ними ресурсов и не является исполняемым приложением. LIBlet-компоненты могут совместно использовать другие LIBlet. LIBlet-компонент упакован в JAR-файл, состоящий из единственного LIBlet-компонента и файла манифеста. В отличие от комплекта MIDlet Suite, где наличие JAD-файла необязательно, JAR-файл LIBlet-компонента должен иметь связанный с ним JAD-файл. LIBlet-компонент определяется наличием специфических атрибутов в JAD-файле и файле манифеста JAR-файла.

Следующие атрибуты должны быть представлены только в JAD-файле LIBlet-компонента:

- `LIBlet-Jar-URL` — URL-адрес, с помощью которого можно загрузить JAR-файл LIBlet-компонента (обязательный);
- `LIBlet-Jar-Size` — размер JAR-файла LIBlet-компонента (обязательный);
- `LIBlet-Jar-SHA1` — хэш-значение JAR-файла LIBlet-компонента;
- `LIBlet-Dependency-JAD-URL-<n>` — URL-адрес, с помощью которого можно загрузить JAR-файл используемого LIBlet-компонента.

Следующие атрибуты должны присутствовать как в JAD-файле, так и в файле манифеста JAR-файла LIBlet-компонента:

- `LIBlet-Name` — имя LIBlet-компонента (обязательный);
- `LIBlet-Version` — версия LIBlet-компонента (обязательный);
- `LIBlet-Vendor` — производитель LIBlet-компонента (обязательный);
- `LIBlet-Description` — описание LIBlet-компонента;
- `LIBlet-Description-<locale>` — локализованное описание LIBlet-компонента. Например, если системное свойство устройства `microedition.locale` имеет значение `RU`, то AMS будет использовать локализованное значение атрибута на русском языке. Вместо `<locale>` надо подставлять локализацию (`RU`);
- `LIBlet-NonShared-Data-Size` — минимальное количество памяти, необходимое для хранения неиспользуемых совместно другими LIBlet данных LIBlet-компонента;
- `LIBlet-Shared-Data-Size` — минимальное количество памяти, необходимое для хранения используемых совместно другими LIBlet данных LIBlet-компонента;
- `LIBlet-Install-Notify` — URL-адрес, по которому отправляется подтверждение успешной инсталляции LIBlet-компонента;
- `LIBlet-Dependency-<n>` — указывает необходимость в стандартных дополнительных API, в специализированных для данного устройства API, в других LIBlet-компонентах;
- `LIBlet-Permission` — имя класса разрешения, ресурса или действия, которые необходимы для функционирования LIBlet-компонента;
- `LIBlet-Permission-Opt` — имя класса разрешения, ресурса или действия, которые могут понадобиться для функционирования LIBlet-компонента;
- `LIBlet-Dependency-Jar-SHA1-<n>` — хэш-значение JAR-файла необходимого LIBlet-компонента.

Следующие атрибуты должны присутствовать только в файле манифеста JAR-файла LIBlet-компонента:

- `LIBlet-Persistent-Data-URL-<n>` — URL-адрес, с помощью которого могут быть загружены сохраняемые данные LIBlet-компонента;
- `LIBlet-Icon` — значок LIBlet-компонента;
- `LIBlet-Delete-Notify` — URL-адрес, по которому отправляется сообщение о деинсталляции LIBlet-компонента;
- `LIBlet-Font` — список файлов, упакованных в JAR-файл LIBlet-компонента и содержащих шрифты.

Для комплектов MIDlet Suite спецификация MIDP 3.0 вводит следующие дополнительные атрибуты файла манифеста и JAD-файла:

- `MIDlet-Name-<locale>` — локализованное имя мидлета (присутствие в JAD-/манифест-файлах);
- `MIDlet-Description-<locale>` — локализованное описание мидлета (присутствие в JAD-/манифест-файлах);
- `MIDlet-<n>-<locale>` — локализованная версия атрибута (присутствие в JAD-/манифест-файлах);
- `MIDlet-Update-URL` — URL-адрес для автоматического обновления мидлета (присутствие в JAD-/манифест-файлах);
- `MIDlet-Minimum-Canvas-Size` — минимальный размер экрана, необходимый для работы приложения (присутствие в JAD-/манифест-файлах);
- `MIDlet-Maximum-Canvas-Size` — максимальный размер экрана, необходимый для работы приложения (присутствие в JAD-/манифест-файлах);
- `MIDlet-Required-IP-Version` — версия протокола Интернета IP (IPv4 или IPv6), которая необходима для работы приложения (присутствие в JAD-/манифест-файлах);
- `MIDlet-<n>-Type` — определяет тип поведения, реализованный приложением. Возможные значения — `screensaver`, `autostart`, `idlescreen` (присутствие в JAD-/манифест-файлах);
- `MIDlet-<n>-Category` — категория приложения. Возможные значения — `games`, `music`, `video`, `location`, `shop`, `messaging`, `internet`, `calls`, `settings`, `camera`, `office`, `contacts`, `pictures` (присутствие в JAD-/манифест-файлах);
- `MIDlet-<n>-UserDenied` — действия, которые запрещены для выполнения пользователем с мидлетом. Возможные значения — `run`, `stop` (присутствие в JAD-/манифест-файлах);

- `MIDlet-UserDenied` — действия, которые запрещены для выполнения пользователем с комплектом MIDlet Suite. Возможные значения — `update`, `delete` (присутствие в JAD-/манифест-файлах);
- `MIDlet-Operator-Allowed` — ограничивает выполнение для сетей приложений, связанных с защищенным доменом оператора мобильной связи, путем перечисления позволенных кодов страны и кодов мобильной связи (MCC-MNC) (присутствие в JAD-/манифест-файлах). Спецификация MIDP 3.0 определяет следующие категории защищенных доменов: неопознанный сторонний защищенный домен (для неподписанных приложений), опознанный сторонний защищенный домен (для подписанных приложений, имеющих сторонний сертификат), защищенный домен оператора мобильной связи (для приложений, имеющих сертификат сотового оператора), защищенный домен производителя устройства (для приложений, имеющих сертификат производителя устройства);
- `MIDlet-Dependency-<n>` — указывает необходимость в стандартных дополнительных API, в специализированных для данного устройства API, в LIBlet-компонентах (присутствие в JAD-/манифест-файлах);
- `MIDlet-Icon-<locale>` — локализованная версия значка комплекта MIDlet Suite (присутствие в файле манифеста);
- `MIDlet-Scalable-Icon` — значок комплекта MIDlet Suite в SVG-формате (присутствие в файле манифеста);
- `MIDlet-Scalable-Icon-<locale>` — локализованная версия значка комплекта MIDlet Suite в SVG-формате (присутствие в файле манифеста);
- `MIDlet-<n>-Scalable-Icon` — значок мидлета в SVG-формате (присутствие в файле манифеста);
- `MIDlet-<n>-Scalable-Icon-<locale>` — локализованная версия значка мидлета в SVG-формате (присутствие в файле манифеста);
- `MIDlet-Font` — список файлов, упакованных в JAR-файл комплекта MIDlet Suite и содержащих шрифты (присутствие в файле манифеста);
- `MIDlet-Splash-Screen-Image` — список файлов изображений различных форматов, представляющих заставку во время загрузки приложений из комплекта (присутствие в файле манифеста);
- `MIDlet-Event-Launch-<n>` — регистрирует мидлет для загрузки в ответ на событие, отправленное другим мидлетом, или в ответ на событие, связанное с изменением состояния системы (присутствие в файле манифеста);
- `MIDlet-Delete-Confirm-<locale>` — локализованная версия текстового сообщения для подтверждения пользователем удаления комплекта MIDlet Suite (присутствие в файле манифеста);

- `MIDlet-Persistent-Data-URL-<n>` — URL-адрес, с помощью которого могут быть загружены сохраняемые данные приложения (присутствие в файле манифеста);
- `MIDlet-Dependency-JAD-URL-<n>` — URL-адрес, с помощью которого можно загрузить JAR-файл используемого LIBlet-компонента (присутствие в JAD-файле);
- `MIDlet-Permission-<n>` — имя класса разрешения, ресурса или действия, которые необходимы для функционирования комплекта MIDlet Suite (присутствие в JAD-/манифест-файлах);
- `MIDlet-Permission-Opt-<n>` — имя класса разрешения, ресурса или действия, которые могут понадобиться для функционирования комплекта MIDlet Suite (присутствие в JAD-/манифест-файлах);
- `MIDlet-Access-Auth-Type-<n>` — позволяет приложению ограничить доступ к своим совместно используемым другими приложениями ресурсам путем перечисления защищенных доменов, из которых происходит доступ к ресурсам, производителей и сертификатов мидлетов, которые получают доступ к ресурсам (присутствие в JAD-/манифест-файлах);
- `MIDlet-Access-Auth-Cert-<n>` — позволяет приложению ограничить доступ к своим совместно используемым другими приложениями ресурсам путем перечисления сертификатов мидлетов, которые получают доступ к ресурсам (присутствие в JAD-/манифест-файлах);
- `MIDlet-Dependency-Jar-SHA1-<n>` — хэш-значение JAR-файла необходимого LIBlet-компонента (присутствие в JAD-файле);
- `MIDlet-Profile-Request` — возможные значения: `true`, `false`. В случае значения `true` атрибут указывает, что JAD-файл должен загружаться, при использовании распространения MIDP-приложений с помощью технологии Over-The-Air (OTA), обеспечивая профиль клиента с помощью технологии UAProf, которая позволяет серверу получить информацию о конфигурации устройства (присутствие в JAD-файле).

Спецификация MIDP 3.0 расширяет список защищенных библиотек API системы безопасности MIDP 2.0, к которым разрешен доступ для проверенных приложений. Так как профиль MIDP 3.0 поддерживает конфигурацию CDC, то спецификация MIDP 3.0 включает в себя классы и интерфейсы, относящиеся к конфигурации CDC. Поэтому система разрешений защищенных доменов MIDP 3.0 основывается на классе `java.security.Permission`, объекты которого включают список разрешенных действий для объекта. Например, для объекта `java.io.FilePermission` это может быть `"read, write"`, в то время как система разрешений MIDP 2.0 обеспечивает только логические разрешения (`true` или `false`). Каждая защищенная библиотека API MIDP 3.0 имеет

подкласс класса `Permission` для разрешения доступа к своей функциональности. LIBlet-компоненты в отличие от комплектов MIDlet Suite не связываются с каким-то защищенным доменом независимо, они исполняются внутри того домена, к которому относится мидлет, связанный с данными LIBlet-компонентами.

Система безопасности MIDP 3.0 определяет следующие разрешения:

- `javax.microedition.io.HttpProtocolPermission("http://*")` — определено в CLDC;
- `javax.microedition.io.HttpsProtocolPermission("https://*")` — определено в CLDC;
- `javax.microedition.io.DatagramProtocolPermission("datagram://*")` — определено в CLDC;
- `javax.microedition.io.DatagramProtocolPermission("datagram://")` — определено в CLDC;
- `javax.microedition.io.SocketProtocolPermission("socket://*")` — определено в CLDC;
- `javax.microedition.io.SocketProtocolPermission("socket://")` — определено в CLDC;
- `javax.microedition.io.SSLProtocolPermission("ssl://*")` — определено в CLDC;
- `javax.microedition.io.CommProtocolPermission("comm:*")` — определено в CLDC;
- `javax.microedition.io.PushRegistryPermission("*","static,dynamic,alarm")` — модифицированное разрешение MIDP 2.0;
- `java.io.FilePermission(path, "read")` — определено в CDC;
- `java.io.FilePermission(path, "write")` — определено в CDC;
- `java.io.SocketPermission(host, action)` — определено в CDC;
- `java.util.PropertyPermission("microedition.deviceid.*", "read")` — определено в CLDC и CDC;
- `java.util.PropertyPermission("microedition.subscriberid.*", "read")` — определено в CLDC и CDC;
- `java.util.PropertyPermission ("microedition.locale", "read")` — применимо для непроверенных приложений стороннего непроверенного домена;
- `java.util.PropertyPermission("microedition.profile", "read")` — применимо для непроверенных приложений стороннего непроверенного домена;

- `java.util.PropertyPermission("microedition.platform", "read")` — применимо для непроверенных приложений стороннего непроверенного домена;
- `java.util.PropertyPermission("microedition.*", "read")` — применимо для непроверенных приложений стороннего непроверенного домена;
- `javax.microedition.event.EventPermission("*", "read")` — применимо для непроверенных приложений стороннего непроверенного домена;
- `javax.microedition.event.EventPermission("*", "register")` — применимо для непроверенных приложений стороннего непроверенного домена;
- `javax.microedition.event.EventPermission("*.*", "post")` — не применимо для непроверенных приложений стороннего непроверенного домена;
- `java.lang.RuntimePermission("exitVM")` — отсутствует как в проверенном стороннем защищенном домене, так и в непроверенном стороннем защищенном домене;
- `java.lang.RuntimePermission("modifyThread")` — отсутствует как в проверенном стороннем защищенном домене, так и в непроверенном стороннем защищенном домене;
- `java.lang.RuntimePermission("*")` — отсутствует как в проверенном стороннем защищенном домене, так и в непроверенном стороннем защищенном домене;
- `java.lang.reflect.ReflectPermission` — отсутствует как в проверенном стороннем защищенном домене, так и в непроверенном стороннем защищенном домене;
- `java.security.AllPermission` — отсутствует как в проверенном стороннем защищенном домене, так и в непроверенном стороннем защищенном домене;
- `java.security.SecurityPermission` — отсутствует как в проверенном стороннем защищенном домене, так и в непроверенном стороннем защищенном домене;
- `java.security.UnresolvedPermission` — отсутствует как в проверенном стороннем защищенном домене, так и в непроверенном стороннем защищенном домене;
- `java.net.NetPermission` — отсутствует как в проверенном стороннем защищенном домене, так и в непроверенном стороннем защищенном домене;
- `javax.microedition.midlet.ActionsDeniedPermission` — отсутствует как в проверенном стороннем защищенном домене, так и в непроверенном стороннем защищенном домене;

- `javax.microedition.midlet.AutoStartPermission` — отсутствует как в проверенном стороннем защищенном домене, так и в непроверенном стороннем защищенном домене.

Пакет `javax.microedition.midlet` MIDP 3.0, помимо новых классов разрешений `ActionsDeniedPermission` (ограничение действий пользователя с мидлетом или MIDlet Suite) и `AutoStartPermission` (разрешение на автостарт мидлета), содержит новый класс `MIDletIdentity`, обеспечивающий идентификацию мидлета и используемый для коммуникации между MIDP-приложениями из комплекта MIDlet Suite с помощью нового пакета `javax.microedition.event`.

Пакет `javax.microedition.event` MIDP 3.0 обеспечивает обмен событиями между MIDP-приложениями из комплекта MIDlet Suite и получение событий, связанных с изменениями состояния системы. Пакет `javax.microedition.event` содержит классы `EventData`, `EventManager`, `EventPermission` и интерфейс `EventDataListener`. Класс `EventData` отвечает за создание события, а также за получение информации, связанной с событием. Поля класса `EventData` определяют системные события, такие как уровень зарядки батареи устройства, нахождение устройства в каком-либо режиме, наличие входящего или исходящего вызовов телефона и др. Класс `EventManager` снабжает методами, необходимыми приложению для того, чтобы послать и получить событие. Кроме того, с помощью метода `registerApplication()` класса `EventManager` можно динамически зарегистрировать приложение для автоматической загрузки в качестве реакции на событие (статическая регистрация обеспечивается атрибутом `MIDlet-Event-Launch-<n>` файла манифеста). Метод `addEventListener()` класса `EventManager` позволяет зарегистрироваться для получения определенного события. В качестве аргумента метод `addEventListener()` использует объект `EventDataListener` — интерфейса, содержащего метод `handleEvent()`, который вызывается в случае получения события.

Также, спецификация MIDP3.0 расширяет пакет `javax.microedition.lcdui` следующими классами и интерфейсами:

- `AnimatedImage` — класс, позволяющий выводить на экран анимированные изображения;
- `CanvasItem` — суперкласс для таких подклассов, как `TextEditor`, который представляет редактируемый текстовый компонент, принадлежащий объекту `Canvas` или `CustomItem`;
- `FileSelector` — класс позволяет пользователю выбрать каталог или файл файловой системы устройства для загрузки или для сохранения файла;
- `FormLayoutPolicy` — абстрактный класс, подклассы которого должны обеспечивать компоновку элементов экрана `Form`;
- `IdleItem` — класс для создания изображения экрана в режиме ожидания;

- `Menu` — класс, представляющий визуальный контейнер меню;
- `Notification` — класс, позволяющий вывести небольшое информационное сообщение пользователю;
- `NotificationType` — класс представляет типы информационных сообщений пользователю и служит для группировки, сортировки и фильтрации объектов `Notification` одного типа;
- `ScalableImage` — класс, представляющий изображения векторной графики (SVG);
- `TabbedPane` — класс обеспечивает для пользователя экранную навигацию для последовательности объектов `Screen`;
- `TableLayoutPolicy` — класс компоновки элементов экрана `Form` в виде таблицы;
- `Text` — класс для компоновки и вывода текста на экран;
- `TextEditor` — класс представляет редактируемый текстовый компонент;
- `CommandLayoutPolicy` — интерфейс, используемый для реализации точного расположения команд;
- `DisplayListener` — интерфейс содержит методы, вызываемые в качестве реакции на события, связанные с объектами `Display`;
- `ItemLayoutHint` — классы, реализующие данный интерфейс, содержат правила контроля компоновки элементов;
- `ItemTraversalListener` — интерфейс обработки событий, связанных с изменением фокуса для элементов экрана `Form`;
- `KeyListener` — интерфейс обработки событий, связанных с клавиатурой устройства;
- `NotificationListener` — интерфейс обработки событий, связанных с изменением состояния объекта `Notification`;
- `TabListener` — интерфейс обработки событий, связанных с изменением состояния объекта `TabbedPane`;
- `TextEditorChangeListener` — интерфейс обработки событий, связанных с изменением состояния объекта `TextEditor`;
- `DisplayCapabilityException` — класс исключения, указывающий, что возможности экрана недостаточны для выполняемых операций;
- `FontFormatException` — класс исключения, указывающий, что свойства шрифта не поддерживаются;
- `NotificationException` — ошибка вывода информационного сообщения для пользователя.

Пакет `javax.microedition.io` MIDP 3.0 включает в себя новый класс `PushRegistryPermission` — класс разрешения для реализации технологии Push, а также новые интерфейсы `IMCConnection` и `IMCServerConnection`. Интерфейс `IMCConnection` определяет исходящее (клиент-серверное) низкоуровневое асинхронное, двунаправленное потоковое соединение для протокола Inter-MIDlet Communication (IMC). Интерфейс `IMCServerConnection` определяет входящее (серверное) соединение для протокола Inter-MIDlet Communication (IMC).

Спецификация MIDP 3.0 представляет протокол Inter-MIDlet Communication (IMC), который обеспечивает прямое соединение между MIDP-приложениями. В отличие от связи "MIDlet — LIBlet", действующей в общей для данного мидлета и LIBlet-компонента изолированной среде выполнения, IMC-протокол позволяет MIDP-приложению получить доступ к совместно используемому компоненту, принадлежащему другой среде выполнения.

Спецификация MIDP 3.0 расширяет библиотеку MIDP 2.0 Media API, представленную пакетом `javax.microedition.media` и являющуюся частью спецификации Mobile Media API (JSR-135), до полной поддержки спецификации JSR 135. Пакет `javax.microedition.media` MIDP 3.0 обеспечивает воспроизведение как аудиотреков, так и видеоклипов, включая необходимые функции управления.

Пакет `javax.microedition.rms` MIDP 3.0 содержит новый класс `RecordStoreInfo`, объект которого представляет информацию о связанном с ним объекте `RecordStore`.

Пакет `java.lang` MIDP 3.0 расширен по сравнению с MIDP 2.0 классами и интерфейсами конфигурации CDC.

Дополнительные пакеты технологии Java ME

Дополнительные пакеты технологии Java ME расширяют среду выполнения приложений поддержкой специфических возможностей, таких как:

- использование технологии Bluetooth. Библиотека Java APIs for Bluetooth, спецификация JSR 82, основывается на конфигурации CLDC;
- отправка и получение SMS- и MMS-сообщений. Библиотека Wireless Messaging API (WMA), спецификация JSR 120 — поддержка бинарных и текстовых сообщений, спецификация JSR 205 — поддержка бинарных, текстовых и составных сообщений, может быть реализована как в CLDC/MIDP-устройствах, так и в CDC-устройствах;

- доступ к персональным данным, таким как адресная и записная книжки, обеспечивается библиотекой Personal Information Management (PIM) API, спецификация JSR 75, реализуется в CLDC-устройствах;
- доступ к файловым системам устройства и карт памяти устройства, обеспечивается библиотекой FileConnection API, спецификация JSR 75, реализуется в CLDC-устройствах;
- использование SIP/SIMPLE-протокола для организации системы мгновенной передачи сообщений (JSR 165 SIMPLE Instant Messaging API) и уведомления о присутствии (JSR 164 SIMPLE Presence API);
- доступ к базам данных и другим источникам данных. Библиотека JDBC API, спецификация JSR 169, реализуется в CDC/FP-устройствах;
- доступ к удаленным SOAP/XML Web-сервисам и разбор XML-данных. Библиотека J2ME Web Services, спецификация JSR 172, предназначена для CDC/CLDC-устройств;
- безопасное хранение таких данных, как персональные и публичные ключи, персональная информация и т. д. Криптографические операции для поддержки платежных протоколов, целостности и конфиденциальности данных. Обеспечение безопасной среды выполнения. Библиотека Security and Trust Services API for J2ME (SATSA), спецификация JSR 177. Образец реализации SATSA 1.0 в наборе с CLDC 1.0.4 RI и MIDP 2.0 RI;
- получение информации о текущем географическом месторасположении устройства для использования в приложениях, основанных на локализации. Библиотека Location API for J2ME, спецификация JSR 179, предназначена для реализации в CDC/CLDC1.1-устройствах;
- использование Session Initiation Protocol (SIP) протокола для создания приложений, реализующих обмен мгновенными сообщениями, чаты, игры и т. д. Библиотека Session Initiation Protocol (SIP) API for J2ME, спецификация JSR 180, предназначена для CDC/CLDC-устройств;
- использование 3D-графики в приложениях. Библиотека Mobile 3D Graphics API for J2ME, спецификация JSR 184, реализуется в CDC/CLDC-устройствах;
- ведение статистики сетевых событий, которая необходима в различных областях, таких как рынок мобильных игр, когда мониторинг и отслеживание событий включает в себя уведомление о загрузки пользователями мобильных игр, что позволяет получать информацию о наиболее популярных играх, или использование телевизионных приставок, где мониторинг событий позволяет провайдеру изучать зрительские предпочтения. Библиотека Event Tracking API for J2ME, спецификация JSR 190, реализуется в CDC/CLDC-устройствах;

- возможность в приложении Java ME вызова другого зарегистрированного приложения, необязательно Java-приложения, для обработки определенного типа данных. Например, при получении сообщения с прикрепленным аудиофайлом приложение может запустить проигрыватель аудиофайлов устройства. Библиотека Content Handler API, спецификация JSR 211, реализуется в CDC/CLDC-устройствах;
- использование 2D векторной графики в SVG-формате. Библиотека 2D Scalable Vector Graphics API for J2ME, спецификация JSR 226, предназначена для устройств CDC/CLDC 1.1;
- возможность интернационализации приложений Java ME путем использования локальных ресурсов. Библиотека Mobile Internationalization API, спецификация JSR 238, реализуется в CDC/CLDC-устройствах;
- поддержка аудио- и видеоконтента, включая необходимые функции управления. Библиотека Mobile Media API, спецификация JSR 135, реализуется в CDC/CLDC-устройствах;
- поддержка графических библиотек OpenGL ES API для 2D- и 3D-графики и EGL API. Библиотека Java Binding for the OpenGL ES® API, спецификация JSR 239, минимальные требования — CLDC 1.1/MIDP 2.0 или CDC 1.1.2/PBP 1.1.2;
- использование возможностей библиотек Swing, Java 2D Graphics, Imaging, и Image I/O. Библиотека Advanced Graphics and User Interface (AGUI) API, спецификация JSR-209, предназначена для Personal Basis Profile 1.1;
- возможность создания распределенных приложений с помощью механизма удаленного вызова методов, позволяющего приложениям, развернутым на различных платформах, взаимодействовать друг с другом. Библиотека RMI Optional Package (RMI OP), спецификация JSR 66, реализуется в CDC-устройствах;
- контроль над функциями телевизионного ресивера. Библиотека Java TV API, спецификация JSR 927;
- полная поддержка обработки XML-документов, используемая в таких приложениях, как Web-сервисы, XML-обмен сообщениями, Web-браузеры и т. д. Библиотека XML API for Java™ ME, спецификация JSR 280, предназначена для устройств CDC/CLDC 1.1.

Технология CDC

Конфигурация Connected Device Configuration (CDC) предназначена для устройств с большими по сравнению с CLDC-устройствами возможностями, таких как мощные карманные персональные компьютеры и телевизионные

приставки. Устройства, в которых реализована конфигурация CLDC, имеют 32-битный микропроцессор/контроллер, 2 Мбайт RAM и 2,5 Мбайт ROM.

Реализация конфигурации CDC обеспечивает полную поддержку языка программирования Java и виртуальной машины Java платформы Java SE.

CDC-технология содержит:

- спецификации CDC 1.0 (JSR 36) и CDC 1.1.2 (JSR 218);
- образцы реализации: CDC reference implementation 1.0.2 — соответствует CDC 1.0 и CDC reference implementation 1.1.2 — соответствует CDC 1.1.2;
- оптимизированную реализацию CDC HotSpot Implementation, которая включает в себя новый динамический компилятор, более надежную поддержку многопоточности и многозадачности, а также быструю расширяемость с поддержкой новых процессоров и операционных систем;
- набор автоматических тестов CDC TCK на соответствие спецификации.

Библиотеки CDC 1.0 API основываются на J2SE 1.3 API и полностью включают в себя все библиотеки конфигурации CLDC 1.0.

Библиотеки CDC 1.1.2 API основываются на J2SE 1.4.2 API и полностью включают в себя все библиотеки конфигурации CLDC 1.1. Основные отличия спецификации CDC 1.1.2 от спецификации CDC 1.0 — это улучшения в системе безопасности, поддержка стандарта Unicode 3.0, протокола IPv6 и утверждений `assertion` языка программирования Java.

Технологии Foundation, Personal Basis Profile и Personal Profile

Для конфигурации CDC существуют три профиля: Foundation Profile, Personal Basis Profile и Personal Profile, которые в сочетании с CDC обеспечивают полную среду выполнения для Java-приложений в определенном классе устройств.

Технология Foundation включает в себя:

- спецификации Foundation Profile 1.0 (JSR 46) и Foundation Profile 1.1.2 (JSR 219), являющиеся основой для остальных профилей. Спецификация Foundation Profile не включает в себя графические библиотеки AWT и Swing и поэтому предназначена для встроенных устройств, для которых нет необходимости в графическом интерфейсе пользователя;
- образцы реализации FP reference implementation 1.0.2 (основывается на Foundation Profile 1.0.), FP reference implementation 1.1.2 (основывается на Foundation Profile 1.1.2), FP reference implementation 1.1.2 с дополнительным пакетом Security Optional Package;

- набор автоматических тестов Foundation Profile TCK на соответствие спецификации;
- оптимизированную реализацию, основанную на CDC HotSpot Implementation.

Спецификация Foundation Profile 1.0 основывается на конфигурации CDC 1.0 и дополняет ее пакеты `java.io`, `java.lang`, `java.net`, `java.security`, `java.text`, `java.util`, `javax.microedition.io`.

Спецификация Foundation Profile 1.1.2 основывается на конфигурации CDC 1.1.2 и дополняет ее пакеты `java.io`, `java.lang`, `java.math`, `java.net`, `java.security`, `java.text`, `java.util`, `javax.microedition.io`, `javax.microedition.pki`.

Технология Personal Basis Profile содержит:

- спецификации Personal Basis Profile 1.0 (JSR 129) и Personal Basis Profile 1.1 (JSR 217);
- образцы реализации — PBP reference implementation 1.0.2 (основывается на Personal Basis Profile 1.0), PBP reference implementation 1.1 (основывается на Personal Basis Profile 1.1);
- набор автоматических тестов Personal Basis Profile TCK на соответствие спецификации.

Спецификация Personal Basis Profile (PBP) расширяет Foundation Profile платформой для создания графического пользовательского интерфейса (неполная графическая библиотека AWT, пакет `java.awt`), а также поддержкой модели программирования компонентов JavaBeans (пакет `java.beans`) и приложений Xlet (пакеты `java.rmi` и `javax.microedition.xlet`).

Профиль PBP предназначен для устройств, требующих наличие графического пользовательского интерфейса, реализованного с помощью неполного набора возможностей графической библиотеки AWT, таких как телевизионные приставки и видеокамеры. Приложения Xlet представляют похожую модель программирования с Java-апплетами и мидлетами и предназначены для цифрового интерактивного телевидения. Так же как и мидлеты, Xlet управляются Application Management Software (AMS), и главный класс Xlet-приложения должен реализовывать интерфейс `javax.tv.xlet.Xlet`. Таким образом, профиль PBP поддерживает две модели программирования приложений: Xlet-приложений и традиционную модель, которая основывается на главном классе, содержащем статический метод `main()`.

Спецификация Personal Basis Profile 1.0 основывается на Foundation Profile 1.0.

Спецификация Personal Basis Profile 1.1 основывается на Foundation Profile 1.1, а также включает в себя расширенный по сравнению с PBP 1.0 набор библиотек AWT.

Технология Personal Profile содержит:

- спецификации Personal Profile 1.0 (JSR 62) и Personal Profile 1.1 (JSR 216);
- образцы реализации — PP reference implementation 1.0 (соответствует Personal Profile 1.0), PP reference implementation 1.1 (соответствует Personal Profile 1.1);
- набор автоматических тестов Personal Profile TCK на соответствие спецификации.

Спецификация Personal Profile (PP) расширяет Personal Basis Profile полной поддержкой графической библиотеки AWT, а также поддержкой модели программирования Java-апплетов (пакет `java.applet`). Профиль PP предназначен для устройств, требующих наличие развитого графического интерфейса пользователя с поддержкой апплетов, таких как мощные персональные карманные компьютеры.

Спецификация Personal Profile 1.0 основывается на Personal Basis Profile 1.0.

Спецификация Personal Profile 1.1 основывается на Personal Basis Profile 1.1.

Графическая библиотека Light Weight User Interface Toolkit (LWUIT)

Графическая библиотека Light Weight User Interface Toolkit (LWUIT) предназначена для мобильных устройств с поддержкой CLDC1.1 MIDP2.0/CDC PBP/SE и позволяет добавлять графические компоненты, обеспечивая привлекательный внешний вид и поведение графического интерфейса пользователя мобильных Java-приложений.

Домашняя страничка открытого проекта LWUIT находится по адресу **https://lwuit.dev.java.net/**.

Интерфейс пользователя мобильного Java-приложения, созданный на основе графической библиотеки LCDUI профиля MIDP, имеет внешний вид и поведение, зависящие от конкретной реализации устройства, и может иметь отличия в шрифте, компоновке графических компонентов и т. д. Графическая библиотека LWUIT позволяет легко создавать интерфейс пользователя мобильных Java-приложений, имеющий одинаковый внешний вид и поведение для всех устройств.

Архитектура библиотеки LWUIT имеет много общего с архитектурой графической библиотеки Swing.

Подобно Swing, библиотека LWUIT содержит базовый набор графических компонентов, менеджеров компоновки и механизм обработки событий. Кроме того, для таких компонентов, как `List`, `Table` и `Tree`, в библиотеке LWUIT, так же как и в Swing, данные отделяются от внешнего вида и поведения, реализуя архитектуру MVC. Также библиотека LWUIT содержит класс `LookAndFeel`, позволяющий разработчику создавать новый внешний вид и поведение графического интерфейса пользователя.

Изменение внешнего вида и поведения графического интерфейса пользователя мобильного Java-приложения, созданного на основе библиотеки LWUIT, возможно динамически, во время выполнения, с помощью специального конфигурационного файла, контролирующего Look and Feel приложения.

Механизм обработки событий библиотеки LWUIT обеспечивает для всех графических компонентов поддержку обработки событий сенсорного экрана мобильного устройства.

Классы `Motion`, `Transition`, `CommonTransitions` и интерфейс `Animation` библиотеки LWUIT позволяют создавать различные визуальные эффекты анимации и перемещений.

Классы `M3G` и `Transition3D` библиотеки LWUIT дают возможность встраивать 3D-эффекты в графический интерфейс пользователя.

Класс `Dialog` обеспечивает интерфейс приложения возможностью отображения диалогового окна для запроса информации у пользователя.

Помимо всего прочего, библиотека LWUIT содержит инструмент Theme Creator, позволяющий редактировать и создавать различного рода ресурсы для графического интерфейса, включая фоновые и другие графические элементы, шрифты, не поддерживаемые изначально конкретным мобильным устройством, а также ресурсы, обеспечивающие локализацию/интернационализацию приложения.

ГЛАВА 6

Создание Java-приложений на основе платформы Java ME

Для создания Java-приложений на основе платформы Java ME наиболее популярны две среды разработки — Java Platform Micro Edition Software Development Kit (Java ME SDK), графический интерфейс которой создан на основе платформы NetBeans, и среда NetBeans IDE, включающая в себя среду Netbeans Mobility Pack. И Java ME SDK, и Netbeans Mobility Pack основываются на наборе инструментов Sun Java Wireless Toolkit, содержащем эмуляторы, имитирующие среду выполнения устройства.

Здесь мы рассмотрим использование среды NetBeans для разработки мобильных Java-приложений.

Среда NetBeans обеспечивает создание кода, тестирование и развертывание приложений для мобильных устройств, в которых реализована технология Java ME. Также среда NetBeans предоставляет набор различных шаблонов, упрощающих создание мобильных приложений, позволяя устанавливать конфигурацию для проекта в его свойствах, что дает возможность осуществлять сборку проекта для широкого диапазона устройств, решая проблему фрагментации устройств (device fragmentation).

> ***Примечание***
>
> Device fragmentation — это термин, обозначающий различия в реализации технологии Java ME в разных устройствах, что препятствует автоматической переносимости мобильного приложения. Различия могут быть как физического характера — размер экрана, памяти и т. д., так и в исполняемых средах приложений — разные сочетания конфигураций, профилей и дополнительных пакетов.

Установка конфигурации проекта Java ME приложения в среде NetBeans позволяет осуществить его сборку для различных сред выполнения и физических возможностей устройств.

Свойства конфигурации проекта Java ME приложения включают в себя:

- платформу эмулятора, позволяющую тестировать переносимость приложения;
- сочетание конфигурации, профиля и дополнительных пакетов;
- разрешения доступа к защищенным библиотекам API;
- установки регистрации Push, которая позволяет мидлету быть запущенным в ответ на входящее сетевое соединение;
- уровень obfuscation (запутывания). Obfuscation — процесс, который усложняет декомпиляцию класс-файлов приложения и уменьшает их размер. Obfuscation производится за счет замены имен пакетов, классов, методов, полей на короткие идентификаторы. Целью запутывания является защита авторских прав и уменьшение размеров работающего кода. По умолчанию среда NetBeans в качестве "запутывателя" использует инструмент ProGuard (домашняя страничка — **http://proguard.sourceforge.net/**);
- установки для электронной подписи и защищенного домена приложения;
- установки для создания Javadoc-документации приложения;
- установки методов развертывания приложения. Развертывание приложения — это процесс передачи созданного в среде разработки приложения в мобильное устройство или сервер для дальнейшего тестирования или внедрения.

Для CLDC/MIDP-приложений среда NetBeans предоставляет графический редактор Visual Mobile Designer (VMD), обеспечивающий визуальную разработку рабочего процесса приложения и графического интерфейса пользователя. Кроме того, среда NetBeans содержит шаблоны для создания клиентских мидлетов, которые могут соединяться с Web-сервисами.

Для облегчения тестирования приложения для различных конфигураций среда NetBeans позволяет включать в код директивы препроцессора, которые модифицируют исходный код перед компиляцией для соответствия конкретной конфигурации. Среда NetBeans поддерживает все директивы препроцессора Antenna (домашняя страничка — **http://antenna.sourceforge.net/**).

Для решения проблемы фрагментации устройств можно также использовать возможности (abilities) — набор идентификаторов, определяющих поддержку физических и программных возможностей для устройства. Возможности можно ассоциировать с конфигурациями и использовать их в директивах препроцессора.

Для CDC-приложений среда NetBeans предоставляет графический редактор GUI Builder для визуальной разработки графического интерфейса пользователя.

В процессе сборки проекта среда NetBeans с помощью соответствующего инструмента обеспечивает preverification — предварительную проверку класс-файлов после компиляции. Проверка класс-файлов является процессом проверки истинности класс-файлов Java в целях безопасности и отклоняет неправильные файлы. Проверка класс-файлов существует и в Java SE, где она осуществляется полностью виртуальной машиной Java. Так как данный процесс требует значительных ресурсов, то для мобильных устройств он разделен на две фазы. Первая фаза — предварительная проверка осуществляется вне устройства, при этом в класс-файлы записываются специальные атрибуты, которые затем используются виртуальной машиной Java устройства во второй фазе — окончательной проверки.

Пример создания приложения для чтения TXT-файлов с использованием высокоуровневой графической библиотеки

В качестве примера приложения Java ME рассмотрим создание MIDP-приложения, позволяющего пользователю читать содержимое текстовых файлов — книги, хранящихся в формате TXT в файловой системе сотового телефона или карманного персонального компьютера.

> ***Примечание***
>
> Файлы с кодами приложения расположены на прилагаемом к книге компакт-диске в папке Часть_II\Глава_6\JBookReader.

Для создания MIDP-приложения в среде NetBeans в меню **File** выберем **New Project | Categories: Java ME | Mobile Application**. Далее введем имя проекта JBookReader и, сбросив флажок **Create Hello MIDlet**, нажмем кнопку **Finish**. В окне **Projects** щелкнем правой кнопкой мыши на названии проекта и выберем пункт **Properties**.

Доступ к файловой системе устройства обеспечивает дополнительный пакет FileConnection API (спецификация JSR 75), который основан на конфигурации CLDC 1.0. Вдобавок, для того чтобы в полной мере воспользоваться классами библиотеки `org.netbeans.microedition`, которая представляет компоненты палитры графического редактора Visual Mobile Designer среды NetBeans, необходим профиль MIDP 2.0. Поэтому в диалоговом окне **Properties** в категории **Platform** установим переключатели **Device Configuration: CLDC 1.0**, **Device Profile: MIDP 2.0** и **Optional Packages: File Connection and PIM Optional Packages 1.0**. При этом платформа эмулятора должна быть Sun Java Wireless Toolkit for CLDC.

Нажмем кнопку **OK** и в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на узле **Source Packages** и выберем **New | Other | Categories: Java | Java Package**. Введем имя пакета jbookreader и нажмем кнопку **Finish**. Далее точно так же щелкнем правой кнопкой мыши на узле **Source Packages** и выберем **New | Other | Categories: MIDP | Visual MIDlet**. Введем имя MIDlet JBookReader и нажмем кнопку **Finish**.

В результате среда NetBeans создаст основу приложения и откроет нам окно **Flow** рабочего потока приложения (рис. 6.1).

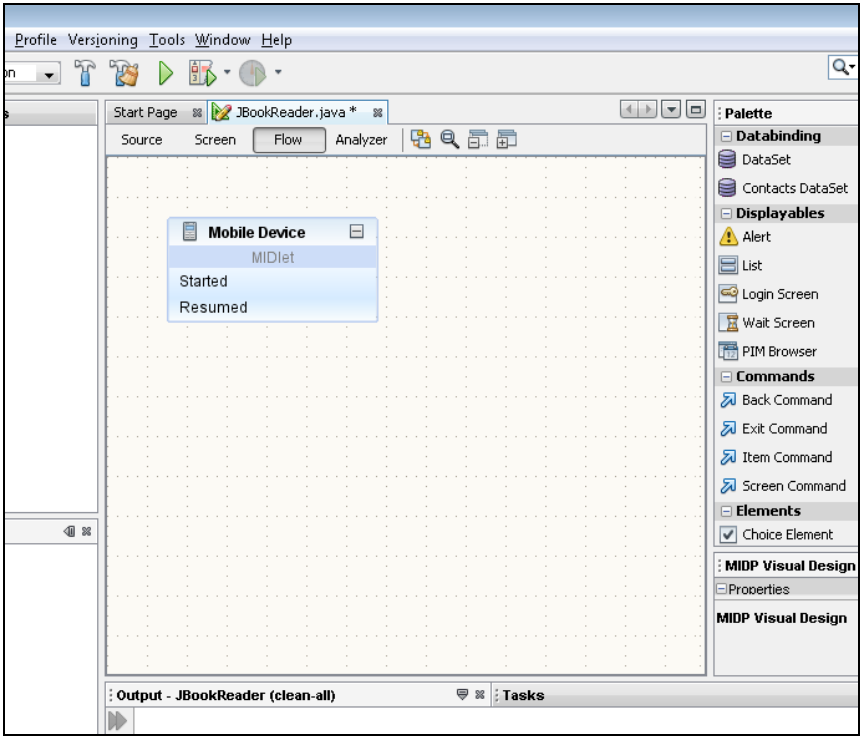

Рис. 6.1. Окно **Flow** рабочего потока приложения

С помощью палитры компонентов **Palette** перенесем в окно **Flow** следующие компоненты.

- **Splash Screen** — класс `org.netbeans.microedition.lcdui.SplashScreen`, созданный на основе класса `javax.microedition.lcdui.Canvas` MIDP-профиля и представляющий заставку, отображаемую при загрузки приложения. Заставка отображается по умолчанию 5000 миллисекунд, и затем запускается команда `DISMISS_COMMAND`. Щелкнув правой кнопкой мыши на компоненте в окне **Flow** и открыв окно **Properties**, можно установить за-

головок, время отображения, текст и изображение заставки. Кроме того, можно установить полноэкранный режим отображения заставки, когда не отображаются команды, заголовок и бегущая строка (метод `setFullScreenMode()` класса `Canvas` профиля MIDP 2.0).

- **File Browser** — класс `org.netbeans.microedition.lcdui.pda.FileBrowser`, использует FileConnection Optional Package (JSR 75) и представляет экран навигации файловой системы устройства. В случае выбора пользователем файла запускается команда `SELECT_FILE_COMMAND`. С помощью палитры компонентов добавим в компонент `fileBrowser` команду **Exit Command** для того, чтобы была возможность выхода из экрана **File Browser**.
- **Wait Screen** — класс `org.netbeans.microedition.lcdui.WaitScreen`, расширяющий класс `javax.microedition.lcdui.Canvas` MIDP 2.0 и представляющий заставку ожидания, отображаемую во время выполнения фоновой задачи. В случае успешного выполнения задачи запускается команда `SUCCESS_COMMAND`, в случае возникновения ошибки — команда `FAILURE_COMMAND`. В окне **Properties** среды NetBeans можно установить заголовок, полноэкранный режим, текст, бегущую строку, фоновую задачу и изображение заставки ожидания.
- **Alert** — класс `javax.microedition.lcdui.Alert` профиля MIDP 1.0 представляет экран, отображающий сообщение пользователю. В окне **Properties** можно установить заголовок, текст, бегущую строку, тип, индикатор прогресса, изображение и время отображения экрана сообщения. В случае установки времени отображения `FOREVER` экран отображается до тех пор, пока пользователь сам его не закроет. При открытии экрана сообщения проигрывается звук, который зависит от того, какой тип сообщения установлен. С помощью окна **Properties** компонента **Alert** установим бегущую строку "ERROR!" экрана сообщения (свойство **Ticker**).
- **Text Box** — класс `javax.microedition.lcdui.TextBox` профиля MIDP 2.0 представляет экран, отображающий редактируемый текст. В окне **Properties** можно установить заголовок, бегущую строку, первоначальный текст, максимальное количество отображаемых символов, ограничения для ввода символов — если установлено значение `ANY`, то пользователь может вводить любой текст, первоначальный режим ввода, определяющий набор используемых при редактировании текста символов Unicode (данная опция зависит от конкретной реализации устройства). С помощью палитры компонентов добавим в компонент `textBox` команды **Exit** и **Item Command**, которые переименуем в `exitCommandRead`, `itemCommandNextPage` и `itemCommandPreviousPage`.
- **Entry Point** — компонент графического редактора Visual Mobile Designer, представляющий метод создаваемого класса. Создадим два метода: `methodRead()` и `methodReadTXT()`.

После добавления вышеперечисленных компонентов соединим их действиями в окне **Flow** (рис. 6.2).

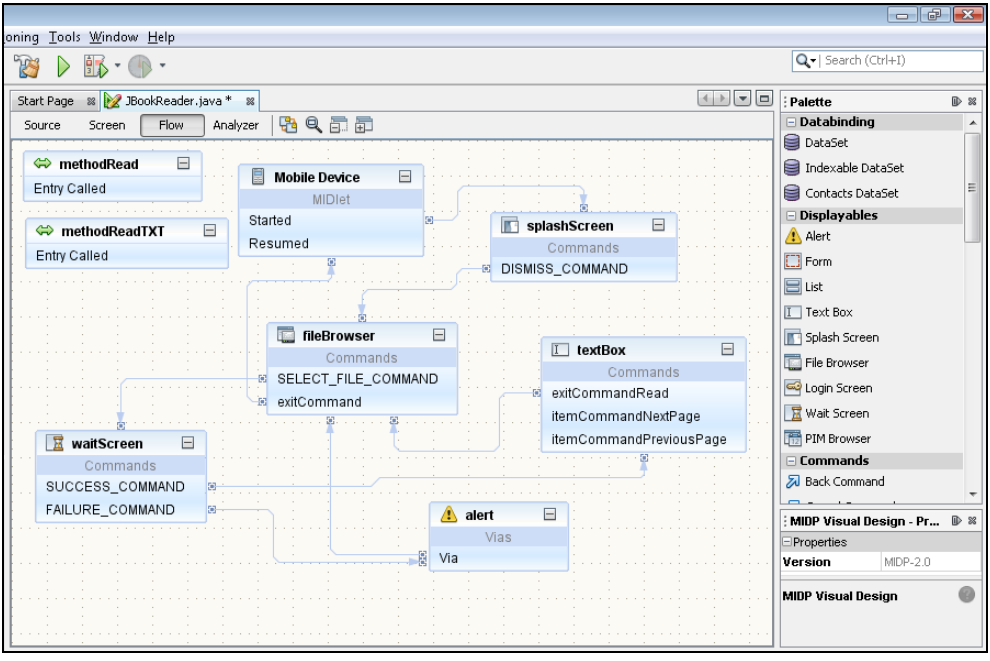

Рис. 6.2. Компоненты рабочего потока приложения

В случае возникновения проблем с соединением компонентов с помощью мыши соединения можно редактировать, используя окно **Properties** (рис. 6.3).

В результате создания рабочего потока приложения среда NetBeans сгенерировала соответствующий код класса `JBookReader`, который мы дополним таким образом, чтобы на экране мобильного телефона осуществлялось отображение выбранного пользователем текстового файла, и чтение электронной книжки было постраничным (листинг 6.1).

Листинг 6.1. Код класса `JBookReader`

```
package jbookreader;
import java.io.InputStream;
import java.io.InputStreamReader;
import javax.microedition.io.file.FileConnection;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
```

```
import org.netbeans.microedition.lcdui.SplashScreen;
import org.netbeans.microedition.lcdui.WaitScreen;
import org.netbeans.microedition.lcdui.pda.FileBrowser;
import org.netbeans.microedition.util.SimpleCancellableTask;
public class JBookReader extends MIDlet implements CommandListener {
  private boolean midletPaused = false;
  private SplashScreen splashScreen;
  private FileBrowser fileBrowser;
  private WaitScreen waitScreen;
  private TextBox textBox;
  private Alert alert;
  private Command exitCommand;
  private Command itemCommandNextPage;
  private Command exitCommandRead;
  private Command itemCommandPreviousPage;
  private SimpleCancellableTask task;
  private Ticker tickerError;
  private Font fontSplash;
  private long instrSkip;
/**
  * The JBookReader constructor.
  */
  public JBookReader() {  }
  private void initialize() {
    instrSkip=0;
  }
  public void startMIDlet() {
    switchDisplayable(null, getSplashScreen());
  }
  public void resumeMIDlet() {  }
  public void switchDisplayable(Alert alert, Displayable nextDisplayable)
{
    Display display = getDisplay();
    if (alert == null) {
      display.setCurrent(nextDisplayable);
    } else {
      display.setCurrent(alert, nextDisplayable);
    }
  }
  public void commandAction(Command command, Displayable displayable) {
    if (displayable == fileBrowser) {
      if (command == FileBrowser.SELECT_FILE_COMMAND) {
        switchDisplayable(null, getWaitScreen());
```

```
    } else if (command == exitCommand) {
      exitMIDlet();
    }
  } else if (displayable == splashScreen) {
    if (command == SplashScreen.DISMISS_COMMAND) {
       switchDisplayable(null, getFileBrowser());
    }
  } else if (displayable == textBox) {
    if (command == exitCommandRead) {
      switchDisplayable(null, getFileBrowser());
    } else if (command == itemCommandNextPage) {
      instrSkip=instrSkip+textBox.getMaxSize();
      switchDisplayable(null, getWaitScreen());

    } else if (command == itemCommandPreviousPage) {
       instrSkip=instrSkip-textBox.getMaxSize();
       switchDisplayable(null, getWaitScreen());
    }
  } else if (displayable == waitScreen) {
    if (command == WaitScreen.FAILURE_COMMAND) {
      switchDisplayable(getAlert(), getFileBrowser());
    } else if (command == WaitScreen.SUCCESS_COMMAND) {
      switchDisplayable(null, getTextBox());
    }
  }
}
public SplashScreen getSplashScreen() {
  if (splashScreen == null) {
    splashScreen = new SplashScreen(getDisplay());
    splashScreen.setTitle("Splash Screen");
    splashScreen.setCommandListener(this);
    splashScreen.setText("Welcome to the Java Book Reader!");
    splashScreen.setTextFont(getFontSplash());
    splashScreen.setTimeout(2000);
  }
  return splashScreen;
}
public FileBrowser getFileBrowser() {
  if (fileBrowser == null) {
    fileBrowser = new FileBrowser(getDisplay());
    fileBrowser.setTitle("File Browser: Select Book");
    fileBrowser.setCommandListener(this);
    fileBrowser.addCommand(FileBrowser.SELECT_FILE_COMMAND);
    fileBrowser.addCommand(getExitCommand());
  }
```

```
    return fileBrowser;
  }
  public WaitScreen getWaitScreen() {
    if (waitScreen == null) {
      waitScreen = new WaitScreen(getDisplay());
      waitScreen.setTitle("Wait Screen");
      waitScreen.setCommandListener(this);
      waitScreen.setText("Please Waite...");
      waitScreen.setTask(getTask());
    }
    return waitScreen;
  }
  public Alert getAlert() {
    if (alert == null) {
      alert = new Alert("ERROR", "Cannot open the file!", null,
                        AlertType.ERROR);
      alert.setTicker(getTickerError());
      alert.setTimeout(Alert.FOREVER);
    }
    return alert;
  }
  public TextBox getTextBox() {
    if (textBox == null) {
      textBox = new TextBox("Book Reader", null, 1024, TextField.ANY);
      textBox.addCommand(getExitCommandRead());
      textBox.addCommand(getItemCommandNextPage());
      textBox.addCommand(getItemCommandPreviousPage());
      textBox.setCommandListener(this);
      textBox.setInitialInputMode("UCB_BASIC_LATIN");
    }
    return textBox;
  }
  public Command getExitCommand() {
    if (exitCommand == null) {
      exitCommand = new Command("Exit", Command.EXIT, 0);
    }
    return exitCommand;
  }
  public Command getExitCommandRead() {
    if (exitCommandRead == null) {
      exitCommandRead = new Command("Exit", Command.EXIT, 0);
    }
    return exitCommandRead;
  }
```

```
public Command getItemCommandNextPage() {
  if (itemCommandNextPage == null) {
    itemCommandNextPage = new Command("Next Page", Command.ITEM, 0);
  }
  return itemCommandNextPage;
}
public Command getItemCommandPreviousPage() {
  if (itemCommandPreviousPage == null) {
    itemCommandPreviousPage =
      new Command("Previous Page", Command.ITEM, 0);
  }
  return itemCommandPreviousPage;
}
public SimpleCancellableTask getTask() {
  if (task == null) {
    task = new SimpleCancellableTask();
    task.setExecutable(new org.netbeans.microedition.util.Executable()
    {
      public void execute() throws Exception {
        methodRead();
      }
    });
  }
  return task;
}
public void methodRead() {
  String bookURL = fileBrowser.getSelectedFileURL();
  getTextBox().setString("");
  textBox.setMaxSize(textBox.getHeight()*textBox.getWidth()/110);
  if (bookURL.indexOf(".txt")>=0){
    methodReadTXT();
  }
  else throw new Error();
}
public void methodReadTXT() {
  try {
    FileConnection book = fileBrowser.getSelectedFile();
    InputStream instr = book.openInputStream();
    InputStreamReader reader=new InputStreamReader(instr);
    if (instrSkip<=0) instrSkip=0;
    reader.skip(instrSkip);
    String str;
    char [] buf=new char[textBox.getMaxSize()];
    reader.read(buf);
```

```
      str=new String(buf);
      textBox.setString(str);
      instr.close();
    } catch (Exception ex) {
    }
  }
  public Ticker getTickerError() {
    if (tickerError == null) {
      tickerError = new Ticker("ERROR!");
    }
    return tickerError;
  }
  public Font getFontSplash() {
    if (fontSplash == null) {
      fontSplash = Font.getDefaultFont();
    }
    return fontSplash;
  }
  public Display getDisplay () {
    return Display.getDisplay(this);
  }
  public void exitMIDlet() {
    switchDisplayable (null, null);
    destroyApp(true);
    notifyDestroyed();
  }
  public void startApp() {
    if (midletPaused) { resumeMIDlet (); }
    else {  initialize();
            startMIDlet();  }
    midletPaused = false;
  }
  public void pauseApp() {
    midletPaused = true;
  }
  public void destroyApp(boolean unconditional) {
  }}
```

Разберем код созданного класса `JBookReader` (см. листинг 6.1).

Используемый в коде класса `JBookReader` класс `java.io.InputStream` содержится в библиотеках конфигурации CLDC 1.0 и необходим нам для управления входящим потоком байтов, открытым с помощью объекта `FileConnection`. Интерфейс `javax.microedition.io.file.FileConnection` со-

держится в библиотеках дополнительного пакета FileConnection Optional Package 1.0 и обеспечивает доступ к файловой системе устройства. Класс `java.io.InputStreamReader` CLDC 1.0 преобразует байтовый входящий поток в символьный.

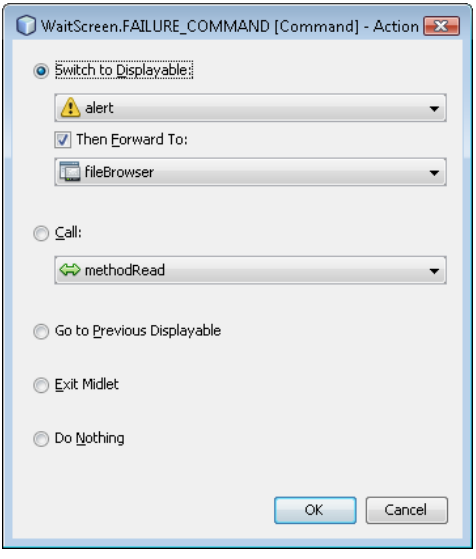

Рис. 6.3. Окно редактирования связей между компонентами, участвующими в рабочем потоке приложения

Главный класс MIDP-приложения должен расширять класс `javax.microedition.midlet.MIDlet` для организации взаимодействия с программным обеспечением устройства Application Management Software (AMS), которое управляет жизненным циклом приложения. Поэтому класс `JBookReader` является подклассом класса `MIDlet`.

Пакет `javax.microedition.lcdui` профиля MIDP содержит необходимые нам классы `Alert` (экран сообщения пользователю), `TextBox` (экран редактируемого текста), `Ticker` (бегущая строка), `Font` (шрифт), `Command` (содержит информацию о действии), интерфейс `CommandListener` (определяет обработчик событий команды), `Display` (менеджер дисплея) и `Displayable` (объект дисплея).

Классы `SplashScreen`, `WaitScreen` и `FileBrowser` библиотек NetBeans MIDP Components и NetBeans MIDP Components PDA представляют компоненты графического редактора Visual Mobile Designer (VMD) среды NetBeans.

Класс `SimpleCancellableTask` библиотеки NetBeans MIDP Components является реализацией интерфейса `CancellableTask`, который используется компонентом Wait Screen VMD для выполнения задачи в фоновом потоке.

Свойство `midletPaused` используется для определения состояния приложения — остановлено выполнение приложения или нет.

Переменную `instrSkip` мы объявили для использования в качестве аргумента метода `skip()` класса `InputStreamReader`, обеспечивающего пропуск указанного количества символов входящего потока, и, таким образом, этот метод позволит нам организовать постраничное чтение электронной книжки.

Метод `initialize()` вызывается в переопределенном методе `startApp()` класса `JBookReader`, который в свою очередь вызывается Application Management Software (AMS) при запуске приложения.

Метод `startMIDlet()` также вызывается в переопределенном методе `startApp()` класса `JBookReader`.

Метод `resumeMIDlet` вызывается в переопределенном методе `startApp()` в случае, если приложение было остановлено (метод `pauseApp()` был вызван AMS), и может быть использован для установки параметров восстановления приложения. Приложение может быть остановлено AMS, например, в случае входящего звонка на мобильный телефон. При восстановлении приложения AMS снова вызывает метод `startApp()`.

Метод `switchDisplayable()` позволяет переключать экраны с помощью метода `setCurrent()` класса `Display`. Объект `Display` — менеджер дисплея существует в единственном экземпляре для каждого приложения, и его можно получить, используя статический метод `getDisplay()` класса `Display`. В качестве аргумента метод `setCurrent()` использует объект `Displayable`, содержащий все элементы графического интерфейса пользователя, отображаемые на данном экране.

Метод `commandAction()` интерфейса `javax.microedition.lcdui.CommandListener` вызывается в случае возникновения событий, связанных с командами. Команды — объекты класса `Command` представлены в пользовательском интерфейсе и связаны с объектами `Displayable` с помощью метода `addCommand(Command cmd)`.

При отображении экрана навигации файловой системы устройства `fileBrowser`, в случае выбора пользователем файла, срабатывает команда `SELECT_FILE_COMMAND`, при этом вызывается метод `commandAction()`, и согласно условию дисплей переключается на экран ожидания `waitScreen`. В случае если пользователь инициирует срабатывание команды `exitCommand`, вызывается метод `exitMIDlet()`, который закрывает приложение.

При отображении заставки `splashScreen` через определенное время срабатывает команда `DISMISS_COMMAND` и дисплей переключается на экран навигации файловой системы `fileBrowser`.

При отображении экрана редактируемого текста `textBox` пользователь может инициировать срабатывание следующих команд:

- `exitCommandRead` — переключение на `fileBrowser`;
- `itemCommandNextPage` — количество пропущенных символов `instrSkip` входящего потока увеличивается на максимальное количество отображаемых на экране `textBox` символов, и дисплей переключается на экран ожидания;
- `itemCommandPreviousPage` — количество пропущенных символов `instrSkip` входящего потока уменьшается на максимальное количество отображаемых на экране `textBox` символов, и дисплей переключается на экран ожидания.

При отображении экрана ожидания, в случае успешного выполнения фоновой задачи, которая в данном случае заключается в считывании одной страницы текста из входящего потока (страница текста определяется максимальным количеством отображаемых на экране `textBox` символов), срабатывает команда `SUCCESS_COMMAND`, и дисплей переключается на экран `textBox`, который и отображает считанную страницу. В случае возникновения исключительной ситуации при выполнении фоновой задачи срабатывает команда `FAILURE_COMMAND`, и дисплей переключается на экран навигации файловой системы устройства `fileBrowser` через отображение сообщения пользователю — экран `alert`.

Методы `getSplashScreen()`, `getFileBrowser()`, `getWaitScreen()`, `getAlert()` и `getTextBox()` класса `JBookReader` возвращают объекты, представляющие экраны **Splash Screen**, **File Browser**, **Wait Screen**, **Alert** и **Text Box** соответственно. Данные методы устанавливают свойства экранов, включая команды пользовательского интерфейса.

Методы `getExitCommand()`, `getExitCommandRead()`, `getItemCommandNextPage()` и `getItemCommandPreviousPage()` класса `JBookReader` возвращают объекты `Command`, содержащие информацию о соответствующих командах. Объект класса `Command` создается с помощью конструктора, который в качестве аргументов принимает:

- отображаемое обозначение команды;
- тип команды; от типа команды зависит, с какой кнопкой устройства данная команда будет связана;
- приоритет команды — целое значение от 0 и больше. Чем меньше значение, тем больше приоритет; при наличии нескольких команд одного типа с кнопкой устройства связывается команда с большим приоритетом — остальные команды помещаются в меню.

Метод `getTask()` класса `JBookReader` возвращает объект класса `SimpleCancellableTask`, используемый в качестве аргумента метода `setTask()`

класса `WaitScreen`, который устанавливает задачу, исполняемую в фоновом потоке. Метод `setExecutable()` класса `SimpleCancellableTask` устанавливает исполняемый объект для данной задачи. Здесь исполняемый объект представлен объектом анонимного класса, реализующего интерфейс `Executable`. Интерфейс `Executable` содержит переопределяемый метод `execute` для выполнения.

В исполняемом методе `execute()` интерфейса `Executable` вызывается метод `methodRead()`, в котором осуществляется очистка экрана `textBox` с помощью метода `setString()` класса `TextBox`. Так как нигде ранее у нас не происходит инициализация объекта `textBox`, то получить его мы должны в методе `methodRead()`, используя метод `getTextBox()`.

В методе `methodRead()` класса `JBookReader` с помощью метода `getSelectedFileURL()` класса `FileBrowser` мы получаем URL-адрес файла, выбранного пользователем, и проверяем в нем наличие расширения txt — если расширение найдено, то вызывается метод `methodReadTXT()` для чтения файла, если нет — генерируется ошибка с помощью ключевого слова `throw`. В результате генерации ошибки выполнение фоновой задачи завершается неудачно и срабатывает команда `FAILURE_COMMAND` экрана **Wait Screen**, которая вызывает отображение экрана сообщения пользователю **Alert** и затем переключение на экран **File Browser**. С помощью методов `getWidth()` и `getHeight()`, унаследованных классом `TextBox` от класса `Displayable`, мы получаем в пикселах ширину и высоту области дисплея, доступной для приложения. Исходя из размеров шрифта Times New Roman 12, установим максимальное количество отображаемых символов методом `setMaxSize()` класса `TextBox`, используя соотношение

```
textBox.getHeight()*textBox.getWidth()/110
```

В методе `methodReadTXT()`, который вызывается методом `methodRead()` класса `JBookReader` в случае успешного выбора пользователем файла для чтения, открываем входящий символьный поток, связанный с выбранным пользователем файлом. Далее осуществляем постраничный пропуск символов при чтении методом `skip()` класса `InputStreamReader`, считываем из входящего потока максимальное количество отображаемых символов и устанавливаем их как содержимое экрана `textBox`.

Методы `getTickerError()` и `getFontSplash()` класса `JBookReader` возвращают объекты, представляющие бегущую строку для экрана сообщения **Alert** и шрифт для текста экрана заставки **Splash Screen** соответственно.

Метод `getDisplay()` класса `JBookReader` возвращает объект `Display` — менеджер дисплея для нашего приложения.

Метод `exitMIDlet()` класса `JBookReader` осуществляет завершение приложения. Метод `switchDisplayable(null, null)`, вызываемый в методе

`exitMIDlet()`, запрашивает AMS о переводе приложения в фоновый режим. Аргумент `true` метода `destroyApp(true)`, который вызывается методом `exitMIDlet()`, устанавливает невозможность приложения избежать состояния `Destroyed`.

Методы `startApp()`, `pauseApp()` и `destroyApp()` класса `JBookReader` вызываются AMS при запуске приложения, при остановке приложения и при завершении приложения соответственно.

После создания класса `JBookReader` мы можем осуществить сборку проекта и запуск созданного нами MIDP-приложения.

Для сборки приложения щелкнем правой кнопкой мыши на названии проекта в окне **Projects** среды NetBeans и выберем пункт **Build**. В результате в папке dist каталога проекта будут созданы JAD- и JAR-файлы приложения

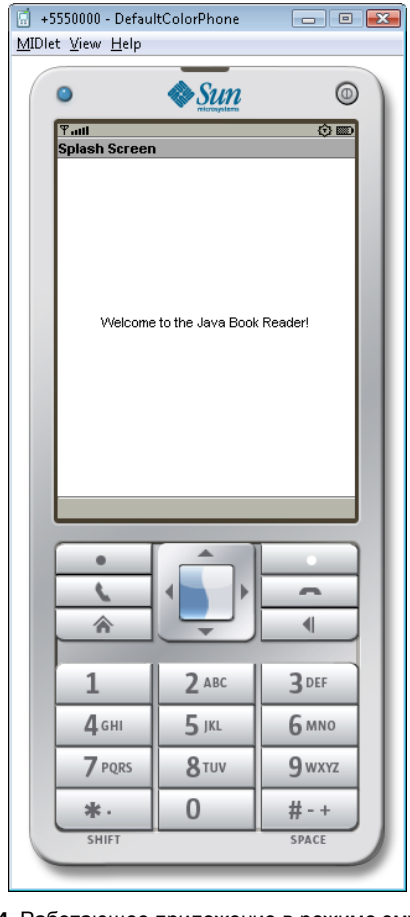

Рис. 6.4. Работающее приложение в режиме эмулятора

JBookReader, которые для инсталляции приложения необходимо передать в устройство.

Размер JAR-файла составил 52,2 Кбайт. Посмотрим, что произойдет с размером JAR-файла после обфускации. В окне **Properties** проекта, в категории **Obfuscating**, выберем максимальный уровень обфускации и снова соберем проект. Размер JAR-файла уменьшится и составит 14 Кбайт.

Для запуска приложения в режиме эмулятора можно воспользоваться меню окна **Projects**, кнопкой **Run Project** панели инструментов среды NetBeans или клавишей <F6> (рис. 6.4).

Пример создания приложения для чтения TXT-файлов с использованием низкоуровневой графической библиотеки

Пакет `javax.microedition.lcdui` MIDP-профиля, обеспечивающий создание графического интерфейса пользователя MIDP-приложений, состоит из двух частей — низкоуровневой и высокоуровневой библиотек.

Высокоуровневая библиотека API обеспечивает высокую переносимость приложений и не позволяет детально контролировать внешний вид и поведение компонентов интерфейса пользователя, которые в данном случае полностью исполняются реализацией устройства. Классы высокоуровневой библиотеки — это подклассы `Alert`, `Form`, `List`, `TextBox` суперкласса `Screen`.

Низкоуровневая библиотека API позволяет иметь доступ к специфическим для данного устройства свойствам и контролировать внешний вид и поведение графических компонентов, но при этом она не гарантирует полную переносимость приложения. Классы низкоуровневой библиотеки — это классы `Canvas` и `Graphics`.

До сих пор мы пользовались API высокого уровня. Модифицируем наше приложение, используя классы `Canvas` и `Graphics` низкоуровневой библиотеки при создании графического интерфейса пользователя.

> ***Примечание***
>
> Файлы с кодами приложения расположены на прилагаемом к книге компакт-диске в папке Часть_II\Глава_6\JBookReaderDraw.

Для создания MIDP-приложения, использующего низкоуровневую графическую библиотеку MIDP-профиля:

1. В среде NetBeans в меню **File** выберем **New Project | Categories: Java ME | Mobile Application**. Введем имя проекта JBookReaderDraw и, сбросив флажок **Create Hello MIDlet**, нажмем кнопку **Finish**.

2. В окне **Projects** щелкнем правой кнопкой мыши на названии проекта и выберем пункт **Properties**. В категории **Platform** установим переключатели **Device Configuration: CLDC 1.0**, **Device Profile: MIDP 2.0** и **Optional Packages: File Connection and PIM Optional Packages 1.0**.
3. В окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на узле **Source Packages** и выберем **New | Other | Categories: Java | Java Package**. Введем имя пакета jbookreaderdraw и нажмем кнопку **Finish**.
4. Щелкнем правой кнопкой мыши на узле пакета jbookreaderdraw и выберем **New | Other | Categories: MIDP | Visual MIDlet**. Введем имя мидлета JBookReaderDraw и нажмем кнопку **Finish**.
5. Снова щелкнем правой кнопкой мыши на узле пакета jbookreaderdraw и выберем **New | Other | Categories: MIDP | MIDP Canvas**. Введем имя класса `JBookReaderCanvas`, расширяющего класс `Canvas`, и нажмем кнопку **Finish**. Класс `JBookReaderCanvas` заменит в модифицированном приложении класс `TextBox` и будет включать в себя функции метода `methodReadTXT()`.

Для того чтобы использовать созданный класс `JBookReaderCanvas` в качестве компонента графического редактора Visual Mobile Designer, при открытом окне **Flow** класса `JBookReaderDraw` щелкнем правой кнопкой мыши на окне палитры компонентов **Palette** и выберем **Palette Manager**. В появившемся диалоговом окне нажмем кнопку **Add to Palette** и выберем класс `JBookReaderCanvas`. В результате в палитре компонентов **Palette** в категории **Custom** появится новый компонент `JBookReaderCanvas`. Если этого не произойдет, то в окне **Palette Manager** надо нажать кнопку **Reset Palette**.

С помощью палитры компонентов **Palette** перенесем в окно **Flow** следующие компоненты: **Splash Screen**; **File Browser**, в который добавим команду **Exit Command**; **Alert** и **canvas** — компонент `JBookReaderCanvas`, в который также добавим команду **Exit Command**.

После добавления компонентов соединим их действиями в окне **Flow** (рис. 6.5).

При создании рабочего потока приложения среда NetBeans сгенерировала соответствующий код класса `JBookReaderDraw`, который мы дополнили, вместе с кодом класса `JBookReaderCanvas`, для обеспечения возможности пользователю читать выбранную им электронную книжку постранично (листинги 6.2 и 6.3).

Листинг 6.2. Код класса `JBookReaderCanvas`

```
package jbookreaderdraw;
import java.io.*;
import javax.microedition.io.file.FileConnection;
```

```
import javax.microedition.lcdui.*;
public class JBookReaderCanvas extends Canvas {
  private Font font;
  private long charSkip;
  private long Skip;
  public JBookReaderCanvas() {
    font=Font.getFont(Font.FACE_MONOSPACE, Font.STYLE_PLAIN,
                      Font.SIZE_MEDIUM);
    Skip=0;
  }
  public void paint(Graphics g) {
    g.setColor(255, 255, 255);
    g.fillRect(0, 0, g.getClipWidth(), g.getClipHeight());
    g.setColor(0, 0, 0);
    g.setFont(font);
    FileConnection book=JBookReaderDraw.getFileConnection();
    InputStream instr;
    charPage=0;
    try {
      instr = book.openInputStream();
      InputStreamReader reader=new InputStreamReader(instr);
      if (Skip<=0) Skip=0;
      reader.skip(Skip);
      String str;
      int height=0;
      while(height<(g.getClipHeight()-20)){
        str="";
        int width=0;
        while(width<(g.getClipWidth()-20)){
          char [] buf=new char [1];
          reader.read(buf);
          str=str+new String(buf);
          charPage++;
          width=width+font.charsWidth(buf, 0, 1);
        }
        g.drawString(str,10,height,Graphics.TOP|Graphics.LEFT);
        height=height+font.getHeight();
      }
      instr.close();
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
```

```
  protected  void keyPressed(int keyCode) {
    if (keyCode==JBookReaderCanvas.KEY_POUND){
      Skip=Skip+charPage;
      this.repaint();
    }
    if (keyCode==JBookReaderCanvas.KEY_STAR){
      Skip=Skip-charPage;
      this.repaint();
    }
    if (keyCode==JBookReaderCanvas.KEY_NUM3){
      font=Font.getFont(Font.FACE_MONOSPACE, Font.STYLE_PLAIN,
                        Font.SIZE_LARGE);
      this.repaint();
    }
    if (keyCode==JBookReaderCanvas.KEY_NUM2){
      font=Font.getFont(Font.FACE_MONOSPACE, Font.STYLE_PLAIN,
                        Font.SIZE_MEDIUM);
      this.repaint();
    }
    if (keyCode==JBookReaderCanvas.KEY_NUM1){
      font=Font.getFont(Font.FACE_MONOSPACE, Font.STYLE_PLAIN,
                        Font.SIZE_SMALL);
      this.repaint();
    }
  }
  protected  void keyReleased(int keyCode) {  }
  protected  void keyRepeated(int keyCode) {  }
  protected  void pointerDragged(int x, int y) {  }
  protected  void pointerPressed(int x, int y) {  }
  protected  void pointerReleased(int x, int y) {  }
}
```

Обсудим код класса `JBookReaderCanvas` (см. листинг 6.2).

Изначально среда NetBeans создала каркас класса `JBookReaderCanvas`, в котором он реализует интерфейс `CommandListener` для обработки высокоуровневых событий, связанных с командами. Но у нас все команды сосредоточены в классе `JBookReaderDraw`, также реализующем интерфейс `CommandListener`. Кроме того, в классе `JBookReaderCanvas` мы намерены обрабатывать только низкоуровневые события, связанные с кнопками устройства. Поэтому в классе `JBookReaderCanvas` уберем реализацию интерфейса `CommandListener` и всю функциональность, связанную с этим.

Классы `InputStream` и `InputStreamReader` пакета `java.io` обеспечат нам чтение входящего потока, открытого с помощью объекта класса `FileConnection`.

Объект `FileConnection`, связанный с выбранным пользователем файлом, мы будем получать, используя созданный нами статический метод `getFileConnection()` класса `JBookReaderDraw`.

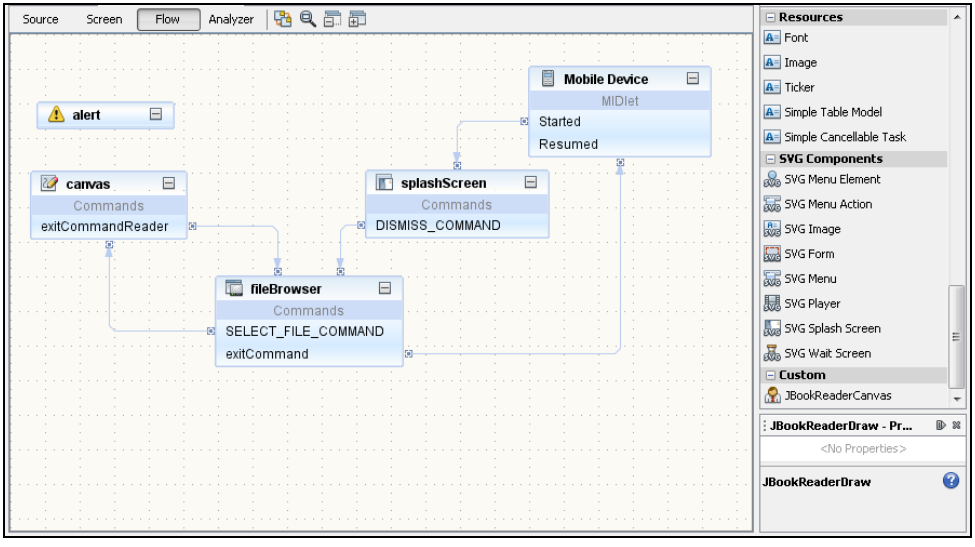

Рис. 6.5. Компоненты рабочего потока модифицированного приложения

Поле `font` класса `JBookReaderCanvas` нам понадобится для управления шрифтом отображаемого текста. Поле `charPage` будет хранить количество символов страницы текста. Поле `Skip` понадобится нам для хранения количества символов, которое нужно пропустить при чтении входящего потока.

В конструкторе класса `JBookReaderCanvas` мы инициализируем объекты `font` и `Skip`. Объект класса `javax.microedition.lcdui.Font` нельзя создать с помощью конструктора, можно только запросить систему устройства с помощью статического метода `getFont()` класса `Font`, и система будет пытаться обеспечить шрифт с требуемыми параметрами. Параметры шрифта состоят из вида, стиля и размера. Изначально мы установим средний размер шрифта.

Абстрактный класс `javax.microedition.lcdui.Canvas`, подклассом которого является созданный нами класс `JBookReaderCanvas`, представляет собой базовый класс для приложений, в которых есть необходимость использования функций рисования с помощью класса `javax.microedition.lcdui.Graphics` и управления низкоуровневыми событиями. Единственный абстрактный метод, который необходимо переопределить в классе, расширяющем класс `Canvas`, — это метод `paint(Graphics g)`. Метод `paint` отвечает за прорисовку экрана `Canvas`, и его нельзя вызвать в приложении. Данный метод вызывается только системой устройства, когда экран `Canvas` устанавливается видимым на

дисплее методом `setCurrent()` класса `Display`. Системой устройства также вызываются следующие методы класса `Canvas`:

- `showNotify()` вызывается перед тем, как экран `Canvas` становится видимым;
- `hideNotify()` вызывается сразу после того, как экран `Canvas` удаляется из дисплея;
- `keyPressed()` вызывается, когда нажимается кнопка устройства;
- `keyRepeated()` вызывается, когда кнопка устройства удерживается в нажатом состоянии;
- `keyReleased()` вызывается, когда кнопка устройства освобождается;
- `pointerPressed()` вызывается, когда указатель устройства нажимается;
- `pointerDragged()` вызывается, когда указатель устройства перемещается;
- `pointerReleased()` вызывается, когда указатель устройства освобождается.

Класс `Graphics` обеспечивает такие низкоуровневые функции рисования, как рисование дуги, символа, изображения, линии, прямоугольника, треугольника и строки.

В переопределенном методе `paint()` класса `JBookReaderCanvas` сначала мы очистим экран дисплея, установив белый цвет рисования методом `setColor(255, 255, 255)`. Затем мы заполним экран дисплея установленным цветом с помощью метода

```
fillRect(int x, int y, int width, int height)
```

Здесь `x` и `y` — координаты левого верхнего угла заполняемого цветом прямоугольника, а `width` и `height` — его ширина и высота. Ширину и высоту экрана дисплея мы получим, используя методы `getClipWidth()` и `getClipHeight()`.

После этого в методе `paint()` класса `JBookReaderCanvas` установим черный цвет рисования методом `setColor(0, 0, 0)` и шрифт, представленный объектом `font`, методом `setFont()`. Далее получим объект `FileConnection`, связанный с выбранным пользователем файлом, и откроем входящий символьный поток. С помощью метода `skip()` класса `InputStreamReader` произведем постраничный пропуск символов при чтении входящего потока и будем посимвольно считывать из него, при этом формируя строку, до тех пор, пока длина формируемой строки не сравняется с шириной экрана плюс отступ. Созданную строку отобразим на экране методом `drawString()`. Будем отображать строки на экране до тех пор, пока не достигнем его нижнего края плюс отступ. Каждый раз смещение по оси *y* при отображении следующей строки будет равняться высоте строки для установленного шрифта, которую мы получим с помощью метода `getHeight()` класса `Font`. Количество изображенных при этом символов мы сохраним в переменной `charPage`.

Переопределенный метод `keyPressed()` класса `JBookReaderCanvas` обеспечит перелистывание страниц и изменение шрифта отображаемого текста при нажатии пользователем кнопок устройства.

Если пользователь нажмет кнопку # мобильного устройства, то количество символов, пропускаемых при чтении входящего потока, увеличится на одну страницу и на экране дисплея отобразится следующая страница текста вызовом метода `repaint()` класса `Canvas`. В результате вызова метода `repaint()` системой устройства будет снова вызван метод `paint()`, только при этом значение поля `Skip` будет уже другим.

Если пользователь нажмет кнопку ***** мобильного устройства, то количество символов, пропускаемых при чтении входящего потока, уменьшится на одну страницу и на экране дисплея отобразится предыдущая страница текста.

При нажатии пользователем кнопок **1**, **2** и **3** мобильного устройства размер шрифта отображаемого текста будет изменяться от самого маленького до самого большого.

Листинг 6.3. Код класса `JBookReaderDraw`

```
package jbookreaderdraw;
import java.io.*;
import javax.microedition.io.file.FileConnection;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import org.netbeans.microedition.lcdui.SplashScreen;
import org.netbeans.microedition.lcdui.pda.FileBrowser;
public class JBookReaderDraw extends MIDlet implements CommandListener {
  private boolean midletPaused = false;
  private SplashScreen splashScreen;
  private FileBrowser fileBrowser;
  private JBookReaderCanvas canvas;
  private Alert alert;
  private Command exitCommand;
  private Command exitCommandReader;
  static FileConnection book=null;
  public JBookReaderDraw() {  }
  private void initialize() {  }
  public void startMIDlet() {
    switchDisplayable(null, getSplashScreen());
  }
  public void resumeMIDlet() {  }
  public void switchDisplayable(Alert alert,
                                Displayable nextDisplayable) {
    Display display = getDisplay();
```

```
    if (alert == null) {
      display.setCurrent(nextDisplayable);
    } else {
      display.setCurrent(alert, nextDisplayable);
    }
  }
  public void commandAction(Command command, Displayable displayable) {
    if (displayable == canvas) {
      if (command == exitCommandReader) {
        switchDisplayable(null, getFileBrowser());
      }
    } else if (displayable == fileBrowser) {
      if (command == FileBrowser.SELECT_FILE_COMMAND) {
        String bookURL = fileBrowser.getSelectedFileURL();
        if (bookURL.indexOf(".txt")>=0){
          try {
            book = fileBrowser.getSelectedFile();
          } catch (IOException ex) {
            ex.printStackTrace();
          }
          switchDisplayable(null, getCanvas());
        }
        else switchDisplayable(getAlert(), getFileBrowser());
      } else if (command == exitCommand) {
        exitMIDlet();
      }
    } else if (displayable == splashScreen) {
      if (command == SplashScreen.DISMISS_COMMAND) {
        switchDisplayable(null, getFileBrowser());
     }     }     }
  public SplashScreen getSplashScreen() {
    if (splashScreen == null) {
      splashScreen = new SplashScreen(getDisplay());
      splashScreen.setTitle("Splash Screen");
      splashScreen.setCommandListener(this);
      splashScreen.setFullScreenMode(true);
      splashScreen.setText("Welcome to the Java Book Reader!");
    }
    return splashScreen;
  }
  public FileBrowser getFileBrowser() {
    if (fileBrowser == null) {
      fileBrowser = new FileBrowser(getDisplay());
```

```
      fileBrowser.setTitle("File Browser: Select Book");
      fileBrowser.setCommandListener(this);
      fileBrowser.addCommand(FileBrowser.SELECT_FILE_COMMAND);
      fileBrowser.addCommand(getExitCommand());
    }
    return fileBrowser;
  }
  public JBookReaderCanvas getCanvas() {
    if (canvas == null) {
      canvas = new JBookReaderCanvas();
      canvas.setTitle("Book Reader:#-Next Page, *-Previous Page");
      canvas.addCommand(getExitCommandReader());
      canvas.setCommandListener(this);
    }
    return canvas;
  }
  public Command getExitCommand() {
    if (exitCommand == null) {
      exitCommand = new Command("Exit", Command.EXIT, 0);
    }
    return exitCommand;
  }
  public Alert getAlert() {
    if (alert == null) {
      alert = new Alert("ERROR!", "Cannot open the file!",
                        null, AlertType.ERROR);
      alert.setTimeout(Alert.FOREVER);
    }
    return alert;
  }
  public Command getExitCommandReader() {
    if (exitCommandReader == null) {
      exitCommandReader = new Command("Exit", Command.EXIT, 0);
    }
    return exitCommandReader;
  }
  public static FileConnection getFileConnection(){
    return book;
  }
  public Display getDisplay () {
    return Display.getDisplay(this);
  }
  public void exitMIDlet() {
    switchDisplayable (null, null);
```

```
    destroyApp(true);
    notifyDestroyed();
  }
  public void startApp() {
    if (midletPaused) {
      resumeMIDlet ();
    } else {
      initialize ();
      startMIDlet ();
    }
    midletPaused = false;
  }
  public void pauseApp() {
    midletPaused = true;
  }
  public void destroyApp(boolean unconditional) {  }}
```

Разберем код созданного нами класса `JBookReaderDraw` (см. листинг 6.3).

В коде класса `JBookReaderDraw` статический объект `book` класса `FileConnection` необходим для обеспечения доступа к выбранному пользователем файлу в классе `JBookReaderCanvas`. Для того чтобы была возможность получить объект `book`, мы создадим соответствующий статический метод `getFileConnection`.

При запуске приложения на экране дисплея появляется заставка — экран **Splash Screen**. Через установленное время срабатывает команда `DISMISS_COMMAND`, и дисплей переключается на экран навигации файловой системы **File Browser**. Если пользователь осуществляет выбор файла и данный файл имеет расширение txt, то в статическое поле `book` заносится объект `FileConnection`, обеспечивающий доступ к выбранному файлу, и дисплей переключается на экран **canvas**, представленный классом `JBookReaderCanvas`. Если выбранный файл имеет другое расширение, то дисплей отображает сообщение пользователю — экран **Alert** и снова переключается на экран **File Browser**. Если пользователь выбирает команду `exitCommand` экрана **File Browser**, то приложение завершает свою работу. Экран **canvas** имеет свою команду `exitCommandReader`, при выборе которой дисплей переключается на экран **File Browser**.

Методы `getSplashScreen()`, `getFileBrowser()`, `getCanvas()`, `getExitCommand()`, `getAlert()`, `getExitCommandReader()` класса `JBookReaderDraw` осуществляют инициализацию и возврат объектов, представляющих соответствующие экраны и команды. Статический метод `getFileConnection()` возвращает статический объект `book`, о котором упоминалось ранее.

Метод `getDisplay()` класса `JBookReaderDraw` возвращает объект — менеджер дисплея для данного приложения. Метод `exitMIDlet()` осуществляет завершение работы приложения. Методы `startApp()`, `pauseApp()` и `destroyApp()` класса `JBookReaderDraw` вызываются Application Management Software соответственно при запуске, остановке и завершении приложения.

Таким образом, созданием классов `JBookReaderDraw` и `JBookReaderCanvas` мы получили MIDP-приложение, использующее низкоуровневую графическую библиотеку MIDP-профиля.

После сборки проекта модифицированного приложения, использующего низкоуровневую библиотеку пакета `javax.microedition.lcdui` профиля MIDP, в папке dist каталога проекта мы получим JAD- и JAR-файлы, необходимые для инсталляции приложения в устройстве.

ЧАСТЬ III

Платформа Java EE

Глава 7	Клиент-серверная архитектура платформы Java EE
Глава 8	Технологии платформы Java EE
Глава 9	Пример приложения Java EE

Платформа Java Platform, Enterprise Edition (Java EE) основывается на платформе Java Platform, Standard Edition (Java SE) и предназначена для создания Web-приложений и клиент-серверных приложений уровня предприятия, позволяя реализовывать сервис-ориентированную архитектуру (SOA).

> ***Примечание***
>
> Сервис-ориентированная архитектура (SOA) позволяет создавать распределенные модульные приложения, использующие заменяемые, слабосвязанные, взаимодействующие посредством стандартных протоколов сервисы со стандартизированными интерфейсами. Программные комплексы, основанные на SOA, могут использовать сервисы, написанные на разных языках программирования и работающие на различных платформах.

Платформа Java EE представляет собой набор технологий, включающий в себя спецификацию самой платформы, технологии Web-сервисов, Web-приложений, корпоративных приложений, а также технологии системы безопасности и управления.

Ключевым понятием Java EE технологии является *контейнер* Java EE.

Как уже было сказано, приложения Java EE имеют клиент-серверную архитектуру, в которой клиентская часть приложения взаимодействует с серверной частью посредством запросов, отправляемых клиентом серверу, который в свою очередь посылает ответы на запросы клиенту. Работу приложения

Java EE обеспечивает сервер приложений Java EE, представляющий собой Java-приложение, выполняемое виртуальной Java-машиной и состоящее из *контейнеров* Java EE, которые обеспечивают среду выполнения для компонентов приложения Java EE. Компоненты приложения Java EE не взаимодействуют между собой напрямую, они используют протоколы и методы своих контейнеров для взаимодействия между собой и со стандартными сервисами платформы Java EE. Сервер приложений Java EE обеспечивает для компонентов приложения Java EE стандартные сервисы: управление транзакциями, систему безопасности, управление жизненным циклом и др.

Спецификация платформы Java EE определяет четыре типа приложений, среды выполнения для которых предоставляют контейнеры Java EE, обеспечивающие поддержку необходимых для выполнения приложений библиотек Java EE API:

- клиентские приложения, как правило, имеющие графический интерфейс пользователя и выполняемые на настольном компьютере;
- апплеты, выполняемые как в Web-браузерах, так и в других приложениях или устройствах, поддерживающих модель программирования апплетов;
- Web-компоненты:
  - сервлеты — Java-приложения, выполняемые на стороне сервера;
  - JSP-приложения — приложения, использующие технологию JavaServer Pages;
  - JSF-приложения — приложения, использующие технологию JavaServer Faces;
  - Web-фильтры, модифицирующие запросы Web-приложению и ответы клиенту от Web-приложения и являющиеся Java-классами, реализующими интерфейс `javax.servlet.Filter`;
  - обработчики Web-событий — классы, реализующие интерфейсы-слушатели событий;
  - Web-сервисы — Web-приложения, обеспечивающие обмен данными с помощью SOAP-/HTTP-протоколов.

Web-компоненты исполняются в Web-контейнере, который включен в Web-сервер. Web-контейнер обеспечивает поддержку библиотек Java EE API для Web-компонентов.

Компоненты Enterprise JavaBeans (EJB) — многократно используемые программные модули, исполняемые внутри EJB-контейнера, который включен в EJB-сервер и организует транзакции, управление жизненным циклом и ресурсами компонентов.

В распределенном приложении Java EE апплеты взаимодействуют с Web-компонентами, клиентские приложения — с Web-компонентами, EJB-компонентами и с базами данных. EJB-компоненты могут связываться с базой данных, а Web-компоненты могут быть связаны с EJB-компонентами и с базой данных.

Большинство стандартных сервисов Java EE, представленных библиотеками Java EE API и Java SE API, обеспечивают совместимость с компонентами, не созданными на основе платформы Java EE.

Приложения платформы Java EE являются многоуровневыми приложениями, функциональность которых распределена по уровням.

Клиентский уровень приложений Java EE представлен клиентскими приложениями различного типа — апплетами, самостоятельными приложениями, клиентскими приложениями, выполняемыми в контейнере сервера приложений Java EE. Клиентские приложения осуществляют запросы к серверу Java EE, который обрабатывает их и дает ответ клиентским приложениям.

Сервер Java EE или сервер приложений Java EE — это программное обеспечение, выполняемое на отдельном сервере (компьютерном оборудовании, исполняющем сервисные функции), реализующее библиотеки Java EE API и обеспечивающее стандартные сервисы Java EE в форме контейнеров, которые являются интерфейсом между Web-компонентами, EJB-компонентами и низкоуровневой функциональностью платформы Java EE. Web- и EJB-компоненты приложения Java EE могут выполняться на одном сервере Java EE или быть распределены по разным серверам и компьютерам.

Web-уровень приложений Java EE — это уровень, состоящий из Web-компонентов, которые динамически генерируют контент в различном формате (HTML-странички, XML-данные) для клиентских приложений и при необходимости взаимодействуют с EJB-компонентами.

Бизнес-уровень приложений Java EE состоит из EJB-компонентов, реализующих бизнес-логику приложения Java EE.

Уровень данных приложений Java EE или уровень Enterprise Information Systems (EIS) состоит из различных ресурсов, таких как серверы баз данных, информационные системы предприятий (например, "1С") и другие ресурсы.

Такая архитектура приложений Java EE упрощает разработку, а отделение контейнерами Java EE бизнес-логики от управления ресурсами и жизненным циклом дает возможность разработчику сосредоточиться на реализации бизнес-функциональности, а не на создании инфраструктуры приложения.

Широкий набор современных технологий делает платформу Java EE промышленным стандартом для разработки решений уровня предприятия, по-

зволяя создавать на своей основе Web-приложения, сетевые и многопоточные приложения, распределенные и многоуровневые программные комплексы, решать задачи параллельного программирования.

Проекты на основе платформы Java EE предназначены для автоматизации бизнес-процессов, создания корпоративных интернет- и интранет-порталов, систем сетевых вычислений, информационных корпоративных систем, торговых площадок и др.

Платформа Java EE обладает достаточной гибкостью и мощностью, чтобы решать задачи во всех областях бизнеса, помогая организовывать и оптимизировать бизнес-процессы, экономить и зарабатывать деньги предприятиям различного уровня.

ГЛАВА 7

Клиент-серверная архитектура платформы Java EE

Структура приложения Java EE

Приложение Java EE поставляется в виде архивного JAR-файла с расширением ear. Данный архив содержит модули Java EE и дополнительный (необязательный) файл описания, дескриптор развертывания (deployment descriptor) — XML-документ application.xml, находящийся в папке META-INF архива.

Информация, содержащаяся в файле дескриптора развертывания приложения, используется сервером Java EE при развертывании приложения. Содержимое файла дескриптора развертывания приложения Java EE регламентируется соответствующей схемой **http://java.sun.com/xml/ns/javaee/application_5.xsd**.

Элементы файла дескриптора развертывания приложения Java EE:

- основной элемент дескриптора приложения, внутри которого находятся все остальные элементы, — `application`;
- элемент `icon` определяет месторасположение значка, представляющего приложение;
- элемент `display-name` определяет отображаемое имя приложения;
- элемент `description` — краткое описание приложения;
- элемент `module` указывает тип и месторасположение каждого модуля;
- элемент `alt-dd` является дополнительным к элементу `module` и указывает URI-адрес файла дескриптора развертывания модуля;
- элемент `ejb` внутри элемента `module` указывает URI-адрес EJB-модуля — JAR-файла с расширением jar;
- элемент `java` внутри элемента `module` указывает URI-адрес клиентского Java-приложения — JAR-файла с расширением jar;

- элемент `web` внутри элемента `module` содержит элементы: `web-uri` — URI-адрес Web-модуля (JAR-файла с расширением war) и `context-root` — идентификатор для данного Web-приложения в сервере Java EE, являющийся частью URL-адреса клиентского запроса к данному Web-приложению;
- элемент `connector` внутри элемента `module` указывает URI-адрес адаптера ресурсов (архивного файла с расширением rar) — драйвера, с помощью которого осуществляется связь со специфической информационной системой предприятия (также как связь с базами данных осуществляется с помощью JDBC-драйвера);
- элемент `security-role` относится к системе безопасности Java EE и содержит элементы: необязательный элемент `description` — описание роли безопасности и элемент `role-name` — имя роли безопасности.

Система безопасности платформы Java EE

Система безопасности платформы Java EE является многоуровневой и обеспечивает авторизированный доступ (контроль доступа, определяющий права клиента) к защищенным ресурсам в Web-модулях и защищенным методам в EJB-модулях.

Система безопасности платформы Java EE на стороне сервера на уровне администратора сервера Java EE подразумевает создание пользователей в группах пользователей с логином и паролем.

В качестве примера создания пользователей сервера Java EE рассмотрим сервер Glassfish.

В среде NetBeans откроем окно **Services**. Щелкнем правой кнопкой мыши на узле **Servers | Glassfish** и выберем **Start**, затем — **View Admin Console**. В результате в Web-браузере, назначенном по умолчанию, откроется страничка, представляющая консоль администратора сервера приложений Glassfish.

Далее введем логин и пароль администратора, установленные при инсталляции сервера. Откроем последовательно узлы **Configuration | Security | Realms**.

Узел **Realms**, в свою очередь, будет содержать три узла: **file**, **certificate** и **admin-realm**, позволяющие управлять регистрацией пользователей сервера с помощью кнопки **Manage Users** консоли администратора и инструментов командной строки.

Realm — это набор пользователей и групп пользователей домена сервера, которые контролируются одной схемой аутентификации/авторизации. Домен

сервера — это область действия, в пределах которой определена одна политика безопасности, предусматривающая разрешения доступа исполняемого Java-кода к защищенным библиотекам API.

Файл политики безопасности домена сервера server.policy находится в папке domains\domain1\config каталога сервера Glassfish. Дополнительные файлы политик безопасности приложений расположены в папке domains\domain1 \generated\policy.

По умолчанию в сервере Glassfish созданы три набора realm — file, certificate и admin-realm.

Набор file realm содержит пользователей, которые связываются с сервером, используя незащищенный протокол HTTP. Логины и пароли пользователей набора file realm хранятся в файле keyfile папки domains\domain1\config. Добавлять пользователей в набор file realm можно с помощью кнопки **Manage Users** странички набора.

Набор certificate realm содержит пользователей, которые связываются с сервером по защищенному протоколу HTTPS. Сертификаты пользователей, используемые сервером для идентификации, хранятся в файле domains\domain1 \config\cacerts.jks и могут быть созданы и добавлены в данный файл с помощью соответствующих инструментов командной строки.

Набор admin-realm содержит перечень администраторов сервера, информация о которых хранится в файле domains\domain1\config\admin-keyfile. Добавлять пользователей в набор admin-realm также можно с помощью кнопки **Manage Users** странички набора.

Система безопасности платформы Java EE на стороне сервера на уровне приложения Java EE подразумевает использование разрешений, определяемых в общем файле политики безопасности (файл domains\domain1\config \server.policy сервера Glassfish) и в файле политики безопасности приложения (папка domains\domain1\generated\policy сервера Glassfish). При этом разрешения могут быть связаны с пользователями сервера.

Система разрешений, основанная на использовании файла политики безопасности, включена в систему безопасности платформы Java SE.

Изначально разрешения были связаны с URL-адресом исполняемого Java-кода (элемент `codeBase` записи `grant`), при этом исполняемый Java-код может быть подписан электронной подписью и иметь сертификат (элемент `signedBy` записи `grant`). Контроль над применением разрешений осуществляется объектом Access Controller (класс `java.security.AccessController`), который используется менеджером безопасности Security Manager (класс `java.lang.SecurityManager`).

Далее система безопасности платформы Java SE была дополнена сервисом Java Authentication and Authorization Service (JAAS) (библиотека `javax.security.auth`), который осуществляет идентификацию и авторизацию пользователя, позволяя связать разрешения с конкретным пользователем (элемент `Principal` записи `grant`).

Сервер Java EE использует сервис JAAS платформы Java SE с помощью интерфейса JACC (Java Authorization Contract for Containers), связывая роли безопасности с разрешениями и, таким образом, дополняя политику безопасности приложения.

Система безопасности платформы Java EE на стороне приложения Java EE, обеспечиваемая контейнерами сервера Java EE, подразумевает определение *ролей безопасности*.

Роль безопасности — это набор разрешений для доступа клиента к защищенным ресурсам в Web-модулях и защищенным методам в EJB-модулях.

Роли безопасности формируют права доступа к защищенным ресурсам в Web-модулях и защищенным методам в EJB-модулях декларативно (файл дескриптора развертывания или аннотации Java-кода) или программно в Java-коде.

Web-модули приложения Java EE

Web-модуль — это архивный JAR-файл с расширением war, содержащий в корневой папке:

- JSP-странички;
- различного рода ресурсы, например файлы изображений;
- HTML-странички;
- JSP-странички и HTML-странички могут включать апплеты, и, соответственно, в корневой папке будут содержаться классы апплетов;
- JSP-странички могут использовать для хранения данных JavaBeans-компоненты, классы которых также будут находиться в корневой папке;
- папку WEB-INF.

Папка WEB-INF Web-модуля в свою очередь содержит:

- файл web.xml дескриптора развертывания;
- папку classes, в которой находятся сервлеты;
- JavaBeans-компоненты и другие класс-файлы, исполняемые на стороне сервера;

- папку lib, содержащую библиотечные JAR-файлы, необходимые для выполнения класс-файлов папки classes;
- папку tags с файлами, содержащими многократно используемые фрагменты JSP-кода, представляющего пользовательские теги;
- дескриптор библиотеки пользовательских тегов — XML-файл с расширением tld.

Web-модуль может представлять не только компонент приложения Java EE, но и самостоятельное Web-приложение. В этом случае JAR-файл Web-модуля развертывается отдельно.

Дескриптор развертывания Web-модуля web.xml (XML Schema for the Servlet 2.5 deployment descriptor) содержит следующие элементы.

- Основной элемент `web-app`, имеющий атрибут `metadata-complete`, который указывает, является ли дескриптор развертывания полным (значение `true`) или существуют аннотации, которые необходимо учитывать контейнеру при развертывании приложения (значение `false`). В Web-приложении классы, использующие аннотации для введения ресурсов, должны располагаться в папке WEB-INF\classes или в JAR-файлах папки WEB-INF\lib архива приложения.
- Элемент `icon` содержит элементы `small-icon` и `large-icon`, определяющие значки Web-приложения.
- Отображаемое имя `display-name`.
- Описание Web-приложения `description`.
- Элемент `distributable` указывает, что Web-приложение является распределенным. При использовании элемента `distributable` только сериализуемые объекты могут сохраняться в объекте `javax.servlet.http.HttpSession`. После первого соединения клиентского приложения с сервером устанавливается сессия, в пределах которой передаются запросы от клиента серверу. Время жизни сессии определяется временем ожидания с момента последнего запроса. Сессионный объект `HttpSession` позволяет передавать информацию между запросами в пределах одной сессии. Объект `HttpSession` обеспечивает средства для хранения и доступа к данным и представляет собой хэш-таблицу, которая содержит пары "ключ — объект". Сессионный объект хранится сервером в пределах сессии персонально для каждого клиента. Сериализуемые объекты `java.io.Serializable` записываются на постоянное хранение в файловой системе в виде последовательности байтов и могут быть восстановлены в любое время, в любой сессии.

- Элемент `context-param` содержит элементы `param-name` и `param-value`, определяющие дополнительные конфигурационные данные Web-приложения, например адрес электронной почты Web-мастера.
- Элемент `filter` содержит элементы, описывающие фильтр Web-приложения: `icon` — значок фильтра, `filter-name` — имя фильтра, `display-name` — отображаемое имя фильтра, `description` — описание фильтра, `filter-class` — полное имя класса фильтра, `init-param` — параметры инициализации фильтра `param-name`, `param-value` и `description`.
- Элемент `filter-mapping` связывает фильтр с сервлетами — элемент `servlet-name` или статическими ресурсами — элемент `url-pattern`. Вложенный элемент `filter-name` указывает имя связываемого фильтра. Вложенный элемент `dispatcher` позволяет связать фильтр с запросами четырех типов: `REQUEST` (запрос от клиента, значение по умолчанию); `FORWARD` (переадресованный запрос от сервлета другим ресурсам — метод `forward()` интерфейса `javax.servlet.RequestDispatcher`); `INCLUDE` (в ответ на запрос включается содержимое других ресурсов — сервлетов, JSP-страниц, HTML-файлов — метод `include()` интерфейса `javax.servlet.RequestDispatcher`); `ERROR` (запрос обрабатывается с помощью странички ошибки).
- Элемент `listener` содержит элементы, описывающие компонент-обработчик событий Web-приложения. Вложенные элементы `icon`, `display-name`, `description` и `listener-class` указывают соответственно значок, отображаемое имя, описание и полное имя класса, реализующего интерфейсы-слушатели событий.
- Элемент `servlet` содержит элементы, описывающие сервлет. Вложенные элементы `icon`, `servlet-name`, `display-name` и `description` определяют значок, имя сервлета, отображаемое имя сервлета и описание сервлета. Вложенные элементы `servlet-class` и `jsp-file` указывают полное имя класса сервлета и путь JSP-файла. Вложенный элемент `init-param` описывает параметры инициализации сервлета с помощью элементов `param-name`, `param-value` и `description`. Если вложенный элемент `load-on-startup` имеет отрицательное значение или отсутствует, то контейнер загружает и инициализирует сервлет по необходимости, если значение элемента начинается с нуля, то контейнер загружает сервлет при развертывании Web-приложения, при этом значение элемента определяет порядок загрузки сервлета. Вложенный элемент `run-as` содержит элементы: `description` — дополнительное описание сервлета; `role-name` — имя роли безопасности сервлета. Вложенный элемент `security-role-ref` позволяет связать имя роли безопасности элемента `role-name` с ролью безопасности, определенной в коде сервлета, с помощью элементов `role-name` (имя роли безопас-

ности кода), `role-link` (имя роли безопасности дескриптора) и `description` (описание).

- Элемент `servlet-mapping` связывает сервлет с определенным типом URL-запроса с помощью элементов `servlet-name` (имя сервлета) и `url-pattern` (URL-шаблон запроса).
- Элемент `session-config` определяет максимальное время ожидания для сессий Web-приложения с помощью элемента `session-timeout`.
- Элемент `mime-mapping` с помощью элементов `extension` и `mime-type` связывает MIME-тип с файловым расширением.
- Элемент `welcome-file-list` содержит элемент `welcome-file`, который определяет перечень файлов, обрабатываемых сервером первыми при запросе клиента. Каталог, имя которого указывается в клиентском URL-запросе, должен содержать один из файлов, указанных в данном перечне.
- Элемент `error-page` связывает исключительные ситуации или ошибки HTTP-протокола с ресурсами Web-приложения. Вложенный элемент `error-code` содержит код ошибки HTTP-протокола, `exception-type` — полное имя Java-класса исключения, `location` — расположение файла Web-приложения.
- Элемент `jsp-config` содержит информацию о JSP-файлах Web-приложения, используя элементы: `taglib` включает в себя элементы `taglib-uri` (относительный URI-адрес библиотеки пользовательских тегов) и `taglib-location` (расположение дескриптора библиотеки пользовательских тегов); `jsp-property-group` представляет собой набор свойств группы JSP-страничек и включает в себя элементы `url-pattern` (тип URL-запроса, определяющий группу JSP-страничек, с которым связываются указанные свойства), `el-ignored` (деактивация языка выражений Expression Language (EL), принимает значения `true`, `false`), `scripting-invalid` (запрет использования Java-операторов в JSP-страницах, принимает значения `true`, `false`), `page-encoding` (кодировка JSP-страниц), `include-prelude` (относительный путь ресурса, который будет включен в начало JSP-страничек группы), `include-coda` (относительный путь ресурса, который будет включен в конец JSP-страничек группы), `is-xml` (обозначает, что JSP-странички группы должны интерпретироваться как XML-документы, принимает значения `true`, `false`), `deferred-syntax-allowed-as-literal` (разрешение использования символа `#`, принимает значения `true`, `false`), `trim-directive-whitespaces` (удаляет лишние пропуски из страничек, принимает значения `true`, `false`).
- Элемент `security-constraint` определяет права доступа к защищенным ресурсам Web-приложения для ролей безопасности.

Данный элемент включает в себя:

- display-name — отображаемое имя ограничения;
- web-resource-collection указывает набор защищенных ресурсов и HTTP-методов с помощью элементов web-resource-name (имя набора), description (описание), url-pattern (тип защищенного URL-запроса, связанный с ресурсами Web-приложения), http-method (HTTP-метод — GET или POST);
- auth-constraint обозначает роль безопасности, которой разрешается доступ к защищенному набору, включает элементы description (описание) и role-name (имя роли безопасности);
- user-data-constraint определяет способ связи клиента с сервером, содержит элементы description (описание), transport-guarantee (NONE — приложение не требует никаких гарантий при передаче данных, INTEGRAL — данные не должны изменяться при транспортировке, CONFIDENTIAL — данные должны передаваться по защищенному каналу).

- Элемент login-config определяет способ идентификации клиента и состоит из элементов: auth-method — формирует механизм идентификации (BASIC — идентификация основана на логине и пароле и использует HTTP-протокол, DIGEST — идентификация основана на логине и пароле и использует HTTPS-протокол, FORM — используются специально созданная страничка с формой для ввода логина и пароля и страничка ошибки, CLIENT-CERT — идентификация основана на сертификате клиента); realm-name — имя realm для идентификации BASIC; form-login-config указывает странички формы и ошибки для идентификации FORM, используя элементы form-login-page и form-error-page.
- Элемент security-role содержит описание роли безопасности, используя элементы description (описание роли безопасности) и role-name (имя роли безопасности).
- Элемент env-entry содержит запись окружения, представляющую собой пару "имя env-entry-name — значение env-entry-value", доступ к которой осуществляется с помощью интерфейса Java Naming and Directory Interface (JNDI) (пакет javax.naming платформы Java SE). Использование записей окружения позволяет изменять логику приложения без корректировки исходного программного кода. Интерфейс JNDI дает возможность осуществлять доступ к системам именования и службам каталогов, обеспечивая хранение и получение любого типа по имени объекта. Элемент description содержит описание записи окружения, элемент env-entry-type — тип записи окружения (например, java.lang.Float).

- Элемент `ejb-ref` позволяет получить ссылку на используемый EJB-компонент в Web-приложении с помощью интерфейса JNDI. Элемент `ejb-ref` содержит элементы: `description` — описание ссылки; `ejb-ref-name` — JNDI-имя ссылки контекста `java:comp/env`; `ejb-ref-type` — тип EJB-компонента (`Entity` представляет информацию, хранящуюся в базе данных, `Session` — существующий в пределах сессии компонент, обеспечивающий необходимую функциональность для клиента); `home` — полное имя домашнего интерфейса EJB-компонента; `remote` — полное имя удаленного интерфейса EJB-компонента; `ejb-link` — связывает ссылку с EJB-компонентом.
- Элемент `ejb-local-ref` позволяет получить ссылку, с помощью интерфейса JNDI, на используемый EJB-компонент, который управляется тем же самым контейнером, что и Web-приложение. Данный элемент включает в себя элементы: `description` — описание ссылки; `ejb-ref-name` — JNDI-имя ссылки; `ejb-ref-type` — тип EJB-компонента; `local-home` — полное имя локального домашнего интерфейса EJB-компонента; `local` — полное имя локального интерфейса EJB-компонента; `ejb-link` — связывает ссылку с EJB-компонентом.
- Элемент `service-ref` описывает ссылку на используемый Web-сервис, доступ к которой возможно получить с помощью интерфейса JNDI. Элемент `service-ref` является контейнером для следующих элементов:
  - `description` — описание ссылки;
  - `display-name` — отображаемое имя ссылки;
  - `icon` — значок;
  - `service-ref-name` — JNDI-имя ссылки;
  - `service-interface` — полное имя интерфейса `javax.xml.rpc.Service` JAX-RPC API или его расширения, возвращаемое методом `lookup()` класса `InitialContext` JNDI API (библиотека Java APIs for XML based RPC (Remote Procedure Call), JAX-RPC). Определяет механизм, позволяющий клиенту, взаимодействующему с удаленным сервером, вызывать удаленные процедуры, причем вызов основан на XML-протоколе SOAP. Интерфейс `Service` обеспечивает доступ со стороны клиента к функциональности Web-сервиса;
  - `wsdl-file` — путь WSDL-файла, который описывает Web-сервис и может быть использован клиентом для доступа к функциональности Web-сервиса;
  - `jaxrpc-mapping-file` — путь JAX-RPC-файла, который описывает соответствие между WSDL-описанием Web-сервиса и его Java-реализацией и используется контейнером при генерации клиентской заглушки (stub)

и серверной заглушки (tie) — программного кода, обеспечивающего SOAP-протокол для клиента и Web-сервера;

- `service-qname` — XML-имя (класс `javax.xml.namespace.QName` платформы Java SE) Web-сервиса, определенное в элементе `<wsdi:service>` WSDL-файла. Данный элемент используется в случае, если WSDL-файл содержит несколько элементов `<wsdi:service>`;
- `port-component-ref` — интерфейс Web-сервиса Service Endpoint Interface (SEI), содержащий методы, обеспечивающие функциональность Web-сервиса;
- `handler` описывает, при наличии, специальную реализацию интерфейса `javax.xml.rpc.handler.Handler`, отвечающую за обработку SOAP-сообщений.

- Элемент `resource-ref` описывает JNDI-ссылку на внешние ресурсы с помощью следующих элементов: `description` — описание ссылки; `res-ref-name` — JNDI-имя ссылки; `res-type` — тип источника данных, определяется полным именем Java-класса или интерфейса, реализованного источником данных (например, `javax.sql.DataSource`); `res-auth` — тип идентификации (приложением — `Application`, или контейнером — `Container`); `res-sharing-scope` определяет, является ресурс совместно используемым (`Shareable`) или нет (`Unshareable`).
- Элемент `resource-env-ref` описывает JNDI-ссылку на системные (администрируемые) объекты системы JMS (Java Messaging System) с помощью следующих элементов: `description` — описание ссылки; `resource-env-ref-name` — JNDI-имя ссылки; `resource-env-ref-type` — тип объекта (например, `javax.jms.ConnectionFactory`).
- Элемент `message-destination-ref` описывает JNDI-ссылку системы JMS на адресата сообщений с помощью следующих элементов: `description` — описание ссылки; `message-destination-ref-name` — JNDI-имя ссылки; `message-destination-type` — тип адресата (например, `javax.jms.Topic`); `message-destination-usage` указывает, является адресат потребителем сообщений (`Consumes`) или/и генератором сообщений (`Produces`); `message-destination-link` связывает ссылку с адресатом, который описывается элементом `message-destination`.
- Элемент `message-destination` описывает адресата сообщений, используя элементы: `description` — описание адресата; `display-name` — отображаемое имя; `icon` — значок; `message-destination-name` — имя адресата.
- Элемент `locale-encoding-mapping-list` обеспечивает локализацию кодировки с помощью элементов `locale` и `encoding`, содержащихся в элементе `locale-encoding-mapping`.

Платформа Java EE 5 позволяет использовать вместо элементов дескриптора развертывания аннотации, указываемые в коде приложения. Аннотации имеют атрибуты, которые заменяют вложенные элементы дескриптора развертывания:

- `@WebServlet` заменяет элемент `servlet` и `servlet-mapping`, имеет атрибуты: `value` или `urlPatterns` — тип URL-запроса для сервлета; `name` — имя сервлета; `loadOnStartup` — порядок загрузки сервлета; `initParams` — параметры инициализации; `icon` — значок; `description` — описание; `asyncSupported` указывает, поддерживает ли сервлет асинхронный режим работы; `asyncTimeout` — время ожидания для асинхронных операций;
- `@ServletFilter` заменяет элемент `filter` и `filter-mapping`, имеет атрибуты: `value` или `urlPatterns` — тип URL-запроса для фильтра; `servletNames` — имя сервлета, к которому применяется фильтр; `initParams` — параметры инициализации; `icon` — значок; `filterName` — имя фильтра; `displayName` — отображаемое имя фильтра; `dispatcherTypes` — тип запроса, с которым связан фильтр; `description` — описание; `asyncSupported` указывает, поддерживает ли фильтр асинхронный режим работы; `asyncTimeout` — время ожидания для асинхронных операций;
- `@WebServletContextListener` заменяет элемент `listener`, имеет атрибут `description` — описание слушателя;
- `@InitParam` заменяет подэлемент `init-param` элементов `servlet` и `filter`, имеет атрибуты `description`, `name` и `value`;
- `@DeclareRoles` заменяет элемент `security-role`, имеет атрибут `value` — имя роли безопасности.
- `@EJB` заменяет `ejb-ref` и `ejb-local-ref`.
- `@Resource` заменяет `resource-ref`, `message-destination-ref`, `resource-env-ref`, `env-entry`, имеет атрибуты: `authenticationType` — тип идентификации ресурса; `description` — описание ресурса; `mappedName` — специфическое для сервера имя, с которым связывается ресурс; `name` — JNDI-имя ресурса; `shareable` — индикатор совместного использования ресурса; `type` — Java-тип ресурса.
- `@PersistenceContext` — аннотация используется для добавления в клиента EJB 3.0 объекта `javax.persistence.EntityManager`, обеспечивающего взаимодействие с контекстом сохранения объектов (persistence context). Persistence context — это набор сохраняемых в базе данных экземпляров сущностей, представленных экземплярами EJB-компонентов типа Entity. В наборе persistence context объекты Entity управляются ассоциированными с данным набором объектами `EntityManager`, которые позволяют до-

бавлять в набор и удалять из набора экземпляры сущностей, осуществлять их поиск в наборе, используя уникальные идентификаторы, и выполнять различного рода запросы.

- `@PersistenceContexts` — позволяет объявить в EJB3.0-клиенте более одной аннотации `@PersistenceContext`.
- `@PersistenceUnit` — аннотация используется для добавления в клиента EJB 3.0 объекта `javax.persistence.EntityManagerFactory`, являющегося фабрикой для объектов `EntityManager`. Разница между объектом `EntityManager`, добавленным с помощью аннотации `@PersistenceContext`, и объектом `EntityManager`, полученным методом `createEntityManager()` интерфейса `EntityManagerFactory`, заключается в том, что добавленный аннотацией объект `EntityManager` полностью управляется контейнером, включая заботу о транзакциях ассоциированного набора persistence context. В то время как созданный фабрикой объект `EntityManager` полностью управляется клиентским приложением, объект `EntityManagerFactory` ассоциируется с конкретной единицей сохранения (persistence unit) с помощью атрибута `unitName` аннотации. Persistence unit определяет набор классов Entity, представляющих данные определенной базы данных. Persistence unit описывается в конфигурационном файле persistence.xml.
- `@PersistenceUnits` позволяет объявить в клиенте EJB 3.0 более одной аннотации `@PersistenceUnit`.
- `@PostConstruct` — аннотацией отмечается метод, который будет вызван контейнером после того, как все ресурсы будут добавлены, но перед вызовом контейнером методов жизненного цикла компонента. Данный метод может быть использован для выполнения какого-либо рода инициализаций после добавления ресурсов, хотя отмеченный аннотацией `@PostConstruct` метод будет вызван контейнером, даже если никакие ресурсы не добавляются в компонент.
- `@PreDestroy` — аннотацией отмечается метод, который будет вызван контейнером перед удалением экземпляра компонента из контейнера. Данный метод обычно используется для освобождения ресурсов, задействованных компонентом.
- `@Resources` является контейнером для аннотаций `@Resource`.
- `@RunAs` заменяет элемент `run-as` дескриптора развертывания Web-модуля и позволяет исполнять сервлет в соответствии с определенной ролью безопасности, связанной с пользователем или группой пользователей сервера.
- `@WebServiceRef` — аннотация заменяет элемент `service-ref` дескриптора развертывания Web-модуля и предназначена для объявления ссылки на

используемый ресурс — JAX-WS 2.0 Web-сервис. Аннотация имеет атрибуты: `name` — JNDI-имя ресурса; `type` — Java-тип ресурса (по умолчанию `java.lang.Object.class`); `mappedName` — позволяет указать связь со специфическим именем ресурса; `value` — класс Web-сервиса, расширяющий класс `javax.xml.ws.Service` и обеспечивающий клиентский доступ к функциональности Web-сервиса; `wsdlLocation` — URL-адрес WSDL-документа Web-сервиса.

- `@WebServiceRefs` позволяет объявить более одной аннотации `@WebServiceRef`.

Таким образом, в Web-модуле роли безопасности, связанные с ними права доступа к защищенным ресурсам и способ идентификации клиента описываются элементами файла дескриптора развертывания `security-role`, `security-constraint` и `login-config`, аннотациями `@DeclareRoles` и `@RunAs`.

Система безопасности Web-приложения может также включать в себя использование в Java-коде методов `getRemoteUser()`, `isUserInRole()` и `getUserPrincipal()` интерфейса `javax.servlet.http.HttpServletRequest`.

EJB-модули приложения Java EE

EJB-модуль — это архивный JAR-файл, содержащий EJB-компоненты и файл дескриптора развертывания ejb-jar.xml в папке META-INF архива.

Дескриптор развертывания EJB-модуля ejb-jar.xml (XML Schema for the EJB 3.0 deployment descriptor) содержит следующие элементы:

- `ejb-jar` — корневой элемент дескриптора развертывания;
- элементы `description`, `display-name` и `icon`;
- `assembly-descriptor` содержит информацию, объединяющую EJB-компоненты в модуле:
  - `security-role` описывает роль безопасности, содержит элементы `description` и `role-name`;
  - `method-permission` определяет для ролей безопасности права доступа к защищенным методам EJB-компонентов. Содержит элементы: `description`; `role-name`; `unchecked` определяет, что метод перед вызовом не проверяется контейнером для авторизации; `method` описывает защищенный метод, содержит элементы `description`, `ejb-name`, `method-intf` (значения `Home`, `Remote`, `LocalHome`, `Local`), `method-name`, `method-params` (вложенные `method-param`);
  - `container-transaction` описывает, как контейнер должен управлять транзакциями при вызове методов EJB-компонентов.

Содержит элементы:

- description;
- method с элементами description, ejb-name, method-intf (значения Home, Remote, LocalHome, Local), method-name, method-params (вложенные method-param);
- trans-attribute определяет, каким образом контейнер должен управлять транзакциями, принимает значения:
  - NotSupported — транзакция не создается;
  - Required — если клиентский вызов поддерживается транзакцией, то вызов методов EJB-компонентов вовлекается в ту же транзакцию; если клиентский вызов не связан с транзакцией, то контейнер создает новую транзакцию перед вызовом методов EJB-компонентов;
  - Supports — если клиентский вызов в транзакции, то контейнер действует по сценарию Required; если нет, то по сценарию NotSupported;
  - RequiresNew — контейнер в любом случае создает новую транзакцию перед вызовом методов EJB-компонентов;
  - Mandatory — клиентский вызов всегда должен быть связан с транзакцией, если нет — генерируется исключение;
  - Never — клиентский вызов не должен быть транзакционным, иначе генерируется исключение;

- interceptor-binding связывает классы, содержащие методы-перехватчики (interceptor classes), с EJB-компонентами. Включает в себя элементы:
  - description;
  - ejb-name;
  - interceptor-class;
  - interceptor-order определяет порядок подключения классов-перехватчиков с помощью вложенных элементов interceptor-class;
  - exclude-default-interceptors определяет, применяются ли классы-перехватчики по умолчанию для EJB-компонента (классы-перехватчики по умолчанию указываются с помощью <interceptor-binding> <ejb-name>*</ejb-name> <interceptor-class> INTERCEPTOR </interceptor-class></interceptor-binding>), принимает значение true/false;

- `exclude-class-interceptors` определяет, применяются ли классы-перехватчики для методов бизнес-интерфейса EJB-компонента, принимает значение `true`/`false`;
- `method` указывает бизнес-метод, к которому применяются классы-перехватчики, включает в себя элементы `method-name` и `method-param`;

- `message-destination` описывает адресата сообщений, используя элементы: `description` — описание адресата; `display-name` — отображаемое имя; `icon` — значок; `message-destination-name` — имя адресата;
- `exclude-list` содержит список методов EJB-компонентов, запрещенных для клиентского вызова, включает в себя элементы `description` и `method` с элементами `description`, `ejb-name`, `method-intf` (значения `Home`, `Remote`, `LocalHome`, `Local`), `method-name`, `method-params` (вложенные `method-param`);
- `application-exception` описывает класс, представляющий нестандартную исключительную ситуацию бизнес-логики EJB-приложения, включает в себя элементы `exception-class` (полное имя класса) и `rollback` (значения `true`/`false` — откат транзакции перед пересылкой исключения клиенту);

- `interceptors` описывает классы, содержащие методы-перехватчики. Включает в себя элементы: `description`; `interceptor` с вложенными элементами `description`, `interceptor-class` (полное имя класса), `around-invoke` (метод класса-перехватчика, вызываемый перед вызовом методов бизнес-интерфейса сессионного EJB-компонента, имеет вложенные элементы `class` и `method-name`), `post-activate` и `pre-passivate` (методы-перехватчики, вызываемые контейнером перед активацией и переводом в пассивное состояние `Stateful` компонента);
- `relationships` описывает связи между EJB 2.1 Entity-компонентами, сохранением которых управляет контейнер (применяется при `<persistence-type>Container</persistence-type> <cmp-version>2.x</cmp-version>`):
  - `description` — описание;
  - `ejb-relation` описывает связь между двумя Entity-компонентами с помощью элементов: `description`; `ejb-relation-name` — имя связи; `ejb-relationship-role` — должно быть два элемента для каждой связи, для каждого из двух компонентов описывает роль в связи, состоит из элементов `description`, `ejb-relationship-role-name` (имя роли), `multiplicity` (тип связи для компонента `One` или `Many`), `cascade-delete` (каскадное удаление для связей типа 1:1 и 1:*M*), `relationship-role-source` (указывает имя компонента с помощью элементов `description` и `ejb-name`), `cmr-field` (описывает поле компонента, участвующее в связи,

с помощью элементов description, cmr-field-name и cmr-field-type — элемент должен быть, если тип поля — java.util.Collection или java.util.Set);

- ejb-client-jar — все интерфейсы и классы, которые необходимы клиенту для доступа к EJB-компонентам приложения, такие как домашние (home) и удаленные (remote) интерфейсы, классы первичных ключей и другие вспомогательные классы, могут быть выделены в отдельный JAR-файл (с расширением ear) файла, относительный путь к которому указывает данный элемент.
- enterprise-beans содержит описание одного или нескольких EJB-компонентов. Содержит вложенные элементы Session, Entity и message-driven.
- Session описывает EJB-компонент типа Session Bean — объект взаимодействует с единственным клиентом, и состояние объекта сохраняется в пределах клиентской сессии (Stateful) или в пределах клиентского вызова (Stateless). Stateless-компонент может быть реализован как Web-сервис:
  - элементы description, display-name и icon;
  - ejb-name — имя EJB-компонента в модуле;
  - mapped-name обеспечивает специфическую информацию, такую как JNDI-имя удаленного (remote) домашнего или бизнес-интерфейса (home/business) компонента;
  - home, remote, local-home, local, business-local, business-remote и service-endpoint — соответствующие интерфейсы сессионного EJB-компонента;
  - ejb-class содержит полное имя класса EJB-компонента;
  - session-type — тип сессионного компонента (Stateful или Stateless);
  - timeout-method — метод EJB-компонента, использующего контейнерный сервис Timer, вызываемый контейнером в заданное время, по истечении заданного временного интервала или с заданной периодичностью. Элемент имеет вложенные элементы — method-name и method-params с элементами method-param. Сервис Timer может использоваться всеми EJB-компонентами, кроме Stateful и EJB 3.0 Entity;
  - init-method — метод EJB 3.0 Stateful-компонента соответствует методу create() домашнего интерфейса EJB 2.1 Stateful-компонента. Элемент имеет вложенные элементы create-method и bean-method с элементами method-name и method-params с элементами method-param;
  - remove-method — метод EJB 3.0 Stateful-компонента, соответствует методу remove() EJB 2.1 Stateful-компонента. Элемент имеет вложенный

элемент `bean-method` с элементами `method-name` и `method-params` с элементами `method-param`;

- `transaction-type` — тип управления транзакциями EJB-компонента (`Bean` или `Container`);
- `around-invoke` — метод класса-перехватчика (interceptor class), вызываемый перед вызовом методов бизнес-интерфейса сессионного EJB-компонента. Элемент имеет вложенные элементы `class` и `method-name`;
- `post-activate`, `pre-passivate` — методы-перехватчики, вызываемые контейнером перед активацией и переводом в пассивное состояние Stateful-компонента. Методы предназначены для работы с используемыми компонентом ресурсами;
- `security-role-ref` — связывает роль безопасности, определенную в коде EJB-компонента с ролью безопасности дескриптора развертывания, включает в себя элементы `description`, `role-name` и `role-link`;
- `security-idEntity` — содержит элементы: `description`; `run-as` — устанавливает, что указанная с помощью элемента `role-name` роль безопасности определяет доступ ко всему EJB-компоненту; `use-caller-idEntity` — взаимоисключающий элемент с `run-as`, устанавливает, что идентификация клиента используется для разрешения доступа к защищенным методам EJB-компонента (является элементом по умолчанию);
- `env-entry` — определяет значения переменных окружения EJB-компонента. Содержит элементы: `description`; `env-entry-name` — JNDI-имя переменной окружения (контекст `java:comp/env`); `env-entry-type` — Java-тип значения переменной окружения (`java.lang.Boolean`, `java.lang.Byte`, `java.lang.Character`, `java.lang.String`, `java.lang.Short`, `java.lang.Integer`, `java.lang.Long`, `java.lang.Float`, `java.lang.Double`); `env-entry-value` — значение переменной окружения;
- `ejb-ref` — описывает JNDI-ссылку на используемый EJB-компонент, содержит элементы: `description`; `ejb-ref-name` — JNDI-имя ссылки; `ejb-ref-type` — тип используемого EJB-компонента (`Entity`, `Session`); `home` — домашний интерфейс используемого EJB-компонента; `remote` — удаленный (бизнес) интерфейс используемого EJB-компонента; `ejb-link` — связывает ссылку с компонентом;
- `ejb-local-ref` — описывает JNDI-ссылку на используемый EJB-компонент в том же контейнере, содержит элементы: `description`; `ejb-ref-name` — JNDI-имя ссылки; `ejb-ref-type` — тип используемого EJB-компонента (`Entity`, `Session`); `local-home` — локальный домашний интерфейс используемого EJB-компонента; `local` — локальный (бизнес)

интерфейс используемого EJB-компонента; `ejb-link` — связывает ссылку с компонентом;

- `resource-ref` — описывает JNDI-ссылку на используемый внешний ресурс, содержит элементы: `description`; `res-ref-name` — JNDI-имя ссылки; `res-type` — Java-тип ресурса; `res-auth` — способ авторизации ресурса (`Application` или `Container`); `res-sharing-scope` — определяет, является ресурс совместно используемым (`Shareable`) или нет `Unshareable`;
- элемент `resource-env-ref` описывает JNDI-ссылку на системные (администрируемые) объекты системы JMS (Java Messaging System) с помощью следующих элементов: `description` — описание ссылки; `resource-env-ref-name` — JNDI-имя ссылки; `resource-env-ref-type` — тип объекта (например, `javax.jms.ConnectionFactory`);
- элемент `message-destination-ref` описывает JNDI-ссылку системы JMS на адресата сообщений с помощью следующих элементов: `description` — описание ссылки; `message-destination-ref-name` — JNDI-имя ссылки; `message-destination-type` — тип адресата (например, `javax.jms.Topic`); `message-destination-usage` — указывает, является адресат потребителем сообщений (`Consumes`) или/и генератором сообщений (`Produces`); `message-destination-link` — связывает ссылку с адресатом, который описывается элементом `message-destination`;
- `service-ref` описывает ссылку на используемый Web-сервис аналогично элементу `service-ref` дескриптора развертывания Web-модуля;

❒ `Entity` описывает EJB-компонент типа Entity — сохраняемый объект представляет данные базы данных:

- `description`, `display name` и `icon`;
- `ejb-name` — имя EJB-компонента в модуле;
- `mapped-name` обеспечивает специфическую информацию, такую как JNDI-имя удаленного домашнего (remote home) интерфейса компонента;
- `home`, `remote`, `local-home`, `local` — соответствующие интерфейсы Entity-компонента;
- `ejb-class` содержит полное имя класса EJB-компонента;
- `persistence-type` определяет тип управления сохранением для EJB 2.1 Entity-компонентов (самим компонентом — `Bean` или контейнером — `Container`);
- `prim-key-class` указывает полное имя класса, представляющего уникальный идентификатор (первичный ключ) для EJB 2.1 Entity-

компонента. Данный элемент необходим в случае `<persistence-type>Bean</persistence-type>` и может представлять как простой первичный ключ (например, `java.lang.String`), так и составной в виде специально созданного класса;

- `reentrant` указывает, может ли EJB-компонент быть повторно вызван в пределах одной транзакции, принимает значения `True` или `False`;
- `cmp-version` указывает версию Entity-компонента в случае `<persistence-type>Container</persistence-type>`, принимает значения 1.*x* или 2.*x*;
- `abstract-schema-name` — в случае `<cmp-version>2.x</cmp-version>` указывает имя таблицы базы данных, которая представлена Entity-компонентом. Данное имя используется в запросах EJB QL (язык запросов, похожий на SQL);
- `cmp-field` — в случае `<persistence-type>Container</persistence-type>` указывает сохраняемые контейнером поля Entity-компонента. Содержит элементы `description` и `field-name`;
- `primkey-field` — в случае `<persistence-type>Container</persistence-type>` указывает поле Entity-компонента, представляющее простой первичный ключ. Данное поле должно быть также указано в элементе `cmp-field`;
- `query` — в случае `<persistence-type>Container</persistence-type>` указывает методы Entity-компонента, представляющие запросы типа `find` (домашний интерфейс) или `select` (класс реализации компонента). Содержит элементы: `description`; `query-method` описывает метод с помощью элементов `method-name` и `method-params` (вложенные `method-param`); `result-type-mapping` — в случае, если метод типа `select` возвращает объект Entity, указывает тип объекта `Local` или `Remote`; `ejb-ql` содержит строку EJB QL-запроса;
- `security-role-ref` связывает роль безопасности, определенную в коде EJB-компонента с ролью безопасности дескриптора развертывания, включает в себя элементы `description`, `role-name` и `role-link`;
- `security-idEntity` содержит элементы: `description`; `run-as` — устанавливает, что указанная с помощью элемента `role-name` роль безопасности определяет доступ ко всему EJB-компоненту; `use-caller-idEntity` — взаимоисключающий элемент с `run-as`, устанавливает, что идентификация клиента используется для разрешения доступа к защищенным методам EJB-компонента (является элементом по умолчанию);
- `env-entry` определяет значения переменных окружения EJB-компонента. Содержит элементы: `description`; `env-entry-name` — JNDI-имя

переменной окружения (контекст java:comp/env); env-entry-type — Java-тип значения переменной окружения (java.lang.Boolean, java.lang.Byte, java.lang.Character, java.lang.String, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double); env-entry-value — значение переменной окружения;

- ejb-ref описывает JNDI-ссылку на используемый EJB-компонент, содержит элементы: description; ejb-ref-name — JNDI-имя ссылки; ejb-ref-type — тип используемого EJB-компонента (Entity, Session); home — домашний интерфейс используемого EJB-компонента; remote — удаленный (бизнес) интерфейс используемого EJB-компонента; ejb-link связывает ссылку с компонентом;
- ejb-local-ref описывает JNDI-ссылку на используемый EJB-компонент в том же контейнере, содержит элементы: description; ejb-ref-name — JNDI-имя ссылки; ejb-ref-type — тип используемого EJB-компонента (Entity, Session); local-home — локальный домашний интерфейс используемого EJB-компонента; local — локальный (бизнес) интерфейс используемого EJB-компонента; ejb-link связывает ссылку с компонентом;
- resource-ref описывает JNDI-ссылку на используемый внешний ресурс, содержит элементы: description; res-ref-name — JNDI-имя ссылки; res-type — Java-тип ресурса; res-auth — способ авторизации ресурса (Application или Container); res-sharing-scope определяет, является ресурс совместно используемым (Shareable) или нет (Unshareable);
- resource-env-ref описывает JNDI-ссылку на системные (администрируемые) объекты системы JMS (Java Messaging System) с помощью следующих элементов: description — описание ссылки; resource-env-ref-name — JNDI-имя ссылки; resource-env-ref-type — тип объекта (например, javax.jms.ConnectionFactory);
- message-destination-ref описывает JNDI-ссылку системы JMS на адресата сообщений с помощью следующих элементов: description — описание ссылки; message-destination-ref-name — JNDI-имя ссылки; message-destination-type — тип адресата (например, javax.jms.Topic); message-destination-usage указывает, является адресат потребителем сообщений (Consumes) или/и генератором сообщений (Produces); message-destination-link связывает ссылку с адресатом, который описывается элементом message-destination;
- service-ref описывает ссылку на используемый Web-сервис аналогично элементу service-ref дескриптора развертывания Web-модуля;

- `message-driven` описывает EJB-компонент типа Message-Driven — объект, асинхронно исполняемый в результате получения сообщения от клиента и сохраняющий свое состояние в рамках запроса (Stateless). Message-Driven-компонент действует как слушатель сообщений сервиса доставки сообщений, такого как Java Message Service (JMS), и отличается от Session- и Entity-компонентов тем, что клиент не может получить доступ к Message-Driven-компоненту через интерфейсы. Message-Driven-компонент состоит только из EJB-класса:
  - `description`, `display name` и `icon`;
  - `ejb-name` — имя EJB-компонента в модуле;
  - `mapped-name` обеспечивает специфическую информацию, такую как JNDI-имя адресата, от которого компонент должен получать сообщения;
  - `ejb-class` — полное имя класса Message-Driven-компонента;
  - `messaging-type` указывает интерфейс слушателя сообщений, реализуемый Message-Driven-компонентом. Например, JMS Message-Driven-компонент должен реализовывать интерфейс `javax.jms.MessageListener`;
  - `timeout-method` — метод Message-Driven-компонента, использующего контейнерный сервис `Timer`. Элемент имеет вложенные элементы — `method-name` и `method-params` с элементами `method-param`;
  - `transaction-type` — тип управления транзакциями EJB-компонента (`Bean` или `Container`);
  - `message-destination-type` — тип адресата (`javax.jms.Topic` или `javax.jms.Queue`);
  - `message-destination-link` — связывает Message-Driven-компонент с источником сообщений — объектом `Topic` или `Queue`;
  - `activation-config` — информация о конфигурационных свойствах компонента. Содержит элементы `description` и `activation-config-property`. Элемент `activation-config-property` содержит элементы `activation-config-property-name` (имя свойства) и `activation-config-property-value` (значение свойства). Для JMS Message-Driven-компонента имена свойств могут быть следующими:
    - `acknowledgeMode` — в случае, если транзакция при получении сообщения создается в коде компонента, т. е. `<transaction-type>Bean</transaction-type>`. Значение свойства определяет способ уведомления об успешном получении сообщения: `AUTO_ACKNOWLEDGE`

(контейнер уведомляет адресата после успешного выполнения метода `onMessage()` интерфейса `javax.jms.MessageListener`) или `DUPS_OK_ACKNOWLEDGE` (контейнер уведомляет адресата тогда, когда ему это удобно, поэтому есть вероятность повторного получения сообщения от адресата);

  - `messageSelector` — с помощью значений свойства можно указать сообщения, которые компонент сможет получать;
  - `destinationType` — значение свойства указывает тип адресата сообщений `javax.jms.Queue` или `javax.jms.Topic`;
  - `subscriptionDurability` — в случае подписки на сообщения, т. е. `<destination-type>javax.jms.Topic</destination-type>`, значение свойства определяет, получает ли компонент все сообщения, даже если они были посланы, когда компонент был неактивен (`Durable`), или только те сообщения, которые были посланы при активном компоненте (`NonDurable`);

- `around-invoke` — метод класса-перехватчика (interceptor class), вызываемый контейнером перед вызовом метода интерфейса слушателя сообщений, в случае JMS метод интерфейса слушателя сообщений — это метод `onMessage()` интерфейса `javax.jms.MessageListener`. Элемент имеет вложенные элементы `class` и `method-name`;

- `security-idEntity` содержит элементы: `description`; `run-as` — устанавливает, что указанная с помощью элемента `role-name` роль безопасности определяет доступ ко всему EJB-компоненту; `use-caller-idEntity` — взаимоисключающий элемент с `run-as`, устанавливает, что идентификация клиента используется для разрешения доступа к защищенным методам EJB-компонента (является элементом по умолчанию);

- `env-entry` определяет значения переменных окружения EJB-компонента. Содержит элементы: `description`; `env-entry-name` — JNDI-имя переменной окружения (контекст `java:comp/env`); `env-entry-type` — Java-тип значения переменной окружения (`java.lang.Boolean`, `java.lang.Byte`, `java.lang.Character`, `java.lang.String`, `java.lang.Short`, `java.lang.Integer`, `java.lang.Long`, `java.lang.Float`, `java.lang.Double`); `env-entry-value` — значение переменной окружения;

- `ejb-ref` описывает JNDI-ссылку на используемый EJB-компонент, содержит элементы: `description`; `ejb-ref-name` — JNDI-имя ссылки; `ejb-ref-type` — тип используемого EJB-компонента (`Entity`, `Session`); `home` — домашний интерфейс используемого EJB-компонента; `remote` — удаленный (бизнес) интерфейс используемого EJB-компонента; `ejb-link` связывает ссылку с компонентом;

- `ejb-local-ref` описывает JNDI-ссылку на используемый EJB-компонент в том же контейнере, содержит элементы: `description`; `ejb-ref-name` — JNDI-имя ссылки; `ejb-ref-type` — тип используемого EJB-компонента (`Entity`, `Session`); `local-home` — локальный домашний интерфейс используемого EJB-компонента; `local` — локальный (бизнес) интерфейс используемого EJB-компонента; `ejb-link` связывает ссылку с компонентом;
- `resource-ref` описывает JNDI-ссылку на используемый внешний ресурс, содержит элементы: `description`; `res-ref-name` — JNDI-имя ссылки; `res-type` — Java-тип ресурса; `res-auth` — способ авторизации ресурса (`Application` или `Container`); `res-sharing-scope` — определяет, является ресурс совместно используемым (`Shareable`) или нет (`Unshareable`);
- элемент `resource-env-ref` описывает JNDI-ссылку на системные (администрируемые) объекты системы JMS (Java Messaging System) с помощью следующих элементов: `description` — описание ссылки; `resource-env-ref-name` — JNDI-имя ссылки; `resource-env-ref-type` — тип объекта (например, `javax.jms.ConnectionFactory`);
- элемент `message-destination-ref` описывает JNDI-ссылку системы JMS на адресата сообщений с помощью следующих элементов: `description` — описание ссылки; `message-destination-ref-name` — JNDI-имя ссылки; `message-destination-type` — тип адресата (например, `javax.jms.Topic`); `message-destination-usage` указывает, является адресат потребителем сообщений (`Consumes`) или/и генератором сообщений (`Produces`); `message-destination-link` связывает ссылку с адресатом, который описывается элементом `message-destination`;
- `service-ref` описывает ссылку на используемый Web-сервис аналогично элементу `service-ref` дескриптора развертывания Web-модуля.

Спецификация EJB 3.0 позволяет использовать в Java-коде компонента вместо элементов дескриптора развертывания следующие аннотации:

- ❒ `@DeclareRoles` заменяет `security-role-ref`;
- ❒ `@RolesAllowed` — `security-role`, `method-permission`;
- ❒ `@PermitAll` и `@DenyAll` — `method-permission`;
- ❒ `@RunAs` — `security-idEntity`;
- ❒ `@TransactionAttribute` — `container-transaction`;
- ❒ `@TransactionManagement` — `transaction-type`;
- ❒ `@Resource` — `resource-env-ref`, `message-destination-ref`, `resource-ref`, `env-entry`;

- `@Resources` является контейнером для аннотаций `@Resource`;
- `@EJB` — `ejb-ref` и `ejb-local-ref`;
- `@EJBs` является контейнером для аннотаций `@EJB`;
- `@PostConstruct`, `@PreDestroy`, `@PostActivate` и `@PrePassivate` определяют методы-перехватчики;
- `@ActivationConfigProperty` заменяет `activation-config`;
- `@ApplicationException` — `application-exception`;
- `@Init` — `init-method`;
- `@Local` указывает локальный бизнес-интерфейс (business-local);
- `@LocalHome` — локальный домашний интерфейс (local-home);
- `@Remote` — удаленный бизнес-интерфейс (business-remote);
- `@RemoteHome` — удаленный домашний интерфейс (home);
- `@Remove` — заменяет `remove-method`;
- `@Stateful` и `@Stateless` — `Session-type`;
- `@Timeout` — `timeout-method`;
- `@MessageDriven` указывает Message-Driven-компонент (message-driven);
- `@Interceptors` заменяет `interceptors` и `interceptor-binding`;
- `@AroundInvoke` — `around-invoke`;
- `@ExcludeDefaultInterceptors` — `exclude-default-interceptors`;
- `@ExcludeClassInterceptors` — `exclude-class-interceptors`;
- `@Entity` — определяет класс как Entity-компонент (`Entity`);
- `@WebServiceRef` заменяет элемент `service-ref`.

Компоненты EJB 3.0 Entity логически группируются в сохраняемые единицы (persistence unit). При этом компоненты сохраняемой единицы управляются объектами `EntityManager` фабрики `EntityManagerFactory`.

Сохраняемая единица определяется в файле persistence.xml папки META-INF каталога JAR-файла, содержащего классы компонентов сохраняемой единицы.

Элементы файла persistence.xml (XML Schema for the persistence configuration file):

- `persistence` — корневой элемент;
- `persistence-unit` — описывает сохраняемую единицу. Атрибуты элемента: `name` — имя сохраняемой единицы; `transaction-type` — тип управления

транзакциями — по умолчанию значение атрибута JTA при выполнении в среде Java EE (используется сервис Java Transaction API сервера Java EE) и `RESOURCE_LOCAL` при выполнении в среде Java SE (в коде используется интерфейс `javax.persistence.EntityTransaction` для управления транзакциями). Элемент имеет следующие вложенные элементы:

- `description` — текстовое описание;
- `provider` указывает класс, реализующий интерфейс `javax.persistence.spi.PersistenceProvider`, если этот класс не является реализацией по умолчанию. Контейнер вызывает интерфейс `PersistenceProvider` для создания фабрики `EntityManagerFactory`. Каждый контейнер EJB 3.0 имеет реализацию интерфейса, используемую по умолчанию;
- `jta-data-source` — в случае `transaction-type="JTA"` указывает специфическое JNDI-имя JTA-источника данных (JNDI-имя объекта `DataSource`, настроенного для подключения к базе данных);
- `non-jta-data-source` — в случае `transaction-type="RESOURCE_LOCAL"` указывает специфическое JNDI-имя не JTA-источника данных (JNDI-имя объекта `DataSource`, настроенного для подключения к базе данных);
- `mapping-file` указывает дополнительный XML-файл, расположенный в папке META-INF и определяющий соответствие между классами сохраняемой единицы и данными базы данных. Содержимое mapping file описывается схемой orm_1_0.xsd 1.0. По умолчанию имя файла — orm.xml;
- `jar-file` указывает JAR-файл, содержащий EJB 3.0 Entity-компоненты для включения их в сохраняемую единицу. JAR-файлы могут содержать дополнительные метаданные, которые будут учтены при развертывании;
- `class` — элемент используется для перечисления классов компонентов, включаемых в сохраняемую единицу;
- `exclude-unlisted-classes` — если данный элемент не указан, то в сохраняемую единицу будут включены все классы, помеченные аннотацией `@Entity` и расположенные в том же каталоге, что и папка META-INF с файлом persistence.xml;
- `properties` — включает в себя вложенные элементы `property` (атрибуты `name` и `value`), которые могут описывать такие свойства, как специфическое JDBC-соединение с базой данных, конфигурационные свойства, необходимые для выполнения в среде Java SE.

Таким образом, в EJB-модуле роли безопасности и связанные с ними права доступа к защищенным методам описываются элементами файла дескрипто-

ра развертывания — `security-role-ref`, `security-idEntity`, `security-role`, `method-permission`, а также аннотациями — `@DeclareRoles`, `@RunAs`, `@RolesAllowed`, `@PermitAll` и `@DenyAll`.

Система безопасности EJB-модуля может включать использование в Java-коде методов `isCallerInRole()` и `getCallerPrincipal()` интерфейса `javax.ejb.EJBContext`.

Клиент приложения Java EE

В качестве J2EE-клиента может выступать Web-клиент или клиентское приложение.

Web-клиент, или "тонкий" клиент, представляет собой Web-браузер, визуализирующий получаемые от J2EE-сервера Web-странички, которые могут содержать встроенные апплеты. Web-клиент называется "тонким", потому что его задача состоит только в установлении соединения с Web-уровнем J2EE-приложения, вся остальная функциональность сосредоточена в компонентах, выполняемых на J2EE-сервере.

Клиентское приложение ("толстый" клиент) представляет собой Java-приложение, имеющее, как правило, графический пользовательский интерфейс и обеспечивающее возможность доступа как к компонентам Web-уровня, так и бизнес-уровня и к уровню данных (EIS) J2EE-приложения. Клиентское приложение может быть создано как для развертывания в контейнере клиентских приложений, так и может быть автономным и выполняться в среде Java SE.

Контейнер клиентских приложений (Application Client Container, ACC) обеспечивает для клиентского приложения стандартные сервисы Java EE, такие как Java Transaction API (JTA), Java Naming and Directory Interface API (JNDI), Java Message Service (JMS), JDBC API и др., и включает в себя весь необходимый для поддержки сервисов набор Java-классов и библиотек. Контейнер клиентских приложений взаимодействует с сервером Java EE с помощью протокола RMI-IIOP, обеспечивая доступ для "толстого" клиента к компонентам Web-уровня и бизнес-уровня. Например, при создании клиентского приложения для работы в контейнере клиентских приложений сервера Java EE GlassFish можно воспользоваться утилитой `package-appclient`, которая упакует JAR-файл клиентского приложения и библиотеки контейнера клиентских приложений (appserv-rt.jar и javaee.jar) в архив appclient.jar, запустить который можно утилитой `appclient`. Утилита `appclient` запускает клиентское приложение в контейнере клиентских приложений.

При создании автономного клиентского приложения, не использующего контейнер клиентских приложений, необходимо программным образом обеспе-

чить все транспортные и коммуникационные протоколы для доступа к компонентам Web-уровня, бизнес-уровня и к уровню данных.

Клиентский модуль EAR-архива приложения Java EE представляет "толстого" клиента, предназначенного для работы в контейнере клиентских приложений. Клиентский JAR-файл содержит Java-классы клиентского приложения, включая главный класс со статическим методом `main()` — точкой входа в приложение, и дополнительный файл дескриптора развертывания application-client.xml, содержимое которого определяется схемой application-client_5.xsds.

Файл дескриптора развертывания application-client.xml содержит следующие элементы:

- `application-client` — корневой элемент, имеет атрибуты `version`, `id` и `metadata-complete` со значениями `true`/`false`. Если атрибут `metadata-complete` имеет значение `false` (по умолчанию), то при развертывании учитываются аннотации кода классов приложения;
- `description`, `display-name` и `icon`;
- `env-entry` описывает переменные окружения клиентского приложения:
  - `description` — описание;
  - `env-entry-name` — JNDI-имя переменной;
  - `env-entry-type` — Java-тип переменной;
  - `env-entry-value` — значение переменной;
  - `mapped-name` связывает ресурс со специфическим именем;
  - `injection-target` — класс, использующий данный ресурс с помощью элементов `injection-target-class` (полное имя класса), `injection-target-name` (экземпляр класса);
- `ejb-ref` — ссылка на EJB-компонент:
  - `description`;
  - `ejb-ref-name` — JNDI-имя EJB-компонента;
  - `ejb-ref-type` — тип EJB-компонента (`Session` или `Entity`);
  - `home` — полное имя домашнего интерфейса;
  - `remote` — полное имя удаленного интерфейса;
  - `ejb-link` — имя `ejb-name` компонента;
  - `mapped-name` связывает ресурс со специфическим именем;

- injection-target — класс, использующий данный ресурс с помощью элементов injection-target-class (полное имя класса), injection-target-name (экземпляр класса);

- resource-ref — ссылка на внешний ресурс:
  - description;
  - res-ref-name — JNDI-имя ресурса;
  - res-type — Java-тип ресурса;
  - res-auth — способ авторизации ресурса (Application или Container);
  - res-sharing-scope определяет, является ресурс совместно используемым (Shareable) или нет (Unshareable);
  - mapped-name связывает ресурс со специфическим именем;
  - injection-target — класс, использующий данный ресурс с помощью элементов injection-target-class (полное имя класса), injection-target-name (экземпляр класса);
- resource-env-ref — JNDI-ссылка на системные (администрируемые) объекты системы JMS (Java Messaging System):
  - description;
  - resource-env-ref-name — JNDI-имя ссылки;
  - resource-env-ref-type — тип объекта (например, javax.jms.ConnectionFactory);
  - mapped-name связывает ресурс со специфическим именем;
  - injection-target — класс, использующий данный ресурс с помощью элементов injection-target-class (полное имя класса), injection-target-name (экземпляр класса);
- message-destination-ref описывает JNDI-ссылку системы JMS на адресата сообщений:
  - description — описание ссылки;
  - message-destination-ref-name — JNDI-имя ссылки;
  - message-destination-type — тип адресата (например, javax.jms.Topic);
  - message-destination-usage указывает, является адресат потребителем сообщений (Consumes) или/и генератором сообщений (Produces);
  - message-destination-link связывает ссылку с адресатом, который описывается элементом message-destination;
  - mapped-name связывает ресурс со специфическим именем;

- `injection-target` — класс, использующий данный ресурс с помощью элементов `injection-target-class` (полное имя класса), `injection-target-name` (экземпляр класса);

- `message-destination` описывает адресата сообщений:
  - `description` — описание адресата;
  - `display-name` — отображаемое имя;
  - `icon` — значок;
  - `message-destination-name` — имя адресата;
- `persistence-unit-ref` — ссылка на сохраняемую единицу:
  - `description`;
  - `persistence-unit-ref-name` — JNDI-имя ссылки;
  - `persistence-unit-name` — имя persistence.xml сохраняемой единицы;
  - `mapped-name` связывает ресурс со специфическим именем;
  - `injection-target` — класс, использующий данный ресурс с помощью элементов `injection-target-class` (полное имя класса), `injection-target-name` (экземпляр класса);
- `post-construct` и `pre-destroy` описывают методы-перехватчики:
  - `lifecycle-callback-class` — полное имя класса-перехватчика;
  - `lifecycle-callback-method` — имя метода-перехватчика;
- `callback-handler` указывает класс приложения, реализующий интерфейс `javax.security.auth.callback.CallbackHandler` сервиса Java Authentication and Authorization Services (JAAS). Экземпляр данного класса будет создаваться контейнером клиентских приложений для получения от пользователя логина и пароля.

Дескрипторы развертывания сервера приложений Java EE

Как в EAR-файл приложения Java EE, так и в его модули могут включаться дополнительные файлы дескрипторов развертывания, специфические для конкретного сервера Java EE. Например, для сервера Java EE GlassFish это дескрипторы развертывания sun-application.xml (EAR-файл), sun-web.xml (WAR-файл), sun-ejb-jar.xml (EJB JAR-файл), sun-application-client.xml (клиентский JAR-файл), sun-cmp-mappings.xml (для EJB 2.1 Entity-компонентов

с сохранением, управляемым контейнером, устанавливает соответствие между компонентами и данными базы данных).

Файл sun-application.xml, структура которого определяется файлом sun-application_5_0-0.dtd, имеет следующие элементы:

- `sun-application` — корневой элемент;
- `web` — Web-модуль;
  - `web-uri` — URI-адрес файла Web-модуля;
  - `context-root` — уникальный идентификатор для данного Web-приложения в сервере Java EE, являющийся частью URL-адреса клиентского запроса к данному Web-приложению;
- `pass-by-reference` — способ, которым сервлет или EJB-компонент вызывает метод удаленного интерфейса другого EJB-компонента, находящегося в том же процессе. Если значение элемента `false` (по умолчанию), то сервер создает копии параметров вызова, что увеличивает издержки; если значение `true`, то вызов происходит не по значению, а по ссылке, без создания копий параметров вызова;
- `unique-id` — уникальный идентификатор приложения Java EE;
- `security-role-mapping` связывает роли безопасности с пользователями или группами пользователей сервера Java EE при развертывании приложения:
  - `role-name` — имя роли приложения Java EE;
  - `principal-name` — имя пользователя сервера Java EE;
  - `group-name` — имя группы сервера Java EE;
- `realm` — имя набора пользователей и групп пользователей домена сервера, контролируемых одной политикой идентификации, используется для обработки всех запросов идентификации, связанных с данным приложением Java EE.

Так как элемент `security-role` дескриптора развертывания Java EE-приложения application.xml сам по себе не определяет никакие права доступа, то он используется для того, чтобы связать определенные в Web- и EJB-модулях роли безопасности с пользователями или группами пользователей сервера Java EE, используя элемент `security-role-mapping` дескриптора сервера sun-application.xml. Такая архитектура позволяет адаптировать приложение Java EE к конкретному серверу, не изменяя составляющие приложение модули.

Файл sun-web.xml, структура которого определяется файлом sun-web-app_2_5-0.dtd, имеет следующие элементы:

- `sun-web-app` — корневой элемент, имеет дополнительный атрибут `httpservlet-security-provider`, указывающий идентификатор серверного

модуля идентификации server authentication module (SAM), который обеспечивает безопасность клиент-серверной коммуникации;

- `context-root` — уникальный идентификатор для данного Web-приложения в сервере Java EE;
- `security-role-mapping` связывает роли безопасности Web-приложения с пользователями или группами пользователей сервера Java EE:
  - `role-name` — имя роли Web-приложения;
  - `principal-name` — имя пользователя сервера Java EE;
  - `group-name` — имя группы сервера Java EE;
- `servlet` — сервлет в Web-модуле:
  - `servlet-name` — имя сервлета в web.xml;
  - `principal-name` — имя пользователя сервера, связывается с ролью безопасности элемента `run-as`;
  - `webservice-endpoint` описывает Web-сервис с помощью следующих элементов:
    - `port-component-name` — имя класса реализации Web-сервиса;
    - `endpoint-address-uri` — относительный путь класса Web-сервиса;
    - `message-security-binding` — вложенный элемент `message-security`, описывает защиту SOAP-сообщений Web-сервиса, используя элементы:
      - `message` указывает метод или операцию, к которым применяются требования безопасности SOAP-сообщений, вложенные элементы — `java-method` (метод описывается элементами `method-name` и `method-params` с `method-param`) и `operation-name` (WDSL-операция Web-сервиса);
      - `request-protection` — требования идентификации для обработки запросов, имеет атрибуты `auth-source` (тип идентификации `sender` — имя и пароль или `content` — цифровая подпись) и `auth-recipient` (идентификация адресата производится до или после идентификации содержимого — `before-content` или `after-content`);
      - `response-protection` определяет требования идентификации для обработки ответов, имеет атрибуты `auth-source` (тип идентификации `sender` — имя и пароль или `content` — цифровая подпись) и `auth-recipient` (идентификация адресата производится до или после идентификации содержимого — `before-content` или `after-content`);

- transport-guarantee — коммуникационные требования между клиентом и сервером, имеет значения NONE, INTEGRAL (требование неизменяемости данных в процессе передачи) и CONFIDENTIAL (передача данных осуществляется по защищенному протоколу);
- service-qname — расширенное имя QName XML-элемента WDSL-файла, описывающего Web-сервис с помощью элементов namespaceURI (пространство имен) и localpart (локальная часть имени);
- tie-class — автоматически генерируемый серверный класс заглушки, обрабатывающий SOAP-сообщения для Web-сервиса;
- servlet-impl-class — имя автоматически генерируемого серверного класса сервлета, который обеспечивает обращение к методам класса реализации Web-сервиса;
- debugging-enabled определяет генерацию тестовой странички для Web-сервиса, имеет значения true (по умолчанию) и false;
- property — дополнительные конфигурационные свойства с помощью атрибутов name и value;

- idempotent-url-pattern — запрос, который повторно может быть адресован другому экземпляру сервера, если запрошенный экземпляр сервера окажется недоступным, при этом повторный запрос не должен создать ошибочную ситуацию приложения. Это может быть поисковый запрос, но не банковская транзакция. Элемент имеет атрибуты url-pattern (URL-шаблон запроса) и no-of-retries (количество повторных запросов);
- session-config содержит конфигурационные параметры клиент-серверной сессии:
  - session-manager имеет атрибут persistence-type (устанавливает тип сохранения состояния сессии memory, file, replicated и ha — High Availability Database (HADB), серверная база данных), содержит элементы:
    - manager-properties — вложенный элемент property с атрибутами name и value, устанавливает такие свойства, как reapIntervalSeconds — интервал между проверками истекших сессий (persistence-type — file, ha или replicated);
    - maxSessions — количество разрешенных сессий;
    - sessionFilename — путь к каталогу, в котором состояние сессии сохраняется между запусками приложения (persistence-type — memory);
    - persistenceFrequency определяет сохранение состояния сессии (web-method — состояние сохраняется после Web-запроса перед отсылкой

ответа клиенту, `time-based` — состояние сохраняется с частотой `reapIntervalSeconds`), применимо при `persistence-type`, равном `ha` или `replicated`;

- `store-properties` — вложенный элемент `property` с атрибутами `name` и `value`, устанавливает такие свойства, как `directory` — путь к каталогу с файлами состояний индивидуальных сессий (`persistence-type` — `file`);
- `persistenceScope` — возможные значения: `session` (каждый раз сохраняется состояние сессии полностью), `modified-session` (состояние сохраняется только при его модификации), `modified-attribute` (сохраняются только модифицированные атрибуты сессии), применимо при `persistence-type` — `ha` или `replicated`;

- `session-properties` — вложенный элемент `property` с атрибутами `name` и `value`, устанавливает такие свойства, как `timeoutSeconds` (максимальный интервал ожидания для всех сессий), `enableCookies` (разрешение для cookies `true`/`false`), `enableURLRewriting` (если браузер не поддерживает cookies, разрешение для сохранения информации внутри URL-адреса `true`/`false`);
- `cookie-properties` описывает свойства cookies (cookies используются для хранения данных и представляют собой строку текста, которая сохраняется на стороне клиента и отправляется на сервер, установивший данный cookie, при каждом запросе), содержит вложенный элемент `property` с атрибутами `name` и `value`, устанавливает такие свойства, как `cookiePath` (путь, по которому браузер посылает cookies, по умолчанию путь инсталляции Web-модуля), `cookieMaxAgeSeconds` (время хранения браузером cookie), `cookieDomain` (указывает домен, для которого действительны cookie), `cookieComment` (комментарии для идентификации сессионных cookie в файле cookie);

- `ejb-ref` связывает (переопределяет) имя ссылки на ресурс файла web.xml с абсолютным JNDI-именем ресурса с помощью элементов `ejb-ref-name` (имя ссылки) и `jndi-name` (JNDI-имя ресурса);
- `resource-ref` связывает (переопределяет) имя ссылки на ресурс файла web.xml с абсолютным JNDI-именем ресурса с помощью элементов `res-ref-name` (имя ссылки), `jndi-name` (JNDI-имя ресурса) и `default-resource-principal` (пользователь по умолчанию для ресурса, вложенные элементы `name` и `password`);
- `resource-env-ref` связывает (переопределяет) имя ссылки на администрируемый объект файла web.xml с абсолютным JNDI-именем ресурса с по-

мощью элементов `resource-env-ref-name` (имя ссылки) и `jndi-name` (JNDI-имя ресурса);

- `service-ref` указывает параметры выполнения для используемого Web-сервиса;
  - `service-ref-name` — JNDI-имя ссылки;
  - `port-info` описывает используемый Web-сервис с помощью элементов:
    - `service-endpoint-interface` — JNDI-имя Web-сервиса;
    - `wsdl-port` — WSDL-имя Web-сервиса, имеет вложенные элементы `namespaceURI` (пространство имен) и `localpart` (локальная часть QName);
    - `stub-property` — вложенные элементы `name` и `value`, определяют свойства объекта `javax.xml.rpc.Stub`, такие как адрес класса реализации Web-сервиса, перед тем, как заглушка будет возвращена клиенту. Программный код — заглушка создается сервером в ответ на первый запрос клиента к Web-сервису и пересылается клиенту для обеспечения SOAP-протокола;
    - `call-property` — вложенные элементы `name` и `value`, определяют, в случае динамического вызова клиентом методов класса реализации Web-сервиса, свойства объекта `javax.xml.rpc.Call` перед тем, как объект будет возвращен сервером клиенту;
    - `message-security-binding` — вложенный элемент `message-security`, который описывает защиту SOAP-сообщений Web-сервиса, используя элементы:
      - `message` — метод или операция, к которым применяются требования безопасности SOAP-сообщений, вложенные элементы — `java-method` (метод описывается элементами `method-name` и `method-params` с `method-param`) и `operation-name` (WDSL-операция Web-сервиса);
      - `request-protection` — требования идентификации для обработки запросов, имеет атрибуты `auth-source` (тип идентификации `sender` — имя и пароль или `content` — цифровая подпись) и `auth-recipient` (идентификация адресата производится до или после идентификации содержимого — `before-content` или `after-content`);
      - `response-protection` — требования идентификации для обработки ответов, имеет атрибуты `auth-source` (тип идентификации `sender` — имя и пароль или `content` — цифровая подпись) и `auth-`

`recipient` (идентификация адресата производится до или после идентификации содержимого — `before-content` или `after-content`);

- `call-property` — вложенные элементы `name` и `value`, определяют, в случае динамического вызова клиентом методов класса реализации Web-сервиса, свойства объекта `javax.xml.rpc.Call` перед тем, как объект будет возвращен сервером клиенту;
- `wsdl-override` переопределяет URL-адрес WSDL-документа Web-сервиса;
- `service-impl-class` — имя сгенерированного класса реализации интерфейса `javax.xml.rpc.Service`, обеспечивающего клиентский доступ к функциональности Web-сервиса;
- `service-qname` — расширенное имя QName XML-элемента WDSL-файла, описывающего Web-сервис с помощью элементов `namespaceURI` (пространство имен) и `localpart` (локальная часть имени);

- `message-destination-ref` выполняет роль `message-destination-link` и связывает имя JNDI-ссылки системы JMS на адресата сообщений файла web.xml с физическим JNDI-именем адресата с помощью элементов `message-destination-ref-name` и `jndi-name`;
- `cache` — конфигурация кэширования для Web-компонентов. Сервер может кэшировать результаты вызовов Web-компонентов для того, чтобы обслужить последующие вызовы быстрее. Элемент имеет атрибуты `max-entries` (максимальное количество записей в кэше, по умолчанию 4096), `timeout-in-seconds` (максимальное время хранения записи в кэше, по умолчанию 30 секунд), `enabled` (устанавливает, возможно ли кэширование для сервлетов или JSP-страниц, по умолчанию `true`).
  - `cache-helper` — класс Web-модуля, реализующий интерфейс `com.sun.appserv.web.cache.CacheHelper` библиотеки Java EE Sun Java System Application Server API. Данный класс создается для определения параметров кэширования программным способом. Элемент имеет атрибуты `name` (строковый идентификатор класса) и `class-name` (полное имя класса), а также вложенный элемент `property` с атрибутами `name` и `value`;
  - `default-helper` — свойства для серверного класса, реализующего интерфейс `CacheHelper` и используемого по умолчанию. Вложенный элемент `property` с атрибутами `name` и `value` устанавливает свойство `cacheKeyGeneratorAttrName` (значение свойства указывает реализацию интерфейса `com.sun.appserv.web.cache.CacheKeyGenerator`, отвечающую за генерацию ключа, который используется для кэширования запроса);

- property имеет атрибуты name и value, устанавливает свойства кэширования, такие как cacheClassName (полное имя класса, реализующего функциональность кэширования), возможные значения:
  - com.sun.appserv.web.cache.LruCache (по умолчанию) — реализован алгоритм удаления записей LRU (Least Recently Used), когда удаляется запись, к которой дольше всего не было обращений;
  - com.sun.appserv.web.cache.BaseCache — кэширование при известном максимальном количестве записей;
  - com.sun.appserv.web.cache.MultiLruCache — кэширование для большого числа записей (более 4096), используется свойство MultiLRUSegmentSize;
  - com.sun.appserv.web.cache.BoundedMultiLruCache — кэширование при ограниченном размере памяти; свойство MultiLRUSegmentSize определяет максимальное количество записей в сегменте кэширования, который использует собственный алгоритм LRU, по умолчанию 4096, свойство MaxSize определяет размер памяти для кэширования;
- cache-mapping связывает имя сервлета или URL-шаблон запроса с условиями кэширования, имеет вложенные элементы:
  - servlet-name — имя сервлета, содержится также в элементе servlet-name файла web.xml; url-pattern — URL-шаблон сервлета;
  - cache-helper-ref — класс, реализующий интерфейс com.sun.appserv.web.cache.CacheHelper;
  - dispatcher (применяется при отсутствии cache-helper-ref) — список методов javax.servlet.RequestDispatcher, для которых производится кэширование;
  - timeout (применяется при отсутствии cache-helper-ref) — максимальное количество времени сохранения записи, атрибуты name (java.lang.Long или java.lang.Integer) и scope (context.attribute, request.header, request.parameter, request.cookie, request.attribute, Session.attribute);
  - refresh-field (применяется при отсутствии cache-helper-ref) определяет поле, которое требует обновления и описывается атрибутами name и scope (область, из которой берется параметр — context.attribute, request.header, request.parameter, request.cookie, request.attribute, Session.attribute);
  - http-method (применяется при отсутствии cache-helper-ref) — метод, для которого производится кэширование, по умолчанию GET;

- `key-field` (применяется при отсутствии `cache-helper-ref`) — компонент ключа, используемый для извлечения записей из кэша, по умолчанию контейнер использует сегмент пути, активирующий сервлет, имеет атрибуты `name` и `scope`;
- `constraint-field` (применяется при отсутствии `cache-helper-ref`) — ограничения кэширования, атрибуты `name`, `scope`, `cache-on-match`, `cache-on-match-failure`, вложенный элемент `constraint-field-value` указывает соответствие значению, которое должно быть у параметра, определяемого атрибутом `name`, имеет атрибуты `match-expr` (тип соответствия — `equals`, `not-equals`, `greater`, `lesser`, `in-range`), `cache-on-match` (если `true`, то кэширование производится при успешном соответствии), `cache-on-match-failure` (если `true`, то кэширование производится в случае несоответствия);

- `class-loader` — конфигурация загрузчика, имеет атрибуты `extra-class-path` (список дополнительных путей `CLASSPATH` для Web-модуля), `delegate` (загрузчики в среде выполнения сервера образуют иерархию от Bootstrap class loader до Web и JSP Engine class loader, если значение атрибута `true` (по умолчанию), то загрузка начинается с родительского для загрузчика Web-модуля class loader, если `false`, то загрузка начинается с class loader Web-модуля). Вложенный элемент `property` с атрибутами `name` и `value` указывает значение свойства `ignoreHiddenJarFiles` (если `true`, то все JAR- и ZIP-файлы в каталоге WEB-INF\lib, начинающиеся с точки (`.`), игнорируются class loader, по умолчанию `false`).
- `jsp-config` — конфигурационная информация для JSP-страниц. Вложенный элемент `property` с атрибутами `name` и `value` указывает значение свойств:
  - `checkInterval` — временной интервал между проверками необходимости перекомпилировать JSP-файл, по умолчанию 0;
  - `classdebuginfo` — компиляция с отладкой, по умолчанию `true`;
  - `classpath` — путь, используемый при компиляции сгенерированных контейнером сервлетов;
  - `compiler` — компилятор, по умолчанию `javac`;
  - `compilerSourceVM` — версия JDK, с которой обеспечивается совместимость сгенерированных сервлетов;
  - `compilerTargetVM` — версия виртуальной машины JVM, для которой генерируются сервлеты;
  - `defaultBufferNone` — `false` по умолчанию; если `true`, то отменяет буферизацию выходных данных JSP-страницы;

- development — true по умолчанию, устанавливает проверку модификации JSP-файла для перекомпиляции;
- dumpSmap — если true, выводится SMAP-информация при отладке, Source Map (SMAP) описывает соответствие между исходным документом (JSP-страничкой) и генерируемым документом (сервлетом), по умолчанию false;
- enablePooling — по умолчанию true, Java-объекты, создаваемые контейнером для пользовательских тегов JSP, объединяются в пул и повторно используются;
- enableTldValidation — если true, то все файлы Tag Library Descriptor (TLD) Web-модуля, объявляющие пользовательские теги, проверяются относительно XML-схем или DTD-файлов, определяющих структуру TLD-файлов, по умолчанию false;
- errorOnUseBeanInvalidClassAttribute — если true, то выдается ошибка при несоответствии значения атрибута class элемента jsp:useBean JSP-странички, по умолчанию false;
- fork — по умолчанию true — компиляция JSP-файлов производится параллельно с работой Web-контейнера, в ресурсно-ограниченных серверах при этом может возникнуть ошибка компиляции;
- genStrAsByteArray — по умолчанию true — если JSP-страничка при кодировании не буферизуется, текстовые строки генерируются в виде байтов;
- genStrAsCharArray — если true, то текстовые строки генерируются в виде символьных массивов, по умолчанию false;
- httpMethods — список поддерживаемых HTTP-методов при запросе к JSP-странички, по умолчанию все методы;
- ieClassId — идентификационный номер плагина для браузера IE, используется тегом jsp:plugin, который запускает выполнение апплета или Bean-компонента, по умолчанию clsid:8AD9C840-044E-11D1-B3E9-00805F499D93;
- ignoreJspFragmentErrors — по умолчанию false, если true, то компилятор игнорирует ошибки предкомпиляции JSP-фрагментов, другой способ избежать предкомпиляции JSP-фрагментов — имена файлов должны иметь расширения jspf;
- initialCapacity — первоначальный объем хэш-таблицы, связывающей JSP-файлы со сгенерированными сервлетами;

- `javaEncoding` — тип кодировки для генерируемых сервлетов, по умолчанию UTF8;
- `keepgenerated` — если `true`, то сохранять сгенерированные Java-файлы сервлетов;
- `mappedfile` — по умолчанию `true` — генерация статического содержимого (HTML-страниц) производится построчно;
- `modificationTestInterval` — частота, с которой проверяется модификация JSP-страниц для перекомпиляции, по умолчанию 0 — проверка происходит при каждом обращении клиента к JSP-странице, используется, если свойство `development` — `true`;
- `reload-interval` — частота, с которой проверяется модификация JSP-страниц для перекомпиляции, по умолчанию 0 — проверка происходит при каждом обращении клиента к JSP-странице;
- `saveBytecode` — если `true`, то генерируемый байт-код сохраняется в файлах с расширением class;
- `scratchdir` — рабочий каталог для хранения генерируемого кода;
- `suppressSmap` — если `true`, то не производится генерация SMAP-информации, по умолчанию `false`;
- `trimSpaces` — если `true`, то удаляются лишние пробелы в генерируемом статическом содержимом (результирующих для JSP HTML-страницах), по умолчанию `false`;
- `usePrecompiled` — если `true`, то при клиентском запросе JSP-файлы не компилируются, а используются сгенерированные при предкомпиляции сервлеты, которые расположены в папках WEB-INF\lib или WEB-INF\classes Web-модуля, по умолчанию `false`;
- `xpoweredBy` — по умолчанию `true`, заголовок `X-Powered-By`, содержащий информацию о версиях сервлета и сервера и др., добавляется в ответ клиенту;

- `parameter-encoding` — кодировка запросов по умолчанию, имеет атрибуты `form-hint-field` (имя скрытого поля в клиентской форме, значения которого декодируются контейнером) и `default-charset` (по умолчанию кодировка ISO-8859-1);
- `property` — атрибуты `name` и `value`, описывают такие свойства, как:
  - `allowLinking` — если `true`, то разрешаются ссылки на ресурсы, которые расположены вне каталога Web-модуля, по умолчанию `false`;
  - `alternatedocroot_n` позволяет Web-приложению обслужить запросы для определенных ресурсов, расположенных в другом корневом

каталоге, <property name="alternatedocroot_1" value="from=[URL-шаблон запроса] dir= [каталог]"/>;

- crossContextAllowed — по умолчанию true, т. е. разрешается доступ к контексту других Web-приложений с помощью метода getContext() интерфейса javax.servlet.ServletContext;
- relativeRedirectAllowed — если true, то разрешается использовать в качестве аргумента метода sendRedirect() интерфейса javax.servlet.http.HttpServletResponse относительный URL-адрес, метод sendRedirect() посылает клиентскому браузеру URL-адрес для перенаправления клиентского запроса другому ресурсу, и браузер создает новый запрос, согласно полученному URL-адресу, к другому ресурсу, по умолчанию false;
- reuseSessionID — если true, то сессия, создаваемая контейнером для Web-приложения, имеет идентификатор ID, определенный в клиентском запросе, по умолчанию false;
- securePagesWithPragma — если false, то при использовании защищенного протокола SSL для браузера Internet Explorer обеспечиваются загрузки, по умолчанию — true и при SSL-загрузках браузер IE будет выдавать ошибку;
- tempdir — временный каталог для Web-модуля, куда контейнер будет размещать рабочие файлы — откомпилированные JSP-файлы и др.;
- useResponseCTForHeaders — если true, то заголовок ответа клиенту будет иметь ту же кодировку, что и ответ, а не по умолчанию UTF-8, по умолчанию false;

- message-destination связывает (переопределяет) логическое имя адресата файла web.xml с абсолютным JNDI-именем адресата с помощью элементов message-destination-name (логическое имя адресата), jndi-name (JNDI-имя адресата);
- webservice-description содержит элементы webservice-description-name (уникальное имя Web-сервиса в Web-модуле) и wsdl-publish-location (URL-адрес каталога, в котором будет находиться WSDL-файл Web-сервиса после развертывания);
- login-config определяет способ идентификации клиента и состоит из элементов:
  - auth-method — механизм идентификации (BASIC — идентификация основана на логине и пароле и использует HTTP-протокол, DIGEST — идентификация основана на логине и пароле и использует HTTPS-протокол, FORM — используется специально созданная страничка с фор-

мой для ввода логина и пароля, а также страничка ошибки, `CLIENT-CERT` — идентификация основана на сертификате клиента);

- `realm-name` — имя realm для идентификации `BASIC`;
- `form-login-config` указывает странички формы и ошибки для идентификации `FORM`, используя элементы `form-login-page` и `form-error-page`.

Файл sun-ejb-jar.xml, структура которого определяется файлом sun-ejb-jar_3_0-0.dtd, имеет следующие элементы:

- `sun-ejb-jar` — корневой элемент;
- `security-role-mapping` связывает роли, определенные элементом `role-name` файла ejb-jar.xml, с пользователями и группами пользователей сервера с помощью элементов `role-name` (имя роли, определенное в файле ejb-jar.xml), `principal-name` (имя пользователя сервера), `group-name` (имя группы пользователей сервера);
- `enterprise-beans`. Вложенные элементы:
  - `name` — имя модуля;
  - `unique-id` — автоматически генерируемый системный идентификатор модуля;
  - `ejb` — свойства для одного EJB-компонента, имеет атрибут `availability-enabled`, при равенстве `true` которого и когда EJB-компонент является Stateful, сервер обеспечивает для него свойства высокой надежности и доступности применительно к сохранению состояния сессии (High Availability Session Persistence), такие как сохранение состояния путем репликации на другие серверы кластера и сохранение в базе данных high-availability database (HADB). В общем сервер Java EE способен обеспечивать свойства высокой доступности и надежности применительно к сохранению сессии (High Availability Session Persistence), сервису сообщений (High Availability Java Message Service) и протоколу коммуникации (RMI-IIOP Load Balancing and Failover). Вложенные элементы:
    - `ejb-name` соответствует имени, определенному в элементе `ejb-name` файла ejb-jar.xml;
    - `jndi-name` — для Entity- и Session-компонентов это JNDI-имя объекта EJBHome, для Message-Driven-компонента это JNDI-имя ресурса JMS, от которого компонент получает сообщения;
    - `ejb-ref` связывает имя, определенное в элементе `ejb-ref-name` файла ejb-jar.xml, с абсолютным JNDI-именем с помощью элементов `ejb-ref-name` и `jndi-name`;

- `resource-ref` связывает имя, определенное в элементе `resource-ref` файла ejb-jar.xml, с абсолютным JNDI-именем и именем пользователя сервера с помощью элементов `res-ref-name`, `jndi-name` и `default-resource-principal`;
- `resource-env-ref` связывает имя, определенное в элементе `resource-env-ref` файла ejb-jar.xml, с абсолютным JNDI-именем с помощью элементов `resource-env-ref-name` и `jndi-name`;
- `service-ref` — параметры выполнения для используемого Web-сервиса. Вложенные элементы:
  - `service-ref-name` — JNDI-имя ссылки;
  - `port-info` — используемый Web-сервис;
  - `service-endpoint-interface` — JNDI-имя Web-сервиса;
  - `wsdl-port` — WSDL-имя Web-сервиса, имеет вложенные элементы `namespaceURI` (пространство имен) и `localpart` (локальная часть QName);
  - `stub-property` — вложенные элементы `name` и `value` определяют свойства объекта `javax.xml.rpc.Stub`, такие как адрес класса реализации Web-сервиса, перед тем, как заглушка будет возвращена клиенту. Программный код — заглушка создается сервером в ответ на первый запрос клиента к Web-сервису и пересылается клиенту для обеспечения SOAP-протокола;
  - `call-property` — вложенные элементы `name` и `value` определяют, в случае динамического вызова клиентом методов класса реализации Web-сервиса, свойства объекта `javax.xml.rpc.Call` перед тем, как объект будет возвращен сервером клиенту;
  - `message-security-binding` — вложенный элемент `message-security`, который описывает защиту SOAP-сообщений Web-сервиса, используя элементы: `message` — метод или операция, к которым применяются требования безопасности SOAP-сообщений, вложенные элементы `java-method` (метод описывается элементами `method-name` и `method-params` с `method-param`) и `operation-name` (WDSL-операция Web-сервиса); `request-protection` — требования идентификации для обработки запросов, имеет атрибуты `auth-source` (тип идентификации `sender` — имя и пароль или `content` — цифровая подпись) и `auth-recipient` (идентификация адресата производится до или после идентификации содержимого — `before-content` или `after-content`); `response-protection` — требования идентификации для обработки ответов,

имеет атрибуты `auth-source` (тип идентификации `sender` — имя и пароль или `content` — цифровая подпись) и `auth-recipient` (идентификация адресата производится до или после идентификации содержимого — `before-content` или `after-content`);

- `call-property` — вложенные элементы `name` и `value` определяют, в случае динамического вызова клиентом методов класса реализации Web-сервиса, свойства объекта `javax.xml.rpc.Call` перед тем, как объект будет возвращен сервером клиенту;
- `wsdl-override` — URL-адрес WSDL-документа Web-сервиса;
- `service-impl-class` — имя сгенерированного класса реализации интерфейса `javax.xml.rpc.Service`, обеспечивающего клиентский доступ к функциональности Web-сервиса;
- `service-qname` — расширенное имя QName XML-элемента WDSL-файла, описывающего Web-сервис с помощью элементов `namespaceURI` (пространство имен) и `localpart` (локальная часть имени);
- `message-destination-ref` — выполняет роль `message-destination-link` и связывает имя JNDI-ссылки системы JMS на адресата сообщений файла ejb-jar.xml с физическим JNDI-именем адресата с помощью элементов `message-destination-ref-name` и `jndi-name`;
- `pass-by-reference` — способ, которым EJB-компонент вызывает метод удаленного интерфейса другого EJB-компонента, находящегося в том же процессе. Если значение элемента `false` (по умолчанию), то сервер создает копии параметров вызова, что увеличивает издержки; если значение `true`, то вызов происходит не по значению, а по ссылке, без создания копий параметров вызова;
- `principal` — имя пользователя для роли безопасности элемента `run-as` файла ejb-jar.xml;
- `mdb-connection-factory` — администрируемый JMS-объект `javax.jms.ConnectionFactory`, связанный с данным Message-Driven-компонентом. Объект `ConnectionFactory` создается администратором сервера и используется клиентом для установления соединения с поставщиком сервиса JMS. Элемент содержит вложенные элементы `jndi-name` (JNDI-имя объекта) и `default-resource-principal` (пользователь ресурса по умолчанию, который описывается элементами `name` и `password`);
- `jms-durable-subscription-name` — для Message-Driven-компонента указывает объект `javax.jms.TopicSubscriber` типа `Durable`, используемый клиентом для получения сообщений, которые публикуются JMS-объектом `Topic`;

- `jms-max-messages-load` — максимальное число сообщений, загружаемых единовременно, по умолчанию 1;
- `ior-security-config` — конфигурация безопасности Interoperable Object Reference (IOR) EJB-компонента, которая обеспечивает безопасность коммуникации между клиентским приложением и EJB-компонентом, развернутыми на разных серверах. EJB-протокол взаимодействия основан на протоколе Internet Inter-ORB Protocol (IIOP/GIOP 1.2) и спецификации Common Secure Interoperability version 2 (CSIv2) CORBA Secure Interoperability specification. Конфигурационные установки CSIv2 определены в IOR EJB-компонента. IOR является ссылкой на объект CORBA/IIOP и содержит информацию об адресе удаленного хоста, номере удаленного порта, классе удаленного объекта и др. CSIv2-спецификация определяет сервис Security Attribute Service (SAS), который обеспечивает идентификацию и авторизацию. Вложенные элементы:
  - `transport-config` — информация о безопасности передачи данных, вложенные элементы: `integrity` — тип обеспечения сохранности сообщений EJB-компонентом, возможные значения: `NONE` (SSL-протокол не используется), `SUPPORTED` (SSL-протокол может использоваться), `REQUIRED` (SSL-протокол должен использоваться при передачи сообщений); `confidentiality` — тип обеспечения конфиденциальности сообщений EJB-компонентом, возможные значения: `NONE` (сообщения не шифруются), `SUPPORTED` (сообщения могут шифроваться), `REQUIRED` (сообщения шифруются при передаче); `establish-trust-in-target` — тип обеспечения идентификации EJB-компонента для клиента (клиентом производится идентификация сервера EJB-компонента), возможные значения: `NONE`, `SUPPORTED`, `REQUIRED`; `establish-trust-in-client` — тип обеспечения идентификации клиента EJB-компонентом, возможные значения: `NONE`, `SUPPORTED`, `REQUIRED`;
  - `as-context` — механизм идентификации клиента, вложенные элементы: `auth-method` — метод идентификации, возможное значение — `USERNAME_PASSWORD`; `realm`; `required` определяет, должен ли использоваться метод, определенный в элементе `auth-method` для идентификации клиента, возможные значения — `true` или `false`;
  - `sas-context` — Security Attribute Service (SAS) состоит из двух уровней. Первый уровень — это идентификационный уровень, использующий имя и пароль клиента. Второй уровень — уровень атрибутов, может быть использован клиентом для того, чтобы послать атрибуты безопасности, содержащие информацию о кли-

енте и его привилегиях, EJB-компоненту. Элемент `sas-context` определяет поддержку первого или второго уровня с помощью элемента `caller-propagation`, возможные значения: `NONE` (идентификация клиента обеспечивается первым уровнем), `SUPPORTED` (второй уровень может использоваться), `REQUIRED` (требуется использование второго уровня);

- `is-read-only-bean` — если `true`, то Entity-компонент доступен только для чтения. По умолчанию `false`;
- `refresh-period-in-seconds` — если значение больше 0, то компонент read-only-bean обновляется через определенный интервал времени. По умолчанию 0 — обновления не происходит;
- `commit-option` определяет для Entity-компонента опцию завершения транзакции, возможные значения: `B` (по умолчанию) и `C`. Существуют три опции завершения транзакции: `A` — по завершению транзакции экземпляр Entity-компонента остается в состоянии готовности (в кэше), но метод `ejbLoad()` интерфейса `javax.ejb.EntityBean` (загружает состояние экземпляра из базы данных) не вызван; `B` — по завершению транзакции экземпляр в кэше, и метод `ejbLoad()` вызван; `C` — по завершению транзакции экземпляр переведен в пассивное состояние и возвращен в пул;
- `cmt-timeout-in-seconds` в случае управления транзакциями контейнером определяет время ожидания окончания транзакции; если транзакция не завершается в течение этого времени, контейнер ее откатывает. По умолчанию — 0. Это означает, что используется установка по умолчанию сервера;
- `use-thread-pool-id` — именованный пул потоков, используемый для удаленного вызова EJB-компонента. Именованный пул потоков может быть создан администратором сервера с помощью команды `create-threadpool`;
- `gen-classes` указывает все сгенерированные сервером классы для компонента с помощью элементов `remote-impl` (имя класса реализации для `EJBObject`), `local-impl` (имя класса реализации для `EJBLocalObject`), `remote-home-impl` (имя класса реализации для `EJBHome`), `local-home-impl` (имя класса реализации для `EJBLocalHome`);
- `bean-pool` определяет свойства пула для компонентов Stateless, Entity и Message-Driven с помощью элементов: `steady-pool-size` — первоначальное и минимальное количество экземпляров компонентов в пуле (по умолчанию 32); `resize-quantity` — количество создаваемых экземпляров компонентов в случае пустого пула; `max-pool-`

size — максимальное количество экземпляров компонентов в пуле (по умолчанию 60); pool-idle-timeout-in-seconds — максимальное количество времени, которое может находиться незадействованный экземпляр компонента в пуле (по умолчанию 10 минут);

- bean-cache определяет свойства кэша для компонентов Stateful и Entity с помощью элементов: max-cache-size — максимальное количество экземпляров компонентов в кэше; cache-idle-timeout-in-seconds — максимальное время, которое может находиться незадействованный экземпляр компонента в кэше в активном состоянии (по умолчанию 10 минут, после этого переводится в пассивное состояние); removal-timeout-in-seconds — время, после которого экземпляр компонента будет удален из кэша; resize-quantity — количество создаваемых экземпляров компонентов в случае пустого пула; victim-selection-policy — механизм перевода Stateful-компонента в пассивное состояние, возможные значения: First In, First Out (FIFO), Least Recently Used (LRU), Not Recently Used (NRU) (по умолчанию);
- mdb-resource-adapter описывает конфигурацию выполнения для Message-Driven-компонента с помощью элементов: resource-adapter-mid — ресурсный адаптер, который поставщик сервиса сообщений использует для доставки сообщений Message-Driven-компоненту (для JMS это jmsra); activation-config описывает конфигурационные свойства выполнения для Message-Driven-компонента, такие как destinationType, SubscriptionDurability и MessageSelector, с помощью элементов description и activation-config-property (вложенные элементы activation-config-property-name и activation-config-property-value);
- webservice-endpoint — EJB-компонент, выполняющий функции Web-сервиса. Вложенные элементы:
  - port-component-name — имя класса реализации Web-сервиса;
  - endpoint-address-uri — относительный путь класса Web-сервиса;
  - message-security-binding — вложенный элемент message-security, который описывает защиту SOAP-сообщений Web-сервиса, используя элементы: message — метод или операция, к которым применяются требования безопасности SOAP-сообщений, вложенные элементы — java-method (метод описывается элементами method-name и method-params с method-param) и operation-name (WDSL-операция Web-сервиса); request-protection — требования идентификации для обработки запросов, имеет атрибуты auth-source (тип идентификации sender —

имя и пароль или content — цифровая подпись) и auth-recipient (идентификация адресата производится до или после идентификации содержимого — before-content или after-content); response-protection — требования идентификации для обработки ответов, имеет атрибуты auth-source (тип идентификации sender — имя и пароль или content — цифровая подпись) и auth-recipient (идентификация адресата производится до или после идентификации содержимого — before-content или after-content);

- transport-guarantee — коммуникационные требования между клиентом и сервером, имеет значения NONE, INTEGRAL (требование неизменяемости данных в процессе передачи) и CONFIDENTIAL (передача данных осуществляется по защищенному протоколу);
- service-qname — расширенное имя QName XML-элемента WDSL-файла, описывающего Web-сервис с помощью элементов namespaceURI (пространство имен) и localpart (локальная часть имени);
- tie-class — автоматически генерируемый серверный класс заглушки, обрабатывающий SOAP-сообщения для Web-сервиса;
- servlet-impl-class — имя автоматически генерируемого серверного класса сервлета, который обеспечивает обращение к методам класса реализации Web-сервиса;
- debugging-enabled — генерация тестовой странички для Web-сервиса, имеет значения true (по умолчанию) и false;
- property — дополнительные конфигурационные свойства с помощью атрибутов name и value;

- flush-at-end-of-method для Entity-компонентов указывает метод, обеспечивающий синхронизацию набора экземпляров компонентов Entity с базой данных с помощью элемента method, который описывает метод, и содержит элементы description, ejb-name, method-intf (Home, Remote, LocalHome, Local), method-name, method-params (вложенные method-param);
- checkpoint-at-end-of-method — устанавливает сохранение состояния Stateful-компонента после выполнения метода, указанного с помощью вложенного элемента method, содержащего элементы description, ejb-name, method-intf (Home, Remote, LocalHome, Local), method-name, method-params (вложенные method-param);
- cmp — вложенный элемент prefetch-disabled запрещает предварительную загрузку состояния Entity-компонента для указанных мето-

дов запросов, содержит элемент `query-method` (указывает метод запроса с помощью элементов `method-name` и `method-params` с элементами `method-param`). Элемент используется для оптимизации использования памяти;

- `cmp-resource` определяет базу данных, используемую для сохранения CMP Entity-компонентов (компонентов, сохранением которых управляет контейнер), с помощью элементов:
  - `jndi-name` — JNDI-имя JDBC-ресурса (доступ приложения Java EE к базе данных обеспечивает JDBC API, JDBC-ресурс, создаваемый администратором сервера, обеспечивает соединение приложения с базой данных);
  - `default-resource-principal` — пользователь ресурса по умолчанию, описывается элементами `name` и `password`;
  - `create-tables-at-deploy` — если `true`, то при развертывании автоматически в базе данных будут созданы таблицы, соответствующие Entity-компонентам;
  - `drop-tables-at-undeploy` — если `true` таблицы, автоматически созданные при развертывании, будут автоматически удалены из базы данных при свертывании модуля;
  - `database-vendor-name` — имя производителя базы данных, возможные значения: `javadb`, `db2`, `mssql`, `oracle`, `postgresql`, `pointbase`, `derby`, `sybase`;
  - `schema-generator-properties` — вложенный элемент `property`, содержащий элементы `name` и `value`, описывает для генерируемых таблиц свойства, такие как `use-unique-table-names` (имена генерируемых таблиц уникальны в домене сервера) и `bean-name.field-name.attribute` (определяет атрибуты столбца таблицы). Атрибуты столбца генерируемой таблицы: `jdbc-type` — JDBC-тип столбца (например, char); `jdbc-maximumlength` — максимальное число символов в столбце; `jdbc-precision` — точность для чисел столбца; `jdbc-scale` — количество знаков после запятой для вещественных чисел столбца; `jdbc-nullable` — указывает, может ли содержать столбец нулевые значения;
- `message-destination` связывает (переопределяет) логическое имя адресата с абсолютным JNDI-именем адресата с помощью элементов `message-destination-name` (логическое имя адресата), `jndi-name` (JNDI-имя адресата);
- `webservice-description` содержит элементы `webservice-description-name` (уникальное имя Web-сервиса) и `wsdl-publish-location` (URL-адрес каталога, в котором будет находиться WSDL-файл Web-сервиса после развертывания);

- `login-config` определяет способ идентификации клиента и состоит из элементов: `auth-method` формирует механизм идентификации (`BASIC` — идентификация основана на логине и пароле и использует HTTP-протокол, `DIGEST` — идентификация основана на логине и пароле и использует HTTPS-протокол, `FORM` — используется специально созданная страничка с формой для ввода логина и пароля, а также страничка ошибки, `CLIENT-CERT` — идентификация основана на сертификате клиента); `realm-name` — имя realm для идентификации `BASIC`; `form-login-config` указывает странички формы и ошибки для идентификации `FORM`, используя элементы `form-login-page` и `form-error-page`.

Файл sun-cmp-mappings.xml, структура которого определяется файлом sun-cmp-mapping_1_2.dtd, имеет следующие элементы:

- `sun-cmp-mappings` — корневой элемент;
- `sun-cmp-mapping` описывает соответствие CMP Entity-компонента схеме базы данных. Вложенные элементы:
  - `schema` — файл, содержащий описание схемы базы данных. Элемент содержит относительный путь файла с именем файла без расширения dbschema. Если элемент пустой, то файл автоматически генерируется при развертывании;
  - `entity-mapping` описывает соответствие столбцам базы данных. Вложенные элементы:
    - `ejb-name` — имя компонента, определенное в файле ejb-jar.xml;
    - `table-name` — имя таблицы;
    - `cmp-field-mapping` связывает поля компонента со столбцами. Разрешается только один элемент `cmp-field-mapping` для каждого элемента `cmp-field` файла ejb-jar.xml. Вложенные элементы:
      - `field-name` — Java-идентификатор поля;
      - `column-name` — имя столбца;
      - `read-only` — если `true`, то столбец доступен только для чтения, по умолчанию `false`;
      - `fetched-with` — по умолчанию EJB-контейнер загружает полностью все CMP-поля CMP Entity-компонента в пул экземпляров компонента. В случае если бизнес-методами задействуются не все CMP-поля, то элемент `fetched-with` дает возможность указать именно те CMP-поля, состояние которых необходимо для загрузки в пул, остальные поля загружаться не будут, что снизит накладные расходы. Элемент содержит взаимоисключающие эле-

менты: `default` — данное поле загружается всякий раз, когда компонент загружается из базы данных; `level` — данное поле относится к иерархической группе загрузки, при этом значение элемента определяет очередность загрузки группы, элемент определяет порядок загрузки полей только в пределах одного Entity-компонента; `named-group` — данное поле относится к именованной группе загрузки — поля Entity-компонента, относящиеся к одной именованной группе загрузки, загружаются одновременно; `none` — поле загрузится только тогда, когда оно будет задействовано в транзакции;

- `cmr-field-mapping` — связывает зависимые поля со столбцами. Разрешается только один элемент `cmr-field-mapping` для каждого элемента `cmr-field` файла ejb-jar.xml. Вложенные элементы:
  - `cmr-field-name` — Java-идентификатор зависимого поля;
  - `column-pair` указывает два столбца связанных таблиц базы данных с помощью двух элементов `column-name`, где первый элемент относится к таблице данного Entity-компонента, а второй элемент — к зависимой таблице;
  - `fetched-with` содержит взаимоисключающие элементы: `default` — данное поле загружается всякий раз, когда компонент загружается из базы данных; `level` — данное поле относится к иерархической группе загрузки, при этом значение элемента определяет очередность загрузки группы, элемент определяет порядок загрузки полей только в пределах одного Entity-компонента;
  - `named-group` — данное поле относится к именованной группе загрузки — поля Entity-компонента, относящиеся к одной именованной группе загрузки, загружаются одновременно;
  - `none` — поле загрузится только тогда, когда оно будет задействовано в транзакции;
- `secondary-table` указывает подчиненные таблицы с помощью элементов: `table-name` — имя таблицы; `column-pair` указывает два столбца связанных таблиц базы данных с помощью двух элементов `column-name`, где первый элемент относится к таблице данного Entity-компонента, а второй элемент — к зависимой таблице;
- `consistency` определяет, каким образом контейнер обеспечивает гарантированную непротиворечивость транзакций для Entity-компонентов, содержит взаимоисключающие элементы:
  - `check-version-of-accessed-instances` устанавливает проверку контейнером соответствия значений столбца, содержащего зна-

чения версии, для состояния компонента до транзакции и по ее окончании, содержит элемент `column-name` (имя столбца, содержащего значения версии — `version column`);

- `lock-when-loaded` устанавливает блокировку обновления строк базы данных всякий раз, когда Entity-компонент загружается из базы данных;
- `check-modified-at-commit` — устанавливает проверку параллельной модификации полей Entity-компонента при завершении транзакции;
- `none` — непротиворечивость не обеспечивается.

Файл sun-application-client.xml, структура которого определяется файлом sun-application-client_5_0-0.dtd, содержит следующие элементы:

- `sun-application-client` — корневой элемент;
- `ejb-ref` связывает имя, определенное в элементе `ejb-ref-name` файла application-client.xml, с абсолютным JNDI-именем с помощью элементов `ejb-ref-name` и `jndi-name`;
- `resource-ref` связывает имя, определенное в элементе `resource-ref` файла application-client.xml, с абсолютным JNDI-именем и именем пользователя сервера с помощью элементов `res-ref-name`, `jndi-name` и `default-resource-principal`;
- `resource-env-ref` связывает имя, определенное в элементе `resource-env-ref` файла application-client.xml, с абсолютным JNDI-именем с помощью элементов `resource-env-ref-name` и `jndi-name`;
- `service-ref` — параметры выполнения для используемого Web-сервиса. Вложенные элементы:
  - `service-ref-name` — JNDI-имя ссылки;
  - `port-info` — используемый Web-сервис. Вложенные элементы:
    - `service-endpoint-interface` — JNDI-имя Web-сервиса;
    - `wsdl-port` — WSDL-имя Web-сервиса, имеет вложенные элементы `namespaceURI` (пространство имен) и `localpart` (локальная часть QName);
    - `stub-property` — вложенные элементы `name` и `value` определяют свойства объекта `javax.xml.rpc.Stub`, такие как адрес класса реализации Web-сервиса, перед тем, как заглушка будет возвращена клиенту. Программный код — заглушка создается сервером в ответ на первый запрос клиента к Web-сервису и пересылается клиенту для обеспечения SOAP-протокола;

- call-property — вложенные элементы name и value определяют, в случае динамического вызова клиентом методов класса реализации Web-сервиса, свойства объекта javax.xml.rpc.Call перед тем, как объект будет возвращен сервером клиенту;
- message-security-binding — вложенный элемент message-security описывает защиту SOAP-сообщений Web-сервиса, используя элементы: message — метод или операция, к которым применяются требования безопасности SOAP-сообщений, вложенные элементы: java-method (метод описывается элементами method-name и method-params с method-param) и operation-name (WDSL-операция Web-сервиса); request-protection — требования идентификации для обработки запросов, имеет атрибуты auth-source (тип идентификации sender — имя и пароль или content — цифровая подпись) и auth-recipient (идентификация адресата производится до или после идентификации содержимого — before-content или after-content); response-protection — требования идентификации для обработки ответов, имеет атрибуты auth-source (тип идентификации sender — имя и пароль или content — цифровая подпись) и auth-recipient (идентификация адресата производится до или после идентификации содержимого — before-content или after-content);

- call-property — вложенные элементы name и value определяют, в случае динамического вызова клиентом методов класса реализации Web-сервиса, свойства объекта javax.xml.rpc.Call перед тем, как объект будет возвращен сервером клиенту;
- wsdl-override переопределяет URL-адрес WSDL-документа Web-сервиса;
- service-impl-class — имя сгенерированного класса реализации интерфейса javax.xml.rpc.Service, обеспечивающего клиентский доступ к функциональности Web-сервиса;
- service-qname — расширенное имя QName XML-элемента WDSL-файла, описывающего Web-сервис с помощью элементов namespaceURI (пространство имен) и localpart (локальная часть имени);

- message-destination связывает (переопределяет) логическое имя адресата с абсолютным JNDI-именем адресата с помощью элементов message-destination-name (логическое имя адресата), jndi-name (JNDI-имя адресата);
- message-destination-ref выполняет роль message-destination-link и связывает имя JNDI-ссылки системы JMS на адресата сообщений файла

application-client.xml с физическим JNDI-именем адресата с помощью элементов `message-destination-ref-name` и `jndi-name`;

- `java-web-start-access` — сервер GlassFish обеспечивает автоматическую поддержку технологии Java Web Start, позволяя развернуть клиентское приложение в клиентском контейнере, просто набрав в браузере URL-адрес клиентского приложения. Используя элемент `java-web-start-access`, можно изменить параметры по умолчанию Java Web Start. Элемент содержит вложенные элементы: `context-root` — часть URL-адреса, относящаяся к клиентскому приложению (по умолчанию **http://host:port/[context-root:app-name/relative-URI-to-appclient-jar]** или **http://host:port/[context-root:module-name]**); `eligible` — если `true` (по умолчанию) обеспечена поддержка Java Web Start, если `false`, то нельзя развернуть клиентское приложение с помощью Java Web Start; `vendor` — имя производителя, которое появляется в JWS-окне загрузки, по умолчанию Application Client.

ГЛАВА 8

Технологии платформы Java EE

Технологии Web-приложений

Технология Java Servlet

Технология Java Servlet описывается спецификациями Java Servlet 2.5 (JSR 54) и Java Servlet 3.0 (JSR 315).

Технология Java Servlet позволяет расширить функциональность Web-сервера путем создания Web-компонентов — *сервлетов*, взаимодействующих с клиентскими приложениями и формирующих ответы на их запросы.

Стандартное программное обеспечение Web-сервера отвечает за предоставление информации, в виде различного рода документов, в ответ на запрос от клиента. Так как сервлеты могут использовать полный набор библиотек Java API, то с их помощью можно существенно расширить возможности Web-сервера. В отличие от скриптов CGI (Common Gateway Interface), сервлеты обрабатывают каждый запрос от клиента в отдельном подпроцессе — потоке, а не в отдельном процессе как CGI-скрипты, что дает преимущество в производительности.

Поддержка технологии Java Servlet обеспечивается путем интеграции сервлет-контейнера и Web-сервера.

Наиболее распространенным сервлет-контейнером является Apache Tomcat — домашняя страничка **http://tomcat.apache.org/**.

Интеграция Tomcat с такими Web-серверами, как Apache Web Server и Microsoft IIS, заключается в настройке переадресации запросов с соответствующими расширениями от Web-сервера сервлет-контейнеру. Такие распространенные серверы Java EE как GlassFish и JBoss изначально содержат в качестве сервлет-контейнера Apache Tomcat.

Сервлет является Java EE Web-компонентом, генерирующим динамический контент и управляемый сервлет-контейнером. Сервлет-контейнер представляет собой расширение Web-сервера или сервера приложений Java EE, которое отвечает за получение запросов и отправку ответов клиентскому приложению, а также за управление жизненным циклом сервлетов, обрабатывающих полученные сервлет-контейнером запросы. Сервлет-контейнер осуществляет связь с клиентским приложением с помощью HTTP- и HTTPS-протоколов.

Подобно модели апплетов, главный класс сервлета должен реализовывать интерфейс `javax.servlet.Servlet` или расширять класс, реализующий данный интерфейс. Библиотека Java Servlet API предлагает два класса, реализующих интерфейс `Servlet` — это классы `javax.servlet.GenericServlet` (протоколонезависимый сервлет) и `javax.servlet.http.HttpServlet` (HTTP-сервлет).

Сервлет связывается с определенным клиентским запросом с помощью элемента `servlet-mapping` файла дескриптора развертывания web.xml.

Реализуя интерфейс `javax.servlet.Filter`, можно решать задачи фильтрации (ограничения) как клиентских запросов сервлетам или статическим ресурсам, так ответов клиенту.

Интерфейс `javax.servlet.RequestDispatcher` позволяет переадресовать клиентский запрос другому ресурсу.

Модульность файла web.xml обеспечивается возможностью добавления файла web-fragment.xml, содержащего дополнительные метаданные, в каталог WEB-INF\lib JAR-файла Web-приложения или в каталог META-INF JAR-файла дополнительного модуля каталога WEB-INF\lib.

Спецификация Java Servlet 3.0 отличается от спецификации Java Servlet 2.5 поддержкой аннотаций и расширяемости путем добавления Web-фрагментов (web fragments) в Web-приложение.

Технология JavaServer Pages

Технология JavaServer Pages (JSP) описывается спецификациями JavaServer Pages 2.1, Expression Language 2.1 (JSR 245) и JavaServer Pages Standard Tag Library (JSTL) (JSR 52).

Технология JSP позволяет создавать динамические Web-странички, не прибегая к языку Java.

JSP-страничка представляет собой текстовый документ, содержащий два типа информации — элементы HTML, SVG, WML и XML, формирующие статический контент, и JSP-элементы, которые создают динамический контент Web-странички.

Файловое расширение для JSP-страниц — jsp, файловое расширение для фрагментов JSP-страниц — jspf.

Для того чтобы JSP-страницы могли выполнять свои функции, они должны быть транслированы в классы сервлетов. Трансляция JSP-страниц может быть произведена предварительно с помощью утилиты `jspc` командной строки сервера, при этом классы реализации включаются в Web-приложение, или JSP-страницы могут быть транслированы контейнером при развертывании или при клиентском запросе к нетранслированной JSP-странице. После трансляции экземпляр класса реализации JSP-страницы обрабатывает клиентские запросы и создает ответы клиенту.

Синтаксис JSP может быть двух несмешиваемых типов — стандартный (JSP-страница с расширением jsp) и XML-синтаксис (JSP-документ с расширением файла jspx).

Элементами JSP являются:

- директивы;
- действия;
- скриптовые элементы, включающие в себя скриптлеты — фрагменты Java-кода, объявления переменных или методов Java-кода, вычисляемые выражения Java-кода;
- комментарии, видимые и невидимые клиентом;
- EL-выражения (Expression Language), являющиеся альтернативой скриптовым элементам;
- стандартные теги библиотеки JavaServer Pages Standard Tag Library (JSTL);
- пользовательские теги.

Подробно описание и синтаксис технологии JSP рассмотрены в *приложении 1*.

Технология JavaServer Faces

Технология JavaServer Faces (JSF) описывается спецификациями JavaServer Faces 2.0 (JSR 314) и JavaServer Faces 1.2 (JSR 252).

Технология JSF представляет собой платформу для создания пользовательского интерфейса Web-приложений, которые выполняются на стороне сервера Java EE и визуализируют свой графический интерфейс пользователя для клиента.

Технология JSF позволяет конструировать пользовательский интерфейс из повторно используемых компонентов, обеспечивать обмен данными с компонентами, управлять состоянием компонентов, обрабатывать события, соз-

давать пользовательские компоненты, которые могут многократно использоваться для создания интерфейса пользователя.

Технология JavaServer Faces включает в себя:

- библиотеку JavaServer Faces API, содержащую стандартные компоненты пользовательского интерфейса, а также классы и интерфейсы, позволяющие управлять обработкой клиентского запроса, состоянием компонентов, обрабатывать события, проверять вводимую пользователем информацию, обеспечивать навигацию, интернационализацию, создавать пользовательские компоненты;
- библиотеки тегов JavaServer Pages для включения JSF-интерфейса в JSP-страницу.

JSF-приложение представляет собой набор Web-страничек, в которые включены JSF-компоненты, набор библиотек тегов, с помощью которых JSF-компоненты включены в Web-странички, набор JavaBeans-компонентов, определяющих свойства и функции JSF-компонентов, конфигурационные файлы faces-config.xml, файл дескриптора развертывания приложения web.xml, пользовательские JSF-компоненты, библиотеки пользовательских тегов, с помощью которых пользовательские JSF-компоненты включены в Web-странички.

Технология JSF поддерживает технологию AJAX.

Технология AJAX (Asynchronous JavaScript and XML) обеспечивает фоновый обмен данными между клиентским браузером и сервером, что позволяет при обновлении данных не перезагружать Web-страничку полностью, а только изменившуюся ее часть. Технология JSF поддерживает обработку AJAX-запроса — объекта `XMLHttpRequest` JavaScript, который управляет взаимодействием клиента с сервером.

Подробно библиотеки технологии JSF рассмотрены в *приложении 12*.

Технологии Web-сервисов

Web-сервис — это приложение, идентифицируемое строкой URI (отличие URL от URI в том, что URL — это URI, который кроме идентификации ресурса указывает его местонахождение), доступ к которому осуществляется с помощью стандартных протоколов.

Технология RESTful

Java-технология Web-сервисов RESTful (JAX-RS) описывается спецификацией JSR 311.

Web-сервис RESTful — это сервис, обмен данными с которым осуществляется по HTTP-протоколу.

JAX-RS-приложение состоит из одного или нескольких ресурсов (resources) и может включать в себя поставщиков (providers).

Ресурсы и поставщики конфигурируются классом, расширяющим абстрактный класс `javax.ws.rs.core.Application`, содержащий переопределяемый метод `getClasses()`, который возвращает набор классов ресурсов и поставщиков.

JAX-RS-приложение упаковывается как сервлет в WAR-файл. Подкласс `Application`, классы ресурсов и поставщиков упаковываются в папку WEB-INF\classes, необходимые библиотеки — в папку WEB-INF\lib. При этом подкласс `Application` указывается в дескрипторе развертывания web.xml JAX-RS-приложения:

```
<web-app>
  <servlet>
    <servlet-name>имя приложения</servlet-name>
    <servlet-class>
      com.sun.jersey.spi.container.servlet.ServletContainer
    </servlet-class>
    <init-param>
      <param-name>javax.ws.rs.Application</param-name>
      <param-value>имя подкласса Application</param-value>
    </init-param>
  </servlet>
. . .
```

Класс ресурса — это Java-класс Web-сервиса, использующий аннотации JAX-RS для обеспечения соответствующего Web-ресурса.

Класс поставщика — это Java-класс, промаркированный аннотацией `@Provider` и реализующий один или несколько интерфейсов пакета `javax.ws.rs.ext` JAX-RS API. Например, спецификацией определены Entity-поставщики, реализующие интерфейсы `javax.ws.rs.ext.MessageBodyReader<T>` или `javax.ws.rs.ext.MessageBodyWriter<T>`.

Интерфейс `MessageBodyReader<T>` обеспечивает преобразование входящего потока в Java-тип и содержит метод `readFrom()`, который считывает Java-тип из входящего потока.

Интерфейс `MessageBodyWriter<T>`, наоборот, осуществляет преобразование Java-типа в выходящий поток с помощью метода `writeTo()`, который пишет Java-тип в HTTP-ответ.

Entity-поставщики создаются, если стандартная реализация не поддерживает необходимый Java-тип.

Аннотации технологии JAX-RS:

- `@MatrixParam` извлекает значение URI параметра `matrix` из тела запроса;
- `@QueryParam` извлекает значение URI параметра `query` из тела запроса;
- `@PathParam` извлекает значение URI параметра по шаблону из тела запроса;
- `@CookieParam` извлекает значение параметра `cookie` из тела запроса;
- `@HeaderParam` извлекает информацию из HTTP-заголовка запроса;
- `@FormParam` извлекает значения параметров HTML-формы, содержащиеся в теле запроса;
- `@DefaultValue` определяет значения по умолчанию для параметров запроса;
- `@Encoded` отключает автоматическое декодирование значений параметров запроса аннотациями `@QueryParam`, `@MatrixParam`, `@PathParam`, `@FormParam`;
- `@Path` идентифицирует URI-путь, который служит запросом к классу ресурса или методу класса ресурса;
- `@GET`, `@PUT`, `@POST`, `@DELETE`, `@HEAD` маркируют Java-методы, которые отвечают на соответствующие HTTP-методы запроса;
- `@Produces` определяет MIME-тип представления ресурса, который отсылается клиенту в ответ на запрос;
- `@Consumes` определяет MIME-тип представления ресурса, который посылается клиентом вместе с запросом;
- `@Context` вводит данные в поля класса, JavaBeans-свойства, параметры метода.

Технология XML-Based RPC (JAX-RPC)

Технология JAX-RPC (JSR 101) позволяет создавать Web-сервисы, обмен XML-сообщениями с которыми производится по протоколу SOAP1.1/HTTP.

JAX-RPC Web-сервисы описываются файлом WSDL (Web Services Description Language) и представляют собой набор процедур удаленного вызова (remote procedure calls, RPC).

В модели JAX-RPC на стороне сервера создаются:

- Java-интерфейс (service endpoint interface, SEI), содержащий методы удаленного вызова. Интерфейс SEI должен расширять интерфейс `java.rmi.Remote`;
- Java-классы, реализующие данный интерфейс;

- файл WSDL, описывающий Web-сервис;
- файл соответствия JAX-RPC (java-wsdl-mapping file), содержащий информацию, которая устанавливает соответствие между Java-интерфейсом Web-сервиса и файлом WSDL. Файл соответствия обеспечивает переносимость Web-сервиса и необходим среде выполнения для отображения документа WSDL на Java-классы, упакованные в WAR-файле приложения.

На стороне клиента создается статический или динамический локальный объект, представляющий Web-сервис, — объект класса `proxy`, методы которого затем вызываются.

Статический класс `proxy` (stub — заглушка) генерируется при компиляции клиентского приложения, используя файл WSDL. Объект статического класса `proxy` создается с помощью интерфейса `javax.xml.rpc.Stub`.

Динамический класс `proxy` генерируется на стороне сервера и пересылается клиенту в ответ на его обращение к Web-сервису по WSDL-описанию. Методы, позволяющие сделать такое обращение, предоставляют интерфейс `javax.xml.rpc.Service`, объект которого создается классом `javax.xml.rpc.ServiceFactory`, используя URL-адрес файла WSDL Web-сервиса.

Интерфейс `javax.xml.rpc.Call` позволяет динамически вызывать удаленные методы Web-сервиса без генерации класса `proxy` и использования файла WSDL.

Клиентское приложение, являющееся частью приложения Java EE, может использовать сервис JNDI для нахождения локального Web-сервиса с помощью метода `lookup` интерфейса `javax.naming.Context`.

На стороне сервера, так же как и на стороне клиента, существует класс `proxy`, генерируется класс серверной заглушки (tie — связка).

Заглушки stub и tie отвечают за сетевое соединение, создание/разбор, сериализацию/десериализацию и пересылку SOAP-сообщений.

Файл WSDL и файл соответствия java-wsdl-mapping.xml генерируются инструментом `wscompile` сервера Java EE. С помощью этого же компилятора генерируются заглушки stub и tie.

Элементы файла WSDL версии 1.1:

- `<wsdl:definitions>` — корневой элемент;
- `<wsdl:document>` — контейнер для документации (комментариев) Web-сервиса;
- `<wsdl:types>` описывает типы данных, используемые Web-сервисом с помощью вложенных элементов `<element>` и `<complexType>`;
- `<wsdl:message>` описывает SOAP-сообщения, атрибут `name` определяет идентификатор сообщения. Вложенные элементы `<part>` описывают неде-

лимые части сообщения, связывая типы данных с идентификаторами сообщений, с помощью атрибутов: `name` — идентификатор части сообщения; `element` — ссылка на элемент `<element>` типа данных, с использованием QName; `type` — ссылка на элемент `<simpleType>` или `<complexType>` типа данных, с использованием QName;

- `<wsdl:portType>` описывает услуги, предоставляемые Web-сервисом. В этой терминологии `port` — это интерфейс Web-сервиса или конечный пункт (endpoint), услуги — это методы интерфейса Web-сервиса или операции. Элемент имеет атрибут `name` (идентификатор порта) и вложенные элементы `<wsdl:operation>` (описывают операции). Элемент `<wsdl:operation>` имеет атрибут `name` (идентификатор операции) и вложенные элементы: `<wsdl:input>`, которые описывают входящее сообщение, атрибуты `name` (идентификатор) и `message` (ссылка на элемент `<wsdl:message>`, с использованием QName); `<wsdl:output>` описывает исходящее сообщение, атрибуты `name` (идентификатор) и `message` (ссылка на элемент `<wsdl:message>`, с использованием QName); `<wsdl:fault>` описывает сообщение об ошибке, атрибуты `name` (идентификатор) и `message` (ссылка на элемент `<wsdl:message>`, с использованием QName);
- `<wsdl:binding>` описывает формат сообщений и детали протокола для операций порта, имеет атрибуты `name` (идентификатор связи) и `type` (ссылка на порт). Вложенные элементы:
  - `<soap:binding>` имеет атрибуты: `transport` — транспортный протокол для SOAP-сообщений (например, `"http://schemas.xmlsoap.org/soap/http"` — HTTP-протокол) и `style` (значения: `rpc` — сообщение содержит параметры и возвращает значения, `document` — сообщение содержит документы);
  - `<soap:operation>` содержит информацию для операции в целом, имеет атрибуты `style` (значения: `rpc` или `document`) и `soapAction` (определяет заголовок операции);
  - `<soap:body>` — информация для тела сообщения, имеет атрибуты: `parts` (указывает часть сообщения); `use` — значения `literal` (кодировка определяется атрибутами `element`, `type` элемента `<part>`) или `encoded` (кодировка определяется атрибутом `type`); `encodingStyle` — список URI, представляющих кодировки, используемые в сообщении; `namespace` — пространство имен тела сообщения;
  - `<soap:fault>` — информация для сообщения об ошибке, атрибуты: `name`, `use`, `encodingStyle` и `namespace`;
  - `<soap:header>` и `<soap:headerfault>` содержат информацию для заголовка сообщения, атрибуты: `message`, `part`, `use`, `namespace` и `encodingStyle`;

- <soap:address> связывает порт с URI-адресом с помощью атрибута location;
- <http:address> — базовый URI порта;
- <http:binding> указывает, что связка использует HTTP-протокол, атрибут verb указывает HTTP-метод (например, GET);
- <http:operation> — относительный URI-адрес для операции, который в комбинации с <http:address> формирует полный адрес HTTP-запроса;
- <http:urlEncoded/> указывает, что HTTP-запрос формируется согласно стандартной кодировке (name1=value&name2=value...);
- <http:urlReplacement/> указывает, что HTTP-запрос формируется согласно алгоритму замещения;
- <mime:content> связывает содержимое сообщения с MIME-типами с помощью атрибутов part и type;
- <mime:multipartRelated> составляет набор частей в одно сообщение MIME-типа multipart/related с помощью вложенных элементов <mime:part>;
- <mime:mimeXml> связывает сообщение с элементом <element name="mimeXml" type="mime:mimeXmlType"/>, имеет атрибут part;

- <wsdl:service> связывает сервисный интерфейс с типом порта, связью и URI-адресом, имеет атрибут name (идентификатор) и вложенный элемент <wsdl:port> — атрибуты name и binding. URI-адрес указывается с помощью вложенного элемента <soap:address>;
- <wsdl:import> связывает пространство имен с адресом документа с помощью атрибутов namespace и location.

Элементы файла java-wsdl-mapping.xml:

- java-wsdl-mapping — корневой элемент;
- package-mapping указывает соответствие между Java-пакетом и пространством имен XML в документе WSDL с помощью элементов package-type (имя пакета) и namespaceURI (пространство имен);
- java-xml-type-mapping указывает соответствие между Java-типами и XML-типами данных документа WSDL с помощью вложенных элементов:
  - java-type — Java-тип;
  - root-type-qname — WSDL QName XML-типа;
  - anonymous-type-qname — WSDL QName анонимного XML-типа;

- `qname-scope` с возможными значениями: `simpleType`, `complexType`, `element`;
- `variable-mapping` имеет вложенные элементы:
  - `java-variable-name` — имя поля или JavaBeans-свойства Java-класса;
  - `data-member` — если `false`, тогда это JavaBeans-свойство;
  - `xml-attribute-name` — значение атрибута `name` или `ref` атрибута WSDL-элемента;
  - `xml-element-name` — значение атрибута `name` или `ref` WSDL-элемента;
  - `xml-wildcard` связывает с элементом `xsd:any`;

- `exception-mapping` указывает соответствие между исключительными ситуациями Web-сервиса и ошибками, определенными в документе WSDL с помощью вложенных элементов:
  - `exception-type` — Java-тип исключительной ситуации;
  - `wsdl-message` — QName WSDL-сообщения;
  - `wsdl-message-part-name` — часть WSDL-сообщения;
  - `constructor-parameter-order` указывает WSDL-элементы `complexType`, соответствующие параметрам Java-конструктора, вложенные элементы — `element-name`;
- `service-interface-mapping` указывает соответствие между Java-типом сервисного интерфейса (`javax.xml.rpc.Service`) и WSDL-сервисом (`<wsdl:service>`) с помощью вложенных элементов: `service-interface` — Java-тип сервисного интерфейса, для статического сервиса это `javax.xml.rpc.Service`; `wsdl-service-name` — QName WSDL-сервиса; `port-mapping` — вложенные элементы `port-name` (WSDL-имя порта) и `java-port-name` (Java-имя порта);
- `service-endpoint-interface-mapping` связывает интерфейс Web-сервиса с WSDL-портом и WSDL-связкой. Вложенные элементы:
  - `service-endpoint-interface` — полное имя Java-интерфейса;
  - `wsdl-port-type` — QName WSDL-порта;
  - `wsdl-binding` — QName WSDL-связки;
  - `service-endpoint-method-mapping` с вложенными элементами:
    - `java-method-name` — Java-метод интерфейса;
    - `wsdl-operation` — WSDL-операция;
    - `wrapped-element` — WSDL-сообщение с одной частью, соответствующее операции;

- `method-param-parts-mapping` с вложенными элементами: `param-position` (позиция параметра метода), `param-type` (Java-тип параметра), `wsdl-message-mapping` (вложенные элементы: `wsdl-message` — QName WSDL-сообщения, `wsdl-message-part-name` — часть WSDL-сообщения, `parameter-mode` — тип параметра `IN`, `OUT`, `INOUT`, `soap-header` — если `true`, тогда параметр соответствует SOAP-заголовку);
- `wsdl-return-value-mapping` — вложенные элементы `method-return-value` (Java-тип возвращаемого методом интерфейса значения), `wsdl-message` — QName WSDL-сообщения, `wsdl-message-part-name` — часть WSDL-сообщения;

- `version` — версия схемы.

Технология XML-Based Web Services (JAX-WS)

Технология JAX-WS (JSR 224) является преемником технологии JAX-RPC.

Технология JAX-WS без дополнительных настроек обеспечивает поддержку протоколов SOAP 1.1, SOAP 1.2 и XML/HTTP, а также поддерживает расширяемость протоколов. Технология JAX-WS интегрирована с технологией Java Architecture for XML Binding (JAXB) (JSR 222), которая обеспечивает преобразование между XML-документами и Java-классами. Кроме того, JAX-WS с помощью асинхронного интерфейса `javax.xml.ws.AsyncHandler<T>` API клиента и интерфейса `javax.xml.ws.Response<T>` позволяет разрабатывать асинхронные Web-службы.

В модели JAX-WS на стороне сервера создаются:

- Java-класс Web-сервиса (web service endpoint), маркированный аннотацией `@WebService` (JSR 181) или `@WebServiceProvider`. В модели JAX-WS создание отдельного интерфейса service endpoint interface (SEI) не требуется. Бизнес-методы Java-класса, видимые клиентом Web-сервиса, маркируются аннотацией `@WebMethod`;
- инструмент `wsgen` сервера Java EE считывает класс web service endpoint и генерирует все необходимые файлы для развертывания Web-сервиса, включая файл WSDL. Инструмент `wsimport` генерирует и компилирует файлы заглушки для клиента.

На стороне клиента используется аннотация `@WebServiceRef`, объявляющая ссылку на Web-сервис, и аннотация `@WebServiceClient`, маркирующая класс, расширяющий класс `javax.xml.ws.Service`. Класс `Service` является аналогом интерфейса `javax.xml.rpc.Service` и обеспечивает клиентский доступ к Web-сервису.

Аннотации технологии JAX-WS (JSR 181, JSR 224, JSR 222, JSR 250):

- `@WebService` маркирует класс или интерфейс Web-сервиса. Атрибуты: `name` — имя Web-сервиса, соответствует имени `wsdl:portType`; `endpointInterface` позволяет отдельно объявить интерфейс Web-сервиса; `portName` соответствует имени `wsdl:port`; `serviceName` соответствует имени `wsdl:service`; `targetNamespace` — пространство имен; `wsdlLocation` — адрес WSDL-файла;
- `@WebMethod` маркирует метод как операцию Web-сервиса. Атрибуты: `operationName` соответствует имени `wsdl:operation`; `action` — поле заголовка soap action; `exclude` — если `true`, тогда метод не рассматривается как операция Web-сервиса;
- `@OneWay` указывает, что у данного метода `@WebMethod` есть только входящие сообщения и нет исходящих;
- `@WebParam` настраивает связь параметров метода с частями сообщений и XML-элементами. Атрибуты: `name` — имя параметра, соответствует `wsdl:part` или имени XML-элемента; `partName` — имя части `wsdl:part`, представляющей параметр; `targetNamespace` — пространство имен параметра; `mode` — тип параметра `IN`, `OUT`, `INOUT`; `header` — если `true`, тогда параметр является параметром из заголовка сообщения;
- `@WebResult` настраивает связь возвращаемого значения метода с частями сообщений и XML-элементами. Атрибуты: `name` — имя возвращаемого значения, соответствует `wsdl:part` или имени XML-элемента; `partName` — имя части `wsdl:part`, представляющей значение; `targetNamespace` — пространство имен значения; `header` — если `true`, тогда значение является значением из заголовка сообщения;
- `@HandlerChain` связывает Web-сервис с внешним конфигурационным файлом управления сообщениями. Пакеты `javax.xml.ws.handler` и `javax.xml.ws.handler.soap` содержат интерфейсы, позволяющие создавать классы обработчиков сообщений запросов и ответов. Атрибут `file` указывает URL-адрес файла handlers.xml;
- `@SOAPBinding` указывает детали SOAP-протокола. Атрибуты: `style` — стиль кодировки сообщений (`DOCUMENT` (по умолчанию) или `RPC`); `use` — стиль форматирования сообщений (`LITERAL` (по умолчанию) или `ENCODED`); `parameterStyle` — параметры метода, вложенные элементы — `WRAPPED` (по умолчанию) или представляют все тело сообщения `BARE`;
- `@Addressing` указывает использование WS-Addressing. Спецификация Web Services Addressing определяет механизм запроса к Web-сервису и обмена сообщениями, независимый от транспортного протокола. Атрибуты: `enabled` — если `true` (по умолчанию), тогда WS-Addressing активировано;

`required` — если `true`, тогда заголовки WS-Addressing обязательны во входящих сообщениях, по умолчанию `false`;

- `@Action` связывает действия WS-Addressing с сообщениями WSDL-операции. Например, можно идентифицировать операцию, вызываемую при получении сообщения запроса. Атрибуты: `input` — URI для входящего сообщения; `output` — URI для исходящего сообщения; `fault` — URI для сообщения об ошибке;
- `@BindingType` определяет тип соединения для класса Web-сервиса. Соединение (binding) — это формальный набор правил для передачи сообщений. Атрибут `value` указывает идентификатор URI соединения, по умолчанию — `value=http://www.w3.org/2003/05/soap/bindings/HTTP/`;
- `@FaultAction` используется в аннотации `@Action` для связи действия WS-Addressing с сообщениями WSDL-операции об ошибке, возникающими в определенной исключительной ситуации. Атрибуты: `className` — имя класса исключения; `value` — URI для сообщения об ошибке;
- `@MTOM` указывает использование MTOM. Message Transmission Optimization Mechanism (MTOM) — это метод, позволяющий более эффективно передавать бинарные данные в SOAP-сообщениях, используя XOP (XML-binary Optimized Packaging). Атрибуты: `enabled` — если `true` (по умолчанию), тогда MTOM активировано; `threshold` — размер бинарных данных, свыше которого производится XOP-кодировка, по умолчанию 0;
- `@RequestWrapper` определяет Java-класс, который является компонентом JavaBeans для параметров метода интерфейса SEI, включаемых в сообщение запроса. Атрибуты: `className` — полное имя класса; `localName` — локальное имя XML-элемента, соответствует имени `wsdl:operation`; `targetNamespace` — пространство имен XML-элемента;
- `@RespectBinding` указывает использование контроля над полным соблюдением содержимого элемента `wsdl:binding` WSDL-файла Web-сервиса. Атрибут — `enabled`, если `true` (по умолчанию), тогда контроль активирован;
- `@ResponseWrapper` определяет Java-класс, который является компонентом JavaBeans для параметров метода интерфейса SEI, включаемых в сообщение ответа. Атрибуты: `className` — полное имя класса; `localName` — локальное имя XML-элемента, соответствует имени `wsdl:operation`; `targetNamespace` — пространство имен XML-элемента;
- `@ServiceMode` указывает для класса Web-сервиса, реализующего интерфейс `javax.xml.ws.Provider<T>` (аннотация `@WebServiceProvider`), доступ к содержимому элемента `<soap:Body>` сообщения — значение атрибута `value=Service.Mode.PAYLOAD`, или доступ к содержимому элемента

`<soap:Envelope>` сообщения — значение атрибута `value=Service.Mode.MESSAGE`;

- `@WebServiceClient` маркирует класс, расширяющий класс `javax.xml.ws.Service`, обеспечивающий клиентский доступ к Web-сервису. Атрибуты: `name` — имя Web-сервиса; `targetNamespace` — пространство имен Web-сервиса; `wsdlLocation` — URL-адрес документа WSDL;
- `@WebEndpoint` — в классе, расширяющем класс `javax.xml.ws.Service`, маркирует метод (`getPort()`), возвращающий экземпляр класса `proxy` (заглушку). Атрибут `name` — имя порта;
- `@WebFault` маркирует класс исключения. Атрибуты: `name` — имя элемента `<wsdl:fault>`; `targetNamespace` — пространство имен элемента; `faultBean` — имя компонента JavaBeans, представляющего параметры сообщения об ошибке;
- `@WebServiceProvider` маркирует класс, реализующий интерфейс `javax.xml.ws.Provider<T>`, который дает возможность Web-сервису работать на уровне XML-сообщений. Атрибуты: `wsdlLocation` — URL-адрес документа WSDL; `serviceName` — имя `wsdl:service`; `portName` — имя порта; `targetNamespace` — пространство имен Web-сервиса;
- `@WebServiceRef` определяет ссылку на Web-сервис. Атрибуты: `name` — JNDI-имя ресурса; `type` — Java-тип ресурса; `mappedName` — специфическое имя ресурса; `value` — сервисный класс (тип, расширяющий `javax.xml.ws.Service`); `wsdlLocation` — URL-адрес документа WSDL;
- `@WebServiceRefs` — контейнер для аннотаций `@WebServiceRef`.

Классы JavaBeans-компонентов, представляющие данные для сообщений, маркируются JAXB-аннотациями, связывающими с XML-элементами:

- `@XmlRootElement` связывает класс верхнего уровня с корневым элементом XML-схемы WSDL-файла Web-сервиса. Атрибуты: `name` (имя элемента) и `namespace` (пространство имен элемента);
- `@XmlAccessorType` указывает сериализацию полей или свойств компонента. Атрибут `value` принимает значение `AccessType.FIELD` или `AccessType.PROPERTY`;
- `@XmlType` связывает компонент с элементом `complexType` WSDL-файла. Атрибуты: `name` — имя элемента; `namespace` — пространство имен элемента; `propOrder` — список идентификаторов свойств компонента;
- `@XmlElement` связывает свойство компонента с элементом `element` WSDL-файла. Атрибуты: `name` — имя элемента; `namespace` — пространство имен элемента.

Общие аннотации:

- `@Resource` вставляет ресурс `javax.xml.ws.WebServiceContext`, обеспечивающий доступ к контексту сообщений и информации системы безопасности;
- `@PostConstruct` маркирует метод, осуществляющий инициализацию;
- `@PreDestroy` маркирует метод, осуществляющий освобождение ресурсов.

Элементы файла WSDL версии 2.0:

- `<wsdl:description>` — корневой элемент, атрибут `targetNamespace` указывает пространство имен Web-сервиса;
- `<wsdl:document>` — комментарии;
- `<wsdl:include>` позволяет включать содержимое других документов WSDL с одинаковым пространством имен; атрибут `location` указывает адрес включаемого документа;
- `<wsdl:import>` позволяет включать содержимое других документов WSDL с другим пространством имен. Атрибуты: `location` — адрес включаемого документа, атрибут `namespace` — пространство имен включаемого документа WSDL;
- `<wsdl:types>` определяет типы данных. Вложенные элементы `<xs:schema>` и `<xs:import>` позволяют включать и импортировать соответственно XML-схемы в WSDL-документ;
- `<wsdl:interface>` описывает услуги, предоставляемые Web-сервисом, имеет атрибуты `name` (идентификатор), `extends` (указывает расширяемый интерфейс) и `styleDefault` (указывает значение по умолчанию атрибутов `style` операций). Вложенный элемент `<wsdl:fault>` объявляет возможные ошибки выполнения методов интерфейса и имеет атрибуты `name` (идентификатор) и `element` (ссылка на элемент `<element>` типа данных с использованием QName). Вложенный элемент `<wsdl:operation>` указывает метод интерфейса Web-сервиса и имеет атрибуты:
  - `name` — идентификатор; `pattern` — URI-адрес операции;
  - `style` — `http://www.w3.org/ns/wsdl/rpc` (RPC-стиль сообщений), `http://www.w3.org/ns/wsdl/style/iri` (IRI-стиль сообщений), `http://www.w3.org/ns/wsdl/style/multipart` (Multipart-стиль сообщений);
  - `wsdlx:safe` принимает значения `true`/`false` и указывает, что данная операция безопасна, т. е. не модифицирует ресурс и может быть вызвана много раз с одинаковым результатом.

Элемент <operation> имеет вложенные элементы:

- <wsdl:input> и <wsdl:output>, которые описывают входящее и исходящее сообщения; имеет атрибуты element (ссылка на элемент <element> типа данных, используя QName) и messageLabel (идентификатор сообщения);
- <wsdl:infault> и <wsdl:outfault> описывают сообщения об ошибках, атрибуты: messageLabel и ref (ссылка на элемент <wsdl:fault>);

- <wsdl:binding> указывает протокол и детали кодировки для отправки/получения сообщений, имеет атрибуты name (идентификатор связи), type (протокол, например, type="http://www.w3.org/ns/wsdl/soap"), interface (ссылка на интерфейс), wsoap:version (версия протокола), wsoap:protocol (транспортный протокол), wsoap:mepDefault (модель обмена сообщениями по умолчанию), whttp:methodDefault (HTTP-метод запроса по умолчанию). Вложенный элемент <wsdl:fault> связывает формат сообщений с ошибками интерфейса, атрибут ref ссылается на элемент <wsdl:fault> элемента <wsdl:interface>; атрибут wsoap:code указывает возможную ошибку SOAP-сообщения для операции. Вложенный элемент <wsdl:operation> связывает операции интерфейса с форматом сообщений, атрибут ref ссылается на элемент <wsdl:operation> элемента <wsdl:interface>, атрибут wsoap:mep указывает модель обмена сообщениями (например, wsoap:mep="http://www.w3.org/2003/05/soap/mep/request-response"), атрибут whttp:location указывает шаблон HTTP-запроса. Элемент <wsdl:operation> имеет вложенные элементы <wsdl:input> (атрибут messageLabel), <wsdl:output> (атрибут messageLabel), <wsdl:infault> (атрибут ref), <wsdl:outfault> (атрибут ref);
- <wsdl:service> определяет интерфейс, который Web-сервис поддерживает, и список адресов конечных точек, обеспечивающих доступ к Web-сервису, имеет атрибуты name (идентификатор) и interface (ссылка на элемент <wsdl:interface>). Вложенные элементы <wsdl:endpoint> описывают конечную точку с помощью атрибутов name (идентификатор), binding (ссылка на связь) и address (URL-адрес запроса к Web-сервису).

Технологии Java EE Enterprise Application

Технология Enterprise JavaBeans

Технология Enterprise JavaBeans позволяет создавать серверные компоненты, реализующие бизнес-логику приложения Java EE.

Технология EJB описывается спецификациями JSR 220 и JSR 318.

Отличие спецификации JSR 220 (EJB 3.0) от предыдущих спецификаций:

- использование Java-аннотаций, упрощающих разработку и исключающих необходимость для разработчика создавать дескриптор развертывания, при этом в отсутствие аннотаций, описывающих общие ситуации, контейнер использует правила по умолчанию;
- ввод объектов и поиск в пространстве имен JNDI с помощью аннотаций;
- исключение требования для сессионных EJB-компонентов иметь домашний и компонентный интерфейсы. Требуемый бизнес-интерфейс для сессионных компонентов может быть простым Java-интерфейсом;
- EJB-компоненты могут объявляться с помощью аннотаций, при этом транзакциями (Java Transaction API (JTA) — JSR 907) управляет контейнер;
- создание Entity-компонентов описывается Java Persistence API;
- возможность перехвата для сессионных и управляемых сообщениями компонентов.

Отличие спецификации JSR 318 (EJB 3.1) от JSR 220 (EJB 3.0):

- упрощенный локальный доступ к сессионным компонентам без использования локального бизнес-интерфейса;
- возможность размещения EJB-компонентов прямо в WAR-файле приложения Java EE (каталог WEB-INF\classes) без использования файла ejb.jar;
- библиотека Embeddable EJB API, позволяющая выполнять EJB-компоненты в EJB-контейнере, встроенном в платформу Java SE;
- сессионные компоненты Singleton, предназначенные для хранения совместно используемых данных уровня приложения;
- аннотация `@Schedule`, реализующая таймер;
- стандартный синтаксис для указания глобальных JNDI-имен EJB-компонентов, не зависящий от конкретного сервера приложений Java EE;
- аннотация `@Asynchronous`, позволяющая легко создавать асинхронные сессионные компоненты;
- ограниченный набор функциональности EJB Lite, который может быть обеспечен в профиле Java EE, таком как профиль Java EE Web.

Архитектура технологии EJB определяет три типа EJB-объектов:

- *сессионные объекты* (Session). Сессионный объект выполняется для одного клиента, может поддерживать транзакции, иметь доступ к базе данных и обновлять ее данные. Время жизни сессионного объекта ограничивается временем работы EJB-контейнера, может быть трех типов: Stateless, Stateful и Singleton. Stateless-объект не сохраняет свое состояние между

вызовами методов. Stateful-объект хранит свое состояние между вызовами методов. Singleton-объект предназначен для хранения совместно используемых данных уровня приложения и поддерживает параллельный к себе доступ со стороны клиентов;

- *управляемые сообщениями объекты* (Message-Driven). MDB-объект асинхронно исполняется при получении единственного клиентского сообщения, поддерживает транзакции, может иметь доступ и обновлять данные базы данных, время жизни объекта ограничивается временем работы EJB-контейнера, является Stateless-объектом;
- *объекты-сущности* (Entity). Entity-объект представляет данные базы данных, время жизни объекта определяется временем жизни данных базы данных.

В модели EJB клиент может осуществлять удаленный или локальный доступ к сессионным EJB-компонентам или же сессионные EJB-компоненты могут выступать для клиента в качестве Web-сервисов.

Для EJB 3.*x* удаленный доступ к сессионным компонентам осуществляется через удаленный бизнес-интерфейс компонентов. Локальный доступ осуществляется через локальный бизнес-интерфейс или без интерфейса на той же виртуальной машине JVM, на которой развернут сессионный компонент. Компоненты Stateless и Singleton могут выступать для клиента в качестве Web-сервисов, при этом они должны соответствовать спецификации JAX-RPC или JAX-WS.

На стороне клиента удаленный или локальный доступ к сессионным компонентам может быть получен двумя способами:

```
@EJB Bean bean;
```

или:

```
@Resource SessionContext ctx;
Beab bean = (Bean)ctx.lookup("bean");
```

где `Bean` — это бизнес-интерфейс или, в случае локального доступа без интерфейса, сам компонент.

На стороне сервера удаленный бизнес-интерфейс сессионного компонента маркируется аннотацией `@Remote`, локальный бизнес-интерфейс маркируется аннотацией `@Local`, в случае локального доступа без интерфейса класс компонента маркируется аннотацией `@LocalBean`. Классы сессионных компонентов маркируются аннотациями `@Stateful`, `@Stateless` или `@Singleton`. В случае если сессионный компонент не реализует бизнес-интерфейс, тогда по умолчанию он доступен локальным доступом без интерфейса.

MDB-компонент вызывается контейнером в результате поступления сообщения в пункт назначения (Java Message Service API — JSR 914) или конечную

точку (Java EE Connector Architecture 1.6 — JSR 322), обслуживаемые MDB-компонентом. MDB-компонент определяется для единственного типа сообщений в соответствии с интерфейсом слушателя сообщений, который он реализует.

Java Message Service (JMS) — это сервис сообщений, реализованный платформой Java EE, определяет две модели системы сообщений. Модель от точки к точке (point-to-point) предполагает, что клиент посылает сообщение в очередь — создаваемый администратором сервера объект `javax.jms.Queue` (пункт назначения). После этого сервер извлекает сообщение из очереди и доставляет его MDB-объекту, который затем обрабатывает сообщение. Модель публикации/подписки (pub/sub) предполагает, что клиент публикует сообщение, используя администрируемый объект `javax.jms.Topic` (пункт назначения). После этого сервер рассылает сообщение MDB-компонентам, подписавшимся на получение сообщений.

На стороне клиента администрируемые объекты `ConnectionFactory`, `Topic` и `Queue` вводятся с помощью аннотации `@Resource`. Администрируемый объект `javax.jms. ConnectionFactory` необходим для установления соединения с сервисом сообщений JMS (метод `createConnection`, возвращающий объект `javax.jms.Connection`). Затем методом `createSession()` объекта `Connection` создается объект `javax.jms.Session` — контекст для производства и потребления сообщений. Метод `createProducer()` объекта `Session` создает объект `javax.jms.MessageProducer`, позволяющий отправить сообщение в пункт назначения методом `send()`.

На стороне сервера класс MDB-компонента маркируется аннотацией `@MessageDriven` и реализует интерфейс `javax.jms.MessageListener` для асинхронного получения JMS-сообщений. Атрибут `mappedName` аннотации `@MessageDriven` связывает MDB-компонент с пунктом назначения.

Спецификация J2EE Connector Architecture описывает архитектуру соединения между серверами и информационными системами EIS. Такое соединение обеспечивает адаптер ресурсов, который встраивается в сервер приложений. Спецификация JCA, начиная с версии 1.5, определяет, что адаптер ресурсов может асинхронно доставлять сообщения в конечную точку. Таким образом, данная архитектура позволяет различным сервисам сообщений (Java Message Service (JMS), Java API for XML Messaging (JAXM) и др.) встраиваться в сервер приложений через адаптер ресурсов. При этом MDB-компонент должен реализовывать специфический для данных сообщений интерфейс.

Компонент Entity представляет таблицу реляционной базы данных. Экземпляр компонента представляет строку таблицы. Класс `Entity` маркируется аннотацией `@Entity`. Аннотация `@PersistenceContext` используется для ввода объекта `javax.persistence.EntityManager`, который управляет набором ком-

понентов Entity, при этом заботу о транзакциях берет на себя контейнер. Набор компонентов Entity, управляемый объектом `EntityManager`, составляет единицу сохраняемости (persistence unit). Единица сохраняемости описывается конфигурационным файлом persistence.xml. Объект `EntityManager` используется для создания и удаления экземпляров Entity, для поиска экземпляров Entity по их первичному ключу, для создания различного рода запросов.

Аннотации технологии EJB:

- пакет `javax.ejb` (JSR 318):
  - `@AccessTimeout` указывает время ожидания для параллельного доступа к компоненту Singleton. Атрибуты: `value` и `unit` (по умолчанию `java.util.concurrent.TimeUnit.MILLISECONDS`);
  - `@MessageDriven` маркирует класс компонента, управляемого сообщениями. Атрибуты: `name` — имя компонента; `messageListenerInterface` — реализуемый компонентом интерфейс; `activationConfig` — с помощью аннотации `@ActivationConfigProperty` (атрибуты `propertyName` и `propertyValue`) указывает свойства конфигурации компонента: способ подтверждения получения сообщений, селектор сообщений, тип пункта назначения и др.; `mappedName` — JNDI-имя пункта назначения для компонента;
  - `@AfterBegin` маркирует метод компонента Stateful, вызываемый при начале новой транзакции;
  - `@AfterCompletion AfterBegin` маркирует метод компонента Stateful, вызываемый при завершении транзакции;
  - `@BeforeCompletion` маркирует метод компонента Stateful, вызываемый перед завершением транзакции;
  - `@ApplicationException` маркирует класс исключения, которое сообщается клиенту напрямую. Атрибуты: `rollback` — если `true`, тогда транзакция откатывается при возникновении исключения, по умолчанию `false`; `inherited` — если `true` (по умолчанию), тогда аннотация применяется ко всем подклассам промаркированного класса;
  - `@Asynchronous` маркирует выполняемые асинхронно методы;
  - `@ConcurrencyManagement` для компонента Singleton указывает тип управления параллельным доступом к компоненту. Значение атрибута `value` по умолчанию — `javax.ejb.ConcurrencyManagementType.CONTAINER`;
  - `@DependsOn` — с помощью атрибута `value` указывает имена компонентов Singleton, чья инициализация должна пройти перед инициализацией данного компонента Singleton;

- `@EJB` указывает ссылку на EJB-компонент. Атрибуты: `name` — имя ссылки; `description` — описание ссылки; `beanName` — имя компонента; `beanInterface` — тип интерфейса компонента; `mappedName` — специфическое имя компонента;
- `@EJBs` — контейнер для аннотаций `@EJB`;
- `@Init` указывает метод сессионного компонента, который соответствует методу `create()` интерфейса `Home` или `Local Home` (EJB 2). Атрибут `value` — имя метода;
- `@Local` указывает с помощью атрибута `value` или маркирует локальный бизнес-интерфейс сессионного компонента;
- `@LocalBean` маркирует сессионный компонент с доступом без интерфейса;
- `@LocalHome` с помощью атрибута `value` объявляет интерфейс `Local Home` для сессионного компонента (EJB 2);
- `@Lock` для компонента Singleton с типом управления параллельным доступом к компоненту `javax.ejb.ConcurrencyManagementType.CONTAINER` связывает метод компонента с типом блокировки. Значение `javax.ejb.LockType.WRITE` (по умолчанию) атрибута `value` означает, что параллельные вызовы метода не разрешены, пока первоначальный вызов метода не будет завершен. Значение `javax.ejb.LockType.READ` атрибута `value` означает, что любое количество параллельных вызовов метода разрешено;
- `@PostActivate` маркирует метод, вызываемый после активации Stateful-компонента;
- `@PrePassivate` маркирует метод, вызываемый перед переводом Stateful-компонента в пассивное состояние;
- `@Remote` указывает с помощью атрибута `value` или маркирует удаленный бизнес-интерфейс сессионного компонента;
- `@RemoteHome` с помощью атрибута `value` объявляет интерфейс `Remote Home` для сессионного компонента (EJB 2);
- `@Remove` маркирует бизнес-метод Stateful-компонента и указывает, что компонент удаляется контейнером после выполнения промаркированного метода. Если значение атрибута `retainIfException` равно `true`, тогда компонент не удаляется в случае возникновения исключительной ситуации при выполнении метода; по умолчанию `false`;
- `@Schedule` устанавливает таймер для вызова метода. Атрибуты: `second`, `minute`, `hour`, `dayOfMonth`, `month`, `dayOfWeek`, `year`, `timezone`, `info`, `persistent` (`true`/`false` — сохранение таймера);

- `@Schedules` — контейнер для аннотаций `@Schedule`;
- `@Singleton` маркирует класс Singleton-компонента. Атрибуты: `name` — имя компонента; `mappedName` — специфическое имя компонента; `description` — описание компонента;
- `@Startup` для Singleton-компонента указывает контейнеру на необходимость инициализации компонента во время запуска приложения;
- `@Stateful` маркирует класс Stateful-компонента;
- `@StatefulTimeout` для Stateful-компонента указывает время ожидания клиентского вызова, по истечении которого компонент будет удален контейнером. Атрибуты: `value` (количество времени) и `unit` (единица времени, по умолчанию `java.util.concurrent.TimeUnit.MINUTES`);
- `@Stateless` маркирует класс Stateless-компонента. Атрибуты: `name` — имя компонента; `mappedName` — специфическое имя компонента; `description` — описание компонента;
- `@Timeout` маркирует метод Stateless- или Message-Driven-компонента, вызываемый по истечении времени таймера `javax.ejb.Timer`;
- `@TransactionAttribute` устанавливает атрибуты транзакции для методов сессионных или управляемых сообщениями компонентов. Атрибут `value` может принимать значения `MANDATORY`, `REQUIRED` (по умолчанию), `REQUIRES_NEW`, `SUPPORTS`, `NOT_SUPPORTED`, `NEVER`;
- `@TransactionManagement` для сессионных или управляемых сообщениями компонентов указывает тип управления транзакциями. Атрибут `value` может принимать значения `CONTAINER` (по умолчанию) или `BEAN`;

❒ пакет `javax.annotation` (JSR 250):

- `@Resource` указывает ссылку на ресурсы, необходимые приложению. Атрибуты: `name` — JNDI-имя ресурса; `type` — Java-тип ресурса; `authenticationType` — тип идентификации ресурса, по умолчанию `CONTAINER`; `shareable` — совместно используемый ресурс или нет, по умолчанию `true`; `mappedName` — специфическое имя ресурса; `description` — описание ресурса;
- `@Resources` — контейнер для аннотаций `@Resource`;
- `@PostConstruct` маркирует метод, вызываемый после ввода ресурсов для выполнения инициализации;
- `@PreDestroy` маркирует метод, вызываемый перед тем, как экземпляр будет удален контейнером для освобождения ресурсов;

- `@ManagedBean` маркирует класс управляемого JavaBeans-компонента. Атрибут `value` указывает имя компонента;
- `@Named` с помощью атрибута `value` указывает имя компонента;
- `@NonBinding` исключает связь;

❐ пакет `javax.annotation.security` (JSR 250):

- `@DeclareRoles` маркирует класс для объявления роли с помощью атрибута `value`;
- `@DenyAll` указывает отсутствие ролей, разрешенных для вызова методов;
- `@PermitAll` указывает полный доступ;
- `@RolesAllowed` указывает список ролей, разрешенных для доступа, с помощью атрибута `value`;
- `@RunAs` указывает выполнение под определенной атрибутом `value` ролью;

❐ пакет `javax.interceptor` (JSR 318):

- `@AroundInvoke` маркирует метод, являющийся методом-перехватчиком для бизнес-методов сессионных компонентов и методов интерфейса-слушателя компонентов, управляемых сообщениями;
- `@AroundTimeout` маркирует метод, являющийся методом-перехватчиком для методов, промаркированных аннотацией `@Timeout`;
- `@Interceptor` маркирует класс-перехватчик;
- `@Interceptors` указывает класс-перехватчик для класса или метода;
- `@InterceptorBindingType` — метааннотация для создания специфических перехватчиков;
- `@ExcludeClassInterceptors` маркирует бизнес-метод и исключает перехватчики, объявленные на уровне класса;
- `@ExcludeDefaultInterceptors` исключает перехватчики по умолчанию;

❐ пакет `javax.persistence` (JSR 317):

- `@Entity` маркирует класс компонента Entity. Атрибут `name` указывает имя компонента, используемое в запросах;
- `@Access` определяет тип доступа к сохраняемому состоянию компонента Entity. Значение `FIELD` атрибута `value` означает, что столбцам таблицы базы данных соответствуют поля класса компонента Entity. Значение `PROPERTY` атрибута `value` означает, что столбцам таблицы базы данных соответствуют JavaBeans-свойства класса компонента Entity;

- `@AssociationOverride` применяется к классу компонента Entity, который расширяет связанный суперкласс, и переопределяет связи один-к-одному (one-to-one) или один-ко-многим (many-to-one), определенные в суперклассе. Атрибуты: `name` — имя связуемого свойства или поля; `joinColumns` — связанный столбец;
- `@AssociationOverrides` — контейнер для аннотаций `@AssociationOverride`;
- `@Basic` определяет простое соответствие свойства или поля столбцу таблицы базы данных. Использование аннотации необязательно для простых Java-типов. Атрибуты: `fetch` — тип загрузки значения поля или свойства, по умолчанию `javax.persistence.FetchType.EAGER` (быстрая загрузка); `optional` — если `true` (по умолчанию), тогда значение поля или свойства может быть нулевым;
- `@AttributeOverride` переопределяет соответствие `@Basic`, определенное в суперклассе или встроенном классе. Атрибуты: `name` — имя свойства или поля; `column` — имя столбца таблицы;
- `@AttributeOverrides` — контейнер для аннотаций `@AttributeOverride`;
- `@Column` определяет соответствие столбца таблицы полю или свойству. Атрибуты: `name` — имя столбца, по умолчанию имя поля или свойства; `unique` — если `true`, тогда столбец является уникальным ключом, по умолчанию `false`; `nullable` — если `true` (по умолчанию), тогда столбец может принимать нулевые значения; `insertable` — если `true` (по умолчанию), тогда столбец включается в SQL-выражение `INSERT`; `updatable` — если `true` (по умолчанию), тогда столбец включается в SQL-выражение `UPDATE`; `columnDefinition` — SQL-фрагмент в SQL-выражении `CREATE TABLE`; `table` — имя таблицы; `length` — длина строк столбца, по умолчанию 255; `precision` — количество цифр в числе столбца, по умолчанию 0; `scale` — количество цифр после запятой в числе столбца, по умолчанию 0;
- `@SqlResultSetMapping` устанавливает соответствие результата SQL-запроса классам компонентов Entity и скалярным переменным. Атрибуты: `name` — имя результата, указанное в методе `createNativeQuery()` объекта `EntityManager`, формирующем SQL-запрос; `entities` — соответствие результата классу компонента Entity; `columns` — соответствие результата скалярной переменной;
- `@SqlResultSetMappings` — контейнер для аннотаций `@SqlResultSetMapping`;
- `@ColumnResult` — в атрибуте `columns` аннотации `@SqlResultSetMapping` указывает соответствие результата SQL-запроса столбцу в случае, ко-

гда выборка идет по одному полю/свойству компонента. Атрибут `name` — имя столбца в выражении `SELECT`;

- `@EntityResult` — в атрибуте `entities` аннотации `@SqlResultSetMapping` указывает соответствие результата SQL-запроса классу компонента Entity. Атрибуты: `entityClass` — класс результата; `fields` — соответствие столбцов, указанных в выражении `SELECT`, полям/свойствам класса компонента; `discriminatorColumn` — имя столбца в списке выражения `SELECT`, который используется для определения типа экземпляра Entity;
- `@FieldResult` — в атрибуте `fields` аннотации `@EntityResult` определяет соответствие столбцов, указанных в выражении `SELECT`, полям/свойствам класса компонента. Атрибуты: `name` — имя поля/свойства класса компонента; `column` — имя столбца в выражении `SELECT`;
- `@Table` определяет первичную таблицу для компонента Entity. Компонент Entity может представлять как одну таблицу базы данных, так и несколько связанных таблиц. Первичная или родительская таблица — это таблица, содержащая первичный ключ, по которому идет связь с другими таблицами. Атрибуты: `name` — имя таблицы; `catalog` — каталог таблицы; `schema` — схема базы данных, к которой относится таблица; `uniqueConstraints` — столбцы с уникальными значениями;
- `@UniqueConstraint` — в атрибуте `uniqueConstraints` аннотации `@Table` указывает столбцы с уникальными значениями, используя атрибут `columnNames`;
- `@SecondaryTable` указывает связанную или вторичную таблицу для класса компонента Entity. Атрибуты: `name` — имя таблицы; `catalog` — каталог таблицы; `schema` — схема базы данных, к которой относится таблица; `uniqueConstraints` — столбцы с уникальными значениями; `pkJoinColumns` — столбец, связанный с первичным ключом родительской таблицы;
- `@SecondaryTables` — контейнер для аннотаций `@SecondaryTable`;
- `@PrimaryKeyJoinColumn` — столбец первичного ключа, который используется как внешний ключ для связи с другой таблицей. Атрибуты: `name` — имя столбца первичного ключа данной таблицы; `referencedColumnName` — имя столбца первичного ключа таблицы, с которой данная таблица связывается; `columnDefinition` — SQL-фрагмент в SQL-выражении `CREATE TABLE`;
- `@PrimaryKeyJoinColumns` — контейнер для аннотаций `@PrimaryKeyJoinColumn`;

- `@Inheritance` — метод наследования, используемый в иерархии классов Entity. Аннотация маркирует корневой класс иерархии. Атрибут `strategy` принимает значения: `avax.persistence.InheritanceType.SINGLE_TABLE` (по умолчанию) — дерево компонентов привязано к одной таблице; `avax.persistence.InheritanceType.TABLE_PER_CLASS` — каждый дочерний класс связан со своей таблицей и содержит столбцы для атрибутов родительского класса; `avax.persistence.InheritanceType.JOINED` — дочерний класс представляет таблицу, которая связана посредством внешнего ключа с таблицей, представленной родительским классом;

- `@DiscriminatorColumn` — для наследования типа `SINGLE_TABLE` и `JOINED` определяет столбец, который указывает, к какому классу относится ряд в таблице. Аннотация маркирует корневой класс иерархии. Атрибуты: `name` — имя столбца; `discriminatorType` — тип значений столбца (`javax.persistence.DiscriminatorType.STRING` (по умолчанию), `javax.persistence.DiscriminatorType.CHAR`, `javax.persistence.DiscriminatorType.INTEGER`); `columnDefinition` — SQL-фрагмент в SQL-выражении `CREATE TABLE`; `length` — длина строки столбца;

- `@DiscriminatorValue` указывает значение столбца `@DiscriminatorColumn` для конкретного класса `Entity`, используя атрибут `value`;

- `@Embeddable` маркирует встраиваемый класс;

- `@Embedded` указывает поля/свойства класса `Entity`, чьи значения определяются экземпляром встроенного класса;

- `@EmbeddedId` маркирует поле/свойство, указывая, что составной первичный ключ представлен встроенным классом;

- `@EntityListeners` указывает класс-слушатель для класса `Entity` или его суперкласса, используя атрибут `value`;

- `@PostLoad` маркирует метод-обработчик событий, вызываемый после того, как экземпляр Entity загрузится из базы данных или будет обновлен. Аннотация маркирует метод класса `Entity`, метод суперкласса или метод класса-слушателя;

- `@PostPersist` маркирует метод-обработчик событий, вызываемый после того, как экземпляр Entity будет сохранен. Аннотация маркирует метод класса `Entity`, метод суперкласса или метод класса-слушателя;

- `@PostRemove` маркирует метод-обработчик событий, вызываемый после того, как экземпляр Entity будет удален. Аннотация маркирует метод класса `Entity`, метод суперкласса или метод класса-слушателя;

- `@PostUpdate` маркирует метод-обработчик событий, вызываемый после обновления данных базы данных компонента Entity. Аннотация маркирует метод класса `Entity`, метод суперкласса или метод класса-слушателя;
- `@PreUpdate` маркирует метод-обработчик событий, вызываемый перед обновлением данных базы данных компонента Entity. Аннотация маркирует метод класса `Entity`, метод суперкласса или метод класса-слушателя;
- `@PrePersist` маркирует метод-обработчик событий, вызываемый перед тем, как экземпляр Entity будет сохранен. Аннотация маркирует метод класса `Entity`, метод суперкласса или метод класса-слушателя;
- `@PreRemove` маркирует метод-обработчик событий, вызываемый перед тем, как экземпляр Entity будет удален. Аннотация маркирует метод класса `Entity`, метод суперкласса или метод класса-слушателя;
- `@Enumerated` указывает, что поле/свойство должно храниться как перечисляемый тип. Атрибут `value` определяет хранение в виде строк (`javax.persistence.EnumType.STRING`) или целых чисел (`javax.persistence.EnumType.ORDINAL` — по умолчанию);
- `@ExcludeDefaultListeners` исключает вызов слушателей по умолчанию;
- `@ExcludeSuperclassListeners` исключает вызов слушателей суперкласса;
- `@Id` маркирует поле/свойство как первичный ключ;
- `@SequenceGenerator` определяет генератор первичного ключа. Атрибуты: `name` — идентификатор генератора; `sequenceName` — имя объекта базы данных, который будет выдавать значения для первичного ключа; `initialValue` — первоначальное значение, с которого будет стартовать объект базы данных при генерации значений, по умолчанию 1; `allocationSize` — шаг генерации, по умолчанию 50;
- `@TableGenerator` определяет генератор первичного ключа, создающий специальную таблицу в базе данных, хранящую последовательность значений для первичного ключа. Атрибуты: `name` — идентификатор генератора; `table` — имя таблицы, хранящей значения первичного ключа; `catalog` — каталог таблицы; `schema` — схема базы данных, к которой относится таблица; `pkColumnName` — имя столбца первичного ключа; `valueColumnName` — имя столбца, хранящего последнее сгенерированное значение; `pkColumnValue` — значение первичного ключа, которое разделяет один сгенерированный набор от другого; `initialValue` — первоначальное значение набора; `allocationSize` — шаг генерации;

`uniqueConstraints` — ограничение по уникальности значений столбцов таблицы (`@UniqueConstraint`);

- `@GeneratedValue` определяет метод генерации значений первичного ключа. Атрибуты: `generator` — идентификатор генератора SequenceGenerator или TableGenerator; `strategy` — метод генерации — `javax.persistence.GenerationType.TABLE` (значения присваиваются первичному ключу с использованием таблицы, хранящей последовательность значений), `javax.persistence.GenerationType.SEQUENCE` (значения присваиваются первичному ключу с использованием столбца, хранящего последовательность значений), `javax.persistence.GenerationType.IDENTITY` (значения присваиваются первичному ключу на основе столбца идентификаторов), по умолчанию `javax.persistence.GenerationType.AUTO` (выбор метода генерации производится автоматически);
- `@IdClass` с помощью атрибута `value` указывает класс, представляющий составной первичный ключ;
- `@JoinColumn` определяет соответствие столбца таблицы, который является внешним ключом, полю или свойству. Атрибуты: `name` — имя внешнего ключа; `referencedColumnName` — имя связанного с внешним ключом столбца; `unique` — если `true`, тогда значения столбца уникальны, по умолчанию `false`; `nullable` — если `true` (по умолчанию), тогда столбец может принимать нулевые значения; `insertable` — если `true` (по умолчанию), тогда столбец включается в SQL-выражение `INSERT`; `updatable` — если `true` (по умолчанию), тогда столбец включается в SQL-выражение `UPDATE`; `columnDefinition` — SQL-фрагмент в SQL-выражении `CREATE TABLE`; `table` — имя таблицы, содержащей столбец;
- `@JoinColumns` — контейнер для аннотаций `@JoinColumn`;
- `@JoinTable` определяет таблицу, содержащую информацию о связях между таблицами. Аннотация используется для компонента Entity, связанного с другими компонентами по типу многие-ко-многим или один-ко-многим. Атрибуты: `name` — имя таблицы; `catalog` — каталог таблицы; `schema` — схема базы данных, к которой относится таблица; `joinColumns` — с помощью аннотации `@JoinColumn` указывает внешний ключ данного компонента; `inverseJoinColumns` — с помощью аннотации `@JoinColumn` указывает внешний ключ связанного компонента; `uniqueConstraints` — ограничение по уникальности значений столбцов таблицы (`@UniqueConstraint`);
- `@Lob` указывает, что поле/свойство хранится как большой объект — специальный тип данных, используемый для хранения изображений, аудио- и видеоданных;

- `@ManyToMany` для поля/свойства указывает связь типа многие-ко-многим. Атрибуты: `targetEntity` — класс связанного компонента; `cascade` указывает каскадное исполнение операций — `javax.persistence.CascadeType.ALL` (всех операций), `javax.persistence.CascadeType.PERSIST` (операций сохранения), `javax.persistence.CascadeType.MERGE` (операции соединения), `javax.persistence.CascadeType.REMOVE` (операции удаления), `javax.persistence.CascadeType.REFRESH` (операции обновления); `fetch` — тип загрузки связи — `javax.persistence.FetchType.LAZY` (по умолчанию) и `javax.persistence.FetchType.EAGER`; `mappedBy` — поле, определяющее связь;
- `@ManyToOne` для поля/свойства указывает связь типа многие-к-одному. Атрибуты: `targetEntity` — класс связанного компонента; `cascade` — указывает каскадное исполнение операций; `fetch` — тип загрузки связи; `optional` — если `true` (по умолчанию), тогда связь необязательна;
- `@OneToMany` для поля/свойства указывает связь типа один-ко-многим. Атрибуты: `targetEntity` — класс связанного компонента; `cascade` указывает каскадное исполнение операций; `fetch` — тип загрузки связи; `mappedBy` — поле, определяющее связь;
- `@OneToOne` для поля/свойства указывает связь типа один-к-одному. Атрибуты: `targetEntity` — класс связанного компонента; `cascade` указывает каскадное исполнение операций; `fetch` — тип загрузки связи; `mappedBy` — поле, определяющее связь; `optional` — если `true` (по умолчанию), тогда связь необязательна;
- `@MapKey` для связи, которая описывается хэш-таблицей, указывает поле/свойство, определяющее ключ хэш-таблицы, используя атрибут `name`;
- `@MappedSuperclass` маркирует класс суперкласса компонентов;
- `@NamedNativeQuery` определяет именованный SQL-запрос. Атрибуты: `name` — идентификатор запроса; `query` — сам запрос; `hints` — специфические подсказки запроса; `resultClass` — класс результата запроса; `resultSetMapping` — имя `SqlResultSetMapping`;
- `@NamedNativeQueries` — контейнер для аннотаций `@NamedNativeQuery`;
- `@NamedQuery` определяет именованный запрос Java Persistence Query Language. Атрибуты: `name` — идентификатор запроса; `query` — сам запрос; `hints` — специфические подсказки запроса; `lockMode` — тип блокировки `javax.persistence.LockModeType.READ` или `javax.persistence.LockModeType.WRITE`;
- `@NamedQueries` — контейнер для аннотаций `@NamedQuery`;

- `@OrderBy` определяет сортировку элементов: восходящую `ASC` и нисходящую `DESC`, используя атрибут `value`;
- `@PersistenceContext` указывает ссылку на объект `EntityManager`. Атрибуты: `name` — JNDI-имя, используемое при получении объекта `EntityManager` методом `lookup` объекта `javax.ejb.SessionContext`; `unitName` — имя единицы сохранения (persistence unit); `type` — тип вызова объекта `EntityManager`: по умолчанию `javax.persistence.PersistenceContextType.TRANSACTION` (вызов производится только в транзакции) и `javax.persistence.PersistenceContextType.EXTENDED` (вызов производится без учета транзакций); `properties` — свойства для контейнера;
- `@PersistenceContexts` — контейнер для аннотаций `@PersistenceContext`;
- `@PersistenceProperty` используется в атрибуте `properties` аннотации `@PersistenceContext` для указания свойств, учитываемых при создании объекта `EntityManager`. Например, это могут быть параметры соединения с базой данных. Атрибуты — `name` и `value`;
- `@PersistenceUnit` указывает ссылку на объект `javax.persistence.EntityManagerFactory`, используемый для получения объекта `EntityManager`, управляемого приложением, а не контейнером. Атрибуты: `name` — JNDI-имя, используемое при получении объекта `EntityManagerFactory` методом `lookup()` объекта `javax.ejb.SessionContext`; `unitName` — имя единицы сохранения (persistence unit);
- `@PersistenceUnits` — контейнер для аннотаций `@PersistenceUnit`;
- `@Temporal` маркирует поля/свойства типа `java.util.Date` и `java.util.Calendar`, используя атрибут `value`;
- `@Transient` маркирует несохраняемые поля/свойства;
- `@Version` маркирует поле/свойство, представляющее версию компонента;
- `@QueryHint` используется в атрибуте `hints` аннотаций `@NamedNativeQuery` и `@NamedQuery`. Атрибуты — `name` и `value`;

❐ пакет `javax.context` (JSR 316):

- `@ApplicationScoped` указывает, что экземпляр Bean-компонента имеет область видимости приложения;
- `@ConversationScoped` указывает, что экземпляр Bean-компонента имеет область видимости сессии приложения (например, жизненного цикла JSF);
- `@Dependent` указывает, что экземпляр Bean-компонента имеет область видимости, определяемую введением ресурса и его удалением;

- `@RequestScoped` указывает, что экземпляр Bean-компонента имеет область видимости обработки клиентского HTTP-запроса;
- `@ScopeType` указывает тип аннотации `ScopeType`;
- `@SessionScoped` указывает, что экземпляр Bean-компонента имеет область видимости HTTP-сессии.

Технология JavaMail

Технология JavaMail 1.4 (JSR 919) предоставляет независящую от протоколов платформу для создания приложений, реализующих задачи отправки и получения сообщений по электронной почте.

Технология JavaMail использует технологию JavaBeans Activation Framework (JAF) 1.1 (JSR 925) для взаимодействия с данными сообщения, которое осуществляется с помощью компонентов JavaBeans, использующих библиотеку JAF API.

Для создания сессии отправки/получения сообщений используется класс `javax.mail.Session`. Объект Session содержит информацию о почтовом сервере, логине и пароле пользователя и др.

Для создания сообщения, предназначенного к отправке, используется класс `javax.mail.internet.MimeMessage`, методы которого позволяют создать содержимое сообщения.

Чтобы послать сообщение адресату, необходим класс `javax.mail.internet.InternetAddress`, представляющий интернет-адрес электронной почты. Методы класса `MimeMessage` позволяют связать интернет-адрес с сообщением.

Класс `javax.mail.Transport` отвечает за отправку сообщения.

За получение сообщения отвечает класс `javax.mail.Store`, метод которого `connect()` осуществляет связь с почтовым сервером, а метод `getFolder()` возвращает объект `javax.mail.Folder`, представляющий папку, содержащую сообщения. Методы `open()` и `close()` класса `Folder` открывают и закрывают папку, а метод `getMessages()` извлекает сообщение из папки.

Объект `javax.mail.Message`, представляющий сообщения, взаимодействует со своим содержимым через промежуточный уровень — JavaBeans Activation Framework (JAF). Такое разделение позволяет управлять любым контентом. Интерфейс `Part`, реализуемый классом `Message`, с помощью метода `getDataHandler()` возвращает JAF-объект `javax.activation.DataHandler`, который дает доступ к таким операциям с контентом, как `view`, `edit` и `print`.

Технологии Hibernate, JDO, Struts, Echo, Spring Framework, Portlet, Google Web Toolkit

Технология Hibernate

Технология Hibernate обеспечивает хранение данных приложения в реляционных базах данных путем отображения обычных объектов Java™ (POJO) на таблицы реляционных СУБД (ORM) с поддержкой транзакций и различных форм запросов к базам данных.

Технология позволяет разрабатывать классы сохраняемых компонентов, используя ассоциации, наследование, полиморфизм, композицию и коллекции.

С Hibernate возможно создавать запросы к базам данных с помощью обычного языка SQL, его Hibernate-расширения Hibernate Query Language (HQL), а также с использованием Criteria и Example API. Технология Hibernate интегрирована с системой JBoss Enterprise Middleware System (JEMS).

Технология Hibernate представлена библиотекой org.hibernate, которую можно скачать по адресу **https://www.hibernate.org/**.

Ранее рассмотренная нами спецификация Java Persistence API (JPA), входящая в спецификацию EJB 3.0 (JSR 220), создавалась на основе технологий Hibernate, TopLink, Java Data Objects и спецификации Container Managed Persistence (EJB-CMP) 2.1.

В технологии Hibernate данные приложения представлены объектами обычных классов Java — объектами POJO (Plain Old Java Objects, простые старые Java-объекты).

Для отображения объектов этих классов в записи соответствующих таблиц базы данных используются конфигурационные файлы Hibernate *.hbm.xml. Соединение с базой данных обычно описывается в конфигурационном файле hibernate.cfg.xml.

Ключевые классы и интерфейсы библиотеки Hibernate:

- интерфейс `org.hibernate.Session` — главный интерфейс, реализующий сервис Hibernate. Содержит методы сохраняющие, удаляющие и обновляющие данные базы данных;
- интерфейс `org.hibernate.SessionFactory` отвечает за создание объектов Session;
- класс `org.hibernate.cfg.Configuration` позволяет определить свойства, используемые при создании объекта `SessionFactory`;
- интерфейс `org.hibernate.Transaction` обеспечивает управление транзакциями;

- интерфейс `org.hibernate.transaction.TransactionFactory` отвечает за создание объекта `Transaction`.

Технология Java Data Objects

Технология Java Data Objects (JDO) — это технология сохранения Java-объектов.

В отличие от технологии JPA, которая ориентирована на взаимодействие с реляционными СУБД, JDO обеспечивает хранение данных в любом хранилище, имеющем соответствующий адаптер. Адаптер, представляющий собой реализацию интерфейса `javax.jdo.PersistenceManager`, предназначен для синхронизации набора сохраняемых объектов Java с хранилищем данных.

Технология JDO является обобщением для произвольных хранилищ данных, требуя реализации специализированного адаптера для каждого вида хранилищ, в то время как для JPA достаточно наличия драйвера JDBC.

В технологии JDO для отображения объектов Java в записи соответствующих таблиц базы данных также используются конфигурационные файлы, только с расширением jdo.

Технология JDO позволяет создавать запросы к данным без использования SQL с помощью языка запросов Java Data Objects Query Language (JDOQL).

Реализация технологии JDO доступна на сайте **http://db.apache.org/jdo/impls.html**.

Основу спецификации JDO составляют следующие классы и интерфейсы:

- класс `javax.jdo.JDOHelper` позволяет получить объект `PersistenceManagerFactory` различными методами;
- интерфейс `javax.jdo.PersistenceManagerFactory` предназначен для получения объектов `PersistenceManager`;
- интерфейс `javax.jdo.PersistenceManager` является главным интерфейсом JDO и содержит методы для управления сохраняемыми объектами Java;
- интерфейс `javax.jdo.Transaction` обеспечивает управление транзакциями;
- интерфейс `javax.jdo.Query` позволяет создавать запросы к хранилищам данных, используя как синтаксис SQL, так и JDOQL.

Технология Struts

Технология Apache Struts представляет собой платформу для создания Java Web-приложений, способных формировать динамические ответы клиенту, взаимодействовать с базами данных и с бизнес-компонентами.

Платформа Apache Struts доступна по адресу **http://struts.apache.org/**.

Обычные Web-приложения, основанные на JavaServer Pages, смешивают доступ к данным, обработку данных и представление данных. Клиент заполняет данные формы, которые отправляются серверу. На сервере данные обрабатываются сервлетом или JSP-страницей, при этом они могут взаимодействовать с базами данных, и после обработки данных сервлет или JSP-страница формируют ответ клиенту. В случае больших проектов такое смешение логики затрудняет их разработку и поддержку.

Технология Struts решает эту проблему путем реализации архитектуры MVC (Model — View — Controller, Модель — Вид — Контроллер). *Модель* отвечает за доступ и обработку данных, *Вид* — за отображение данных, а *Контроллер* — за взаимодействие между Моделью и Видом. Кроме того, Struts предоставляет собственную библиотеку тегов для создания JSP-страниц, обеспечивает поддержку интернационализации, проверку данных как на стороне клиента, так и на стороне сервера, а также механизм создания шаблонов.

Последняя версия Struts 2 создана слиянием с платформой OpenSymphony Web Works Framework и основана на Servlet API 2.4, JSP API 2.0 и Java 5.

В технологии Struts 2 Модель реализована действиями (`Action`), Вид — результатами (`Result`) и Контроллер — `FilterDispatcher`.

В Struts 2 результаты реализуются с помощью JSP-страниц или других технологий уровня представления.

Контроллер `FilterDispatcher` (класс фильтра, реализующий интерфейс `javax.servlet.Filter`) поставляется вместе с платформой и отвечает за получение запроса от клиента, который затем направляет объектам перехватчиков (Interceptors).

Объекты перехватчиков обеспечивают пред- и постобработку с обработкой исключений, загрузкой файлов с клиентской машины, извещением слушателей о событиях жизненного цикла, проверкой данных, вводом необходимых ресурсов и т. д. Платформа Struts 2 содержит набор стандартных перехватчиков. Контроллер `FilterDispatcher` подключается в дескрипторе развертывания приложения web.xml. Перехватчики регистрируются и описываются в конфигурационном файле struts.xml. После обработки перехватчиками запрос передается объекту `Action`.

Действия `Action` связываются с определенными запросами и результатами в конфигурационном файле struts.xml. Действие `Action` создается разработчиком приложения в виде класса POJO и отвечает за обработку данных. После выполнения действия `Action` генерируется результат, связанный с действием в конфигурационном файле.

Результат `Result` создается разработчиком приложения с использованием библиотеки пользовательских тегов Struts и языка выражений Object-Graph

Navigation Language (OGNL). В Struts 2 данные приложения хранятся в объекте `ValueStack`. Язык выражений OGNL позволяет извлекать данные из объекта `ValueStack` и отображать их в результате.

Технология Echo

Технология Echo — это платформа для разработки Web-приложений с богатым графическим интерфейсом пользователя.

Для создания графического интерфейса пользователя платформа Echo предоставляет собственную библиотеку графических компонентов.

Адрес сайта технологии Echo — **http://nextapp.com/.**

Платформа Echo состоит из двух независимых частей — клиентской JavaScript-платформы и серверной Java-платформы. Клиентские приложения на основе клиентской JavaScript-платформы разрабатываются с использованием кода JavaScript, который выполняется в клиентском браузере. Для работы серверного приложения на основе серверной Java-платформы Echo необходим контейнер Java Servlet. Серверное приложение создается на языке Java и взаимодействует с "тонким" клиентом, который синхронизирует состояние своих компонентов с серверным приложением и посылает генерируемые пользователем события серверу.

Для создания серверного приложения Echo необходимо наличие двух классов. Первый класс должен расширять абстрактный класс `ApplicationInstance`, содержащий метод `init()`, который инициализирует состояние графического интерфейса пользователя. Метод `init()` класса приложения должен создавать и возвращать объект `Window`, представляющий главное окно приложения. Второй класс приложения расширяет сервлет WebContainerServlet, отвечающий за взаимодействие с клиентским приложением.

При разработке клиентского приложения необходимо создать класс, производный от класса `Echo.Application`, представляющего состояние экземпляра приложения и имеющего свойство `rootComponent`, которое отвечает за отображение области приложения. Для запуска клиентского приложения в браузере нужно объявить функцию `init()`, которая должна создавать экземпляр класса, производного от `Echo.Application`, и инсталлировать его в классе `Echo.FreeClient`, после чего вызывать метод `init()` для запуска клиента.

Технология Spring Framework

Технология Spring Framework представляет собой модульную платформу для создания приложений Java EE.

Адрес сайта технологии Spring — **http://www.springsource.org/.**

Модульность Spring Framework позволяет использовать только те части платформы, которые необходимы для разработки конкретного приложения.

Платформа Spring Framework состоит более чем из 20 модулей, которые сгруппированы в платформы:

- Core Container;
- Data Access/Integration;
- Web;
- Aspect Oriented Programming (AOP) и Instrumentation;
- Test.

Платформа Core Container

Платформа Core Container состоит из модулей Core, Beans, Context, Expression Language.

Модули Core и Beans составляют основу Spring Framework и обеспечивают сервис Inversion of Control (IoC) или Dependency Injection, объединяющий компоненты в готовое к работе приложение. Модуль Beans содержит интерфейс `org.springframework.beans.factory.BeanFactory`, который дает возможность конфигурировать объекты таким образом, что позволяет осуществлять поиск объектов по их именам и управлять связями между объектами.

Модуль Context основан на модулях Core и Beans и добавляет поддержку интернационализации, механизма обработки событий, загрузку ресурсов, создание контекстов выполнения, EJB, Java Management Extensions (JMX) и удаленное взаимодействие. Интерфейс `ApplicationContext` является ключевым для модуля Context.

Модуль Expression Language обеспечивает язык выражений, который представляет собой расширение языка EL спецификации JSP 2.1. Язык выражений Spring поддерживает доступ и установку свойств компонентов JavaBeans, вызов методов компонентов, доступ к коллекциям и массивам, управление списками, логические и арифметические операции, именованные переменные и поиск объектов по их именам с помощью IoC-контейнера.

Платформа Data Access/Integration

Платформа Data Access/Integration состоит из модулей JDBC, ORM, OXM, JMS и Transaction.

Модуль JDBC обеспечивает взаимодействие с системами управления реляционными базами данных и предоставляет API, облегчающее доступ к базам данных, включая управление исключительными ситуациями.

Модуль ORM (Object-Relational Mapping) — модуль отображения Java-объектов в таблицы реляционной базы данных с поддержкой технологий JPA, JDO, Hibernate и других продуктов ORM. Вместе с поддержкой большого набора продуктов ORM Spring обеспечивает управление сессиями, ресурсами, транзакциями, исключительными ситуациями, тестированием.

Модуль OXM (Object/XML Mapping) — модуль отображения Java-объектов на XML-документы с поддержкой технологий JAXB, Castor, XMLBeans, JiBX и XStream.

Модуль JMS обеспечивает сервис сообщений Java Messaging Service (JMS).

Модуль Transaction обеспечивает декларативное и программное управление транзакциями.

Платформа Web

Платформа Web состоит из модулей Web, Web-Servlet и Web-Portlet.

Модуль Web обеспечивает базовую интеграцию Web-компонентов.

Модуль Web-Servlet реализует модель MVC для Web-приложений. Spring-реализация MVC близка к технологии Struts.

Модуль Web-Portlet предназначен для интеграции технологии Portlet.

Платформы Aspect Oriented Programming (AOP) и Instrumentation

Модуль AOP реализует аспектно-ориентированное программирование для приложений Spring. Spring AOP обеспечивает сервисы Java EE для объектов POJO, включая декларативное управление транзакциями и определение методов-перехватчиков.

Модуль Aspect интегрирует Spring с технологией AspectJ.

Модуль Instrumentation обеспечивает поддержку инструментов и реализации загрузки classloader для различных серверов приложений.

Платформа Test

Модуль Test предназначен для тестирования компонентов Spring с помощью технологии JUnit или TestNG.

Технология Portlet

Технология Portlet описывается спецификациями Java Portlet API (JSR 168) и Java Portlet API 2.0 (JSR 286).

Портлет — это приложение, используемое в качестве подключаемого компонента пользовательского интерфейса, отображаемого в странице портала.

Портал — Web-приложение, обеспечивающее объединение содержимого из различных источников в Web-странице.

Портлет управляется специальным контейнером, который получает клиентские запросы от портала и с помощью портлетов обрабатывает их, генерируя в ответ динамический контент. Таким образом, применение технологии Portlet позволяет отображать пользователю Web-страницу, состоящую из областей, содержимое которых меняется без перезагрузки всей Web-страницы. Каждая такая область представляет собой окно, отображающее генерируемое портлетом содержимое и имеющее заголовок и кнопки, управляющие параметрами окна.

Содержимое, генерируемое портлетом, называется *фрагментом*. Как правило, пользователь взаимодействует с фрагментами с помощью ссылок или форм. При получении клиентского запроса портал определяет, содержит ли запрос действие, которое должно выполняться каким-либо портлетом, связанным со страницей портала. Если такое действие есть, портал адресует запрос портлет-контейнеру. После этого портлет-контейнер вызывает портлеты для обработки запроса. Далее сгенерированные фрагменты передаются порталу для объединения в странице портала, которая генерируется сервером портала.

Портлеты могут использовать сервлеты и JSP-страницы для генерации фрагментов. Портлет, сервлеты и JSP-страницы, связанные вместе, образуют портлет-приложение.

Портлет-контейнер является расширением контейнера сервлетов и обеспечивает интерфейс между порталом и портлетами, которые должны быть развернуты в портлет-контейнере.

Проект Apache Pluto (**http://portals.apache.org/pluto/**) — эталонная реализация спецификации Portlet. По сути, Pluto — это Web-приложение, развернутое в сервере Tomcat и выполняющее функции портала и портлет-контейнера. Существуют и другие решения порталов — Jetspeed, uPortal, Liferay и др.

Портлет представляет собой класс, реализующий интерфейс `javax.portlet.Portlet`.

Портлет-приложение упаковывается в WAR-файл и развертывается в портлет-контейнере. Каждое портлет-приложение, помимо дескриптора развертывания web.xml, имеет дескриптор развертывания portlet.xml, содержащий информацию о портлете.

Простейший портал может быть создан путем развертывания портлет-приложения в контейнере Pluto и регистрации его в конфигурационных фай-

лах pageregistry.xml и portletentityregistry.xml контейнера для добавления в страницу портала. При соответствующем URL-запросе к серверу Pluto будет отображаться Web-страница, содержащая окно созданного портлета.

Технология Google Web Toolkit

Технология Google Web Toolkit — это платформа для создания AJAX-приложений.

Технология GWT позволяет создавать AJAX-приложения на языке Java. Компилятор GWT переводит код Java в код JavaScript, легко создавая клиентскую часть приложения, работающую в клиентском браузере, при этом технология GWT обеспечивает поддержку широкого спектра браузеров.

GWT-компилятор имеет свои ограничения, в частности он не поддерживает динамическую загрузку классов, финализацию и сериализацию. При написании клиентской части приложения необходимо использовать такие Java-конструкции, которые могут быть переведены на JavaScript.

Для создания серверной части приложения GWT предлагает механизм Remote Procedure Call (RPC) удаленного вызова процедур для двухстороннего обмена объектами Java между клиентом и сервером. Кроме того, GWT позволяет осуществлять HTTP-запросы для получения JSON-данных с сервера. JSON (JavaScript Object Notation) — текстовый формат обмена данными, не зависящий от языка программирования.

Технология GWT содержит GWT Hosted Web Browser — встроенный браузер, который дает возможность отлаживать клиентскую часть приложения без преобразования в JavaScript.

Библиотека GWT Web UI снабжает разработчика широким ассортиментом классов и интерфейсов для создания богатого графического пользовательского интерфейса приложения.

Домашняя страничка технологии Google Web Toolkit находится по адресу **http://code.google.com/intl/ru/webtoolkit/**.

ГЛАВА 9

Пример приложения Java EE

В качестве примера приложения Java EE рассмотрим создание приложения уровня предприятия, решающее задачи взаимодействия между отделом продаж и складом предприятия.

> ***Примечание***
>
> Файлы с кодами приложения расположены на прилагаемом к книге компакт-диске в папке Часть_III\Глава_9\WebApplicationSaleStore.

Составные части процесса взаимодействия между отделом продаж и складом:

- отдел продаж получает из базы данных предприятия информацию о наличии товара;
- отдел продаж формирует заказ на отгрузку товара и ставит отгружаемый товар в резерв с передачей информации в базу данных;
- отдел продаж передает заказ на отгрузку товара складу;
- склад после отгрузки товара изменяет информацию о наличии товара с сохранением в базе данных.

Приложение будем создавать на базе платформы Java EE 6 с использованием сервера GlassFish 3.

Создание основы приложения

Для создания проекта приложения в среде NetBeans в меню **File** выберем **New Project | Java Web | Web Application**. Введем имя проекта приложения WebApplicationSaleStore, выберем платформу **JavaServer Faces** и нажмем кнопку **Finish.**

Для того чтобы открывать Web-странички приложения из среды NetBeans, в меню **Tools** выберем **Options | General** и установим Web-браузер по умолчанию.

Далее создадим базу данных предприятия. Для этого в среде NetBeans откроем окно **Services**. В узле **Databases** щелкнем правой кнопкой мыши на системе управления базами данных Java DB.

Примечание

Java DB — это система управления реляционными базами данных с открытым исходным кодом, полностью созданная на языке Java.

Выберем **Create DataBase** и создадим свою базу данных `Products` со своим логином и паролем, которая будет хранить информацию о товаре. При этом будет создано соединение с базой данных, отображаемое в узле **Databases**.

Щелкнем правой кнопкой мыши на названии проекта WebApplicationSaleStore и выберем **New | Other | GlassFish | JDBC Resource**. Создадим администрируемый объект сервера GlassFish — источник данных DataSource — jdbc/Products, связанный с созданным нами соединением с базой данных.

После этого щелкнем правой кнопкой мыши на названии проекта и выберем **New | Other | Persistence | Persistence Unit**. Создадим единицу сохранения, связанную с созданным источником данных jdbc/Products и представленную файлом persistence.xml.

Создадим компонент Entity, который будет представлять таблицу `Products` в базе данных. Для этого щелкнем правой кнопкой мыши на названии проекта и выберем **New | Other | Persistence | Entity Class**. Введем имя класса `Products` и имя пакета `market.store`.

В редакторе исходного кода введем дополнительно свойства класса `Products`, представляющие столбцы таблицы:

```
private String Products;
private Double Amount;
private Double Reserve;
```

Где `Products` — наименование товара, `Amount` — общее количество товара, `Reserve` — количество зарезервированного для отгрузки товара.

Далее в меню **Source** среды NetBeans выберем **Insert Code | Getter and Setter**, отметим галочками выше введенные свойства класса `Products` и сгенерируем для них методы `get` и `set`.

Щелкнем правой кнопкой мыши на названии проекта и выберем **New | Other | Persistence | JSF Pages from Entity Classes**. Выберем класс `market.store.Products` и сгенерируем JSF-странички и другие ресурсы для созданного нами Entity-компонента `Products`.

Среда NetBeans в зависимости от версий может по-разному генерировать исходный код. Изменим сгенерированный исходный код класса `Products` в соответствии с кодом, который отображен в листинге 9.1.

Листинг 9.1. Код класса `Products`

```
package market.store;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Products implements Serializable {
  private static final long serialVersionUID = 1L;
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  private String Products;
  private Double Amount;
  private Double Reserve;

  public Double getAmount() {
    return Amount;
  }

  public void setAmount(Double Amount) {
    this.Amount = Amount;
  }

  public String getProducts() {
    return Products;
  }

  public void setProducts(String Products) {
    this.Products = Products;
  }

  public Double getReserve() {
    return Reserve;
  }
```

```
public void setReserve(Double Reserve) {
  this.Reserve = Reserve;
}

public Long getId() {
  return id;
}

public void setId(Long id) {
  this.id = id;
}

@Override
public int hashCode() {
  int hash = 0;
  hash += (id != null ? id.hashCode() : 0);
  return hash;
}

@Override
public boolean equals(Object object) {
  // TODO: Warning — this method won't work
  //       in the case the id fields are not set
  if (!(object instanceof Products)) {
    return false;
  }
  Products other = (Products) object;
  if ((this.id == null && other.id != null) ||
      (this.id != null && !this.id.equals(other.id))) {
    return false;
  }
  return true;
}

@Override
public String toString() {
  return "market.store.Products[id=" + id + "]";
}}
```

Код класса `Products` (см. листинг 9.1) содержит рассмотренные нами ранее в *главе 8* аннотации, методы `get` и `set` для свойств класса, а также переопределенные методы класса `java.lang.Object`.

В каталоге WEB-INF проекта приложения средой NetBeans были сгенерированы конфигурационные файлы faces-config.xml, sun-web.xml, web.xml.

В дескрипторе развертывания web.xml указан класс сервлета для JSF-страничек, порядок его загрузки, URL-шаблон для вызова сервлета, максимальное время ожидания для сессий Web-приложения, файл, обрабатываемый сервером GlassFish первым при клиентском запросе.

В дескрипторе развертывания sun-web.xml указан уникальный идентификатор /WebApplicationSaleStore для данного Web-приложения в сервере GlassFish, являющийся частью URL-адреса клиентского запроса, а также описана конфигурация загрузчика.

В конфигурационном файле faces-config.xml указана связь ресурсного файла Bundle.properties с именем bundle и его относительный путь. Ресурсный файл Bundle.properties содержит ресурсы для JSF-страничек приложения.

При клиентском запросе первой открывается страничка index.xhtml, в коде которой тегом `h:commandLink` определена гиперссылка с текстом "Show All Products Items", при этом при нажатии на нее вызывается страничка List.xhtml.

В коде JSF-страницы List.xhtml тегом `ui:composition` указывается использование шаблона template.xhtml, атрибут тега `template` определяет относительный URI-адрес шаблона. Тег `ui:composition` является контейнером для отображаемого клиенту содержимого. Все, что находится вне тега, клиенту не отображается.

Тег `ui:define` JSF-страницы List.xhtml определяет вставляемый контент, атрибут тега `name` указывает идентификатор тега `ui:insert` шаблона template.xhtml.

Тегом `h:outputText` JSF-страницы List.xhtml выводятся значения свойств ресурсного файла Bundle.properties и возвращаемые значения методов связанного JavaBean-компонента `productsController` (класс `ProductsController`) — методы указываются без `get`.

Тег `h:commandLink` JSF-страницы List.xhtml выводит гиперссылки, при нажатии на которые вызываются соответствующие методы компонента `productsController`.

Тег `h:dataTable` JSF-страницы List.xhtml создает HTML-таблицу, представляющую таблицу `Products` базы данных.

Файл template.xhtml определяет шаблон для JSF-страничек приложения.

Тег `h:outputStylesheet` файла template.xhtml задает внешнюю CSS-таблицу стилей (файл jsfcrud.css) для JSF-страничек приложения. Тег `ui:insert` файла template.xhtml вставляет содержимое в шаблон страницы.

JSF-страницы приложения используют связанный или управляемый ManagedBean-компонент `productsController`.

ManagedBean-компоненты автоматически инициализируются контейнером во время запуска приложения и определяют свойства и методы, используемые приложением.

ManagedBean-компонент `productsController` (листинг 9.2) использует вспомогательный Stateless-компонент `ProductsFacade` (листинг 9.3), который реализует функциональность `EntityManager`, а также вспомогательные классы `PaginationHelper` (листинг 9.4) и `JsfUtil` (листинг 9.5).

Класс `PaginationHelper` обеспечивает разбивку на страницы при визуализации таблицы `Products`, а класс `JsfUtil` — вывод сообщений при изменении таблицы `Products`.

Листинг 9.2. Код класса `ProductsController`

```
package market.store;
import market.store.util.JsfUtil;
import market.store.util.PaginationHelper;
import java.util.ResourceBundle;
import javax.ejb.EJB;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.FacesConverter;
import javax.faces.model.DataModel;
import javax.faces.model.ListDataModel;
import javax.faces.model.SelectItem;
@ManagedBean (name="productsController")
@SessionScoped
public class ProductsController {
  private Products current;
  private DataModel items = null;
  @EJB private market.store.ProductsFacade ejbFacade;
  private PaginationHelper pagination;
  private int selectedItemIndex;
  public ProductsController() {  }
  public Products getSelected() {
    if (current == null) {
      current = new Products();
      selectedItemIndex = -1;
    }
```

```
    return current;
  }
  private ProductsFacade getFacade() {
    return ejbFacade;
  }
  public PaginationHelper getPagination() {
    if (pagination == null) {
      pagination = new PaginationHelper(10) {
        @Override
        public int getItemsCount() {
          return getFacade().count();
        }
        @Override
        public DataModel createPageDataModel() {
          return new ListDataModel(getFacade().findRange(
             new int[]{getPageFirstItem(),
               getPageFirstItem()+getPageSize()}));
        }};  }
    return pagination;
  }
  public String prepareList() {
    recreateModel();
    return "List";
  }
  public String prepareView() {
    current = (Products)getItems().getRowData();
    selectedItemIndex = pagination.getPageFirstItem() +
                        getItems().getRowIndex();
    return "View";
  }
  public String prepareCreate() {
    current = new Products();
    selectedItemIndex = -1;
    return "Create";
  }
  public String create() {
    try {
      getFacade().create(current);
      JsfUtil.addSuccessMessage(ResourceBundle.getBundle("/Bundle")
       .getString("ProductsCreated"));
      return prepareCreate();
    } catch (Exception e) {
```

```
      JsfUtil.addErrorMessage(e, ResourceBundle.getBundle("/Bundle")
        .getString("PersistenceErrorOccured"));
      return null;
    }    }
  public String prepareEdit() {
    current = (Products)getItems().getRowData();
    selectedItemIndex = pagination.getPageFirstItem() +
                        getItems().getRowIndex();
    return "Edit";
  }
  public String update() {
    try {
      getFacade().edit(current);
      JsfUtil.addSuccessMessage(ResourceBundle.getBundle("/Bundle")
        .getString("ProductsUpdated"));
      return "View";
    } catch (Exception e) {
      JsfUtil.addErrorMessage(e, ResourceBundle.getBundle("/Bundle")
        .getString("PersistenceErrorOccured"));
      return null;
    }    }
  public String destroy() {
    current = (Products)getItems().getRowData();
    selectedItemIndex = pagination.getPageFirstItem() +
                        getItems().getRowIndex();
    performDestroy();
    recreateModel();
    return "List";
  }
  public String destroyAndView() {
    performDestroy();
    recreateModel();
    updateCurrentItem();
    if (selectedItemIndex >= 0) {
      return "View";
    } else {
      // all items were removed — go back to list
      recreateModel();
      return "List";
    }    }
  private void performDestroy() {
    try {
      getFacade().remove(current);
```

```
      JsfUtil.addSuccessMessage(ResourceBundle.getBundle("/Bundle")
        .getString("ProductsDeleted"));
    } catch (Exception e) {
      JsfUtil.addErrorMessage(e, ResourceBundle.getBundle("/Bundle")
        .getString("PersistenceErrorOccured"));
    }
  }
  private void updateCurrentItem() {
    int count = getFacade().count();
    if (selectedItemIndex >= count) {
      // selected index cannot be bigger than number of items:
      selectedItemIndex = count-1;
      // go to previous page if last page disappeared:
      if (pagination.getPageFirstItem() >= count) {
        pagination.previousPage();
      }
    }
    if (selectedItemIndex >= 0) {
      current = getFacade().findRange(
        new int[]{selectedItemIndex, selectedItemIndex+1}).get(0);
    }
  }
  public DataModel getItems() {
    if (items == null) {
      items = getPagination().createPageDataModel();
    }
    return items;
  }
  private void recreateModel() {
    items = null;
  }
  public String next() {
    getPagination().nextPage();
    recreateModel();
    return "List";
  }
  public String previous() {
    getPagination().previousPage();
    recreateModel();
    return "List";
  }
  public SelectItem[] getItemsAvailableSelectMany() {
    return JsfUtil.getSelectItems(ejbFacade.findAll(), false);
  }
```

```
public SelectItem[] getItemsAvailableSelectOne() {
  return JsfUtil.getSelectItems(ejbFacade.findAll(), true);
}
@FacesConverter(forClass=Products.class)
public static class ProductsControllerConverter implements Converter {
  public Object getAsObject(FacesContext facesContext,
            UIComponent component, String value) {
    if (value == null || value.length() == 0) {
      return null;
    }
    ProductsController controller = (ProductsController)facesContext
       .getApplication().getELResolver()
       .getValue(facesContext.getELContext(), null,
                 "productsController");
          return controller.ejbFacade.find(getKey(value));
  }
  java.lang.Long getKey(String value) {
    java.lang.Long key;
    key = Long.valueOf(value);
    return key;
  }
  String getStringKey(java.lang.Long value) {
    StringBuffer sb = new StringBuffer();
    sb.append(value);
    return sb.toString();
  }
  public String getAsString(FacesContext facesContext,
                    UIComponent component, Object object) {
    if (object == null) {
       return null;
    }
    if (object instanceof Products) {
      Products o = (Products) object;
      return getStringKey(o.getId());
    } else {
      throw new IllegalArgumentException("object " +
         object + " is of type " + object.getClass().getName() +
         "; expected type: "+ProductsController.class.getName());
     }       }     }     }
```

Обсудим код класса `ProductsController` (см. листинг 9.2).

Метод `getSelected()` класса `ProductsController` создает новый экземпляр компонента `Products`, представляющий строку в таблице `Products`.

Метод `getFacade()` класса `ProductsController` возвращает экземпляр класса `ProductsFacade`.

Метод `getPagination()` класса `ProductsController` создает и возвращает экземпляр анонимного класса, расширяющего абстрактный класс `PaginationHelper()`. При этом в конструкторе класса закладывается, что каждая страница будет отображать не более 10 строк таблицы `Products`.

Методы `prepareList()`, `prepareView()`, `prepareCreate()` и `prepareEdit()` класса `ProductsController` обеспечивают активацию соответственно страниц List.xhtml, View.xhtml, Create.xhtml и Edit.xhtml.

Метод `update()` класса `ProductsController` обеспечивает редактирование данных таблицы.

Метод `destroy()` класса `ProductsController` обеспечивает удаление строк таблицы.

Метод `destroyAndView()` класса `ProductsController` обеспечивает удаление экземпляра компонента `Products` и, соответственно, строки таблицы.

Методы `performDestroy()` и `updateCurrentItem()` класса `ProductsController` используются методом `destroyAndView()`.

Метод `getItems()` класса `ProductsController` возвращает набор объектов, представляющих строки таблицы `Products`, для использования JSF-компонентами.

Метод `recreateModel()` класса `ProductsController` обнуляет набор `items`.

Метод `next()` класса `ProductsController` обеспечивает переход на следующую страницу, отображающую часть таблицы `Products`, в случае, если таблица не помещается на одной странице.

Метод `previous()` класса `ProductsController` обеспечивает переход на предыдущую страницу отображения таблицы `Products`.

Методы `getItemsAvailableSelectMany()` и `getItemsAvailableSelectOne()` класса `ProductsController` возвращают массивы элементов, представляющих экземпляры компонента `Products` и их строковое значение, для использования компонентами `UISelectMany` или `UISelectOne` соответственно.

Внутренний класс `ProductsControllerConverter` класса `ProductsController` конвертирует из строки в Java-объект и, наоборот, для класса `Products`, обеспечивает обработку вводимых данных от клиента и визуализацию для клиента строк таблицы.

Листинг 9.3. Код класса `ProductsFacade`

```
package market.store;

import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;

@Stateless
public class ProductsFacade {
  @PersistenceContext(unitName = "WebApplicationSaleStorePU")
  private EntityManager em;
  public void create(Products products) {
    em.persist(products);
  }

  public void edit(Products products) {
    em.merge(products);
  }

  public void remove(Products products) {
    em.remove(em.merge(products));
  }

  public Products find(Object id) {
    return em.find(Products.class, id);
  }

  public List<Products> findAll() {
    CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
    cq.select(cq.from(Products.class));
    return em.createQuery(cq).getResultList();
  }

  public List<Products> findRange(int[] range) {
    CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
    cq.select(cq.from(Products.class));
    Query q = em.createQuery(cq);
    q.setMaxResults(range[1] — range[0]);
```

```
    q.setFirstResult(range[0]);
    return q.getResultList();
  }

  public int count() {
    CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
    Root<Products> rt = cq.from(Products.class);
    cq.select(em.getCriteriaBuilder().count(rt));
    Query q = em.createQuery(cq);
    return ((Long) q.getSingleResult()).intValue();
  }}
```

Обсудим код класса `ProductsFacade` (см. листинг 9.3).

Метод `create()` класса `ProductsFacade` сохраняет экземпляр `Products` в базе данных.

Метод `edit()` класса `ProductsFacade` редактирует состояние экземпляра `Products`.

Метод `remove()` класса `ProductsFacade` удаляет экземпляр `Products`.

Метод `find()` класса `ProductsFacade` находит экземпляр `Products` по его идентификатору.

Метод `findAll()` класса `ProductsFacade` возвращает список экземпляров `Products`. Используемый в методе `findAll()` интерфейс `CriteriaQuery` — это интерфейс, обеспечивающий функциональность запросов к базе данных. Пакет `javax.persistence.criteria` представляет библиотеку Criteria API, которая позволяет динамически конструировать запросы на основе вызова методов.

Метод `findRange()` класса `ProductsFacade` возвращает список экземпляров `Products` в заданном диапазоне (используется при разбивке на страницы отображения таблицы `Products`).

Метод `count()` класса `ProductsFacade` возвращает количество экземпляров `Products`.

Листинг 9.4. Код класса `PaginationHelper`

```
package market.store.util;
import javax.faces.model.DataModel;
public abstract class PaginationHelper {
  private int pageSize;
  private int page;
  public PaginationHelper(int pageSize) {
```

```
    this.pageSize = pageSize;
  }
  public abstract int getItemsCount();
  public abstract DataModel createPageDataModel();
  public int getPageFirstItem() {
    return page*pageSize;
  }
  public int getPageLastItem() {
    int i = getPageFirstItem() + pageSize -1;
    int count = getItemsCount() — 1;
    if (i > count) {
      i = count;
    }
    if (i < 0) {
      i = 0;
    }
    return i;
  }
  public boolean isHasNextPage() {
    return (page+1)*pageSize+1 <= getItemsCount();
  }
  public void nextPage() {
    if (isHasNextPage()) {
      page++;
  }}
  public boolean isHasPreviousPage() {
    return page > 0;
  }
  public void previousPage() {
    if (isHasPreviousPage()) {
      page--;
  }}
  public int getPageSize() {
    return pageSize;
  }}
```

Обсудим код класса `PaginationHelper` (см. листинг 9.4).

В конструкторе `PaginationHelper` устанавливается размер страницы для отображения таблицы `Products` (количество строк на странице).

Метод `getItemsCount()` класса `PaginationHelper` возвращает общее количество строк для отображения.

Метод `getPageFirstItem()` класса `PaginationHelper` возвращает номер первой строки на странице.

Метод `getPageLastItem()` класса `PaginationHelper` возвращает номер последней строки на странице.

Метод `isHasNextPage()` класса `PaginationHelper` определяет, есть ли следующая страница.

Метод `nextPage()` класса `PaginationHelper` возвращает номер следующей страницы.

Метод `isHasPreviousPage()` класса `PaginationHelper` определяет, есть ли предыдущая страница.

Метод `previousPage()` класса `PaginationHelper` возвращает номер предыдущей страницы.

Метод `getPageSize()` класса `PaginationHelper` возвращает количество строк на странице.

Листинг 9.5. Код класса `JsfUtil`

```
package market.store.util;

import java.util.List;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.model.SelectItem;

public class JsfUtil {
  public static SelectItem[] getSelectItems(List<?> entities,
      boolean selectOne) {
    int size = selectOne ? entities.size() + 1 : entities.size();
    SelectItem[] items = new SelectItem[size];
    int i = 0;
    if (selectOne) {
      items[0] = new SelectItem("", "---");
        i++;
    }
    for (Object x : entities) {
       items[i++] = new SelectItem(x, x.toString());
    }
    return items;
  }
  public static void addErrorMessage(Exception ex, String defaultMsg) {
    String msg = ex.getLocalizedMessage();
```

```
    if (msg != null && msg.length() > 0) {
       addErrorMessage(msg);
    } else {
      addErrorMessage(defaultMsg);
    }
  }
  public static void addErrorMessages(List<String> messages) {
    for (String message : messages) {
      addErrorMessage(message);
    }
  }
  public static void addErrorMessage(String msg) {
    FacesMessage facesMsg =
      new FacesMessage(FacesMessage.SEVERITY_ERROR, msg, msg);
    FacesContext.getCurrentInstance().addMessage(null, facesMsg);
  }
  public static void addSuccessMessage(String msg) {
    FacesMessage facesMsg =
      new FacesMessage(FacesMessage.SEVERITY_INFO, msg, msg);
    FacesContext.getCurrentInstance().addMessage("successInfo",
                                                 facesMsg);
  }
  public static String getRequestParameter(String key) {
    return FacesContext.getCurrentInstance().getExternalContext()
            .getRequestParameterMap().get(key);
  }
  public static Object getObjectFromRequestParameter(
              String requestParameterName,
              Converter converter, UIComponent component) {
    String theId = JsfUtil.getRequestParameter(requestParameterName);
    return converter.getAsObject(
      FacesContext.getCurrentInstance(), component, theId);
  }}
```

Обсудим код класса `JsfUtil` (см. листинг 9.5).

Метод `getSelectItems()` класса `JsfUtil` возвращает массив элементов, представляющих экземпляры компонента `Products` и их строковое значение.

Методы `addErrorMessage()` и `addSuccessMessage()` класса `JsfUtil` обеспечивают вывод сообщений для клиента об ошибках и об успешном выполнении операций.

Метод `getRequestParameter()` класса `JsfUtil` возвращает значение параметра клиентского запроса по его имени как строку.

Метод `getObjectFromRequestParameter()` класса `JsfUtil` возвращает значение параметра клиентского запроса как Java-объект.

Как уже было сказано, методы `prepareList()`, `prepareView()`, `prepareCreate()` и `prepareEdit()` класса `ProductsController` обеспечивают активацию соответственно страниц List.xhtml, View.xhtml, Create.xhtml и Edit.xhtml.

Страница `Create` отображает для клиента поля для ввода значений свойств создаваемого экземляра `Products`, гиперссылку для сохранения созданного экземпляра в базе данных, гиперссылки активации страниц `index` и `List`.

Страница `Edit` отображает клиенту поля для редактирования значений свойств экземпляра компонента `Products`, гиперссылку для сохранения отредактированного экземпляра в базе данных, гиперссылки активации страниц `index`, `List` и `View`.

Страница `View` отображает клиенту значения свойств экземпляра компонента `Products`, гиперссылку для удаления экземпляра, гиперссылки активации страниц `index`, `List`, `Create` и `Edit`.

В самом начале мы создали единицу сохранения, связанную с созданным источником данных jdbc/Products и представленную файлом persistence.xml (листинг 9.6).

Листинг 9.6. Код конфигурационного файла единицы сохранения persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
   xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

  <persistence-unit name="WebApplicationSaleStorePU" transaction-
type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>jdbc/Products</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
    <property name="eclipselink.ddl-generation" value="create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

Конфигурационный файл persistence.xml (см. листинг 9.6) определяет имя единицы сохранения — `WebApplicationSaleStorePU`, тип управления транзак-

циями JTA (Java Transaction API), указывает класс `org.eclipse.persistence.jpa.PersistenceProvider`, реализующий интерфейс `javax.persistence.spi.PersistenceProvider`, вызываемый контейнером для создания фабрики `EntityManagerFactory`, определяет источник данных jdbc/Products.

С помощью элемента `<exclude-unlisted-classes>false</exclude-unlisted-classes>` в сохраняемую единицу включаются все классы, помеченные аннотацией `@Entity` и расположенные в том же каталоге, что и папка META-INF с файлом persistence.xml.

В свойствах `<properties>` файла persistence.xml указано свойство `eclipselink.ddl-generation` реализации Java Persistence API (JPA) — EclipseLink JPA, используемое реализацией для определения метода генерации таблиц при развертывании приложения. Возможные значения свойства: `none` (таблицы не создаются в базе данных), `create-tables` (таблицы создаются) и `drop-and-create-tables` (таблицы удаляются и создаются заново).

Созданное нами соединение с базой данных `Products` позволяет напрямую из среды NetBeans сделать запрос и посмотреть содержимое таблицы `Products`.

Для того чтобы сделать запрос и посмотреть содержимое таблицы `Products`, в среде NetBeans в окне **Services** в узле **Databases** щелкнем правой кнопкой мыши на соединении **jdbc:derby://localhost:1527/Products** и выберем пункт **Connect**, установив соединение с базой данных `Products`.

После этого там же выберем **Execute Command**. Появится окно SQL-запроса. Наберем

```
SELECT * FROM Products;
```

нажмем кнопку **Run SQL** и увидим содержимое таблицы (рис. 9.1).

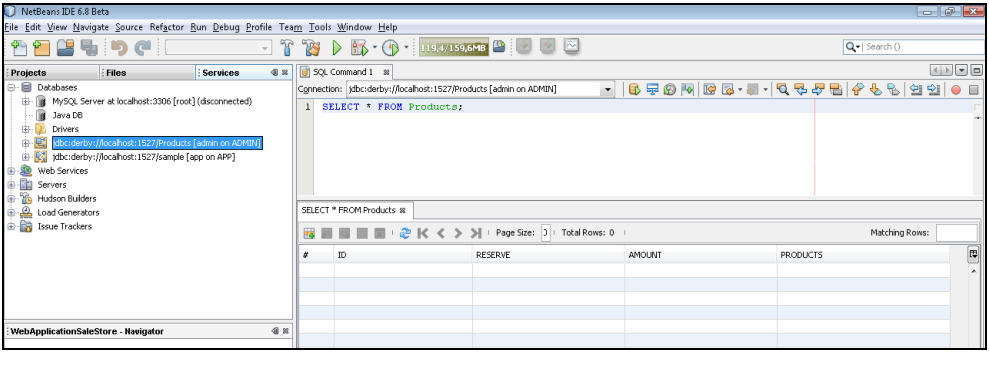

Рис. 9.1. Отображение таблицы `Products` в среде NetBeans

Кроме того, из среды NetBeans можно зайти в консоль администратора сервера GlassFish (рис. 9.2). Для этого в окне **Services** в узле **Servers** щелкнем

правой кнопкой мыши на сервере **GlassFish v3 Domain** и выберем пункт **View Admin Console**.

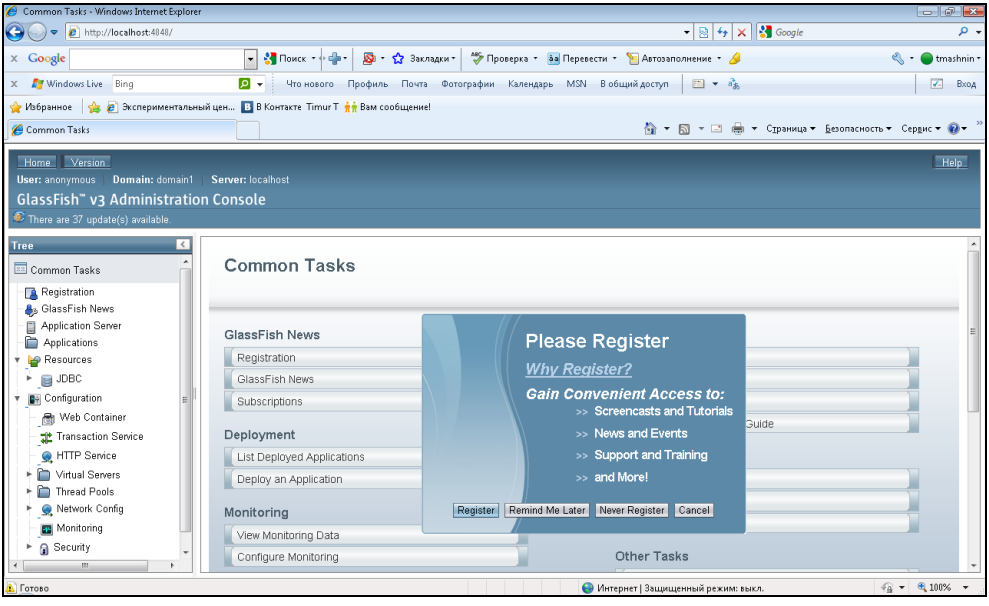

Рис. 9.2. Консоль администратора сервера GlassFish

Для того чтобы запустить приложение из среды NetBeans (рис. 9.3), необходимо щелкнуть правой кнопкой мыши на проекте в окне **Projects** и выбрать пункт **Run**.

Рис. 9.3. Стартовая страничка приложения

Создание "тонкого" клиента

После создания основы нашего приложения Java EE создадим единого "тонкого" клиента для администратора базы данных, для отдела продаж и склада предприятия.

Для начала зарегистрируем администратора базы данных, пользователей отдела продаж и пользователей склада в сервере. Для этого откроем консоль администратора сервера, раскроем узел **Security**, затем узел **Realms** и щелкнем на **realm file**. В открывшемся окне **Edit Realm** нажмем кнопку **Manage Users**, затем в окне **File Users** нажмем кнопку **New**. Введем имя пользователя admin_database, пароль и нажмем кнопку **OK**. То же самое проделаем для отдела продаж (имя **sale**) и склада (имя **store**).

Далее изменим код начальной страницы index.xhtml, введя гиперссылки для администратора базы данных, отдела продаж и склада, открывающие для них рабочие странички (листинг 9.7).

Листинг 9.7. Измененный код начальной страницы index.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:body>
    Отдел продаж — склад
    <h:form>
      <br/>
      <br/>
      <a
href="http://localhost:8080/WebApplicationSaleStore/faces/products/ListSa
leStore.xhtml">
      Отдел продаж</a>
      <br/>
      <br/>
      <a
href="http://localhost:8080/WebApplicationSaleStore/faces/products/ListSa
leStore.xhtml">
Склад</a>
      <br/>
      <br/>
      <a
href="http://localhost:8080/WebApplicationSaleStore/faces/products/List.x
html">
Администратор базы данных</a>
      </h:form>
    </h:body>
</html>
```

В коде начальной страницы index.xhtml URL-адрес ресурсов включает в себя адрес сервера и идентификатор приложения в сервере.

Теперь изменим файл web.xml, введя авторизацию доступа к страницам List.xhtml и ListSaleStore.xhtml (листинг 9.8).

Листинг 9.8. Код файла дескриптора развертывания приложения web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>
  <welcome-file-list>
    <welcome-file>faces/index.xhtml</welcome-file>
  </welcome-file-list>
  <security-constraint>
    <display-name>ConstraintList</display-name>
    <web-resource-collection>
      <web-resource-name>List</web-resource-name>
      <description/>
      <url-pattern>/faces/products/List.xhtml</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <description/>
```

```
      <role-name>admin_database</role-name>
    </auth-constraint>
  </security-constraint>
  <security-constraint>
    <display-name>ConstraintListSaleStore</display-name>
    <web-resource-collection>
      <web-resource-name>ListSaleStore</web-resource-name>
      <description/>
      <url-pattern>/faces/products/ListSaleStore.xhtml</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <description/>
      <role-name>store</role-name>
      <role-name>sale</role-name>
    </auth-constraint>
  </security-constraint>
  <security-role>
    <description/>
    <role-name>admin_database</role-name>
  </security-role>
  <security-role>
    <description/>
    <role-name>sale</role-name>
  </security-role>
  <security-role>
    <description/>
    <role-name>store</role-name>
  </security-role>
</web-app>
```

Редактирование конфигурационных файлов можно осуществлять с помощью удобного визуального редактора среды NetBeans.

Элементом `security-constraint` файла web.xml мы ввели ограничение доступа к ресурсам, при этом элементом `auth-constraint` определили разрешение доступа с авторизацией для ролей `store`, `sale` и `admin_database`.

Далее свяжем роли с зарегистрированными пользователями сервера, используя файл sun-web.xml (листинг 9.9).

Листинг 9.9. Код файла дескриптора развертывания сервера sun-web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Application
Server 9.0 Servlet 2.5//EN"
"http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-0.dtd">
```

```
<sun-web-app error-url="">
  <context-root>/WebApplicationSaleStore</context-root>
  <security-role-mapping>
    <role-name>admin_database</role-name>
    <principal-name>admin_database</principal-name>
  </security-role-mapping>
  <security-role-mapping>
    <role-name>sale</role-name>
    <principal-name>sale</principal-name>
  </security-role-mapping>
  <security-role-mapping>
    <role-name>store</role-name>
    <principal-name>store</principal-name>
  </security-role-mapping>
  <class-loader delegate="true"/>
  <jsp-config>
    <property name="keepgenerated" value="true">
      <description>Keep a copy of the generated servlet class' java
code.</description>
    </property>
  </jsp-config>
</sun-web-app>
```

В коде файла sun-web.xml для связывания ролей безопасности с пользователями сервера применяется элемент `security-role-mapping`.

Теперь при нажатии гиперссылки начальной страницы сервер будет запрашивать логин и пароль.

В папке products проекта приложения создадим странички ListSaleStore.xhtml (листинг 9.10), ViewSaleStore.xhtml (листинг 9.11) и EditSaleStore.xhtml (листинг 9.12), представляющие интерфейс пользователя для отдела продаж и склада.

Страницы List.xhtml, View.xhtml, Edit.xhtml и Create.xhtml представляют интерфейс пользователя для администратора базы данных и отличаются от страниц отдела продаж и склада наличием функций удаления и создания строк в таблице `Products` базы данных.

Листинг 9.10. Код страницы ListSaleStore.xhtml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

  <ui:composition template="/template.xhtml">
    <ui:define name="title">
      <h:outputText value="#{bundle.ListProductsTitle}"></h:outputText>
    </ui:define>
    <ui:define name="body">
      <h:form styleClass="jsfcrud_list_form">
        <h:panelGroup id="messagePanel" layout="block">
          <h:messages errorStyle="color: red"
                      infoStyle="color: green" layout="table"/>
        </h:panelGroup>
        <h:outputText value="#{bundle.ListProductsEmpty}"
           rendered="#{productsController.items.rowCount == 0}"/>
        <h:panelGroup
           rendered="#{productsController.items.rowCount > 0}">
          <h:outputText
value="#{productsController.pagination.pageFirstItem +
1}..#{productsController.pagination.pageLastItem +
1}/#{productsController.pagination.itemsCount}"/> 
          <h:commandLink action="#{productsController.previous}"
    value="#{bundle.Previous} #{productsController.pagination.pageSize}"
    rendered="#{productsController.pagination.hasPreviousPage}"/> 
          <h:commandLink action="#{productsController.next}"
        value="#{bundle.Next} #{productsController.pagination.pageSize}"
        rendered="#{productsController.pagination.hasNextPage}"/> 
          <h:dataTable value="#{productsController.items}" var="item"
                       border="0" cellpadding="2" cellspacing="0"
                       rowClasses="jsfcrud_odd_row,jsfcrud_even_row"
                       rules="all" style="border:solid 1px">
            <h:column>
              <f:facet name="header">
                <h:outputText
                   value="#{bundle.ListProductsTitle_amount}"/>
              </f:facet>
              <h:outputText value="#{item.amount}"/>
            </h:column>
            <h:column>
              <f:facet name="header">
                <h:outputText
                   value="#{bundle.ListProductsTitle_products}"/>
              </f:facet>
```

```
            <h:outputText value="#{item.products}"/>
          </h:column>
          <h:column>
            <f:facet name="header">
              <h:outputText
                 value="#{bundle.ListProductsTitle_reserve}"/>
            </f:facet>
            <h:outputText value="#{item.reserve}"/>
          </h:column>
          <h:column>
            <f:facet name="header">
              <h:outputText value="#{bundle.ListProductsTitle_id}"/>
            </f:facet>
            <h:outputText value="#{item.id}"/>
          </h:column>
          <h:column>
            <f:facet name="header">
              <h:outputText value=" "/>
            </f:facet>
            <h:commandLink
               action="#{productsController.prepareViewSaleStore}"
               value="#{bundle.ListProductsViewLink}"/>
            <h:outputText value=" "/>
            <h:commandLink
               action="#{productsController.prepareEditSaleStore}"
               value="#{bundle.ListProductsEditLink}"/>
            <h:outputText value=" "/>
          </h:column>
        </h:dataTable>
      </h:panelGroup>
      <br />
      <br />
      <br />
      <h:commandLink value="#{bundle.ListProductsIndexLink}"
                     action="/index" immediate="true" />
      <br />
      <br />
      <h:commandLink value="#{bundle.ShipList}"
         action="#{shipController.prepareList}" immediate="true" />
    </h:form>
  </ui:define>
 </ui:composition>
</html>
```

В коде страницы ListSaleStore.xhtml имеется гиперссылка на страницу ListShip.xhtml, отображающую таблицу заказа на отгрузку, о которой пойдет речь позже.

Листинг 9.11. Код страницы ViewSaleStore.xhtml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
  <ui:composition template="/template.xhtml">
    <ui:define name="title">
      <h:outputText value="#{bundle.ViewProductsTitle}"></h:outputText>
    </ui:define>
    <ui:define name="body">
      <h:panelGroup id="messagePanel" layout="block">
        <h:messages errorStyle="color: red"
                    infoStyle="color: green" layout="table"/>
      </h:panelGroup>
      <h:form>
        <h:panelGrid columns="2">
          <h:outputText value="#{bundle.ViewProductsLabel_amount}"/>
          <h:outputText value="#{productsController.selected.amount}"
             title="#{bundle.ViewProductsTitle_amount}"/>
          <h:outputText value="#{bundle.ViewProductsLabel_products}"/>
          <h:outputText value="#{productsController.selected.products}"
             title="#{bundle.ViewProductsTitle_products}"/>
          <h:outputText value="#{bundle.ViewProductsLabel_reserve}"/>
          <h:outputText value="#{productsController.selected.reserve}"
             title="#{bundle.ViewProductsTitle_reserve}"/>
          <h:outputText value="#{bundle.ViewProductsLabel_id}"/>
          <h:outputText value="#{productsController.selected.id}"
             title="#{bundle.ViewProductsTitle_id}"/>
        </h:panelGrid>
        <br />
        <br />
        <br />
        <h:commandLink action="EditSaleStore"
                       value="#{bundle.ViewProductsEditLink}"/>
        <br />
```

```
        <br />
        <h:commandLink
           action="#{productsController.prepareListSaleStore}"
           value="#{bundle.ViewProductsShowAllLink}"/>
        <br />
        <br />
        <h:commandLink value="#{bundle.ViewProductsIndexLink}"
           action="/index" immediate="true" />
      </h:form>
    </ui:define>
  </ui:composition>
</html>
```

Листинг 9.12. Код страницы EditSaleStore.xhtml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
  <ui:composition template="/template.xhtml">
    <ui:define name="title">
      <h:outputText value="#{bundle.EditProductsTitle}"></h:outputText>
    </ui:define>
    <ui:define name="body">
      <h:panelGroup id="messagePanel" layout="block">
        <h:messages errorStyle="color: red" infoStyle="color: green"
                    layout="table"/>
      </h:panelGroup>
      <h:form>
        <h:panelGrid columns="2">
          <h:outputLabel value="#{bundle.EditProductsLabel_id}"
             for="id" />
          <h:inputText id="id" value="#{productsController.selected.id}"
             title="#{bundle.EditProductsTitle_id}" />
          <h:outputLabel value="#{bundle.EditProductsLabel_products}"
             for="products" />
          <h:inputText id="products"
             value="#{productsController.selected.products}"
             title="#{bundle.EditProductsTitle_products}" />
          <h:outputLabel value="#{bundle.EditProductsLabel_amount}"
             for="amount" />
```

```
            <h:inputText id="amount"
               value="#{productsController.selected.amount}"
               title="#{bundle.EditProductsTitle_amount}" />
            <h:outputLabel value="#{bundle.EditProductsLabel_reserve}"
               for="reserve" />
            <h:inputText id="reserve"
               value="#{productsController.selected.reserve}"
               title="#{bundle.EditProductsTitle_reserve}" />
         </h:panelGrid>
         <h:commandLink action="#{productsController.updateSaleStore}"
            value="#{bundle.EditProductsSaveLink}"/>
         <br />
         <br />
         <h:commandLink action="ViewSaleStore"
            value="#{bundle.EditProductsViewLink}" immediate="true"/>
         <br />
         <h:commandLink
            action="#{productsController.prepareListSaleStore}"
            value="#{bundle.EditProductsShowAllLink}" immediate="true"/>
         <br />
         <br />
         <h:commandLink value="#{bundle.EditProductsIndexLink}"
           action="/index" immediate="true" />
      </h:form>
    </ui:define>
  </ui:composition>
</html>
```

Далее создадим компонент Entity, представляющий таблицу `Ship` (листинг 9.13) заказа на отгрузку, которая будет автоматически заполняться при заполнении отделом продаж столбца `Reserve` (резерв товара) и которую сможет просматривать склад.

Листинг 9.13. Код класса `Ship`

```
package market.store;
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
@Entity
```

```
public class Ship implements Serializable {
  private static final long serialVersionUID = 1L;
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  private String Products;
  private Double Amount;
  public Double getAmount() {
    return Amount;
  }
  public void setAmount(Double Amount) {
    this.Amount = Amount;
  }
  public String getProducts() {
    return Products;
  }
  public void setProducts(String Products) {
    this.Products = Products;
  }
  public Long getId() {
    return id;
  }
  public void setId(Long id) {
    this.id = id;
  }
  @Override
  public int hashCode() {
    int hash = 0;
    hash += (id != null ? id.hashCode() : 0);
    return hash;
  }
  @Override
  public boolean equals(Object object) {
    // TODO: Warning — this method won't work
    //       in the case the id fields are not set
    if (!(object instanceof Ship)) {
      return false;
    }
    Ship other = (Ship) object;
    if ((this.id == null && other.id != null) ||
        (this.id != null && !this.id.equals(other.id))) {
      return false;
    }
```

```
    return true;
  }
  @Override
  public String toString() {
    return "market.store.Ship[id=" + id + "]";
  }}
```

После создания кода класса `Ship` создадим в папке products проекта приложения JSF-страницы для компонента `Ship`: ListShip.xhtml, ViewShip.xhtml, EditShip.xhtml, CreateShip.xhtml. Код данных страниц аналогичен коду страниц интерфейса администратора базы данных для таблицы `Product`.

Гиперссылка на страницу ListShip.xhtml, отображающую таблицу `Ship` базы данных, находится на странице ListSaleStore.xhtml.

Создадим для JSF-страниц компонента `Ship` управляемый компонент `ShipController`, вспомогательный Stateless-компонент `ShipFacade`, реализующий функциональность `EntityManager`, а также вспомогательные классы `PaginationHelperShip` и `JsfUtilShip`. Функциональность классов `ShipController`, `ShipFacade`, `PaginationHelperShip` и `JsfUtilShip` аналогична функциональности рассмотренных ранее классов `ProductsController`, `ProductsFacade`, `PaginationHelper` и `JsfUtil`.

После этого изменим код класса `ProductsController` (листинг 9.14) таким образом, чтобы он обслуживал страницы ListSaleStore.xhtml, ViewSaleStore.xhtml и EditSaleStore.xhtml, а также обеспечивал автоматическое создание и сохранение строки в таблице `Ship` заказа на отгрузку товара при редактировании таблицы `Products` отделом продаж.

Листинг 9.14. Измененный код класса `ProductsController`

```
package market.store;
import market.store.util.JsfUtil;
import market.store.util.PaginationHelper;
import java.util.ResourceBundle;
import javax.ejb.EJB;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.FacesConverter;
import javax.faces.model.DataModel;
import javax.faces.model.ListDataModel;
import javax.faces.model.SelectItem;
```

```
@ManagedBean (name="productsController")
@SessionScoped
public class ProductsController {
  private Products current;
  private DataModel items = null;
  @EJB private market.store.ProductsFacade ejbFacade;
  @EJB private market.store.ShipFacade ejbShipFacade;
  private PaginationHelper pagination;
  private int selectedItemIndex;
  public ProductsController() {  }
  public Products getSelected() {
    if (current == null) {
      current = new Products();
      selectedItemIndex = -1;
    }
    return current;
  }
  private ProductsFacade getFacade() {
    return ejbFacade;
  }
  private ShipFacade getShipFacade() {
    return ejbShipFacade;
  }
  public PaginationHelper getPagination() {
    if (pagination == null) {
      pagination = new PaginationHelper(10) {
        @Override
        public int getItemsCount() {
          return getFacade().count();
        }
        @Override
        public DataModel createPageDataModel() {
          return new ListDataModel(getFacade()
            .findRange(new int[]{getPageFirstItem(),
                  getPageFirstItem()+getPageSize()}));
        }  };  }
    return pagination;
  }
  public String prepareListSaleStore() {
    recreateModel();
    return "ListSaleStore";
  }
  public String prepareList() {
    recreateModel();
```

```
    return "List";
  }
  public String prepareView() {
    current = (Products)getItems().getRowData();
    selectedItemIndex = pagination.getPageFirstItem() +
                        getItems().getRowIndex();
    return "View";
  }
  public String prepareViewSaleStore() {
    current = (Products)getItems().getRowData();
    selectedItemIndex = pagination.getPageFirstItem() +
                        getItems().getRowIndex();
    return "ViewSaleStore";
  }
  public String prepareCreate() {
    current = new Products();
    selectedItemIndex = -1;
    return "Create";
  }
  public String create() {
    try {
      getFacade().create(current);
      JsfUtil.addSuccessMessage(ResourceBundle.getBundle("/Bundle")
        .getString("ProductsCreated"));
      return prepareCreate();
    } catch (Exception e) {
      JsfUtil.addErrorMessage(e, ResourceBundle.getBundle("/Bundle")
        .getString("PersistenceErrorOccured"));
      return null;
    }
  }
  public String prepareEdit() {
    current = (Products)getItems().getRowData();
    selectedItemIndex = pagination.getPageFirstItem() +
                        getItems().getRowIndex();
    return "Edit";
  }
  public String prepareEditSaleStore() {
    current = (Products)getItems().getRowData();
    selectedItemIndex = pagination.getPageFirstItem() +
                        getItems().getRowIndex();
    return "EditSaleStore";
  }
```

```
public String update() {
  try {
    getFacade().edit(current);
    JsfUtil.addSuccessMessage(ResourceBundle
      .getBundle("/Bundle").getString("ProductsUpdated"));
    return "View";
  } catch (Exception e) {
    JsfUtil.addErrorMessage(e, ResourceBundle.getBundle("/Bundle")
      .getString("PersistenceErrorOccured"));
    return null;
  }
}
public String updateSaleStore() {
  try {
    getFacade().edit(current);
    Ship ship = new Ship();
    ship.setId(current.getId());
    ship.setProducts(current.getProducts());
    ship.setAmount(current.getReserve());
    if (getShipFacade().find(current.getId())==null){
      getShipFacade().create(ship);
    }
    JsfUtil.addSuccessMessage(ResourceBundle
        .getBundle("/Bundle").getString("ProductsUpdated"));
    return "ViewSaleStore";
  } catch (Exception e) {
    JsfUtil.addErrorMessage(e, ResourceBundle.getBundle("/Bundle")
       .getString("PersistenceErrorOccured"));
    return null;
  }    }
public String destroy() {
  current = (Products)getItems().getRowData();
  selectedItemIndex = pagination.getPageFirstItem() +
                      getItems().getRowIndex();
  performDestroy();
  recreateModel();
  return "List";
}
public String destroyAndView() {
  performDestroy();
  recreateModel();
  updateCurrentItem();
  if (selectedItemIndex >= 0) {
    return "View";
```

```
    } else {
      // all items were removed — go back to list
      recreateModel();
      return "List";
    }    }
  private void performDestroy() {
    try {
      getFacade().remove(current);
      JsfUtil.addSuccessMessage(ResourceBundle
          .getBundle("/Bundle").getString("ProductsDeleted"));
    } catch (Exception e) {
      JsfUtil.addErrorMessage(e, ResourceBundle.getBundle("/Bundle")
          .getString("PersistenceErrorOccured"));
    }    }
  private void updateCurrentItem() {
    int count = getFacade().count();
    if (selectedItemIndex >= count) {
      // selected index cannot be bigger than number of items:
      selectedItemIndex = count-1;
      // go to previous page if last page disappeared:
      if (pagination.getPageFirstItem() >= count) {
        pagination.previousPage();
      }
    }
    if (selectedItemIndex >= 0) {
      current = getFacade().findRange(
        new int[]{selectedItemIndex, selectedItemIndex+1}).get(0);
    }
  }
  public DataModel getItems() {
    if (items == null) {
      items = getPagination().createPageDataModel();
    }
    return items;
  }
  private void recreateModel() {
    items = null;
  }
  public String next() {
    getPagination().nextPage();
    recreateModel();
    return "List";
  }
  public String previous() {
    getPagination().previousPage();
```

```
    recreateModel();
    return "List";
  }
  public SelectItem[] getItemsAvailableSelectMany() {
    return JsfUtil.getSelectItems(ejbFacade.findAll(), false);
  }
  public SelectItem[] getItemsAvailableSelectOne() {
    return JsfUtil.getSelectItems(ejbFacade.findAll(), true);
  }
  @FacesConverter(forClass=Products.class)
  public static class ProductsControllerConverter implements Converter {
    public Object getAsObject(FacesContext facesContext,
              UIComponent component, String value) {
      if (value == null || value.length() == 0) {
        return null;
      }
      ProductsController controller = (ProductsController)facesContext
          .getApplication().getELResolver().
      getValue(facesContext.getELContext(), null, "productsController");
      return controller.ejbFacade.find(getKey(value));
    }
    java.lang.Long getKey(String value) {
      java.lang.Long key;
      key = Long.valueOf(value);
      return key;
    }
    String getStringKey(java.lang.Long value) {
      StringBuffer sb = new StringBuffer();
      sb.append(value);
      return sb.toString();
    }
    public String getAsString(FacesContext facesContext,
              UIComponent component, Object object) {
      if (object == null) {
        return null;
      }
      if (object instanceof Products) {
        Products o = (Products) object;
        return getStringKey(o.getId());
      } else {
        throw new IllegalArgumentException("object " +
          object + " is of type " + object.getClass().getName() +
          "; expected type: "+ProductsController.class.getName());
      }  }  }}
```

В коде класса `ProductsController` (см. листинг 9.14) метод `getShipFacade()` возвращает экземпляр класса `ShipFacade`, используемый здесь для сохранения строки таблицы `Ship`.

Методы `prepareListSaleStore()`, `prepareViewSaleStore()` и `prepareEditSaleStore()` класса `ProductsController` вызываются в JSF-страницах интерфейса отдела продаж и склада.

При вызове метода `updateSaleStore()` класса `ProductsController` в странице EditSaleStore.xhtml создается экземпляр `Ship`, в который записываются значения свойств `Id`, `Products` и `Reserve` редактируемого экземпляра `Products`, и если такая строка отсутствует в таблице `Ship` базы данных, то созданный экземпляр `Ship` сохраняется.

В качестве завершающего аккорда осуществим локализацию ресурсного файла Bunle.

Для этого изменим код конфигурационного файла faces-config.xml (листинг 9.15).

Листинг 9.15. Измененный код конфигурационного файла faces-config.xml

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
 http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">
 <application>
  <resource-bundle>
    <base-name>/Bundle</base-name>
    <var>bundle</var>
  </resource-bundle>
  <locale-config>
    <default-locale>ru</default-locale>
    <supported-locale>en</supported-locale>
  </locale-config>
  </application>
</faces-config>
```

С помощью элемента `locale-config` конфигурационного файла faces-config.xml определим русскую локализацию по умолчанию — элемент `default-locale` и поддерживаемую английскую локализацию — элемент `supported-locale`.

Далее создадим два ресурсных файла, Bundle_en.properties (листинг 9.16) и Bundle_ru.properties (листинг 9.17), обеспечивающих английскую и русскую локализации соответственно.

Листинг 9.16. Код файла Bundle_en.properties

```
PersistenceErrorOccured=A persistence error occurred.
Previous=Previous
Next=Next

ProductsCreated=Products was successfully created.
ProductsUpdated=Products was successfully updated.
ProductsDeleted=Products was successfully deleted.
CreateProductsTitle=Create New Products
CreateProductsSaveLink=Save
CreateProductsShowAllLink=Show All Products Items
CreateProductsIndexLink=Index
CreateProductsLabel_amount=Amount:
CreateProductsTitle_amount=Amount
CreateProductsLabel_products=Products:
CreateProductsTitle_products=Products
CreateProductsLabel_reserve=Reserve:
CreateProductsTitle_reserve=Reserve
CreateProductsLabel_id=Id:
CreateProductsTitle_id=Id
EditProductsTitle=Edit Products
EditProductsSaveLink=Save
EditProductsViewLink=View
EditProductsShowAllLink=Show All Products Items
EditProductsIndexLink=Index
EditProductsLabel_amount=Amount:
EditProductsTitle_amount=Amount
EditProductsLabel_products=Products:
EditProductsTitle_products=Products
EditProductsLabel_reserve=Reserve:
EditProductsTitle_reserve=Reserve
EditProductsLabel_id=Id:
EditProductsTitle_id=Id
ViewProductsTitle=View
ViewProductsDestroyLink=Destroy
ViewProductsEditLink=Edit
ViewProductsCreateLink=Create New Products
ViewProductsShowAllLink=Show All Products Items
ViewProductsIndexLink=Index
ViewProductsLabel_amount=Amount:
ViewProductsTitle_amount=Amount
ViewProductsLabel_products=Products:
ViewProductsTitle_products=Products
```

```
ViewProductsLabel_reserve=Reserve:
ViewProductsTitle_reserve=Reserve
ViewProductsLabel_id=Id:
ViewProductsTitle_id=Id
ListProductsTitle=List
ListProductsEmpty=(No Products Items Found)
ListProductsDestroyLink=Destroy
ListProductsEditLink=Edit
ListProductsViewLink=View
ListProductsCreateLink=Create New Products
ListProductsIndexLink=Index
ListProductsTitle_amount=Amount
ListProductsTitle_products=Products
ListProductsTitle_reserve=Reserve
ListProductsTitle_id=Id
ShiList=Shipping

ShipCreated=Ship was successfully created.
ShipUpdated=Ship was successfully updated.
ShipDeleted=Ship was successfully deleted.
CreateShipTitle=Create New Ship
CreateShipSaveLink=Save
CreateShipShowAllLink=Show All Ship Items
CreateShipIndexLink=Index
CreateShipLabel_amount=Amount:
CreateShipTitle_amount=Amount
CreateShipLabel_products=Products:
CreateShipTitle_products=Products
CreateShipLabel_id=Id:
CreateShipTitle_id=Id
EditShipTitle=Edit Ship
EditShipSaveLink=Save
EditShipViewLink=View
EditShipShowAllLink=Show All Ship Items
EditShipIndexLink=Index
EditShipLabel_amount=Amount:
EditShipTitle_amount=Amount
EditShipLabel_products=Products:
EditShipTitle_products=Products
EditShipLabel_id=Id:
EditShipTitle_id=Id
ViewShipTitle=View
ViewShipDestroyLink=Destroy
```

```
ViewShipEditLink=Edit
ViewShipCreateLink=Create New Ship
ViewShipShowAllLink=Show All Ship Items
ViewShipIndexLink=Index
ViewShipLabel_amount=Amount:
ViewShipTitle_amount=Amount
ViewShipLabel_products=Products:
ViewShipTitle_products=Products
ViewShipLabel_id=Id:
ViewShipTitle_id=Id
ListShipTitle=List
ListShipEmpty=(No Ship Items Found)
ListShipDestroyLink=Destroy
ListShipEditLink=Edit
ListShipViewLink=View
ListShipCreateLink=Create New Ship
ListShipIndexLink=Index
ListShipTitle_amount=Amount
ListShipTitle_products=Products
ListShipTitle_id=Id
```

Листинг 9.17. Код файла Bundle_ru.properties

```
PersistenceErrorOccured=Ошибка сохранения.
Previous=Предыдущий
Next=Следующий

ProductsCreated=Строка создана.
ProductsUpdated=Строка обновлена.
ProductsDeleted=Строка удалена.
CreateProductsTitle=Создать новую строку.
CreateProductsSaveLink=Сохранить.
CreateProductsShowAllLink=Показать таблицу.
CreateProductsIndexLink=В начало.
CreateProductsLabel_amount=Количество:
CreateProductsTitle_amount=Количество
CreateProductsLabel_products=Товар:
CreateProductsTitle_products=Товар
CreateProductsLabel_reserve=Резерв:
CreateProductsTitle_reserve=Резерв
CreateProductsLabel_id=Id:
CreateProductsTitle_id=Id
EditProductsTitle=Редактировать строку.
```

```
EditProductsSaveLink=Сохранить.
EditProductsViewLink=Показать.
EditProductsShowAllLink=Показать таблицу.
EditProductsIndexLink=В начало.
EditProductsLabel_amount=Количество:
EditProductsTitle_amount=Количество
EditProductsLabel_products=Товар:
EditProductsTitle_products=Товар
EditProductsLabel_reserve=Резерв:
EditProductsTitle_reserve=Резерв
EditProductsLabel_id=Id:
EditProductsTitle_id=Id
ViewProductsTitle=Показать.
ViewProductsDestroyLink=Удалить.
ViewProductsEditLink=Редактировать.
ViewProductsCreateLink=Создать новую строку.
ViewProductsShowAllLink=Показать таблицу.
ViewProductsIndexLink=В начало.
ViewProductsLabel_amount=Количество:
ViewProductsTitle_amount=Количество
ViewProductsLabel_products=Товар:
ViewProductsTitle_products=Товар
ViewProductsLabel_reserve=Резерв:
ViewProductsTitle_reserve=Резерв
ViewProductsLabel_id=Id:
ViewProductsTitle_id=Id
ListProductsTitle=Таблица Товара
ListProductsEmpty=Таблица пуста
ListProductsDestroyLink=Удалить.
ListProductsEditLink=Редактировать.
ListProductsViewLink=Показать.
ListProductsCreateLink=Создать новую строку.
ListProductsIndexLink=В начало.
ListProductsTitle_amount=Количество.
ListProductsTitle_products=Товар.
ListProductsTitle_reserve=Резерв.
ListProductsTitle_id=Id
ShipList=Заказ на отгрузку

ShipCreated=Строка создана.
ShipUpdated=Строка обновлена.
ShipDeleted=Строка удалена.
CreateShipTitle=Создать новую строку.
CreateShipSaveLink=Сохранить.
```

```
CreateShipShowAllLink=Показать таблицу.
CreateShipIndexLink=В начало.
CreateShipLabel_amount=Количество:
CreateShipTitle_amount=Количество
CreateShipLabel_products=Товар:
CreateShipTitle_products=Товар
CreateShipLabel_id=Id:
CreateShipTitle_id=Id
EditShipTitle=Редактировать.
EditShipSaveLink=Сохранить.
EditShipViewLink=Показать.
EditShipShowAllLink=Показать таблицу.
EditShipIndexLink=В начало.
EditShipLabel_amount=Количество:
EditShipTitle_amount=Количество
EditShipLabel_products=Товар:
EditShipTitle_products=Товар
EditShipLabel_id=Id:
EditShipTitle_id=Id
ViewShipTitle=Показать.
ViewShipDestroyLink=Удалить.
ViewShipEditLink=Редактировать.
ViewShipCreateLink=Создать новую строку.
ViewShipShowAllLink=Показать таблицу.
ViewShipIndexLink=В начало.
ViewShipLabel_amount=Количество:
ViewShipTitle_amount=Количество
ViewShipLabel_products=Товар:
ViewShipTitle_products=Товар
ViewShipLabel_id=Id:
ViewShipTitle_id=Id
ListShipTitle=Таблица Заказа
ListShipEmpty=Таблица пуста
ListShipDestroyLink=Удалить.
ListShipEditLink=Редактировать.
ListShipViewLink=Показать.
ListShipCreateLink=Создать новую строку.
ListShipIndexLink=В начало.
ListShipTitle_amount=Количество.
ListShipTitle_products=Товар.
ListShipTitle_id=Id
```

Теперь контейнер сервера будет извлекать значения свойств из ресурсного файла русской локализации Bundle_ru.properties.

Для того чтобы запустить созданное нами приложение, в окне **Projects** среды NetBeans щелкнем правой кнопкой мыши на названии проекта и выберем пункт **Run**.

Наше приложение готово к использованию (рис. 9.4).

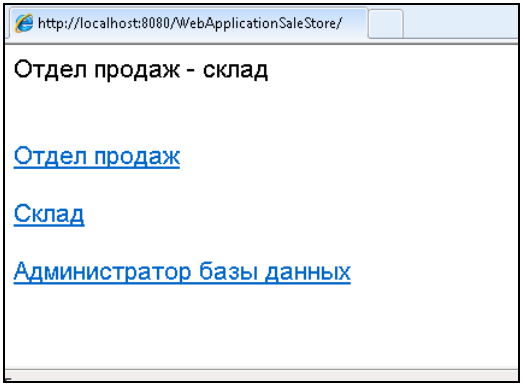

Рис. 9.4. Стартовая страничка приложения

Приложения

Приложение 1	Структура JRE и JDK
Приложение 2	Структура интерфейса программирования платформы Java SE
Приложение 3	Проекты Eclipse
Приложение 4	Основные библиотеки NetBeans API
Приложение 5	Спецификации платформы Java SE
Приложение 6	Структура графической библиотеки Swing
Приложение 7	Коллекция классов пакета java.io
Приложение 8	Справочная система JavaHelp
Приложение 9	Архитектура технологии OSGi
Приложение 10	Библиотеки спецификации CLDC 1.0
Приложение 11	Синтаксис JSP
Приложение 12	Библиотеки технологии JavaServer Faces
Приложение 13	Описание компакт-диска

ПРИЛОЖЕНИЕ 1

Структура JRE и JDK

Структура файловой системы среды выполнения Java Runtime Environment (JRE)

Таблица П1.1. *Исполняемые файлы и динамически подключаемые библиотеки DLL инструментов и библиотек платформы Java, расположенные в папке bin*

Папки и инструменты	Описание
`java`	Является загрузчиком Java-приложений
`javacpl`	Запускает контрольную панель Java Control Panel, которая показывает и позволяет устанавливать набор параметров работы платформы Java в операционной системе
`java-rmi`	Обеспечивает поддержку технологии удаленного вызова процедур Java Remote Method Invocation (RMI). Принимает RMI/HTTP-запросы и направляет их серверу RMI
`javaw`	Запускает Java-приложения так же, как и инструмент `java`, только без создания консольного окна
`javaws`	Используется для загрузки Java-приложений с помощью технологии Java Web Start и установки различных опций
`jbroker`	Обеспечивает поддержку технологии CORBA (Common Object Request Broker Architecture)
`jp2launcher`	Часть загрузчика Java Plug-in
`jqs` и `jqsnotify`	Реализуют механизм Java Quick Starter (JQS), уменьшающий время первоначального запуска для апплетов и Java-приложений
`jucheck`	Проверяет через Интернет наличие обновлений
`jureg`	Java Update RegisterTask, часть механизма обновлений
`jusched`	Sun Java Update Scheduler, часть механизма обновлений

Таблица П1.1 *(окончание)*

Папки и инструменты	**Описание**
`keytool`	Управляет хранилищами ключей и сертификатов. Генерирует ключи, добавляет и удаляет из хранилища сертификаты
`kinit`	Используется для регистрации в системе аутентификации/авторизации Kerberos
`klist`	Считывает и отображает список доступа системы Kerberos
`ktab`	Позволяет управлять логинами и паролями системы Kerberos
`orbd` (Object Request Broker Daemon)	Обеспечивает для клиента поиск и вызов сохраняемых объектов в CORBA-окружении
`pack200`	Трансформирует JAR-файл в высокосжатый pack200-файл. Инструмент `unpack200` производит обратную трансформацию
`policytool`	Создает и изменяет конфигурационный файл системы безопасности Java
`Rmid`	Производит системную активацию RMI
`rmiregistry`	Запускает регистрацию удаленных объектов
`servertool`	Обеспечивает интерфейс управления сервером
`ssvagent`	Часть загрузчика Java Plug-in
`tnameserv`	Обеспечивает доступ к сервису Java IDL Transient Naming Service
Папка client	Содержит файлы, используемые виртуальной машиной Client VM
Папка server	Содержит файлы, используемые виртуальной машиной Server VM

Таблица П1.2. *Базовые библиотеки, ресурсные и конфигурационные файлы в папке lib*

Папки и библиотеки	**Описание**
Папка applet	Предназначена для размещения классов апплетов, что сокращает время запуска больших апплетов в браузере
Папка audio	Содержит ресурсные файлы, используемые Java Sound API
Папка cmm	Содержит цветовые профили
Папка deploy	Содержит файлы, используемые при развертывании приложений
Папка ext	Каталог по умолчанию для JAR-файлов расширений платформы Java
Папка fonts	Содержит файлы шрифтов
Папка i386	Среда выполнения JRE операционной системы Windows содержит конфигурационный файл виртуальной машины JVM

Таблица П1.2 *(окончание)*

Папки и библиотеки	Описание
Папка im	Содержит JAR-файлы, обеспечивающие взаимодействие между текстовыми компонентами приложения и методами ввода, что в частности позволяет связать стандартную клавиатуру с такими раскладками, как тайская и индусская
Папка images	Содержит файлы изображений, используемые средой выполнения
Папка management	Содержит файлы, используемые при контроле и управлении JVM с помощью технологии Java Management Extensions (JMX)
Папка security	Содержит файлы управления системой безопасности Java
Папка servicetag	Содержит конфигурационный файл инсталляции JRE
Папка zi	Содержит информацию о временных зонах
JAR-файл charsets	Содержит кодировки, используемые средой выполнения
JAR-файл deploy	Библиотеки, используемые при развертывании приложений
JAR-файл javaws	Библиотеки одноименного инструмента
JAR-файл jce	Библиотеки технологии Java Cryptographic Extension
JAR-файл jsse	Библиотеки технологии Java Secure Socket Extension
JAR-файл management-agent	Агент технологии Java Management Extensions (JMX)
JAR-файл plugin	Библиотеки технологии Java Plug-in
JAR-файл resources	Ресурсные файлы среды выполнения
JAR-файл rt	Базовые библиотеки Java Application Programming Interface (API) платформы Java

Структура файловой системы комплекта разработки Java Development Kit (JDK)

Таблица П1.3. *Инструменты разработки Java-приложений, расположенные в папке* bin

Инструменты	Описание
`apt`	Инструмент обработки аннотаций
`extcheck`	Проверяет JAR-файл на конфликт с уже инсталлированным JAR-файлом расширения платформы Java. Инструмент позволяет перед инсталляцией JAR-файла расширения проверить его на свежесть версии по сравнению с уже инсталлированным JAR-файлом

Таблица П1.3 *(продолжение)*

Инструменты	Описание
`HtmlConverter`	Конвертирует HTML-страницу, содержащую апплет, в формат определенного Java Plug-in, что позволяет настроить запуск апплета в конкретном браузере
`idlj`	Генерирует Java-классы из файла IDL (язык описания интерфейсов). Инструмент `idlj` является важной частью технологии Java IDL, которая интегрирует платформу Java с технологией CORBA (Common Object Request Broker Architecture)
`jar`	Производит все необходимые операции с JAR-файлами, включая их создание и обновление
`jarsigner`	Средство контроля и создания электронной подписи для JAR-файлов
`javac`	Компилирует исходный Java-код в байт-код
`javadoc`	Генерирует документацию из комментариев, содержащихся в исходном коде
`javah`	Создает заголовочные и исходные C-файлы из Java-класса, необходимые для обращения из Java-кода к C-коду, является инструментом механизма Java Native Interface (JNI)
`javap`	Декомпилятор байт-кода
`jconsole`	Графический интерфейс Java Management Extensions (JMX) контроля работы виртуальной машины JVM, позволяет контролировать как локальную, так и удаленную виртуальную машину
`jdb`	Используется для отладки Java-приложений
`jhat`	Java Heap Analysis Tool — средство анализа использования памяти при работе приложения, служит для поиска утечек памяти
`jinfo`	Выводит конфигурационную информацию работающего Java-процесса, файла памяти или удаленного сервера отладки
`jmap`	Выводит статистику использования памяти
`jps`	Выводит список виртуальных машин указанной с помощью URI-адреса системы
`jrunscript`	Используется для запуска скриптов. По умолчанию язык написания скриптов — JavaScript
`jstack`	Выводит состояние Java-потоков, используется для поиска заблокированных потоков
`jstat`	JVM Statistics Monitoring Tool — средство диагностики производительности и потреблении ресурсов для указанной виртуальной машины
`jstatd`	JVM jstat Daemon — RMI-приложение, позволяющее подключаться удаленным инструментам мониторинга к локальной виртуальной машине
`jvisualvm`	Графический интерфейс, предоставляющий детальную информацию о работающих Java-приложениях
`native2ascii`	Конвертирует не Latin-1 и не Unicode-символы, содержащиеся в файле, в символы кодировки Unicode для того, чтобы файл мог быть обработан инструментами Java

Таблица П1.3 *(окончание)*

Инструменты	Описание
`packager`	JavaBean Packager — обеспечивает использование компонентов JavaBeans в качестве ActiveX-компонентов, создавая для компонента регистрационный и библиотечный файлы REG и TLB
`rmic`	Генерирует классы заглушек и файл IDL для удаленных объектов
`schemagen`	Генерирует XML-схему для Java-файлов
`serialver`	Выводит идентификатор версии для одного или нескольких классов
`wsgen`	Генерирует классы заглушек и файл WSDL для Web-сервисов JAX-WS
`wsimport`	Обрабатывает Web Services Description Language (WSDL) файл и генерирует требуемые артефакты для Web-сервиса JAX-WS
`xjc`	JAXB-компилятор — генерирует из XML-схемы Java-классы

Таблица П1.4. *Другие папки JDK*

Папки	Описание
Папка include	Содержит заголовочные C-файлы, обеспечивающие поддержку технологий Java Native Interface и Java Virtual Machine Debugger Interface
Папка jre	Содержит среду выполнения JRE
Папка lib	Содержит библиотеки классов, используемые инструментами разработки
Папка sample	Содержит примеры использования Java-технологий

ПРИЛОЖЕНИЕ 2

Структура интерфейса программирования платформы Java SE

Таблица П2.1. *Базовые библиотеки и технологии платформы Java SE*

Библиотеки и технологии	Описание
Библиотеки ввода/вывода I/O	Обеспечивают систему ввода/вывода, сериализацию объектов, доступ к файловой системе
Библиотека Networking	Обеспечивает сетевое взаимодействие
Библиотека Security	Представляет систему безопасности платформы, включая аутентификацию/авторизацию, криптографическую защиту, поддержку цифровых подписей и безопасных протоколов
Библиотека интернационализации Internationalization	Позволяет адаптировать приложения для различных языков и регионов
Технология JavaBeans Component	Позволяет разрабатывать компоненты, основанные на архитектуре JavaBeans
Технология Java Management Extensions (JMX)	Обеспечивает мониторинг и управление ресурсами
Технология XML (JAXP)	Позволяет обрабатывать XML-документы и данные
Технология Java Native Interface (JNI)	Обеспечивает взаимодействие Java-кода с C-кодом
Технология расширения Extension Mechanism	Используется для расширения функциональности платформы
Технология Endorsed Standards Override Mechanism	Обеспечивает механизм включения новых версий классов и интерфейсов, созданных вне процесса Java Community Process, в платформу Java

Таблица П2.2. *Интегрированные библиотеки и технологии платформы Java SE*

Библиотеки и технологии	Описание
Технология Remote Method Invocation (RMI)	Позволяет разрабатывать распределенные приложения
Технология Java IDL (CORBA)	Интегрирует технологию CORBA (Common Object Request Broker Architecture) с платформой Java
Технология RMI-IIOP	Обеспечивает поддержку протокола Java Remote Method Invocation over Internet Inter-ORB Protocol
Технология Scripting for the Java Platform	Позволяет выполнять различного рода скрипты
Технология Java Naming and Directory Interface (JNDI)	Обеспечивает поиск объектов по их именам в среде выполнения

Таблица П2.3. *Библиотеки и технологии интерфейса пользователя платформы Java SE*

Библиотеки и технологии	Описание
Технология Accessibility	Позволяет разрабатывать приложения для людей с ограниченными возможностями
Технология Print Service	Представляет сервис печати
Технология Sound	Обеспечивает ввод, обработку и проигрывание аудио- и MIDI-файлов (Musical Instrument Digital Interface)
Технология Drag and Drop Data Transfer	Облегчает передачу данных между приложениями и внутри приложения
Технология Image I/O	Обеспечивает обработку изображений
Технология Java 2D Graphics and Imaging	Позволяет работать с 2D-графикой и изображениями
Графические библиотеки AWT и Swing	Предназначены для создания графического интерфейса пользователя

ПРИЛОЖЕНИЕ 3

Проекты Eclipse

- **Business Intelligence and Reporting Tools (BIRT)** — система формирования отчетов данных для Web-приложений.
- **Data Tools Platform (DTP)** — расширяемая платформа для создания приложений, ориентированных на работу с базами данных.
- **Device Software Development Platform (DSDP)** — расширяемая платформа для разработки программного обеспечения для различных устройств, включающая в себя разные независимые технологии, ориентирующиеся на распределенные и мобильные системы.
- **Eclipse Project** — платформа для создания высокоинтегрированных инструментов и насыщенных клиентских приложений, состоит из следующих подпроектов:
  - **Platform** — подпроект объединяет платформы и сервисы, составляющие ядро Eclipse Project, на основе которого создаются все остальные модульные расширения. Ядро, которое является основой для создания различных инструментов и приложений, состоит из:
    - **Workspaces** — рабочая область, которая является контейнером для проектов, обеспечивая доступ и управление локальными и удаленными ресурсами.
    - **Workbench** — рабочее пространство, представляющее собой пользовательский интерфейс (UI) Eclipse Platform, построено на основе двух наборов инструментов — SWT и JFace. SWT — это графическая библиотека, содержащая компоненты для создания пользовательского интерфейса, тесно интегрированная с операционной системой. SWT определяет общий интерфейс API, который доступен для различных операционных систем. Для каждой операционной системы реализация SWT использует соответствующие графические элементы. В случае их отсутствия реализация SWT обеспечивает подходящую эмуляцию. JFace — это платформно-независимый набор инструментов для разработки графического интерфейса приложения, построенный на основе на SWT и реализующий модель MVC (Model — View — Controller). Workbench базируется на редакторах, видах и перспективах. Редакторы позволяют пользователю открывать, редактировать и со-

хранять объекты. Виды используются для отображения информации об активном объекте, с которым пользователь работает в рабочем пространстве. Окно рабочего пространства может иметь несколько отдельных перспектив, при этом только одна из них видима в любой данный момент. Каждая перспектива имеет собственные редакторы и виды. Возможно расширение Workbench путем добавления новых редакторов, видов и перспектив или расширением существующих.

- **Team** — сервис, позволяющий размещать проекты в рабочей области согласно их версиям и связывать проекты с соответствующими хранилищами, которые хранят изменения в проекте. В случае необходимости возможно подключение новых хранилищ. Данный сервис позволяет организовывать командную работу над проектом, при этом результаты работы членов команды становятся совместно используемыми с помощью обмена через хранилище.
- **Help** — сервис, реализующий справочную систему для подключенного набора инструментов, содержит механизм, позволяющий инструментам добавлять документацию в виде онлайновых книг.

Кроме того, Platform содержит модуль **Platform Runtime**, который при запуске определяет набор доступных плагинов, читает их файлы манифеста и создает реестр информации о них. При этом плагины загружаются динамически, по требованию, что сокращает время старта и использование оперативной памяти. Также производится проверка соответствия по именам объявлений расширений соответствующим им объявлениям точек расширения. Все обнаруженные при этом конфликты протоколируются. Результирующий реестр плагинов доступен через Platform API. Плагины могут добавляться, заменяться или удаляться уже после запуска.

- **Eclipse Java development tools (JDT)** — подпроект представляет собой инструментальный плагин, который реализует интегрированную среду разработки Java. Данная среда поддерживает разработку Java-приложений всех типов, включая создание плагинов для Eclipse. При загрузке плагина JDT в рабочее пространство Eclipse добавляется тип проекта Java и Java-перспектива, содержащая виды, редакторы, мастера, генерирующий код, сборщика и инструменты объединения и рефакторинга кода. Подпроект состоит в свою очередь из модулей: JDT APT, который обеспечивает поддержку аннотаций; JDT Core — компилятор, включающий инкрементальную сборку, т. е. компиляцию сделанных изменений проекта при его сохранении; JDT Debug — обеспечивает выполнение и отладку кода; JDT Text — редактор исходного кода Java с выделением ключевых слов и синтаксиса, форматированием и завершением кода, контекстной помощью и т. д.; JDT UI — добавляет специфические для Java элементы в рабочее пространство Eclipse.
- **PDE — Plugin Development Environment** является расширением JDT и обеспечивает инструментарий для разработки, отладки, тестирования и установки плагинов Eclipse, а также других расширений, основанных на технологии OSGi.

- **e4 Project** — подпроект, предназначен для создания Eclipse Platform следующего поколения.

- **Eclipse Modeling Project** объединяет ряд технологий моделирования и предоставляет набор платформ и инструментов моделирования.
- **Eclipse RT Project** — проект, поддерживающий различные runtime-технологии, такие как: Eclipse Communication Framework — платформа для разработки распределенных инструментов и приложений, основанных на Eclipse; EclipseLink — платформа, обеспечивающая работу практически с любым типом данных; Equinox — набор пакетов, реализующих OSGi-сервисы и обеспечивающих инфраструктуру для выполнения систем, базирующихся на технологии OSGi; embedded Rich Client Platform (eRCP) — расширение Eclipse Rich Client Platform (RCP) для встраиваемых устройств; Rich Ajax Platform (RAP) — дает возможность создавать AJAX-приложения, используя инструменты Eclipse; Riena Platform Project — платформа для создания многоуровневых корпоративных клиент-серверных приложений; SMILA — расширяемая платформа, обеспечивающая работу с неструктурированной корпоративной информацией; Swordfish SOA Runtime Framework Project — реализация расширяемой SOA-платформы, базирующейся на технологии Equinox.
- **SOA Tools Platform Project** — проект, объединяющий платформы и инструменты для планирования, конфигурирования, сборки, развертывания, мониторинга и управления программного обеспечения, созданного на основе Service Oriented Architecture (SOA).
- **Eclipse Technology Project** — проект, состоящий из трех разделов: академические научные исследования (Research Stream); учебные материалы (Education Stream); инкубатор для небольших проектов, добавляющих новые возможности программному обеспечению (Incubators Stream).
- **Tools Project** — проект, предназначенный для развития широкого диапазона инструментов программирования, таких как C/C++ IDE, COBOL IDE, PHP IDE, инструментов для аспектно-ориентированного программирования (Aspect-Oriented Programming) и др.
- **Test and Performance Tools Platform Project** — проект, содержащий платформы и сервисы, предназначенные для разработки тестов программного обеспечения.
- **Eclipse Web Tools Platform Project** — проект, объединяющий инструменты для разработки приложений Web и Java EE.

ПРИЛОЖЕНИЕ 4

Основные библиотеки NetBeans API

Таблица П4.1. *Основные библиотеки NetBeans API*

Название библиотеки	**Описание**
Actions API	Обеспечивает управление системными действиями, такими как копирование и удаление, доступными из меню, панели управления, с помощью "горячих" клавиш или их комбинации
Auto Update Services API	Обеспечивает загрузку, инсталляцию и деинсталляцию, подключение и отключение модулей, обновляющих или добавляющих новую функциональность приложению
Command Line Parsing API	С помощью данной библиотеки модули могут производить синтаксический анализ командной строки при запуске приложения с различными опциями
Common Palette API	Обеспечивает доступ к общей палитре графических компонентов, которые должны отражаться в редакторе кода
Datasystems API	Позволяет работать с объектами `DataObject`, которые представляют собой один или несколько файлов файловой системы вместе с их характеристиками. Эта библиотека используется для работы с такими типами данных, как файлы UML-моделей, шаблонами, логическими группировками физических файлов и т. д.
Dialogs API	Библиотека классов для обработки сообщений пользователя, отображения диалоговых окон и окон мастеров
Editor Code Completion API	Обеспечивает автозавершение кода в редакторе
Explorer & Property Sheet API	Библиотека для визуального представления иерархии узлов и их свойств в таких структурах, как деревья, списки, комбинированные списки, таблицы и меню
File System API	Библиотека предназначена для управления виртуальной файловой системой, которая обеспечивает доступ к объектам класса `FileObject`, объединяющего информацию об объектах файловой системы и операциях над ними

Таблица П4.1 *(окончание)*

Название библиотеки	**Описание**
I/O APIs	Обеспечивает доступ к окну стандартного вывода (Output window), включая возможность создания гиперссылок, связанных со строками редактора
JavaHelp Integration API	Используется для добавления справочной информации в модули и для отображения справочной информации
Lexer API	Позволяет создавать последовательности лексем для различных входящих источников. Используется, например, для организации подсветки синтаксиса в редакторе, автозавершения кода, организации гиперссылок в редакторе
Module System API	Обеспечивает механизм динамического расширения приложений посредством подключения модулей, представляющих собой JAR-файлы, содержащие расширения классов, при этом файл манифеста модуля определяет инсталляцию модуля
Navigator API	Предназначается для представления структуры документов в специальном окне, обеспечивая быструю навигацию и контроль документов
Nodes API	Контролирует использование и создание узлов (nodes). Узлы используются для визуального представления различных объектов, формируемых в иерархическое дерево, при этом узлы не хранят реальные данные. Узлы являются расширением концепции JavaBeans, имея дополнительные черты, такие как полная поддержка иерархии, обеспечение действий и cookies, динамическая модификация набора свойств, поддерживаемых объектом и др.
Options Dialog and SPI	Позволяет инсталлировать дополнительные разделы в диалоговом окне **Options**
Progress API	Обеспечивает визуализацию прогресса для исполняемых в течение длительного времени задач с помощью специального компонента, интегрированного в правом нижнем углу графического интерфейса приложения
Task List API	Обеспечивает специальное окно для отображения задач или проблем, таких как ошибки и замечания, генерируемые подключаемыми модулями для выделенных файлов и/или папок
UI Utilities API	Библиотека классов, определяющих графический пользовательский интерфейс приложения
Utilities API	Модуль, содержащий классы, которые определяют общую инфраструктуру приложения
Visual Library API	Обеспечивает набор стандартных, повторно используемых, графических элементов (widgets) пользовательского интерфейса (кнопка, ползунок, окно редактирования и т. д.) вместе с их свойствами, такими как компоновка, рамки, определенные действия и т. д.
Window System API	Позволяет модулям создавать window-подобные компоненты, при этом окна не могут создаваться непосредственно, а только с помощью класса `org.openide.windows.TopComponent`. Создаваемые окна могут быть вложенными, иметь меню, управление активацией, выбором узлов, копированием и др.

ПРИЛОЖЕНИЕ 5

Спецификации платформы Java SE

Таблица П5.1. *Спецификации платформы Java SE*

Спецификация	Описание
Java API for XML Parsing (JAXP) (JSR 5)	Спецификация позволяет производить анализ XML-документов, используя XML-анализаторы, являющиеся реализацией данной спецификации. XML-документы обеспечивают кросс-платформенный формат данных. Технология Java является кросс-платформенной технологией, поэтому естественно, что технология Java широко использует формат XML для работы с данными
Unified Printing API (Java Print Service API) (JSR 6)	Спецификация описывает сервис печати, который позволяет настраивать параметры печати и выводить на печать тексты и изображения
Preferences API (JSR 10)	Спецификация описывает способ сохранения и применения конфигурационных данных Java-приложения, позволяющих адаптировать приложения для различного рода пользователей и сред выполнения. Реализации спецификации могут быть разными, как правило, конфигурационные данные хранятся в XML-файлах каталога пользователя — таким образом, обеспечивается простое и надежное управление ими. Хранение конфигурационных данных может также осуществляться и в системных реестрах, и в базах данных — это зависит от конкретной реализации данной спецификации. Альтернативой использования Preferences API является применение Properties API и JNDI API. Однако использование Properties API подразумевает хранение конфигурационных данных в ресурсных файлах, расположенных в произвольном месте файловой системы компьютера, что увеличивает риски их повреждения. Кроме того, Properties API поддерживает только строковые типы данных, а Preferences API обеспечивает хранение строк, чисел, логических значений, простых массивов. Технология Preferences API близка к технологии JNDI API, но по сравнению с ней является более простым способом управления конфигурационными данными, обеспечивая хранение и извлечение небольшого количества данных, представляющих пользовательские и системные установки работы приложения
Decimal Arithmetic Enhancement (JSR 13)	Спецификация добавляет вычисления с десятичными числами с плавающей запятой, улучшая точность вычислений и позволяя избежать ошибок округления

Таблица П5.1 *(продолжение)*

Спецификация	Описание
Add Generic Types To The Java Programming Language (JSR 14)	Спецификация расширяет язык программирования Java обобщенными или параметризируемыми типами
Image I/O Framework Specification (JSR 15)	Спецификация обеспечивает модульную архитектуру для загрузки и сохранения изображений. Модульность выражается в том, что набор поддерживаемых форматов изображений не ограничен. Стандартная реализация поддерживает GIF-, JPEG- и PNG-форматы изображений. Для поддержки других форматов можно добавить соответствующие плагины, например библиотеку Java Advanced Imaging Image I/O Tools. Технология Image I/O может использоваться как клиентскими приложениями для загрузки изображений из файловой системы компьютера или через Интернет, так и серверными приложениями для динамической генерации изображений в ответ на клиентский запрос
A Simple Assertion Facility (JSR 41)	Спецификация позволяет использовать утверждения (assertion), содержащие логические выражения. При вычислении такого выражения, если выражение не верно, система выдает ошибку. Данная конструкция используется при отладке и тестировании приложений
Logging API Specification (JSR 47)	Спецификация позволяет генерировать отчеты об ошибках работы приложения для последующего анализа конечными пользователями, системными администраторами и разработчиками
New I/O APIs for the Java Platform (JSR 51)	Спецификация обеспечивает низкоуровневый доступ к операциям ввода/вывода, включая работу с буфером, каналами, кодировкой/декодировкой символов и др.
JDBC 3.0 Specification (JSR 54)	Спецификация обеспечивает программный доступ к реляционным базам данных
Certification Path API (JSR 55)	Спецификация расширяет систему безопасности Java возможностью управления цепочками сертификатов
Java Network Launching Protocol and API (JSR 56)	Спецификация описывает технологию развертывания Java-приложений Java Web Start
Long-Term Persistence for JavaBeans Specification (JSR 57)	Спецификация обеспечивает поддержку механизма сериализации/десериализации компонентов JavaBeans на основе XML-формата
J2SE Merlin Release Contents (JSR 59)	Спецификация платформы Java SE версии 1.4 под кодовым названием Merlin
Java GSS API (JSR 72)	Java Generic Security Services API обеспечивает защиту обмена сообщениями между приложениями, используя механизм Kerberos V5
Java Platform Profiling Architecture (JSR 163)	Спецификация позволяет контролировать выполнение приложений и извлекать различного рода статистику, относящуюся к работающей виртуальной машине JVM
Concurrency Utilities (JSR 166)	Спецификация позволяет производить параллельные вычисления, предоставляя пул потоков, платформу для асинхронного выполнения задач, высокоточное определение времени и др.

Таблица П5.1 *(продолжение)*

Спецификация	Описание
Monitoring and Management Specification for the Java Virtual Machine (JSR 174)	Спецификация обеспечивает мониторинг и управление виртуальной машиной JVM, позволяя контролировать и наблюдать загрузку/выгрузку классов, распределение памяти, сборку мусора, создание потоков и др.
A Metadata Facility for the Java Programming Language (JSR 175)	Спецификация позволяет определять новые типы аннотаций и маркировать ими поля, методы и классы, при этом инструменты разработки и развертывания производят обработку аннотаций, генерируя дополнительный Java-код, документы и другие артефакты, влияющие на выполнение программы
J2SE 5.0 (Tiger) Release Contents (JSR 176)	Спецификация платформы Java SE версии 5.0 под кодовым названием Tiger
Java Compiler API (JSR 199)	Спецификация позволяет компилировать программным образом исходный Java-код
Network Transfer Format for Java Archives (JSR 200)	Спецификация определяет формат архивирования Pack200, используемый для загрузки Java-приложений. Формат Pack200 обеспечивает дополнительное сжатие по сравнению с форматом JAR
Extending the Java Programming Language with Enumerations, Autoboxing, Enhanced for loops and Static Import (JSR 201)	Спецификация определяет для языка Java новые свойства — перечисления, автоматическую конвертацию/деконвертацию примитивных типов данных в соответствующие оберточные типы, улучшенный цикл `for`, использование конструкции `import static` и переменного списка аргументов
Java Class File Specification Update (JSR 202)	Спецификация представляет обновление формата класс-файла, содержащего байт-код
More New I/O APIs for the Java Platform ("NIO.2") (JSR 203)	Спецификация представляет улучшенный доступ к файловой системе, обновленную функциональность сокет-каналов и асинхронные операции ввода/вывода для сокетов и файлов
Unicode Supplementary Character Support (JSR 204)	Спецификация обеспечивает поддержку дополнительных символов спецификации Unicode 3.1
Java API for XML Processing (JAXP) 1.3 (JSR 206)	Следующая версия технологии JAXP для обработки XML-документов
Java Business Integration (JBI) (JSR 208)	Спецификация расширяет платформу Java SE реализацией архитектуры SOA (Service-Oriented Architecture), позволяя создавать модульные приложения на основе модели Web-сервисов
Common Annotations for the Java Platform (JSR 250)	Спецификация представляет набор аннотаций, созданных на основе спецификации JSR 175
Java Smart Card I/O API (JSR 268)	Спецификация позволяет Java-приложениям взаимодействовать с приложениями, работающими на смарт-картах

Таблица П5.1 *(окончание)*

Спецификация	Описание
Pluggable Annotation Processing API (JSR 269)	Спецификация позволяет создавать новые обработчики аннотаций
Java SE 6 Release Contents (JSR 270)	Спецификация платформы Java SE версии 6
Java Module System (JSR 277)	Спецификация описывает архитектуру создания и развертывания модульных приложений и библиотек. Основные элементы архитектуры — Java-модуль и его метаданные, схема версий модуля, хранилище модулей и среда выполнения модулей
Supporting Dynamically Typed Languages on the Java Platform (JSR 292)	Спецификация вводит поддержку динамической генерации типов данных, основанную на данных, получаемых при работе приложения
Improved Modularity Support in the Java Programming Language (JSR 294)	Дальнейшее развитие модульной системы Java (JSR 277). Спецификация вводит понятие суперпакета, описывая его синтаксис и использование
Post mortem JVM Diagnostics API (JSR 326)	Спецификация обеспечивает доступ к данным, содержащимся в артефактах, сгенерированных после неудачного завершения работы приложения
Java Language Specification (JSR 901)	Спецификация языка программирования Java
J2SE 1.4 (Merlin) Maintenance Review (JSR 915)	Изменения версии 1.4 платформы Java SE
J2SE 1.4 (Merlin) Beta 2 Maintenance Review (JSR 916)	Изменения версии 1.4 платформы Java SE
J2SE 1.4 (Merlin) Beta 3 Maintenance Review (JSR 917)	Изменения версии 1.4 платформы Java SE
J2SE 1.4 Release Candidate (JSR 918)	Основные черты платформы Java SE версии 1.4
J2SE 1.4.2 (Mantis) Beta (JSR 923)	Изменения версии 1.4 платформы Java SE
Java Virtual Machine Specification (JSR 924)	Обзор изменений спецификации виртуальной машины JVM для платформы J2SE 1.5

ПРИЛОЖЕНИЕ 6

Структура графической библиотеки Swing

Таблица П6.1. *Пакеты классов и интерфейсов библиотеки Swing API*

Пакет	Описание
`javax.accessibility`	Библиотека классов и интерфейсов Java Accessibility API (JAAPI), позволяющая создавать приложения для людей с ограниченными возможностями. Библиотека JAAPI обеспечивает взаимодействие графических компонентов интерфейса пользователя с такими технологиями доступа, как чтение с экрана и системы распознавания речи
`javax.swing`	Библиотека представляет набор Swing-компонентов интерфейса пользователя, а также вспомогательные классы и интерфейсы
`javax.swing.border`	Классы и интерфейсы для создания рамок Swing-компонентов
`javax.swing.colorchooser`	Классы и интерфейсы, используемые классом `javax.swing.JColorChooser`, который представляет панель выбора цвета
`javax.swing.event`	Классы и интерфейсы модели обработки событий Swing, дополняющие AWT-библиотеку `java.awt.event`
`javax.swing.filechooser`	Классы и интерфейсы, используемые классом `javax.swing.JFileChooser`, который представляет графический интерфейс навигации файловой системы компьютера
`javax.swing.plaf`	Библиотека, обеспечивающая расширяемые возможности Look and Feel (внешний вид/поведение) графического интерфейса Swing
`javax.swing.plaf.basic`	Реализация библиотеки `javax.swing.plaf`, представляющая базовый внешний вид/поведение графического интерфейса Swing
`javax.swing.plaf.metal`	Реализация библиотеки `javax.swing.plaf`, представляющая внешний вид/поведение графического интерфейса Swing по умолчанию, под именем *Metal*

Таблица П6.1 *(окончание)*

Пакет	Описание
`javax.swing.plaf.multi`	Реализация библиотеки `javax.swing.plaf`, позволяющая дополнять основной Look and Feel одним или несколькими вспомогательными Look and Feel. Например, вид/поведение *Metal* — поддержкой трансформации текста в речь
`javax.swing.plaf.synth`	Реализация библиотеки `javax.swing.plaf`, позволяющая создавать новые Look and Feel с помощью файлов метаданных
`javax.swing.table`	Библиотека для работы с классом `javax.swing.JTable`, представляющим таблицу. Классы и интерфейсы библиотеки позволяют контролировать создание, обновление и отображение таблиц, включая отображение и управление данными таблиц
`javax.swing.text`	Библиотека для работы с такими текстовыми компонентами, как текстовые поля и области отображения текста. Классы и интерфейсы библиотеки обеспечивают выделение текста, редактирование и др.
`javax.swing.text.html`	Библиотека обеспечивает поддержку HTML-формата для класса `javax.swing.JEditorPane`, представляющего текстовый компонент для редактирования различного типа содержимого. Библиотека позволяет создавать редакторы HTML-документов
`javax.swing.text.html.parser`	Пакет обеспечивает анализатор HTML-формата по умолчанию
`javax.swing.text.rtf`	Библиотека обеспечивает поддержку RTF-формата для класса `javax.swing.JEditorPane`, представляющего текстовый компонент для редактирования различного типа содержимого. Библиотека позволяет создавать редакторы RTF-документов
`javax.swing.tree`	Библиотека для работы с классом `javax.swing.JTree`, представляющим графический компонент для отображения деревьев (наборов иерархических данных). Классы и интерфейсы библиотеки позволяют контролировать создание, обновление и отображение деревьев, включая отображение и управление связанных с ними данных
`javax.swing.undo`	Библиотека обеспечивает поддержку функциональности undo/redo (перевод в предыдущее состояние) для таких приложений, как текстовые редакторы

ПРИЛОЖЕНИЕ 7

Коллекция классов пакета *java.io*

Таблица П7.1. *Классы пакета* `java.io`

Класс	Иерархия наследования	Функциональность

`BufferedInputStream`	`java.lang.Object`
	`└java.io.InputStream`
	`└java.io.FilterInputStream`
	`└java.io.BufferedInputStream`
		Класс организует буфер в оперативной памяти при чтении байтового потока

`BufferedOutputStream`	`java.lang.Object`
	`└java.io.OutputStream`
	`└java.io.FilterOutputStream`
	`└java.io.BufferedOutputStream`
		Класс организует буфер в оперативной памяти при записи байтового потока

`BufferedReader`	`java.lang.Object`
	`└java.io.Reader`
	`└java.io.BufferedReader`
		Класс организует буфер в оперативной памяти при чтении символьного потока

`BufferedWriter`	`java.lang.Object`
	`└java.io.Writer`
	`└java.io.BufferedWriter`
		Класс организует буфер в оперативной памяти при записи символьного потока

`ByteArrayInputStream`	`java.lang.Object`
	`└java.io.InputStream`
	`└java.io.ByteArrayInputStream`
		Класс организует чтение из массива байтов, при этом массив уже существует в оперативной памяти

`ByteArrayOutputStream`	`java.lang.Object`
	`└java.io.OutputStream`
	`└java.io.ByteArrayOutputStream`
		Класс организует запись в массив байтов, при этом массив уже существует в оперативной памяти

`CharArrayReader`	`java.lang.Object`
	`└java.io.Reader`
	`└java.io.CharArrayReader`
		Класс организует чтение из массива символов, при этом массив уже существует в оперативной памяти

`CharArrayWriter`	`java.lang.Object`
	`└java.io.Writer`
	`└java.io.CharArrayWriter`
		Класс организует запись в массив символов, при этом массив уже существует в оперативной памяти

Таблица П7.1 *(продолжение)*

Класс	Иерархия наследования	Функциональность
`Console`	`java.lang.Object` `└java.io.Console`	Класс организует ввод и вывод в консольном приложении Java

`DataInputStream`	`java.lang.Object`
	`└java.io.InputStream`
	`└java.io.FilterInputStream`
	`└java.io.DataInputStream`
		Класс используется для чтения примитивных типов данных `int`, `char`, `long` и т. д. в машинно-независимом формате

`DataOutputStream`	`java.lang.Object`
	`└java.io.OutputStream`
	`└java.io.FilterOutputStream`
	`└java.io.DataOutputStream`
		Класс используется для записи примитивных типов данных `int`, `char`, `long` и т. д. в машинно-независимом формате

`File`	`java.lang.Object` `└java.io.File`	Класс для работы с файлами и каталогами в файловой системе
`FileDescriptor`	`java.lang.Object` `└java.io.FileDescriptor`	Класс, позволяющий протестировать правильность описания файла, сокета и т. д., с участием которых создан поток, а также синхронизировать с ними все существующие буферы в оперативной памяти. Внутренний класс для использования другими классами при работе с файлами

`FileInputStream`	`java.lang.Object`
	`└java.io.InputStream`
	`└java.io.FileInputStream`
		Класс для чтения байтов из файла в файловой системе

`FileOutputStream`	`java.lang.Object`
	`└java.io.OutputStream`
	`└java.io.FileOutputStream`
		Класс для записи байтов в файл файловой системы

`FilePermission`	`java.lang.Object`
	`└java.security.Permission`
	`└java.io.FilePermission`
		Класс используется для управления доступом к файлам

`FileReader`	`java.lang.Object`
	`└java.io.Reader`
	`└java.io.InputStreamReader`
	`└java.io.FileReader`
		Класс для чтения символов из файла в файловой системе

`FileWriter`	`java.lang.Object`
	`└java.io.Writer`
	`└java.io.OutputStreamWriter`
	`└java.io.FileWriter`
		Класс для записи символов в файл файловой системы

Таблица П7.1 *(продолжение)*

Класс	Иерархия наследования	Функциональность
FilterInputStream	java.lang.Object └java.io.InputStream └java.io.FilterInputStream	Базовый класс для классов: DataInputStream, BufferedInputStream, LineNumberInputStream, PushbackInputStream. Самостоятельно класс не используется
FilterOutputStream	java.lang.Object └java.io.OutputStream └java.io.FilterOutputStream	Базовый класс для классов: DataOutputStream, BufferedOutputStream, PrintStream. Самостоятельно класс не используется
FilterReader	java.lang.Object └java.io.Reader └java.io.FilterReader	Базовый класс для класса PushbackReader. Самостоятельно класс не используется
FilterWriter	java.lang.Object └java.io.Writer └java.io.FilterWriter	На базе классов FilterWriter и FilterReader возможно создание классов, предназначенных для фильтрации или обработки символьных потоков
InputStream	java.lang.Object └java.io.InputStream	Суперкласс для всех классов, организующих входящие байтовые потоки
InputStreamReader	java.lang.Object └java.io.Reader └java.io.InputStreamReader	Класс, являющийся мостом между байтовыми потоками и символьными потоками. Организует чтение байтов из входного потока и преобразование их в символы, используя указанную или принятую по умолчанию кодировку
LineNumberInputStream	java.lang.Object └java.io.InputStream └java.io.FilterInputStream └java.io.LineNumberInputStream	Данный класс больше не используется
LineNumberReader	java.lang.Object └java.io.Reader └java.io.BufferedReader └java.io.LineNumberReader	Класс позволяет отслеживать нумерацию строк во входном потоке
ObjectInputStream	java.lang.Object └java.io.InputStream └java.io.ObjectInputStream	Класс осуществляет обратный процесс после сериализации примитивных данных и объектов, предварительно записанных, используя класс ObjectOutputStream. Сериализация — это особый вид чтения и записи, при котором обеспечивается сохранность данных или объектов. Это необходимо при передаче по сети или использовании архивированного объекта при повторном вызове программы

Таблица П7.1 *(продолжение)*

Класс	Иерархия наследования	Функциональность
`ObjectInputStream.GetField`	`java.lang.Object` `└java.io.ObjectInputStream.GetField`	Класс используется для десериализации объекта при чтении значений именованных полей указанных типов

`ObjectOutputStream`	`java.lang.Object`
	`└java.io.OutputStream`
	`└java.io.ObjectOutputStream`
		Класс используется для сериализации объекта, при этом объект должен реализовывать интерфейс `java.io.Serializable`

`ObjectOutputStream.PutField`	`java.lang.Object` `└java.io.ObjectOutputStream.PutField`	Класс используется для сериализации объекта, при записи значений именованных полей, что позволяет четко контролировать процесс сериализации
`ObjectStreamClass`	`java.lang.Object` `└java.io.ObjectStreamClass`	Класс содержит информацию о классе, проходящем сериализации. Это имя и версия класса, имя и тип полей
`ObjectStreamField`	`java.lang.Object` `└java.io.ObjectStreamField`	Класс содержит информацию об именованных полях указанного типа при сериализации класса с записью набора полей отличных от полей собственной реализации
`OutputStream`	`java.lang.Object` `└java.io.OutputStream`	Суперкласс для всех классов, представляющих исходящие байтовые потоки

`OutputStreamWriter`	`java.lang.Object`
	`└java.io.Writer`
	`└java.io.OutputStreamWriter`
		Класс, являющийся мостом между символьными потоками и байтовыми потоками. Организует преобразование символов в байты, используя указанную или принятую по умолчанию кодировку, а затем записывает их в выходной поток

`PipedInputStream`	`java.lang.Object`
	`└java.io.InputStream`
	`└java.io.PipedInputStream`
		Класс для организации чтения байтов в канальном потоке. Канальный поток — это поток, созданный для вывода данных из одной программы и ввода их в другую программу. Должен работать вместе с соответствующим классом `PipedOutputStream`

`PipedOutputStream`	`java.lang.Object`
	`└java.io.OutputStream`
	`└java.io.PipedOutputStream`
		Класс для записи байтов в канальном потоке

Таблица П7.1 *(продолжение)*

Класс	Иерархия наследования	Функциональность
PipedReader	java.lang.Object └java.io.Reader └java.io.PipedReader	Класс для чтения символов в канальном потоке
PipedWriter	java.lang.Object └java.io.Writer └java.io.PipedWriter	Класс для записи символов в канальном потоке
PrintStream	java.lang.Object └java.io.OutputStream └java.io.FilterOutputStream └java.io.PrintStream	Класс используется для печати примитивных типов данных
PrintWriter	java.lang.Object └java.io.Writer └java.io.PrintWriter	Класс используется для печати символов
PushbackInputStream	java.lang.Object └java.io.InputStream └java.io.FilterInputStream └java.io.PushbackInputStream	Класс реализует буфер для возврата в поток прочитанных байтов. Класс используется при синтаксическом анализе
PushbackReader	java.lang.Object └java.io.Reader └java.io.FilterReader └java.io.PushbackReader	Класс реализует буфер для возврата в поток прочитанных символов. Класс используется при синтаксическом анализе
RandomAccessFile	java.lang.Object └java.io.RandomAccessFile	Класс обеспечивает произвольный доступ к файлу при чтении и записи данных
Reader	java.lang.Object └java.io.Reader	Суперкласс для всех классов, представляющих входящие символьные потоки
SequenceInputStream	java.lang.Object └java.io.InputStream └java.io.SequenceInputStream	Класс используется для сцепления нескольких входящих байтовых потоков в один входящий поток
SerializablePermission	java.lang.Object └java.security.Permission └java.security.BasicPermission └java.io.SerializablePermission	Класс управляет разрешениями и другими правами в процессах сериализации и десериализации
StreamTokenizer	java.lang.Object └java.io.StreamTokenizer	Класс используется для лексического анализа входящего потока и разбивает его на лексемы
StringBufferInputStream	java.lang.Object └java.io.InputStream └java.io.StringBufferInputStream	Этот класс больше не используется

Таблица П7.1 *(окончание)*

Класс	Иерархия наследования	Функциональность

`StringReader`	`java.lang.Object`
	`└java.io.Reader`
	`└java.io.StringReader`
		Класс предназначен для организации символьного потока при чтении из объекта `String`

`StringWriter`	`java.lang.Object`
	`└java.io.Writer`
	`└java.io.StringWriter`
		Класс предназначен для организации исходящего символьного потока, при этом данные накапливаются в буфере, из которого в дальнейшем возможно создание объекта `String`

`Writer`	`java.lang.Object` `└java.io.Writer`	Суперкласс для всех классов, представляющих исходящие символьные потоки

ПРИЛОЖЕНИЕ 8

Справочная система JavaHelp

Справочная система JavaHelp — это открытый проект, позволяющий встроить HTML-справку в Java-приложение.

Адрес домашней странички проекта JavaHelp — **https://javahelp.dev.java.net/**. Страничку компании Sun, посвященной системе JavaHelp, можно найти по адресу **http://java.sun.com/javase/technologies/desktop/javahelp/**.

Справочная система JavaHelp представляет собой интерактивную справочную систему (online help), предназначенную для платформы Java и реализованную на языке Java.

Существует большое количество разнообразных справочных систем, таких как Microsoft WinHelp, AP Help, Oracle Help, JavaHelp и др. Отличие JavaHelp в том, что это платформно-независимая справочная система, написанная на языке Java и предназначенная для интеграции с Java-приложениями.

Для Java-приложений система JavaHelp обеспечивает два типа контекстной справки.

> ***Примечание***
>
> В случае, когда справочная система связана с состояниями приложения и зависит от действий пользователя, она называется *контекстной справочной системой* (context-sensitive help), при этом обеспечивая справку именно для того состояния, с которым она связана. Контекстная справка может быть доступна с помощью появляющейся контекстной подсказки или с помощью нажатия кнопки.

Первый тип контекстной справки системы JavaHelp — это вызываемая пользователем справка. Например, при нахождении фокуса в окне приложения пользователь может вызвать справку о данном окне, нажав клавишу <F1>.

Второй тип контекстной справки системы JavaHelp — это вызываемая системой справка. В этом случае приложение обеспечивает информацию автоматически при выполнении пользователем определенного действия.

JavaHelp дает возможность добавить документацию оперативной помощи к любой системе с Java Runtime Environment (JRE). При помощи JavaHelp можно обеспечить оперативную информацию для Java-компонентов, апплетов и Java-приложений.

Проект JavaHelp состоит из:

- каталога javahelp, содержащего папки bin и lib. Папка bin содержит инструменты `jhindexer` и `jhsearch`, позволяющие в командной строке соответственно создать базу данных для полнотекстового поиска и проверить ее работоспособность. Папка lib содержит папку dtd с DTD-файлами (документируют содержание XML-файлов) для HelpSet-файла, файла содержания, предметного указателя, закладок, навигационного файла, а также JAR-файлы:
  - jh.jar — библиотека содержит все пакеты системы JavaHelp за исключением классов, необходимых для создания базы данных для полнотекстового поиска;
  - jhall.jar — библиотека содержит все пакеты системы JavaHelp (включая классы), необходимые для создания базы данных полнотекстового поиска;
  - jhbasic.jar — библиотека с минимальной конфигурацией, не содержит классы для обеспечения полнотекстового поиска;
  - jsearch.jar — библиотека содержит классы для создания базы данных и реализации полнотекстового поиска;
- каталога demos, содержащего примеры и демонстрационные файлы, а также приложение Help Viewer, представленное файлом hsviewer.jar и позволяющее визуализировать справочную систему JavaHelp;
- каталога doc, содержащего документацию;
- JAR-файла src с исходными файлами JavaHelp API.

Справочная система JavaHelp состоит из данных, которые представлены HTML- и XML-файлами, а также графического интерфейса, который создается на основе библиотеки JavaHelp API.

Библиотека JavaHelp API позволяет осуществлять навигацию, поиск и отображение информации.

Окно графического интерфейса JavaHelp-справки содержит меню, панель инструментов, окно содержания, окно предметного указателя, окно закладок, окно полнотекстового поиска и окно справки.

Файлы HTML-справки, файл содержания, предметный указатель и навигационный файл составляют набор HelpSet, представленный классом `javax.help.HelpSet` библиотеки JavaHelp API.

Файлы HTML-справки набора HelpSet содержат справочную информацию оперативной помощи.

Навигационный файл (map file) набора HelpSet идентифицирует каждый файл HTML-справки с помощью идентификатора `ID` и связывает `ID` с относительным URL-адресом файла HTML-справки. Навигационный файл имеет формат XML и расширение jhm.

Элементы навигационного файла:

- тег `<map>` определяет границы документа;
- тег `<mapID>` связывает индекс файла HTML-справки с его относительным адресом. Атрибут `target` тега `<mapID>` определяет индекс, атрибут `url` — относительный адрес.

Файл содержания (TOC file) набора HelpSet связывает названия разделов содержания справки с индексами файлов HTML-справки и имеет формат XML.

Элементы файла содержания:

- тег `<toc>` определяет границы документа;
- тег `<tocitem>` связывает название раздела содержания с индексом файла HTML-справки. Атрибут `text` тега `<tocitem>` определяет название раздела содержания, атрибут `target` — индекс.

Файл предметного указателя (index file) набора HelpSet связывает содержимое указателя с идентификаторами `ID` файлов HTML-справки и имеет формат XML.

Элементы файла предметного указателя:

- тег `<index>` определяет границы документа;
- тег `<indexitem>` связывает названия элементов предметного указателя с индексами файлов HTML-справки. Атрибут `text` тега `<indexitem>` определяет отображаемое название элемента указателя, атрибут `target` — индекс.

Точкой входа является файл HelpSet, который имеет формат XML и расширение hs. Файл HelpSet указывает относительные URL-адреса навигационного файла, файла содержания, предметного указателя. Кроме того, файл HelpSet указывает классы, ответственные за обработку этих файлов и за реализацию функций полнотекстового поиска и закладок.

Элементы файла HelpSet:

- тег `<helpset>` определяет границы HelpSet-документа;
- тег `<title>` определяет заголовок справочной системы;
- внутри тега `<maps>` содержится информация о навигационном файле;
- тег `<homeID>` определяет идентификатор файла HTML-справки, который отображается по умолчанию;
- тег `<mapref>` указывает относительный URL-адрес навигационного файла;
- внутри тега `<view>` содержится информация об отображаемых областях справочной системы (содержание, указатель и др.);
- тег `<name>` содержит идентификатор области отображения;
- тег `<label>` определяет появляющийся текст, описывающий область отображения;
- тег `<type>` содержит внутри себя имя класса, расширяющего класс `javax.help.NavigatorView` и формирующего данные, которые этот класс визуализирует с помощью класса, расширяющего класс `javax.help.JHelpNavigator` и представляющего графический компонент `JTree` или `JList`;
- тег `<data>` содержит относительный адрес файла данных. Для полнотекстового поиска атрибут `engine` тега `<data>` определяет имя класса, представляющего систему поиска.

Библиотека JavaHelp API предлагает набор классов, расширяющих класс `NavigatorView` и реализующих функции навигации:

- класс `FavoritesView` формирует с использованием файла закладок данные, которые визуализируются с помощью класса `JHelpFavoritesNavigator`. Файл закладок (Favorites) автоматически создается при добавлении первой закладки и хранится в папке JavaHelp каталога пользователя;
- класс `IndexView` формирует с использованием файла предметного указателя данные, которые визуализируются с помощью класса `JHelpIndexNavigator`;
- класс `TOCView` формирует с использованием файла содержания данные, которые визуализируются с помощью класса `JHelpTOCNavigator`;
- класс `SearchView` обеспечивает создание визуального представления данных полнотекстового поиска с помощью класса `JHelpSearchNavigator`, который использует класс, расширяющий абстрактный класс `javax.help.search.SearchEngine`, для формирования данных полнотекстового поиска с помощью базы данных, расположенной по умолчанию в папке JavaHelpSearch.

Библиотека JavaHelp API для взаимодействия с базой данных предлагает класс `com.sun.java.help.search.DefaultSearchEngine`, расширяющий абстрактный класс `SearchEngine`.

Для того чтобы создать базу данных для полнотекстового поиска, необходимо воспользоваться классом `com.sun.java.help.search.Indexer`, который запускается с помощью команды `jhindexer` и индексирует содержимое файлов HTML-справки, сохраняя индексы в файлах папки JavaHelpSearch.

Класс `DefaultSearchEngine` использует файлы папки JavaHelpSearch для поиска информации в файлах HTML-справки, связанной с запросом пользователя.

Класс `Indexer` использует классы `com.sun.java.help.search.DefaultIndexBuilder`, расширяющий абстрактный класс `javax.help.search.IndexBuilder`, и `com.sun.java.help.search.DefaultIndexerKit`, расширяющий абстрактный класс `javax.help.search.IndexerKit`.

Класс `IndexBuilder` обеспечивает создание и сохранение индексов для базы данных.

Класс `IndexerKit` отвечает за разбор (parse) содержимого файлов HTML-справки, исходя из их MIME-типа.

Визуальным представлением справочной системы JavaHelp в целом управляет интерфейс `javax.help.HelpBroker`. Класс `javax.help.DefaultHelpBroker` является реализацией интерфейса `HelpBroker` по умолчанию.

Справочные системы JavaHelp могут объединяться в одну динамически и статически. Для объединения JavaHelp-справок должен существовать первоначальный набор HelpSet, в который будут добавляться остальные наборы и который называется мастер HelpSet.

Статическое объединение осуществляется с помощью тега `<subhelpset>` файла мастера HelpSet с расширением hs. Атрибут `location` тега `<subhelpset>` определяет относительный URL-адрес файла HelpSet добавляемой JavaHelp-справки.

Динамическое объединение возможно с помощью метода `add()` класса `HelpSet` и метода `merge()` класса `JHelpNavigator` JavaHelp API.

Классы `javax.help.UniteAppendMerge`, `javax.help.SortMerge`, `javax.help.AppendMerge` и `javax.help.NoMerge` контролируют способ объединения.

Для того чтобы интегрировать JavaHelp-справку в приложение, необходимо:

1. Добавить библиотеку jh.jar в список `CLASSPATH` приложения, например, используя файл MANIFEST.MF.
2. Получить объект `ClassLoader` с помощью метода `getClassLoader()` класса `Class<T>`.
3. С помощью статического метода `findHelpSet()` класса `javax.help.HelpSet` получить URL-адрес файла HelpSet JavaHelp-справки, используя в качестве аргументов объект `ClassLoader` и имя файла HelpSet.
4. Для данного файла HelpSet создать с помощью конструктора объект класса `HelpSet`, используя URL-адрес файла HelpSet.
5. Для созданного объекта HelpSet с помощью метода `createHelpBroker()` создать объект `HelpBroker`.
6. В обработчике событий, связанном с компонентом управления графического интерфейса приложения, создать с помощью конструктора объект класса `javax.help.CSH.DisplayHelpFromSource`, используя в качестве аргумента объект `HelpBroker`. При этом будет визуализирована JavaHelp-справка.

Для того чтобы создать контекстную справку, необходимо связать с компонентом графического интерфейса приложения идентификатор файла HTML-справки. Это можно сделать с помощью статического метода `setHelpIDString()` класса `javax.help.CSH` библиотеки JavaHelp API. Методы `enableHelpKey()`, `enableHelpOnButton()` и `showID()` интерфейса `HelpBroker`, а также создание с помощью конструктора объектов классов `CSH.DisplayHelpAfterTracking` и `CSH.DisplayHelpFromFocus` позволяет визуализировать контекстную справку.

Платформа NetBeans предлагает свой способ интеграции JavaHelp-справки в модульную систему.

Для интеграции JavaHelp-справки в модульную систему NetBeans необходимо зарегистрировать набор HelpSet в файловой системе NetBeans. Для этого создается XML-файл, содержащий URL-адрес файла HelpSet (листинг П8.1).

Листинг П8.1. Код XML-файла, регистрирующего набор HelpSet в файловой системе NetBeans

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE helpsetref PUBLIC "-//NetBeans//DTD JavaHelp Help Set Reference 1.0//EN"
"http://www.netbeans.org/dtds/helpsetref-1_0.dtd">
<helpsetref url="nbdocs:/...docs/NameFileHelpSet"/>
```

URL-протокол nbdocs подразумевает, что файлы набора HelpSet должны находиться в папке docs каталога модульной системы NetBeans.

Далее этот файл должен быть зарегистрирован в папке Services\JavaHelp\ файловой системы NetBeans. Поэтому в файле layer.xml модуля необходимо указать URL-адрес созданного XML-файла:

```
<folder name="Services">
  <folder name="JavaHelp">
    <file name="Name.xml" url="url">
      <attr name="position" intvalue="number"/>
    </file>
  </folder>
</folder>
```

Для визуализации контекстной JavaHelp-справки платформа NetBeans предлагает:

- класс `org.openide.util.HelpCtx` библиотеки Utilities API, который связывает идентификатор файла HTML-справки со свойствами модуля, при этом справка визуализируется нажатием клавиши <F1>;
- класс `org.netbeans.api.javahelp.Help` библиотеки JavaHelp integration API, метод `showHelp()` которого позволяет программным образом визуализировать JavaHelp-справку.

Проект Javadoc2Help, домашняя страничка которого находится по адресу **http://sourceforge.net/projects/javadoc2help/**, позволяет конвертировать Javadoc-документацию в набор HelpSet.

ПРИЛОЖЕНИЕ 9

Архитектура технологии OSGi

Архитектура технологии OSGi состоит из следующих уровней:

- операционная система компьютера;
- виртуальная Java-машина;
- платформа OSGi. Библиотека классов API платформы OSGi представлена пакетом `org.osgi.framework`;
- OSGi-компоненты (bundles), созданные и развертываемые на основе платформы OSGi. OSGi-компоненты являются элементами, составляющими OSGi-приложения. Каждый bundle, представленный JAR-файлом, исполняется платформой OSGi;
- система безопасности, которая тесно связана с платформой OSGi. Система безопасности OSGi основана на системе безопасности платформы Java SE и включает в себя реализацию технологии электронной подписи и управление разрешениями. Система безопасности OSGi расширяет систему безопасности платформы Java SE, предоставляя API для динамического управления разрешениями.

Ядром технологии OSGi является платформа OSGi (OSGi Framework), которая в свою очередь состоит из следующих уровней.

- *Среда выполнения* (execution environment) — это стандартная среда выполнения Java-платформы.
- *Модульный уровень* (modules layer) определяет политику загрузки классов и ресурсов, которая обеспечивает как загрузку классов для каждого OSGi-компонента, так и связь между ними. Таким образом, гарантируется возможность совместного использования Java-пакетов между bundle-экспортерами и bundle-импортерами. OSGi-компонент (bundle) представляет собой JAR-файл, содержащий необходимые ресурсы (class-файлы, HTML-файлы, значки, дополнительные JAR-файлы и др.), а также MANIFEST-файл с описанием содержимого JAR-файла и информацией о JAR-файле как OSGi-компоненте (табл. П9.1).
- *Жизненный цикл* (life cycle layer) обеспечивает библиотеки API для инсталляции, запуска, остановки, обновления и деинсталляции bundles. Основные классы и интерфейсы API жизненного цикла — это `Bundle`, `BundleActivator`, `BundleContext`,

`BundleEvent`, `FrameworkEvent`, `BundleListener`, `SynchronousBundleListener`, `FrameworkListener` и `BundleException` (табл. П9.2). Инсталляция bundle может быть выполнена либо другим bundle, уже инсталлированным, либо с помощью инструментов реализации платформы OSGi. Интерфейс `BundleContext` обеспечивает метод `installBundle` для инсталляции bundle. После инсталляции bundle создается объект `Bundle` и все оставшиеся операции жизненного цикла выполняются с созданным объектом `Bundle`. Метод `getBundle()` интерфейса `BundleContext` позволяет получить объект `Bundle`, представляющий инсталлированный bundle, чтобы затем с помощью методов `start()`, `stop()`, `uninstall()` и `update()` управлять жизненным циклом bundle.

- *Сервисный уровень* (service layer) обеспечивает динамическую связь OSGi-компонентов с помощью сервисной модели "публикации — обнаружения — связывания". Использование элементов файла манифеста для регистрации позволяет обеспечить только статическую связь между bundle-экспортерами и bundle-импортерами Java-пакетов. Для динамической связи технология OSGi определяет архитектуру реестра сервисов Service Registry, являющуюся формой сервис-ориентированной архитектуры (SOA). Данная модель вводит понятие сервиса, который представляет собой объект Java-класса, реализующего один или несколько Java-интерфейсов и таким образом несущего определенную функциональность. Сервисная модель OSGi определяет взаимодействие минимум трех участников:
  - bundle, поставщик сервиса, регистрирует сервис в методе `start()` класса, определенного элементом `Bundle-Activator` файла манифеста, используя метод `registerService()` интерфейса `BundleContext`. При этом сервис регистрируется в реестре сервисов платформы OSGi;
  - реестр сервисов, который платформа OSGi обеспечивает на уровне виртуальной машины Java и который является сервис-брокером;
  - bundle, потребитель сервиса, следит за изменениями в реестре сервисов и в случае регистрации необходимого сервиса загружает его, таким образом, образуя динамическую связь с bundle, поставщиком сервиса.

Система безопасности технологии OSGi включает в себя четыре класса API платформы OSGi: `AdminPermission`, `BundlePermission`, `PackagePermission` и `ServicePermission`, позволяющие устанавливать разрешения в файле политики безопасности для OSGi-компонентов.

Кроме того, система безопасности предоставляет стандартные сервисы, позволяющие динамически управлять разрешениями:

- сервис Package Admin Service (пакет `org.osgi.service.packageadmin`), регистрируемый системным bundle платформы OSGi (системный bundle представляет саму платформу OSGi);
- сервис Permission Admin Service (пакеты `org.osgi.service.condpermadmin` и `org.osgi.service.permissionadmin`).

Помимо сервисов системы безопасности, технология OSGi предоставляет набор системных и протокольных сервисов (табл. П9.3).

Таблица П9.1. *Элементы файла MANIFEST.MF OSGi-компонента*

Элемент	Описание
`Bundle-ActivationPolicy:`	Определяет политику запуска bundle. Возможное значение `lazy` указывает, что bundle после старта не активируется до тех пор, пока не потребуется загрузка классов из него. Директивы `include` и `exclude` могут уточнять, какие пакеты запускают или не запускают активацию bundle при загрузке класса из указанного пакета
`Bundle-Activator:`	Определяет имя класса, используемого для запуска и остановки bundle. Например, `com.acme.fw.Activator`
`Bundle-Category:`	Указывает имя категории bundle. Например, `communication` — модуль предназначен для коммуникации, `games` — модуль относится к играм, и т. д.
`Bundle-Classpath:`	Определяет список JAR-файлов и каталогов внутри bundle, содержащих классы и ресурсы
`Bundle-ContactAddress:`	Содержит почтовый адрес производителя bundle
`Bundle-Copyright:`	Указывает авторские права
`Bundle-Description:`	Содержит короткое описание bundle
`Bundle-DocURL:`	URL-адрес документации
`Bundle-Localization:`	Местонахождение внутри bundle ресурсного properties-файла
`Bundle-ManifestVersion:`	Версия спецификации
`Bundle-Name:`	Имя bundle
`Bundle-NativeCode:`	Содержит спецификацию библиотек операционной системы, включенных в bundle
`Bundle-RequiredExecutionEnvironment:`	Ограничивает список сред выполнения для выполнения bundle
`Bundle-SymbolicName:`	Идентификатор bundle, который должен быть обязательно указан
`Bundle-UpdateLocation:`	Указывает URL-адрес для обновления bundle
`Bundle-Vendor:`	Указывает производителя bundle
`Bundle-Version:`	Версия bundle
`DynamicImport-Package:`	Содержит список пакетов, импортируемых при необходимости
`Export-Package:`	Содержит список экспортируемых пакетов
`Fragment-Host:`	Определяет корневой (host) bundle для данного fragment bundle. Fragment — это bundle, который прикрепляется к host bundle. Fragment bundle и host bundle обрабатываются платформой OSGi вместе, при этом элементы манифеста fragment bundle добавляются к элементам манифеста host bundle

Таблица П9.1 *(окончание)*

Элемент	Описание
`Import-Package:`	Содержит список импортируемых пакетов
`Require-Bundle:`	Прямо указывает идентификаторы bundle-экспортеров, пакеты которых импортируются

Таблица П9.2. *Основные классы и интерфейсы библиотеки API уровня Life Cycle платформы OSGi*

Класс или интерфейс	Описание
Интерфейс `Bundle`	Представляет инсталлированный bundle. Для каждого инсталлированного bundle платформой OSGi создается объект Bundle. Объект `Bundle` используется для управления жизненным циклом bundle. Каждому инсталлированному bundle платформа OSGi присваивает идентификационный номер, который не изменяется в течение всего жизненного цикла, даже если bundle обновляется. При новой инсталляции присваивается новый идентификационный номер. Интерфейс `Bundle` имеет следующие поля, характеризующие состояние bundle: • `INSTALLED` — bundle уже инсталлирован в платформу OSGi, но еще не разрешен; • `RESOLVED` — платформа OSGi успешно разрешила все зависимости bundle, определенные в файле манифеста (`Bundle-Classpath`, `Export-Package`, `Import-Package`, `Require-Bundle` и `Fragment-Host`), и bundle готов стартовать; • `STARTING` — вызван метод `start()` интерфейса `Bundle` либо метод `start()` класса, определенного элементом `Bundle-Activator` файла манифеста; • `ACTIVE` — bundle успешно стартовал; • `STOPPING` — вызван метод `stop()` интерфейса `Bundle` либо метод `stop()` класса, определенного элементом `Bundle-Activator` файла манифеста, и bundle переходит в состояние `RESOLVED`; • `UNINSTALLED` — bundle деинсталлирован и не может быть использован
Интерфейс `BundleActivator`	Используется в случае, если необходимо управлять запуском и остановкой bundle. Для этого необходимо создать класс, имеющий конструктор без параметров (для того, чтобы платформа OSGi могла автоматически создать экземпляр класса) и реализующий интерфейс `BundleActivator`, переопределить методы `start()` и `stop()` интерфейса, и зарегистрировать созданный класс с помощью элемента `Bundle-Activator` файла манифеста

Таблица П9.2 *(окончание)*

Класс или интерфейс	Описание
Интерфейс `BundleContext`	Представляет исполняемый контекст bundle и обеспечивает методы, с помощью которых bundle может взаимодействовать с платформой OSGi. Объект `BundleContext` создается при вызове метода `start()` интерфейса `BundleActivator`, при этом создается связь объекта `Bundle` с объектом `BundleContext`, представляющая связь bundle с его исполняемым контекстом. Объект `BundleContext` действителен при состояниях `STARTING`, `STOPPING` и `ACTIVE` объекта `Bundle`
Класс `BundleEvent`	Представляет события платформы OSGi, описывающие изменения жизненного цикла bundle. Объект `BundleEvent` создается и передается объекту `BundleListener` в случае изменений в жизненном цикле bundle
Класс `FrameworkEvent`	Представляет события, связанные с самой платформой OSGi
Интерфейс `BundleListener`	Содержит метод `bundleChanged`, вызываемый асинхронно в случае изменений в жизненном цикле bundle
Интерфейс `SynchronousBundleListener`	Содержит метод `bundleChanged`, вызываемый синхронно в случае изменений в жизненном цикле bundle
Интерфейс `FrameworkListener`	Содержит метод `frameworkEvent()`, вызываемый асинхронно в случае генерации объекта `FrameworkEvent`
Класс `BundleException`	Представляет исключительную ситуацию, связанную с жизненным циклом bundle

Таблица П9.3. *Стандартные сервисы OSGi*

Сервис	Описание
Permission Admin	Управляет разрешениями
Package Admin	Управляет зависимостями bundles
Start Level	Устанавливает набор bundles, которые должны быть запущены вместе или инициализированы перед запуском остальных bundles
URL Handler	Обеспечивает управление URL-адресами
Log Service	Получает информацию об ошибках, предупреждениях и т. д. и направляет ее другим bundles, которые подписаны на получение такого рода информации
Configuration Admin Service	Обеспечивает конфигурационную информацию
Device Access Service	Загружает bundle, реализующий необходимый драйвер
User Admin Service	Обеспечивает аутентификацию/авторизацию пользователей
IO Connector Service	Реализует пакет CDC/CLDC `javax.microedition.io`

Таблица П9.3 *(окончание)*

Сервис	**Описание**
Preferences Service	Обеспечивает доступ к базе данных свойств, определенных по умолчанию
Component Runtime	Обеспечивает динамическую загрузку сервисов
Deployment Admin	Обеспечивает развертывание bundle вместе с дополнительными ресурсами
Event Admin	Обеспечивает общий механизм обработки событий
Application Admin	Обеспечивает запуск и остановку всего OSGi-приложения
Http Service	Обеспечивает поддержку HTTP-протокола
UPnP Service	Обеспечивает поддержку протокола Universal Plug and Play (UPnP)
DMT Admin	Обеспечивает поддержку API для управления мобильными устройствами
Wire Admin Service	Обеспечивает связь между поставщиками сервисов и потребителями сервисов
XML Parser Service	Обеспечивает анализ XML-документов
Foreign Application Access	Позволяет не OSGi-приложениям, таким как MIDP, Xlets, Applets и др., иметь доступ к платформе OSGi
Monitoring	Обеспечивает контроль выполнения bundles

ПРИЛОЖЕНИЕ 10

Библиотеки спецификации CLDC 1.0

Таблица П10.1. *Библиотеки CLDC 1.0 в сравнении с библиотеками Java SE*

Пакет	CLDC 1.0	Java SE
java.lang	Interface Runnable. Class Boolean, Byte, Character, Class, Integer, Long, Math, Object, Runtime, Short, String, StringBuffer, System, Thread, Throwable. Exception ArithmeticException, ArrayIndexOutOfBounds Exception, ArrayStoreException, ClassCastException, ClassNotFoundException, Exception, IllegalAccessException, IllegalArgumentException, IllegalMonitorStateException, IllegalThreadStateException, IndexOutOfBoundsException, InstantiationException, InterruptedException, NegativeArraySizeException, NullPointerException, NumberFormatException, RuntimeException, SecurityException, StringIndexOutOfBounds Exception. Error Error, OutOfMemoryError, VirtualMachineError	Interface Appendable, CharSequence, Cloneable, Comparable<T>, Iterable<T>, Readable, Runnable, Thread.UncaughtExceptionHandler. Class Boolean, Byte, Character, Character.Subset, Character.UnicodeBlock, Class<T>, ClassLoader, Compiler, Double, Enum<E extends Enum<E>>, Float, InheritableThreadLocal<T>, Integer, Long, Math, Number, Object, Package, Process, ProcessBuilder, Runtime, RuntimePermission, SecurityManager, Short, StackTraceElement, StrictMath, String, StringBuffer, StringBuilder, System, Thread, ThreadGroup, ThreadLocal<T>, Throwable, Void. Enum Thread.State. Exception ArithmeticException, ArrayIndexOutOfBoundsException, ArrayStoreException, ClassCastException, ClassNotFoundException, CloneNotSupportedException, EnumConstantNotPresentException, Exception, IllegalAccessException, IllegalArgumentException, IllegalMonitorStateException, IllegalStateException, IllegalThreadStateException, IndexOutOfBoundsException, InstantiationException, InterruptedException, NegativeArraySizeException, NoSuchFieldException, NoSuchMethodException, NullPointerException, NumberFormatException, RuntimeException, SecurityException,

Таблица П10.1 *(продолжение)*

Пакет	CLDC 1.0	Java SE
		StringIndexOutOfBoundsException, TypeNotPresentException, UnsupportedOperationException. Error AbstractMethodError, AssertionError, ClassCircularityError, ClassFormatError, Error, ExceptionInInitializerError, IllegalAccessError, IncompatibleClassChangeError, InstantiationError, InternalError, LinkageError, NoClassDefFoundError, NoSuchFieldError, NoSuchMethodError, OutOfMemoryError, StackOverflowError, ThreadDeath, UnknownError, UnsatisfiedLinkError, UnsupportedClassVersionError, VerifyError, VirtualMachineError. Annotation Deprecated, Override, SuppressWarnings
java.io	Interface DataInput, DataOutput. Class ByteArrayInputStream, ByteArrayOutputStream, DataInputStream, DataOutputStream, InputStream, InputStreamReader, OutputStream, OutputStreamWriter, PrintStream, Reader, Writer. Exception EOFException, InterruptedIOException, IOException, UnsupportedEncodingException, UTFDataFormatException.	Interface Closeable, DataInput, DataOutput, Externalizable, FileFilter, FilenameFilter, Flushable, ObjectInput, ObjectInputValidation, ObjectOutput, ObjectStreamConstants, Serializable. Class BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter, ByteArrayInputStream, ByteArrayOutputStream, CharArrayReader, CharArrayWriter, Console, DataInputStream, DataOutputStream, File, FileDescriptor, FileInputStream FileOutputStream, FilePermission, FileReader, FileWriter, FilterInputStream, FilterOutputStream, FilterReader, FilterWriter, InputStream, InputStreamReader, LineNumberInputStream, LineNumberReader, ObjectInputStream, ObjectInputStream.GetField, ObjectOutputStream, ObjectOutputStream.PutField, ObjectStreamClass, ObjectStreamField, OutputStream, OutputStreamWriter, PipedInputStream, PipedOutputStream, PipedReader, PipedWriter, PrintStream, PrintWriter, PushbackInputStream, PushbackReader, RandomAccessFile, Reader, SequenceInputStream, SerializablePermission, StreamTokenizer, StringBufferInputStream, StringReader, StringWriter, Writer. Exceptions CharConversionException, CharConversionException,

Таблица П10.1 *(продолжение)*

Пакет	CLDC 1.0	Java SE
		EOFException, FileNotFoundException, InterruptedIOException, InvalidClassException, InvalidObjectException, IOException, NotActiveException, NotSerializableException, ObjectStreamException, OptionalDataException, StreamCorruptedException, SyncFailedException, UnsupportedEncodingException, UTFDataFormatException, WriteAbortedException. Errors IOError
java.util	Interface Enumeration. Class Calendar, Date, Hashtable, Random, Stack, TimeZone, Vector. Exception EmptyStackException, NoSuchElementException	Interface Collection, Comparator, Deque, Enumeration, EventListener, Formattable, Iterator, List, ListIterator, Map, Map.Entry, NavigableMap, NavigableSet, Observer, Queue, RandomAccess, Set, SortedMap, SortedSet. Class AbstractCollection, AbstractList, AbstractMap, bstractMap.SimpleEntry, AbstractMap.SimpleImmutableEntry, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayDeque, ArrayList, Arrays, BitSet, Calendar, Collections, Currency, Date, Dictionary, EnumMap, EnumSet, EventListenerProxy, EventObject, FormattableFlags, Formatter, GregorianCalendar, HashMap, HashSet, Hashtable, IdentityHashMap, LinkedHashMap, LinkedHashSet, LinkedList, ListResourceBundle, Locale, Observable, PriorityQueue, Properties, PropertyPermission, PropertyResourceBundle, Random, ResourceBundle, ResourceBundle.Control, Scanner, ServiceLoader, SimpleTimeZone, Stack, StringTokenizer, Timer, TimerTask, TimeZone, TreeMap, TreeSet, UUID, Vector, WeakHashMap. Enum Formatter.BigDecimalLayoutForm. Exception ConcurrentModificationException, DuplicateFormatFlagsException, EmptyStackException, FormatFlagsConversionMismatch Exception, FormatterClosedException, IllegalFormatCodePointException, IllegalFormatConversionException, IllegalFormatException, IllegalFormatFlagsException, IllegalFormatPrecisionException,

Таблица П10.1 *(окончание)*

Пакет	CLDC 1.0	Java SE
		IllegalFormatWidthException, InputMismatchException, InvalidPropertiesFormatException, MissingFormatArgumentException, MissingFormatWidthException, MissingResourceException, NoSuchElementException, TooManyListenersException, UnknownFormatConversionException, UnknownFormatFlagsException. Error ServiceConfigurationError
javax. microedition.io	Interface Connection, ContentConnection, Datagram, DatagramConnection, InputConnection, OutputConnection, StreamConnection, StreamConnectionNotifier. Class Connector. Exception ConnectionNotFoundException	

ПРИЛОЖЕНИЕ 11

Синтаксис JSP

Директивы

- Директива **page.**

  Стандартный синтаксис:

  ```
  <%@ page [attributes]%>
  ```

  Директива `page` устанавливает для контейнера ряд свойств JSP-страницы, определенных атрибутами директивы:

  - `autoFlush="true | false"` — если `true` (по умолчанию), тогда буфер очищается автоматически при переполнении;
  - `buffer="none | 8kb | sizekb"` — размер буфера, используемого при пересылке ответа клиентскому Web-браузеру;
  - `contentType="mimeType [ ;charset=characterSet ]" | "text/html;charset=ISO-8859-1"` — указывает формат ответа клиенту;
  - `deferredSyntaxAllowedAsLiteral` — если `false` (по умолчанию), тогда возникает ошибка трансляции при использовании EL-выражений `#{expr}` (отложенные вычисления) в той части JSP-страницы, которая формирует статический контент. Если `true`, тогда выражение `#{expr}` воспринимается контейнером как строка;
  - `errorPage="relativeURL"` — относительный путь JSP-страницы, к которой обращается данная JSP-страница в случае возникновения исключительных ситуаций;
  - `extends="package.class"` — указывает полное имя класса, который будет расширен транслированным из JSP-страницы классом;
  - `import="{package.class | package.* }, ..."` — список импортируемых Java-пакетов, используемых в скриптлетах, объявлениях и выражениях JSP-страницы;
  - `info="text"` — текстовая информация о JSP-странице, которую можно затем получить программным способом с помощью метода `getServletInfo()` интерфейса `javax.servlet.Servlet`;

- `isELIgnored` — если `false` (по умолчанию), тогда выражения Expression Language распознаются контейнером;
- `isErrorPage="true | false"` — если `true`, тогда данная JSP-страница используется другими JSP-страницами для обработки исключительных ситуаций. По умолчанию `false`;
- `isThreadSafe="true | false"` — больше не используется, `true` по умолчанию — контейнер может посылать параллельные клиентские запросы JSP-странице;
- `language="java"` — язык, используемый в скриптлетах, объявлениях и выражениях JSP-страницы;
- `pageEncoding` — определяет кодировку для JSP-страницы;
- `session="true | false"` — если `true` (по умолчанию), тогда контейнер создает HTTP-сессию при клиентском запросе к JSP-странице;
- `trimDirectiveWhitespaces` — если `true`, тогда все пустые строки удаляются из ответа клиенту, по умолчанию `false`.

XML-синтаксис:

```
<jsp:directive.page pageDirectiveAttrList />
```

❒ Директива **`taglib`**.

Стандартный синтаксис:

```
<%@ taglib [attributes]%>
```

Директива `taglib` объявляет, что JSP-страница использует пользовательские теги и указывает используемую библиотеку пользовательских тегов и префикс пользовательского тега с помощью атрибутов:

- `prefix="tagPrefix"` — префикс пользовательского тега (пространство имен) `<myPrefix:myTag>`;
- `uri="URIToTagLibrary"` — URI-адрес дескриптора библиотеки пользовательских тегов;
- `tagdir="tagDir"` — взаимоисключающий с `uri` атрибут, указывает, что библиотека пользовательских тегов находится в каталоге или подкаталоге \WEB-INF\tags\.

XML-синтаксис: директива `taglib` не используется, вместо нее применяется атрибут `xmlns`, включаемый в любой XML-элемент.

```
<anyxmlelement xmlns:prefix="{uri|urn:jsptld:path|urn:jsptagdir:
/WEB-INF/tags[/subdir]+}">
```

❒ Директива **`include`**.

Стандартный синтаксис:

```
<%@ include file="relativeURL" %>
```

Директива `include` вставляет в JSP-страницу другую JSP-страницу, HTML-страницу или текстовый файл на стадии трансляции.

XML-синтаксис:

```
<jsp:directive.include file="relativeURLspec"/>
```

- Директива **tag**.

Стандартный синтаксис:

```
<%@ tag [attributes]%>
```

Директива `tag` описывает свойства пользовательского тега и используется вместо директивы `page` в файлах пользовательских тегов, которые представляют пользовательские теги с помощью только JSP-синтаксиса, не прибегая к Java-библиотеке Tag Extension API. Файлы пользовательских тегов имеют расширения tag (стандартный синтаксис), tagx (XML-синтаксис) и tagf (фрагменты).

Атрибуты директивы `tag`:

- `display-name="name of the tag file | display-name"` — отображаемое имя, по умолчанию имя файла тегов без расширения;
- `body-content="scriptless|tagdependent|empty"` — тип содержимого тела тега. Если тело тега содержит другие пользовательские теги, стандартные теги и HTML-текст, то значение атрибута — `scriptless` (по умолчанию). Значение `tagdependent` означает, что тело тега не обрабатывается как JSP-контент и разработчик пользовательского тега должен обеспечить обработку тела тега, например исполнение SQL-запроса;
- `dynamic-attributes="page-scoped attribute"` — атрибуты (параметры), которые обрабатываются в файле тегов одинаковым способом, но конкретные имена атрибутов неизвестны на стадии разработки. Указывается временное имя атрибута, вместо которого подставляются конкретные имена при использовании пользовательского тега;
- `description="text"`, `small-icon="relativeURL"` и `large-icon="relativeURL"` — описание и значки пользовательского тега;
- `example="text"` — пример использования тега;
- `language="java"` — язык, используемый в скриптлетах, объявлениях и выражениях тега;
- `import="{package.class | package.*}, ... "` — список импортируемых Java-пакетов, используемых в скриптлетах, объявлениях и выражениях тега;
- `pageEncoding="{characterSet | ISO-8859-1}"` — кодировка для тега;
- `isELIgnored` — если `false` (по умолчанию), тогда выражения Expression Language распознаются контейнером.

XML-синтаксис:

```
<jsp:directive.tag tagDirectiveAttrList />
```

- Директива **attribute**.

Стандартный синтаксис:

```
<%@ attribute [attributes]%>
```

Директива `attribute` объявляет атрибуты (параметры), которые определяют поведение пользовательского тега так же, как параметры определяют поведение метода.

Атрибуты директивы `attribute`:

- `name = "attribute-name"` — уникальное имя атрибута;
- `required="true | false"` — если `false` (по умолчанию), тогда атрибут необязателен при использовании пользовательского тега;
- `fragment="true | false"` — файл пользовательского тега транслируется контейнером в объект реализации (tag handler). Если атрибут `fragment` имеет значение `true`, тогда параметр, объявленный директивой `attribute`, вычисляется объектом реализации, если `false` (по умолчанию) — параметр вычисляется контейнером перед тем, как быть посланным в объект реализации. Если значение `true`, тогда контейнер фиксирует тип параметра как `javax.servlet.jsp.tagext.JspFragment` (атрибут `type`) и значение атрибута `rtexprvalue` как `true`, соответственно значения атрибутов `type` и `rtexprvalue` указывать не требуется. Значение `true` атрибута `fragment` позволяет использовать с помощью действия `<jsp:invoke>` в теле пользовательского тега фрагмент JSP-кода, представляющий собой исполняемый набор динамических элементов (действий и EL-выражений), определенный в JSP-странице, которая использует данный пользовательский тег;
- `rtexprvalue="true | false"` — если `true` (по умолчанию), тогда значение параметра вычисляется динамически во время исполнения;
- `type="java.lang.String | a non-primitive type"` — тип значения параметра, по умолчанию `java.lang.String`;
- `description="text"` — описание параметра.

XML-синтаксис:

```
<jsp:directive.attribute attributeDirectiveAttrList/>
```

❒ Директива **`variable`**.

Стандартный синтаксис:

```
<%@ variable [attributes]%>
```

Директива `variable` объявляет переменные, значения которых передаются пользовательским тегом вызывающей его JSP-странице.

Атрибуты директивы `variable`:

- `name-given="scripting variable"` — имя переменной, определенной в JSP-странице, которая вызывает пользовательский тег;
- `name-from-attribute="scripting variable"` — имя параметра, значение которого при вызове тега дает имя переменной. Используется вместе с атрибутом `alias`;
- `alias="locally-scoped attribute"` — имя переменной тега, значение которой присваивается переменной `name-from-attribute`;

- variable-class="java.lang.String" | name of the variable class" — Java-класс переменной, по умолчанию java.lang.String;
- declare="true | false" — если true (по умолчанию), тогда переменная объявляется в вызывающей тег JSP-странице;
- scope="AT_BEGIN | AT_END | NESTED" — область видимости переменной в вызывающей тег JSP-странице. NESTED — область видимости ограничена тегом, AT_BEGIN — область видимости от открывающего тега и до конца JSP-страницы, AT_END — область видимости после завершения вызова тега и до конца страницы;
- description="text" — описание переменной.

XML-синтаксис:

```
<jsp:directive.variable variableDirectiveAttrList/>
```

Стандартные действия

❒ Действие **<jsp:attribute>.**

Стандартный синтаксис и XML-синтаксис:

```
<jsp:attribute name="attributeName" [ trim= "true | false" ] />
```

Действие <jsp:attribute> определяет значение параметра в JSP-странице. Атрибут name указывает имя параметра, если значение атрибута trim равно true (по умолчанию), тогда пропуски в начале и в конце тела действия отбрасываются.

❒ Действие **<jsp:body>.**

Стандартный синтаксис и XML-синтаксис:

```
<jsp:body>any elements or text</jsp:body>
```

Действие <jsp:body> определяет тело вызываемого тега и должно использоваться, если одно или несколько действий jsp:attribute присутствует в теле вызываемого тега.

❒ Действие **<jsp:element>.**

Стандартный синтаксис и XML-синтаксис:

```
<jsp:element name="elementName"
  ( /> | >  ( any elements or text </jsp:element> ))
| ( <jsp:element name="elementName">
       [ <jsp:attribute name="attributeName" [ trim="true | false" ]
      ( /> | (any elements or text </jsp:attribute> ) ) ]+
      [ <jsp:body> any elements or text </jsp:body> ]
</jsp:element>
```

Действие <jsp:element> динамически генерирует XML и используется в JSP-страницах, файлах тегов, JSP-документах. Атрибут name="elementName" указывает, какой XML-элемент генерируется.

- Действие **<jsp:doBody>**.

  Стандартный синтаксис и XML-синтаксис:

```
<jsp:doBody ({var="scopedAttributeName"  |
    varReader="scopedAttributeName" }
   [scope="page | request | session | application" ] />) | />
```

  Действие `<jsp:doBody>` используется в файлах пользовательских тегов и исполняет тело пользовательского тега, определенное в вызывающей тег JSP-странице.

  Атрибуты действия `<jsp:doBody>`:

  - `var="scopedAttributeName"` — имя переменной, хранящей результат исполнения тела тега. Тип переменной — объект `java.lang.String`. Используется для манипулирования результатом исполнения;
  - `varReader="scopedAttributeName"` — имя переменной, хранящей результат исполнения тела тега. Тип переменной — объект `java.io.Reader`. Используется для манипулирования результатом исполнения;
  - `scope="page | request | session | application"` — область видимости переменной, по умолчанию `page`.

  Если действие `<jsp:doBody>` используется без атрибутов, тогда результат исполнения тела тега сразу передается обратно JSP-странице.

- Действие **<jsp:forward>**.

  Стандартный синтаксис:

```
<jsp:forward page="{relativeURL | '${' Expression'}' |
   <%= expression %>}" { /> | > [<jsp:param name="parameterName"
value="{parameterValue | '${' Expression '}' | <%= expression %>}" }
/> ]+ </jsp:forward>}
```

  XML-синтаксис:

```
<jsp:forward page="{relativeURL | '${' Expression '}' |
   %= expression % }" { /> | > [<jsp:param  name="parameterName"
   value="{parameterValue | '${' Expression '}' | %= expression %}" }
/> ]+ </jsp:forward> }
```

  Действие `<jsp:forward>` переадресовывает клиентский запрос JSP-странице другому ресурсу (JSP-страница, сервлет и т. д.).

  Атрибуты действия `<jsp:forward>`:

  - `page="{relativeURL | <%= expression %>}"` — относительный URL-адрес ресурса, которому переадресовывается клиентский запрос;
  - `<jsp:param name="parameterName" value="{parameterValue | <%= expression %>}" />+` — параметры, передаваемые динамическому ресурсу.

- Действие **<jsp:getProperty>**.

  Стандартный синтаксис и XML-синтаксис:

```
<jsp:getProperty name="beanInstanceName" property="propertyName" />
```

Действие `<jsp:getProperty>` получает свойство JavaBeans-компонента и вставляет его в ответ. Перед использованием действия `<jsp:getProperty>` необходимо применить действие `<jsp:useBean>`.

Атрибуты действия `<jsp:getProperty>`:

- `name="beanInstanceName"` — имя экземпляра JavaBeans-компонента, указанное в действии `<jsp:useBean>`;
- `property="propertyName"` — имя свойства JavaBeans-компонента.

❒ Действие **`<jsp:include>`**.

Стандартный синтаксис:

```
<jsp:include page="{relativeURL | '${' Expression '}' |
   <%= expression %>}"
   [ flush="true| false" ]
   { /> | > [ <jsp:param name="parameterName"
      value="{parameterValue | '${' Expression '}' |
      <%= expression %>}" />] +
</jsp:include> }
```

XML-синтаксис:

```
<jsp:include page="{relativeURL | '${' Expression '}' |
   %= expression %}"
   [ flush="true| false" ]
   { /> | > [ <jsp:param name="parameterName"
      value="{parameterValue | '${' Expression '}' |
      %= expression %}" />] +
</jsp:include> }
```

Действие `<jsp:include>` позволяет включать статические и динамические ресурсы в JSP-страницу.

Атрибуты действия `<jsp:include>`:

- `page="{ relativeURL | <%= expression %> }"` — URL-адрес включаемого ресурса;
- `flush="true | false"` — если `true`, тогда буфер исходящей страницы очищается перед включением ресурса, по умолчанию `false`;
- `<jsp:param name="parameterName" value="{parameterValue | <%= expression %>}" />+` — параметры, передаваемые включаемому ресурсу.

❒ Действие **`<jsp:invoke>`**.

Стандартный синтаксис и XML-синтаксис:

```
<jsp:invoke fragment="fragmentName"
({var="scopedAttributeName" |
  varReader="scopedAttributeName"}
 [scope="page | request | session | application" ] />) | />
```

Действие `<jsp:invoke>` позволяет использовать в теле пользовательского тега фрагмент JSP-кода, представляющий собой исполняемый набор динамических

элементов (действий и EL-выражений), определенный в JSP-странице, которая использует пользовательский тег.

Атрибуты действия `<jsp:invoke>`:

- `fragment="fragmentName"` — имя параметра пользовательского тега с атрибутом `fragment="true"`;
- `var="scopedAttributeName"` — имя переменной, хранящей результат действия. Тип переменной — объект `java.lang.String`;
- `varReader="scopedAttributeName"` — имя переменной, хранящей результат действия. Тип переменной — объект `java.io.Reader`;
- `scope="page | request | session | application"` — область видимости переменной, по умолчанию `page`.

❒ Действие **`<jsp:output>`**.

Только XML-синтаксис:

```
<jsp:output ( omit-xml-declaration="yes|no|true|false" )
   { doctypeDecl } />
   doctypeDecl ::= ( doctype-root-element="rootElement"
   doctype-public="PubidLiteral"
   doctype-system="SystemLiteral" )
   | ( doctype-root-element="rootElement"
       doctype-system="SystemLiteral" )
```

Действие `<jsp:output>` используется в JSP-документах и файлах тегов для модификации свойств исходящего документа.

Атрибуты действия `<jsp:output>`:

- `omit-xml-declaration="yes | no | true | false"` — указывает, включается (`yes` или `true`) или нет (`no` или `false`) XML-декларация (`<?xml version="1.0" encoding="UTF-8" ?>` — XML-декларация по умолчанию) в исходящий документ. Если в JSP-документе используется действие `<jsp:root>`, тогда значение атрибута `omit-xml-declaration` по умолчанию `yes`, если в JSP-документе отсутствует действие `<jsp:root>`, тогда значение атрибута `omit-xml-declaration` по умолчанию `no`. Для файла тегов значение атрибута `omit-xml-declaration` по умолчанию `yes`;
- `doctype-root-element="rootElement"` — используется вместе с атрибутом `doctype-system` и указывает корневой элемент исходящего XML-документа;
- `doctype-system="SystemLiteral"` — указывает URI-ссылку на DTD-документ, который определяет структуру исходящего XML-документа. Использование атрибута `doctype-system` не вызывает проверку контейнером соответствия XML-документа DTD-документу;
- `doctype-public="PubidLiteral"` — необязательный атрибут, используется вместе с атрибутом `doctype-system` и указывает значение Public ID для генерируемого `DOCTYPE` XML-документа.

- Действие **<jsp:plugin>**.

  Стандартный синтаксис:

```
<jsp:plugin
   type="bean|applet"
   code="classFileName"
   codebase="classFileDirectoryName"
   [ name="instanceName" ]
   [ archive="URIToArchive, ..." ]
   [ align="bottom|top|middle|left|right" ]
   [ height="{displayPixels |  <%= expression %>}"]
   [ width="{displayPixels |  <%= expression %>}"]
   [ hspace="leftRightPixels" ]
   [ vspace="topBottomPixels" ]
   [ jreversion="JREVersionNumber | 1.2" ]
   [ nspluginurl="URLToPlugin" ]
   [ iepluginurl="URLToPlugin" ] >
   [ <jsp:params>
      [ <jsp:param name="parameterName"
         value="{parameterValue | '${' Expression '}' |
         <%= expression %>}" /> ]+
     </jsp:params> ]
     [ <jsp:fallback> text message if plugin download fails
       </jsp:fallback> ]
</jsp:plugin>
```

  XML-синтаксис:

```
<jsp:plugin
   type="bean|applet" code="classFileName"
   codebase="classFileDirectoryName"
   [ name="instanceName" ]  [ archive="URIToArchive, ..." ]
   [ align="bottom|top|middle|left|right" ]
   [ height="{displayPixels | '${' Expression '}' | %= expression %}"]
   [ width="{displayPixels | '${' Expression '}' | %= expression %}"]
   [ hspace="leftRightPixels" ]  [ vspace="topBottomPixels" ]
   [ jreversion="JREVersionNumber | 1.2" ]
   [ nspluginurl="URLToPlugin" ]
   [ iepluginurl="URLToPlugin" ]  >
   [ <jsp:params>
      [ <jsp:param name="parameterName"
         value="{parameterValue | '${' Expression '}' |
         %= expression %}" /> ]+
     </jsp:params> ]
   [ <jsp:fallback>  text message if plugin download fails
     </jsp:fallback>  ]
</jsp:plugin>
```

  Действие `<jsp:plugin>` вызывает исполнение апплета или компонента JavaBeans в клиентском Web-браузере.

Атрибуты действия `<jsp:plugin>`:

- `type="bean|applet"` — тип исполняемого объекта;
- `code="classFileName"` — имя исполняемого Java-класса с расширением class;
- `codebase="classFileDirectoryName"` — путь к каталогу, содержащему исполняемый Java-класс;
- `name="instanceName"` — имя экземпляра исполняемого Java-класса, которое может быть использовано в коде JSP-страницы;
- `archive="URIToArchive, ..."` — список путей библиотечных файлов, которые должны быть загружены перед исполнением Java-класса;
- `align="bottom|top|middle|left|right"` — расположение изображения исполняемого объекта относительно базовой строки текста;
- `height="{displayPixels | <%= expression %>}" width="{displayPixels | <%= expression %>}"` — размеры изображения исполняемого объекта;
- `hspace="leftRightPixels" vspace="topBottomPixels"` — величина отступа изображения исполняемого объекта;
- `jreversion="JREVersionNumber | 1.2"` — требуемая версия Java Runtime Environment (JRE) для исполнения объекта;
- `nspluginurl="URLToPlugin"` — URL-адрес, по которому клиент может загрузить JRE-плагин для браузера Netscape Navigator;
- `iepluginurl="URLToPlugin"` — URL-адрес, по которому клиент может загрузить JRE-плагин для браузера Internet Explorer;
- `<jsp:params> [ <jsp:param name="parameterName" value="{parameterValue | <%= expression %>}" /> ]+ </jsp:params>` — передаваемые исполняемому объекту параметры;
- `<jsp:fallback>` *`text message for user`* `</jsp:fallback>` — сообщение клиенту в случае, если JRE-плагин не запустится.

❐ Действие **`<jsp:root>`**.

Только XML-синтаксис:

```
<jsp:root
   xmlns:jsp="http://java.sun.com/JSP/Page"
   [ xmlns:taglibPrefix="URI" ]+ ...
   version="1.2 | 2.0">
      JSP Page
</jsp:root>
```

Действие `<jsp:root>` является необязательным и может использоваться как корневой элемент в JSP-документе или файле пользовательского тега. Если `<jsp:root>` используется в JSP-файле, тогда контейнер воспринимает его как JSP-документ, даже если JSP-файл не имеет расширения jspx.

Атрибуты действия `<jsp:root>`:

- `xmlns:jsp="http://java.sun.com/JSP/Page"` — делает возможным использование стандартных JSP-элементов;
- `version="1.2 | 2.0"` — указывает версию JSP-спецификации;
- `xmlns:taglibPrefix="URI"` — указывает используемую библиотеку тегов. `URI` — местонахождение TLD-файла, описывающего используемый набор тегов, который соответствует указанному префиксу.

❐ Действие **`<jsp:setProperty>`**.

Стандартный синтаксис:

```
<jsp:setProperty name="beanInstanceName"
{  property="*" |
   property="propertyName" [ param="parameterName" ] |
   property="propertyName" value="{stringLiteral|
      '${' Expression '}' | <%= expression %>}"  }
/>
```

XML-синтаксис:

```
<jsp:setProperty name="beanInstanceName"
{  property="*" |
   property="propertyName" [ param="parameterName" ] |
   property="propertyName" value="{stringLiteral |
      '${' Expression '}' | %= expression %}"  }
/>
```

Действие `<jsp:setProperty>` используется вместе с действием `<jsp:useBean>` и устанавливает свойства используемого компонента JavaBeans.

❐ Действие **`<jsp:useBean>`**.

Стандартный синтаксис:

```
 <jsp:useBean id="beanInstanceName"
    scope="page|request|session|application"
 {  class="package.class" [ type="package.class" ]|
    beanName="{package.class | '${' Expression '}' |
       <%= expression %>}" type="package.class" |
    type="package.class"  }
 { /> | > other elements </jsp:useBean> }
```

XML-синтаксис:

```
<jsp:useBean id="beanInstanceName"
   scope="page|request|session|application"
{  class="package.class"  [ type="package.class" ]  |
   beanName="{package.class | '${' Expression '}' |
      %= expression %}" type="package.class" |
      type="package.class"  }
{ /> | > other elements </jsp:useBean>  }
```

Действие <jsp:useBean> обеспечивает использование экземпляра компонента JavaBeans.

Атрибуты действия <jsp:useBean>:

- id="beanInstanceName" — имя экземпляра компонента JavaBeans;
- scope="page|request|session|application" — область видимости экземпляра. По умолчанию — page;
- class="package.class" — Java-класс, экземпляр которого создается контейнером;
- type="package.class" — Java-тип экземпляра, используется, если экземпляр уже существует;
- class="package.class" type="package.class" — создается экземпляр указанного Java-класса и определяется Java-тип экземпляра, который может быть таким же, как и класс, как суперкласс или как интерфейс, реализуемый классом;
- beanName="{package.class | <%= expression %>}" type="package.class" — создается экземпляр указанного класса или имя класса создаваемого экземпляра вычисляется выражением, после этого определяется Java-тип экземпляра.

❒ Действие **<jsp:text>**.

Стандартный синтаксис и XML-синтаксис:

```
<jsp:text>
   template data
</jsp:text>
```

Действие <jsp:text> включает указанный контент в исходящий документ.

Комментарии

Комментарии в JSP-странице (стандартный синтаксис):

❒ <!-- comment --> — комментарии включаются в исходящий документ (ответ клиенту);

❒ <%-- comment --%> — комментарии не включаются в ответ клиенту.

Комментарии в JSP-документе (XML-синтаксис):

<!-- comment --> — комментарии не включаются в ответ клиенту.

Скриптовые элементы

Скриптлеты

Стандартный синтаксис:

```
<% code fragment %>
```

XML-синтаксис:

```
<jsp:scriptlet> code fragment </jsp:scriptlet>
```

Скриптлет — это элемент, исполняемый контейнером при запросе и содержащий Java-операторы и другие скриптовые элементы — объявления и выражения.

Объявления

Стандартный синтаксис:

```
<%! declaration; [ declaration; ]+ ... %>
```

XML-синтаксис:

```
<jsp:declaration> code fragment [ declaration; ]+ ... </jsp:declaration>
```

Объявления описывают переменные и методы, используемые в Java-коде JSP-страницы.

Выражения

Стандартный синтаксис:

```
<%= expression %>
```

XML-синтаксис:

```
<jsp:expression> expression </jsp:expression>
```

Выражения — это элементы, содержащие вычисляемые Java-выражения, которые после вычисления контейнером конвертируются в строки и вставляются в исходящий документ.

EL-выражения

Стандартный синтаксис и XML-синтаксис:

- `${expr}` — выражение вычисляется сразу при вызове JSP-страницы;
- `#{expr}` — вычисление выражения может быть отложено. Выражение с отложенным вычислением (deferred evaluation) вычисляется тогда, когда потребуется значение выражения. Выражения с отложенным вычислением нельзя использовать в статическом контенте.

Язык Expression Language включает в себя:

- арифметические операторы (`+`, `-`, `*`, `/` и `div`, `%` и `mod`);
- логические операторы (`&&` и `and`, `||` и `or`, `!` и `not`);
- операторы отношения (`==` и `eq`, `!=` и `ne`, `<` и `lt`, `>` и `gt`, `<=` и `le`, `>=` и `ge`);
- оператор условия (`A ? B : C`);
- `empty`-оператор (`empty A` возвращает `true`, если `A` — ноль);

- переменные;
- неявные объекты (`pageContext`, `servletContext`, `session`, `request`, `response`, `param`, `paramValues`, `header`, `headerValues`, `cookie`, `initParam`, `pageScope`, `requestScope`, `sessionScope`, `applicationScope`);
- литералы (`true` и `false`, целые, вещественные, строки, `null`);
- функции (функция определяется так же, как и пользовательский тег и может быть вызвана в выражении).

Стандартные теги библиотеки JavaServer Pages Standard Tag Library (JSTL)

JSTL-библиотека представляет набор действий, которые могут быть использованы в JSP-странице.

JSTL-библиотека интегрирована с языком Expression Language. EL-выражения могут использоваться в JSTL-тегах как с помощью конструкции `${expr}`, так и с помощью конструкции `#{expr}`.

Для того чтобы использовать JSTL-теги в JSP-страницах, необходимо применить директиву `taglib` (стандартный синтаксис) или атрибут `xmlns` (XML-синтаксис):

- для тега с префиксом `c`:
  - стандартный синтаксис:
    ```
    <%@ taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
    ```
  - XML-синтаксис:
    ```
    <anyxmlelement xmlns:c="http://java.sun.com/jsp/jstl/core" />
    ```
- для тега с префиксом `x`:
  - стандартный синтаксис:
    ```
    <%@ taglib uri="http://java.sun.com/jstl/xml" prefix="x" %>
    ```
  - XML-синтаксис:
    ```
    <anyxmlelement xmlns:x="http://java.sun.com/jsp/jstl/xml" />
    ```
- для тега с префиксом `fmt`:
  - стандартный синтаксис:
    ```
    <%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
    ```
  - XML-синтаксис:
    ```
    <anyxmlelement xmlns:fmt ="http://java.sun.com/jsp/jstl/fmt" />
    ```
- для тега с префиксом `sql`:
  - стандартный синтаксис:
    ```
    <%@ taglib uri="http://java.sun.com/jstl/sql" prefix="sql"%>
    ```
  - XML-синтаксис:
    ```
    <anyxmlelement xmlns:sql ="http://java.sun.com/jsp/jstl/sql" />
    ```

- для тега с префиксом `fn`:
  - стандартный синтаксис:
    ```
    <%@taglib uri="http://java.sun.com/jstl/functions" prefix="fn"%>
    ```
  - XML-синтаксис:
    ```
    <anyxmlelement xmlns:fn ="http://java.sun.com/jsp/jstl/functions" />
    ```

Теги библиотеки JSTL

- Тег с префиксом `c` обеспечивает использование базовой библиотеки JSTL.
- Тег с префиксом `x` обеспечивает XML-обработку.
- Тег с префиксом `fmt` обеспечивает форматирование и интернационализацию.
- Тег с префиксом `sql` обеспечивает доступ к базам данных.
- Тег с префиксом `fn` позволяет использовать JSTL-функции.

Тег <c>

- Действие **`<c:out>`**.

  Синтаксис:

  ```
  <c:out value="value" [escapeXml="{true|false}"]
  [default="defaultValue"] />
  ```

  С телом:

  ```
  <c:out value="value" [escapeXml="{true|false}"]>
    default value
  </c:out>
  ```

  Действие `<c:out>` вычисляет выражение и помещает результат вычисления в исходящий документ.

  Атрибуты действия `<c:out>`:

  - `default` — значение выражения по умолчанию, в случае, если результат выражения — ноль;
  - `escapeXml` — если `true` (по умолчанию), тогда символы `<`, `>`, `&`, `'`, `"` в результате выражения интерпретируются исходящим документом как символы, а не как элементы синтаксиса HTML или XML;
  - `value` — вычисляемое выражение.

- Действие **`<c:set>`**.

  Действие устанавливает значение переменной с областью видимости, используя атрибут `value`:

  ```
  <c:set value="value"
  var="varName" [scope="{page|request|session|application}"]/>
  ```

или:

```
<c:set var="varName" [scope="{page|request|session|application}"]>
  body content
</c:set>
```

Действие устанавливает значение свойства объекта (объект указан атрибутом `target`), используя атрибут `value`:

```
<c:set value="value"
target="target" property="propertyName"/>
```

или:

```
<c:set target="target" property="propertyName">
  body content
</c:set>
```

Действие устанавливает отложенное значение переменной:

```
<c:set var="varName" value="deferred-value"/>
```

В качестве значения атрибута `value` может выступать вычисляемое выражение.

- Действие **`<c:remove>`**.

  Синтаксис:

  ```
  <c:remove var="varName"
  [scope="{page|request|session|application}"]/>
  ```

  Действие `<c:remove>` удаляет переменную с областью видимости.

- Действие **`<c:catch>`**.

  Синтаксис:

  ```
  <c:catch [var="varName"]>
    nested actions
  </c:catch>
  ```

  Действие `<c:catch>` перехватывает исключительную ситуацию, вызванную действием, включенным в тело `<c:catch>`, и сохраняет информацию об ошибке в переменной, определенной атрибутом `var`. При использовании действия `<c:catch>`, в случае возникновения исключительной ситуации, не генерируется страничка ошибки (error page).

- Действие **`<c:if>`**.

  Синтаксис:

  ```
  <c:if test="testCondition"
  var="varName" [scope="{page|request|session|application}"]/>
  ```

  или:

  ```
  <c:if test="testCondition"
  [var="varName"] [scope="{page|request|session|application}"]>
  body content
  </c:if>
  ```

Действие <c:if> вычисляет выражение, определенное атрибутом test, и если результат вычисления (сохраняется в переменной, определенной атрибутом var) — true, тогда исполняется тело тега <c:if> и результат исполнения помещается в исходящий документ.

- Действие **<c:choose>**.

  Синтаксис:

```
<c:choose>
  body content (<when> and <otherwise> subtags)
</c:choose>
```

  Действие <c:choose> исполняет тело первого тега <c:when>, чье условие (определенное атрибутом test) дает значение true. Если ни одно действие <c:when> не дает значение условия true, тогда исполняется тело тега <c:otherwise>. Результат действия <c:choose> помещается в исходящий документ.

- Действие **<c:when>**.

  Синтаксис:

```
<c:when test="testCondition">
  body content
</c:when>
```

  Действие <c:when> является вложенным для действия <c:choose>.

- Действие **<c:otherwise>**.

  Синтаксис:

```
<c:otherwise>
  conditional block
</c:otherwise>
```

  Действие <c:otherwise> является вложенным для действия <c:choose>.

- Действие **<c:forEach>**.

  Цикл в коллекции объектов:

```
<c:forEach[var="varName"] items="collection"
[varStatus="varStatusName"]
[begin="begin"] [end="end"] [step="step"]>
body content
</c:forEach>
```

  Обычный цикл:

```
<c:forEach [var="varName"]
[varStatus="varStatusName"]
begin="begin" end="end" [step="step"]>
body content
</c:forEach>
```

  Действие <c:forEach> организует цикл, в котором содержимое тела тега исполняется заданное количество раз, при этом результат исполнения помещается в исходящий документ.

Атрибуты действия `<c:forEach>`:

- `var` — переменная цикла, тип переменной определяется типом объектов коллекции. В случае обычного цикла это целое число;
- `Items` — коллекция объектов: массивы, реализация `java.util.Collection`, реализация `java.util.Iterator`, реализация `java.util.Enumeration`, реализация `java.util.Map`, список строк;
- `varStatus` — переменная статуса цикла — экземпляр класса `javax.servlet.jsp.jstl.core.LoopTagStatus`, обеспечивающего информацией о цикле;
- `begin` — индекс начала цикла;
- `end` — индекс конца цикла;
- `step` — шаг цикла.

❐ Действие **`<c:forTokens>`**.

Синтаксис:

```
<c:forTokens items="stringOfTokens" delims="delimiters"
[var="varName"]
[varStatus="varStatusName"]
[begin="begin"] [end="end"] [step="step"]>
body content
</c:forTokens>
```

Действие `<c:forTokens>` организует цикл в строке лексем, разделенных указанными разделителями. В цикле содержимое тела тега исполняется заданное количество раз, при этом результат исполнения помещается в исходящий документ.

Атрибуты действия `<c:forTokens>`:

- `items` — строка лексем;
- `delims` — набор разделителей;
- `var` — переменная цикла;
- `varStatus` — переменная статуса цикла — экземпляр класса `javax.servlet.jsp.jstl.core.LoopTagStatus`, обеспечивающего информацией о цикле;
- `begin` — индекс начала цикла;
- `end` — индекс конца цикла;
- `step` — шаг цикла.

❐ Действие **`<c:redirect>`**.

Синтаксис:

```
<c:redirect url="value" [context="context"]/>
```

или:

```
<c:redirect url="value" [context="context"]>
  <c:param> subtags
</c:redirect>
```

Действие <c:redirect> посылает ответ клиенту, содержащий URL-адрес ресурса для переадресации клиентского запроса. Перед отправкой ответа клиенту контейнер выполняет вложенные действия <c:param>, которые добавляют параметры клиентского запроса в URL-адрес. Отличие от действия <jsp:forward> состоит в том, что при использовании действия <c:redirect> переадресация осуществляется клиентским браузером, а при использовании действия <jsp:forward> переадресация выполняется сервером. Без атрибута context URL-адрес ссылается на ресурс контекста данного сервлета, использование же атрибута context позволяет сослаться на ресурс именованного контекста.

- Действие **<c:param>.**

Синтаксис:

```
<c:param name="name" value="value"/>
```

или:

```
<c:param name="name">
  parameter value
</c:param>
```

Действие <c:param> добавляет параметры запроса в URL-адрес.

- Действие **<c:url>.**

Синтаксис:

```
<c:url value="value" [context="context"]
[var="varName"] [scope="{page|request|session|application}"]/>
```

или:

```
<c:url value="value" [context="context"]
[var="varName"] [scope="{page|request|session|application}"]>
  <c:param> subtags
</c:url>
```

Действие <c:url> создает URL-адрес и сохраняет его в переменной, имя которой определяется атрибутом var и область видимости которой определяется атрибутом scope. Атрибут context указывает именованный контекст ресурса, относительный URL-адрес которого создается. Действия <c:param> добавляют параметры клиентского запроса в URL-адрес.

- Действие **<c:import>.**

Содержимое ресурса включается как объект String:

```
<c:import url="url" [context="context"]
[var="varName"] [scope="{page|request|session|application}"]
[charEncoding="charEncoding"]>
optional body content for <c:param> subtags
</c:import>
```

Содержимое ресурса включается как объект Reader:

```
<c:import url="url" [context="context"]
varReader="varReaderName"
```

```
[charEncoding="charEncoding"]>
body content where varReader is consumed by another action
</c:import>
```

Основное отличие действия `<c:import>` от действия `<jsp:include>` состоит в том, что действие `<jsp:include>` позволяет включать в JSP-страницу статические и динамические ресурсы, относящиеся только к данному приложению. Действие же `<c:import>` позволяет включать в JSP-страницу контент, расположенный вне данного приложения.

Атрибуты действия `<c:import>`:

- `url` — относительный URL-адрес ресурса текущего контекста, относительный URL-адрес ресурса именованного (стороннего) контекста, абсолютный URL-адрес внешнего ресурса;
- `context` — именованный (сторонний) контекст включаемого ресурса;
- `var` — имя переменной, сохраняющей содержимое ресурса, тип переменной `String`;
- `scope` — область видимости переменной `var`;
- `charEncoding` — кодировка содержимого включаемого ресурса;
- `varReader` — имя переменной, сохраняющей содержимое ресурса, тип переменной `Reader`.

Тег *<fmt>*

JSTL-библиотека fmt решает задачи локализации и форматирования.

Локализация — это адаптация языка и времени для пользователей определенного региона.

Для настольных приложений виртуальная машина JVM обеспечивает локализацию и установки по умолчанию путем взаимодействия с локальной операционной системой. В случае распределенных приложений HTTP-протокол обеспечивает информацию для сервера о локальных установках языка клиента (но не обеспечивает информацию о локальных установках времени). Если в клиентском HTTP-запросе отсутствует заголовок `Accept-Language`, содержащий информацию о локальных установках языка клиента, тогда используются локальные установки по умолчанию, которые виртуальная машина Java контейнера получает от операционной системы сервера. JSTL-библиотека fmt позволяет переопределить локальные установки для исходящего документа клиенту.

❒ Действие **`<fmt:setLocale>`**.

Синтаксис:

```
<fmt:setLocale value="locale"
[variant="variant"]
[scope="{page|request|session|application}"]/>
```

Действие <fmt:setLocale> сохраняет и тем самым устанавливает указанную локализацию в конфигурационной переменной с именем javax.servlet.jsp.jstl.fmt.locale (тип переменной java.util.Locale).

Атрибуты действия <fmt:setLocale>:

- value — код языка (например, en_US);
- scope — область видимости конфигурационной переменной;
- variant — специфический вариант для платформы Web-браузера (например, для Microsoft Windows — WIN).

❒ Действие **<fmt:message>**.

Синтаксис без тела:

```
<fmt:message key="messageKey"
[bundle="resourceBundle"]
[var="varName"]
[scope="{page|request|session|application}"]/>
```

Синтаксис с телом, определяющим параметры сообщения:

```
<fmt:message key="messageKey"
[bundle="resourceBundle"]
[var="varName"]
[scope="{page|request|session|application}"]>
  <fmt:param> subtags
</fmt:message>
```

Синтаксис с телом, определяющим ключ и параметры сообщения:

```
<fmt:message [bundle="resourceBundle"]
[var="varName"]
[scope="{page|request|session|application}"]>
  key
  optional <fmt:param> subtags
</fmt:message>
```

Действие <fmt:message> находит текстовое сообщение, связанное с указанным ключом, в ресурсном файле и или сохраняет его в переменной, имя которой определяется атрибутом var и область видимости которой определяется атрибутом scope, или выводит сообщение сразу в исходящий документ клиенту. Атрибут key указывает ключ сообщения в ресурсном файле.

Действие <fmt:message> осуществляет поиск ресурсного файла с помощью экземпляра класса javax.servlet.jsp.jstl.fmt.LocalizationContext, который хранит информацию о ресурсном файле и локализации. Ссылка на экземпляр класса LocalizationContext может храниться либо в конфигурационной переменной javax.servlet.jsp.jstl.fmt.localizationContext, либо в переменной с определенной областью видимости. Экземпляр LocalizationContext может быть создан с помощью действия <fmt:setBundle> или действия <fmt:bundle>. Используя атрибут bundle действия <fmt:message>, можно указать экземпляр LocalizationContext, ко-

торый будет использоваться действием <fmt:message> для поиска ресурсного файла. Если атрибут bundle не указан, тогда для поиска ресурсного файла используется конфигурационная переменная. Можно также переопределить поиск ресурсного файла, вложив действие <fmt:message> в действие <fmt:bundle>.

- Действие **<fmt:bundle>**.

  Синтаксис:

```
<fmt:bundle basename="basename"
[prefix="prefix"]>
  body content
</fmt:bundle>
```

  Действие <fmt:bundle> создает экземпляр класса LocalizationContext, который используется вложенными в <fmt:bundle> действиями для поиска ресурсного файла. Имя ресурсного файла состоит из базовой части, локализации и расширения (например, message_en.properties). Атрибут basename указывает базовую часть имени ресурсного файла. С помощью необязательного атрибута prefix можно указать общий для всех используемых в теле <fmt:bundle> ключей ресурсного файла префикс, при этом в теле <fmt:bundle> можно указывать уже короткие имена ключей.

- Действие **<fmt:setBundle>**.

  Синтаксис:

```
<fmt:setBundle basename="basename"
[var="varName"]
[scope="{page|request|session|application}"]/>
```

  Действие <fmt:setBundle> создает экземпляр LocalizationContext и сохраняет его или в переменной, имя которой определяется атрибутом var и область видимости которой определяется атрибутом scope, или в конфигурационной переменной javax.servlet.jsp.jstl.fmt.localizationContext.

- Действие **<fmt:param>**.

  Синтаксис:

```
<fmt:param value="messageParameter"/>
```

  или:

```
<fmt:param>
  body content
</fmt:param>
```

  Действия <fmt:param> являются вложенными для действия <fmt:message> и определяют параметры сообщения, указанные в ресурсном файле, согласно порядку расположения тегов <fmt:param>.

- Действие **<fmt:requestEncoding>**.

  Синтаксис:

```
<fmt:requestEncoding [value="charsetName"]/>
```

Действие <fmt:requestEncoding> может быть использовано для установки кодировки клиентского запроса в целях корректной декодировки значений параметров запроса.

- Действие **<fmt:timeZone>**.

Синтаксис:

```
<fmt:timeZone value="timeZone">
  body content
</fmt:timeZone>
```

Действие <fmt:timeZone> определяет временную зону для тела тега, используя атрибут value, который указывает идентификатор ID временной зоны согласно java.util.TimeZone.

- Действие **<fmt:setTimeZone>**.

Синтаксис:

```
<fmt:setTimeZone value="timeZone"
[var="varName"]
[scope="{page|request|session|application}"]/>
```

Действие <fmt:setTimeZone> сохраняет идентификатор временной зоны в переменной, которая затем может быть использована в действии <fmt:timeZone>. Атрибут var указывает имя переменной, атрибут scope — область видимости переменной. Если переменная не указана, тогда временная зона сохраняется в конфигурационной переменной javax.servlet.jsp.jstl.fmt.timeZone, тем самым определяя указанную временную зону используемой по умолчанию.

- Действие **<fmt:formatNumber>**.

Синтаксис:

```
<fmt:formatNumber value="numericValue"
[type="{number|currency|percent}"]
[pattern="customPattern"]
[currencyCode="currencyCode"]
[currencySymbol="currencySymbol"]
[groupingUsed="{true|false}"]
[maxIntegerDigits="maxIntegerDigits"]
[minIntegerDigits="minIntegerDigits"]
[maxFractionDigits="maxFractionDigits"]
[minFractionDigits="minFractionDigits"]
[var="varName"]
[scope="{page|request|session|application}"]/>
```

или:

```
<fmt:formatNumber [type="{number|currency|percent}"]
[pattern="customPattern"]
[currencyCode="currencyCode"]
[currencySymbol="currencySymbol"]
[groupingUsed="{true|false}"]
```

```
[maxIntegerDigits="maxIntegerDigits"]
[minIntegerDigits="minIntegerDigits"]
[maxFractionDigits="maxFractionDigits"]
[minFractionDigits="minFractionDigits"]
[var="varName"]
[scope="{page|request|session|application}"]>
  numeric value to be formatted
</fmt:formatNumber>
```

Действие `<fmt:formatNumber>` используется для отображения данных в исходящем документе в виде чисел (по умолчанию), денежных сумм или процентов.

Атрибуты действия `<fmt:formatNumber>`:

- `value` — числовое значение, которое необходимо отобразить;
- `type` — тип отображаемого значения — число (по умолчанию), денежная сумма или проценты;
- `pattern` — позволяет форматировать значение согласно шаблонам класса `java.text.DecimalFormat`;
- `currencyCode` — в случае значения атрибута `type="currency"` указывает код денежной единицы согласно ISO 4217 Currency Code List (например, `EUR`);
- `currencySymbol` — в случае значения атрибута `type="currency"` указывает символ денежной единицы (например, `"$"` для US);
- `maxIntegerDigits` — максимальное количество знаков перед десятичной запятой;
- `minIntegerDigits` — минимальное количество знаков перед десятичной запятой;
- `maxFractionDigit` — максимальное количество знаков после десятичной запятой;
- `minFractionDigits` — минимальное количество знаков после десятичной запятой;
- `groupingUsed` — если `true` (по умолчанию), тогда знаки перед десятичной запятой группируются определенным образом для установленной локализации. Например, в английской версии знаки перед десятичной точкой группируются по три и разделяются запятой;
- `var` — имя переменной, в которой сохраняется отформатированный результат;
- `scope` — область видимости переменной.

❒ Действие **`<fmt:parseNumber>`**.

Синтаксис:

```
<fmt:parseNumber value="numericValue"
[type="{number|currency|percent}"]
[pattern="customPattern"]
[parseLocale="parseLocale"]
```

```
[integerOnly="{true|false}"]
[var="varName"]
[scope="{page|request|session|application}"]/>
```

или:

```
<fmt:parseNumber [type="{number|currency|percent}"]
[pattern="customPattern"]
[parseLocale="parseLocale"]
[integerOnly="{true|false}"]
[var="varName"]
[scope="{page|request|session|application}"]>
  numeric value to be parsed
</fmt:parseNumber>
```

Действие `<fmt:parseNumber>` разбирает указанное значение и возвращает результат в виде экземпляра класса `java.lang.Number`.

Атрибуты действия `<fmt:parseNumber>`:

- `value` — разбираемое значение;
- `type` — тип значения — число (по умолчанию), денежная сумма или проценты;
- `pattern` — отформатированное значение согласно шаблонам класса `java.text.DecimalFormat`;
- `parseLocale` — указывает локализацию или экземпляр класса `java.util.Locale`;
- `integerOnly` — если `true`, тогда только целая часть значения разбирается, по умолчанию — `false`;
- `var` — имя переменной, в которой сохраняется результат разбора;
- `scope` — область видимости переменной.

❒ Действие **`<fmt:formatDate>`**.

Синтаксис:

```
<fmt:formatDate value="date"
[type="{time|date|both}"]
[dateStyle="{default|short|medium|long|full}"]
[timeStyle="{default|short|medium|long|full}"]
[pattern="customPattern"]
[timeZone="timeZone"]
[var="varName"]
[scope="{page|request|session|application}"]/>
```

Действие `<fmt:formatDate>` позволяет форматировать дату и время согласно локализации и отображать результат форматирования в исходящем документе.

Атрибуты действия `<fmt:formatDate>`:

- `value` — форматируемые дата/время;
- `type` — форматируется и отображается время, дата (по умолчанию) или время и дата;

- `dateStyle` — стиль для даты согласно `java.text.DateFormat`;
- `timeStyle` — стиль для времени согласно `java.text.DateFormat`;
- `pattern` — позволяет форматировать согласно шаблонам класса `java.text.SimpleDateFormat`;
- `timeZone` — временная зона для форматирования;
- `var` — имя переменной, в которой сохраняется результат форматирования;
- `scope` — область видимости переменной.

❒ Действие **`<fmt:parseDate>`**.

Синтаксис:

```
<fmt:parseDate value="dateString"
[type="{time|date|both}"]
[dateStyle="{default|short|medium|long|full}"]
[timeStyle="{default|short|medium|long|full}"]
[pattern="customPattern"]
[timeZone="timeZone"]
[parseLocale="parseLocale"]
[var="varName"]
[scope="{page|request|session|application}"]/>
```

или:

```
<fmt:parseDate [type="{time|date|both}"]
[dateStyle="{default|short|medium|long|full}"]
[timeStyle="{default|short|medium|long|full}"]
[pattern="customPattern"]
[timeZone="timeZone"]
[parseLocale="parseLocale"]
[var="varName"]
[scope="{page|request|session|application}"]>
  date value to be parsed
</fmt:parseDate>
```

Действие `<fmt:parseDate>` позволяет разобрать отформатированные время/дату.

Атрибуты действия `<fmt:parseDate>`:

- `value` — строка, содержащая время/дату, для разбора;
- `type` — строка содержит время, дату (по умолчанию) или время и дату;
- `dateStyle` — разбор даты осуществляется согласно стилю `java.text.DateFormat`;
- `timeStyle` — разбор времени осуществляется согласно стилю `java.text.DateFormat`;
- `pattern` — разбор осуществляется согласно шаблонам класса `java.text.SimpleDateFormat`;
- `timeZone` — временная зона для разбора;
- `parseLocale` — локализация, чей стиль используется при разборе;

- `var` — имя переменной, в которой сохраняется результат разбора;
- `scope` — область видимости переменной.

Тег <*sql*>

Действия `<sql>` обеспечивают взаимодействие с реляционными базами данных.

❒ Действие **`<sql:query>`**.

Синтаксис без тела:

```
<sql:query sql="sqlQuery"
var="varName" [scope="{page|request|session|application}"]
[dataSource="dataSource"]
[maxRows="maxRows"]
[startRow="startRow"]/>
```

Синтаксис с телом, определяющим аргументы запроса:

```
<sql:query sql="sqlQuery"
var="varName" [scope="{page|request|session|application}"]
[dataSource="dataSource"]
[maxRows="maxRows"]
[startRow="startRow"]>
  <sql:param> actions
</sql:query>
```

Синтаксис с телом, определяющим запрос и его параметры:

```
<sql:query var="varName"
[scope="{page|request|session|application}"]
[dataSource="dataSource"]
[maxRows="maxRows"]
[startRow="startRow"]>
  query
  optional <sql:param> actions
</sql:query>
```

Действие `<sql:query>` осуществляет запрос к базе данных и результат запроса, содержащий строки данных, сохраняет в указанной переменной.

Атрибуты действия `<sql:query>`:

- `sql` — SQL-выражение, определяющее запрос к базе данных;
- `var` — имя переменной, сохраняющей результат запроса;
- `scope` — область видимости переменной;
- `dataSource` — указывает администрируемый на стороне сервера объект `javax.sql.DataSource`, обеспечивающий соединение с базой данных. Значение атрибута в виде строки указывает JNDI-имя объекта `DataSource` или URL-адрес базы данных с параметрами в виде `jdbc:subprotocol:host[,[driver][,[user][,password]]]` как аргумент метода `getConnection()` класса `java.sql.DriverManager`;

- `maxRows` — максимальное количество строк, включаемых в результат запроса;
- `startRow` — индекс строки, начиная с которого строки включаются в результат запроса.

❒ Действие **`<sql:update>`**.

Синтаксис без тела:

```
<sql:update sql="sqlUpdate"
[dataSource="dataSource"]
[var="varName"] [scope="{page|request|session|application}"]/>
```

Синтаксис с телом, определяющим параметры обновления базы данных:

```
<sql:update sql="sqlUpdate"
[dataSource="dataSource"]
[var="varName"] [scope="{page|request|session|application}"]>
  <sql:param> actions
</sql:update>
```

Синтаксис с телом, определяющим оператор обновления и параметры обновления базы данных:

```
<sql:update [dataSource="dataSource"]
[var="varName"] [scope="{page|request|session|application}"]>
  update statement
  optional <sql:param> actions
</sql:update>
```

Действие `<sql:update>` исполняет SQL-операторы `INSERT`, `UPDATE`, `DELETE` или другие SQL-выражения, которые ничего не возвращают.

Атрибуты действия `<sql:update>`:

- `sql` — SQL-выражение, определяющее обновление базы данных;
- `var` — имя переменной, сохраняющей результат обновления в виде количества обновленных строк базы данных;
- `scope` — область видимости переменной;
- `dataSource` — указывает администрируемый на стороне сервера объект `DataSource`, обеспечивающий соединение с базой данных. Значение атрибута в виде строки указывает JNDI-имя объекта `DataSource` или URL-адрес базы данных с параметрами в виде `jdbc:subprotocol:host[,[driver][,[user][,password]]]` как аргумент метода `getConnection()` класса `java.sql.DriverManager`.

❒ Действие **`<sql:transaction>`**.

Синтаксис:

```
<sql:transaction [dataSource="dataSource"]
[isolation=isolationLevel]>
<sql:query> and <sql:update> statements
</sql:transaction>
```

```
isolationLevel ::= "read_committed"
                 | "read_uncommitted"
                 | "repeatable_read"
                 | "serializable"
```

Действие `<sql:transaction>` устанавливает транзакцию для действий `<sql:query>` и `<sql:update>`. При использовании атрибута `dataSource` в действии `<sql:transaction>`, атрибуты `dataSource` действий `<sql:query>` и `<sql:update>` не используются. Атрибут `isolation` указывает параметры транзакции согласно `java.sql.Connection`.

- Действие **`<sql:param>`**.

Синтаксис:

```
<sql:param value="value"/>
```

или:

```
<sql:param>
  parameter value
</sql:param>
```

Действие `<sql:param>` определяет значения параметров `"?"`, указанных в SQL-выражении, согласно порядку расположения тегов `<sql:param>`.

- Действие **`<sql:setDataSource>`**.

Синтаксис:

```
<sql:setDataSource
{dataSource="dataSource" |
url="jdbcUrl"
[driver="driverClassName"]
[user="userName"]
[password="password"]}
[var="varName"]
[scope="{page|request|session|application}"]/>
```

Действие `<sql:setDataSource>` сохраняет конфигурационную информацию об объекте `DataSource` в переменной с областью видимости или, если переменная не указана, в конфигурационной переменной `javax.servlet.jsp.jstl.sql.dataSource`.

Атрибуты действия `<sql:setDataSource>`:

- `dataSource` — администрируемый на стороне сервера объект `DataSource`, обеспечивающий соединение с базой данных. Значение атрибута в виде строки указывает JNDI-имя объекта `DataSource` или URL-адрес базы данных с параметрами в виде `jdbc:subprotocol:host[,[driver][,[user][,password]]]` как аргумент метода `getConnection()` класса `java.sql.DriverManager`;
- `url` — URL-адрес базы данных;
- `driver` — полное имя класса JDBC-драйвера базы данных;
- `user` — имя пользователя базы данных;
- `password` — пароль пользователя базы данных;

- var — имя переменной, сохраняющей информацию об объекте DataSource;
- scope — область видимости переменной.

❐ Действие **<sql:dateParam>**.

Синтаксис:

```
<sql:dateParam value="value" [type="{timestamp (по умолчанию) |time|date}"]/>
```

Действие <sql:dateParam> устанавливает значения параметров "?" в SQL-выражении для значений типа java.util.Date. Атрибут value указывает значение параметра для столбцов DATE, TIME или TIMESTAMP в таблице базы данных. Атрибут type соответственно указывает тип значения параметра. Действие <sql:dateParam> является вложенным для действий <sql:query> и <sql:update>.

Тег <x>

Тег <x> представляет XML-действия, обеспечивающие обработку XML-документов. Доступ в XML-действиях библиотеки JSTL к содержимому XML-документов основан на языке выражений XPath (XML Path Language, язык запросов к элементам XML-документа).

❐ Действие **<x:parse>**.

XML-документ определен как объект String или Reader:

```
<x:parse {doc="XMLDocument"}
{var="var" [scope="scope"]|varDom="var" [scopeDom="scope"]}
[systemId="systemId"]
[filter="filter"]/>
```

XML-документ определен в теле тега:

```
<x:parse
{var="var" [scope="scope"]|varDom="var" [scopeDom="scope"]}
[systemId="systemId"]
[filter="filter"]>
  XML Document to parse
</x:parse>
```

Где scope — {page|request|session|application}.

Действие <x:parse> производит разбор указанного XML-документа из XML-формата в формат, который затем может быть использован языками XPath и XSLT (язык преобразований XML-документов), интегрированными с JSTL-библиотекой.

Атрибуты действия <x:parse>:

- doc — исходный XML-документ как объект String или Reader;
- var — имя переменной, сохраняющей результат разбора;
- scope — область видимости переменной var;
- varDom — имя переменной, сохраняющей результат разбора. Тип переменной — org.w3c.dom.Document;

- `scopeDom` — область видимости переменной `varDom`;
- `systemId` — указывает URI-адрес разбираемого документа для того, чтобы разрешить все относительные URL-адреса других документов или ресурсов, представленные в разбираемом документе;
- `filter` — указывает объект `org.xml.sax.XMLFilter`, используемый для фильтрации документа перед разбором.

❒ Действие **`<x:out>`**.

Синтаксис:

```
<x:out select="XPathExpression" [escapeXml="{true|false}"]/>
```

Действие `<x:out>` выполняет XPath-выражение и результат помещает в исходящий документ. Атрибут `select` указывает XPath-выражение, атрибут `escapeXml` — если `true` (по умолчанию), тогда символы `<`, `>`, `&`, `'`, `"` в результате выражения интерпретируются исходящим документом как символы, а не как элементы синтаксиса HTML или XML.

❒ Действие **`<x:set>`**.

Синтаксис:

```
<x:set select="XPathExpression"
var="varName" [scope="{page|request|session|application}"]/>
```

Действие `<x:set>` выполняет XPath-выражение и результат сохраняет в переменной, имя которой определяется атрибутом `var` и область видимости которой определяется переменной `scope`.

❒ Действие **`<x:if>`**.

Синтаксис:

```
<x:if select="XPathExpression"
var="varName" [scope="{page|request|session|application}"]/>
```

или:

```
<x:if select="XPathExpression"
[var="varName"] [scope="{page|request|session|application}"]>
  body content
</x:if>
```

Действие `<x:if>` вычисляет XPath-выражение и результат вычисления — `true` или `false` — сохраняет в переменной, имя которой определяется атрибутом `var` и область видимости которой определяется переменной `scope`. Если результат вычисления `true`, тогда выполняется тело тега и результат выполнения помещается в исходящий документ.

❒ Действие **`<x:choose>`**.

Синтаксис:

```
<x:choose>
  body content (<x:when> and <x:otherwise> subtags)
</x:choose>
```

Действие `<x:choose>` исполняет тело первого тега `<x:when>`, чье условие (определенное атрибутом `select`) дает значение `true`. Если ни одно действие `<x:when>` не дает значение условия `true`, тогда исполняется тело тега `<x:otherwise>`. Результат действия `<x:choose>` помещается в исходящий документ.

- Действие **`<x:when>`**.

  Синтаксис:

```
<x:when select="XPathExpression">
  body content
</x:when>
```

  Действие `<x:when>` является вложенным для действия `<x:choose>`.

- Действие **`<x:otherwise>`**.

  Синтаксис:

```
<x:otherwise>
  conditional block
</x:otherwise>
```

  Действие `<x:otherwise>` является вложенным для действия `<x:choose>`.

- Действие **`<x:forEach>`**.

  Синтаксис:

```
<x:forEach[var="varName"] select="XPathExpression">
[varStatus="varStatusName"]
[begin="begin"] [end="end"] [step="step"]>
  body content
</x:forEach>
```

  Действие `<x:forEach>` используется для организации цикла в выбранных, с помощью XPath-выражения атрибута `select`, элементах XML-документа. Тело тега выполняется при каждой итерации цикла, а результат выполнения помещается в исходящий документ.

  Атрибуты действия `<x:forEach>`:

  - `var` — переменная цикла, тип переменной зависит от XPath-выражения атрибута `select`;
  - `select` — вычисляемое XPath-выражение;
  - `varStatus` — переменная статуса цикла — экземпляр класса `javax.servlet.jsp.jstl.core.LoopTagStatus`, обеспечивающего информацией о цикле;
  - `begin` — индекс начала цикла;
  - `end` — индекс конца цикла;
  - `step` — шаг цикла.

- Действие **`<x:transform>`**.

  Синтаксис без тела:

```
<x:transform
{doc="XMLDocument"} xslt="XSLTStylesheet"
```

```
[{docSystemId="XMLSystemId"}]
[xsltSystemId="XSLTSystemId"]
[{var="varName" [scope="scopeName"]|result="resultObject"}] />
```

Синтаксис с телом, определяющим параметры трансформации:

```
<x:transform
{doc="XMLDocument"} xslt="XSLTStylesheet"
[{docSystemId="XMLSystemId"}]
[xsltSystemId="XSLTSystemId"]
[{var="varName" [scope="scopeName"]|result="resultObject"}]
  <x:param> actions
</x:transform>
```

Синтаксис с телом, содержащим трансформируемый XML-документ и определяющим параметры трансформации:

```
<x:transform
xslt="XSLTStylesheet"
[{docSystemId="XMLSystemId"}]
xsltSystemId="XSLTSystemId"
[{var="varName" [scope="scopeName"]|result="resultObject"}]
  XML Document to parse
  optional <x:param> actions
</x:parse>
```

Где `scopeName` — `{page|request|session|application}`.

Действие `<x:transform>` выполняет XSLT-трансформацию XML-документа в другой XML-документ, HTML-документ или обычный текст. XSLT-трансформация производится в соответствии с указанной в атрибуте `xslt` таблицей стилей XSLT. Результат трансформации помещается в исходящий документ или сохраняется в переменной, указанной атрибутом `var`, либо сохраняется в объекте, указанном атрибутом `result`.

Атрибуты действия `<x:transform>`:

- `doc` — объект, представляющий исходный XML-документ;
- `xslt` — объект, представляющий таблицу стилей XSLT;
- `docSystemId` — указывает URI-адрес трансформируемого документа для того, чтобы разрешить все относительные URL-адреса других документов или ресурсов, представленные в трансформируемом документе;
- `xsltSystemId` — указывает URI-адрес таблицы стилей XSLT для того, чтобы разрешить все относительные URL-адреса других документов или ресурсов, представленные в таблице стилей XSLT;
- `var` — имя переменной, сохраняющей результат трансформации. Тип переменной `org.w3c.dom.Document`;
- `scope` — область видимости переменной;
- `result` — объект `javax.xml.transform.Result`, сохраняющий результат трансформации.

- Действие **<x:param>**.

  Синтаксис:

```
<x:param name="name" value="value"/>
```

  или:

```
<x:param name="name">
  parameter value
</x:param>
```

  Действие <x:param> устанавливает значения параметров, используемых в таблице стилей XSLT. Действие <x:param> является вложенным для действия <x:transform>.

Функции библиотеки JSTL

- Функция **fn:contains**.

  Синтаксис:

```
fn:contains(string, substring) → boolean
```

  Функция fn:contains возвращает true, если подстрока *substring* входит в строку *string*.

- Функция **fn:containsIgnoreCase**.

  Синтаксис:

```
fn:containsIgnoreCase(string, substring) → boolean
```

  Функция fn:containsIgnoreCase возвращает true, если подстрока *substring* входит в строку *string* без учета регистра.

- Функция **fn:endsWith**.

  Синтаксис:

```
fn:endsWith(string, suffix) → boolean
```

  Функция fn:endsWith возвращает true, если строка *string* заканчивается последовательностью символов *suffix* (обычно используется для проверки расширения файлов).

- Функция **fn:escapeXml**.

  Синтаксис:

```
fn:escapeXml(string) → String
```

  Функция fn:escapeXml работает аналогично действию <c:out> с атрибутом escapeXml="true".

- Функция **fn:indexOf**.

  Синтаксис:

```
fn:indexOf(string, substring) → int
```

Функция `fn:indexOf` возвращает первый индекс вхождения подстроки *substring* в строку *string*.

- Функция **fn:join**.

  Синтаксис:

  `fn:join(`*array*`, `*separator*`) → String`

  Функция `fn:join` конвертирует массив *array* в строку, при этом элементы массива разделяются символами *separator*.

- Функция **fn:length**.

  Синтаксис:

  `fn:length(`*input*`) → integer`

  Функция `fn:length` возвращает количество элементов в коллекции *input* или количество символов в строке *input*.

- Функция **fn:replace**.

  Синтаксис:

  `fn:replace(`*inputString*`, `*beforeSubstring*`, `*afterSubstring*`) → String`

  Функция `fn:replace` заменяет в строке *inputString* подстроку *beforeSubstring* на подстроку *afterSubstring*.

- Функция **fn:split**.

  Синтаксис:

  `fn:split(`*string*`, `*delimiters*`) → String[]`

  Функция `fn:split` разделяет строку *string* на лексемы, определяемые символами *delimiters*, и формирует массив подстрок, при этом разделители *delimiters* не включаются в подстроки.

- Функция **fn:startsWith**.

  Синтаксис:

  `fn:startsWith(`*string*`, `*prefix*`) → boolean`

  Функция `fn:startsWith` возвращает `true`, если строка *string* начинается с символов *prefix*.

- Функция **fn:substring**.

  Синтаксис:

  `fn:substring(`*string*`, `*beginIndex*`, `*endIndex*`) → String`

  Функция `fn:substring` возвращает подстроку строки *string*, определяемую начальным и конечным индексами.

- Функция **fn:substringAfter**.

  Синтаксис:

  `fn:substringAfter(`*string*`, `*substring*`) → String`

Функция `fn:substringAfter` возвращает остаток строки *string* после подстроки *substring*.

- Функция **fn:substringBefore**.

  Синтаксис:

  `fn:substringBefore(`*string*`, `*substring*`) → String`

  Функция `fn:substringBefore` возвращает начало строки *string* перед подстрокой *substring*.

- Функция **fn:toLowerCase**.

  Синтаксис:

  `fn:toLowerCase(`*string*`) → String`

  Функция `fn:toLowerCase` конвертирует символы строки *string* в нижний регистр.

- Функция **fn:toUpperCase**.

  Синтаксис:

  `fn:toUpperCase(`*string*`) → String`

  Функция `fn:toUpperCase` конвертирует символы строки *string* в верхний регистр.

- Функция **fn:trim**.

  Синтаксис:

  `fn:trim(`*string*`) → String`

  Функция `fn:trim` удаляет пустые пространства перед началом и в конце строки *string*.

Пользовательские теги

Пользовательские теги можно создавать с помощью JSP-синтаксиса или с использованием Java-библиотеки Tag Extension API.

Файлы пользовательских тегов, созданные с помощью JSP-синтаксиса, имеют расширения tag (стандартный синтаксис), tagx (XML-синтаксис) и tagf (фрагменты).

Файлы пользовательских тегов составляют библиотеки тегов, которые размещаются в папках \META-INF\tags\ JAR-файлов, расположенных в каталоге WEB-INF\lib Web-приложения, или же файлы пользовательских тегов могут быть помещены в каталог \META-INF\tags\ самого Web-приложения. Java-классы, представляющие пользовательские теги и созданные с помощью Tag Extension API, составляют библиотеки тегов, которые в Web-приложении размещаются в папке WEB-INF\lib в виде JAR-файлов или в папке WEB-INF\classes в виде класс-файлов.

Контейнер распознает библиотеки пользовательских тегов с помощью файлов дескрипторов, имеющих расширение tdl и расположенных в папке META-INF\ Web-приложения или в папках META-INF\ JAR-файлов библиотек пользовательских тегов.

Для того чтобы использовать пользовательские теги в JSP-страницах, необходимо применить директиву `taglib` (стандартный синтаксис) или атрибут `xmlns` (XML-синтаксис), указывающие контейнеру URI-адрес дескриптора библиотеки тегов.

В случае если файлы пользовательских тегов, созданные с помощью JSP-синтаксиса, размещаются в каталоге \META-INF\tags\ самого Web-приложения, писать файл дескриптора библиотеки тегов не требуется. В этом случае контейнер сам генерирует файл дескриптора библиотеки тегов, при этом в директиве `taglib` или атрибуте `xmlns` необходимо использовать атрибут `tagdir`.

Файл дескриптора библиотеки тегов (TLD) — это XML-файл, структура которого определяется XML-схемой **http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd**.

Элементы TLD-файла:

- `taglib` — корневой элемент;
- `description`, `display-name` и `icon` — описание, отображаемое имя и значок библиотеки тегов;
- `tlib-version` — версия библиотеки;
- `short-name` — обычно значение атрибута `prefix` в директиве `taglib` или атрибуте `xmlns`;
- `uri` — URI-идентификатор библиотеки;
- `validator` — для проверки правильности использования данной библиотеки пользовательских тегов и выявления ошибок в JSP-странице на стадии трансляции можно создать класс, расширяющий `javax.servlet.jsp.tagext.TagLibraryValidator`, метод `validate()` которого осуществляет проверку JSP-страницы на предмет правильности применения пользовательских тегов и возвращает сообщение о результатах проверки. Элемент `validator` позволяет связать проверочный класс с библиотекой пользовательских тегов, содержит вложенные элементы: `description` — описание класса; `validator-class` — полное имя класса; `init-param` — параметры инициализации класса, которые определяются вложенными элементами `description`, `param-name` и `param-value`;
- `listener` — библиотека пользовательских тегов может содержать классы-прослушиватели событий (классы, реализующие интерфейсы пакетов `javax.servlet` и `javax.servlet.http` — `ServletContextListener`, `ServletRequestAttributeListener`, `ServletRequestListener`, `ServletContextAttributeListener`, `HttpSessionActivationListener`, `HttpSessionAttributeListener`, `HttpSessionBindingListener`, `HttpSessionListener`). Элемент `listener` указывает классы-прослушиватели событий для того, чтобы контейнер создал их экземпляры и зарегистрировал их. Элемент `listener` содержит вложенные элементы: `description`, `display-name` и `icon`; `listener-class` — полное имя класса-прослушивателя;
- `tag` — элемент описывает пользовательский тег, созданный с использованием Tag Extension API, имеет атрибут `id` — идентификатор тега в библиотеке. Атрибут `id`

имеют также все элементы, которые могут встречаться в файле дескриптора более одного раза. Вложенные элементы:

- `description`, `display-name` и `icon` — описание, отображаемое имя и значок тега;
- `name` — имя тега, которое указывается после префикса;
- `tag-class` — имя класса, определяющего пользовательский тег (Tag Handler), реализует интерфейс `javax.servlet.jsp.tagext.JspTag`;
- `tei-class` — если пользовательский тег использует скриптовые переменные, может быть создан класс, расширяющий `javax.servlet.jsp.tagext.TagExtraInfo`, который вызывается контейнером при JSP-компиляции для получения информации о скриптовых переменных, а также для проверки параметров тега. Элемент `tei-class` указывает полное имя класса, расширяющего класс `javax.servlet.jsp.tagext.TagExtraInfo`;
- `body-content` — указывает контейнеру тип содержимого тела тега. Возможные значения: `empty` — тело тега должно быть пустым; `scriptless` — в теле тега должны отсутствовать скриптовые элементы; `JSP` — тело тега определяется JSP-синтаксисом; `tagdependent` — тело тега интерпретируется и обрабатывается реализацией тега (Tag Handler), а не JSP-транслятором;
- `variable` — альтернативный по отношению к использованию элемента `tei-class` способ описания скриптовых переменных, определяемых тегом. Элемент `variable` описывает скриптовые переменные с помощью вложенных элементов:
  - `description` — текстовое описание переменной;
  - `name-given` — имя скриптовой переменной;
  - `name-from-attribute` — имя параметра тега, который дает имя переменной;
  - `variable-class` — Java-класс переменной, по умолчанию `java.lang.String`;
  - `declare` — если `true` (по умолчанию), тогда переменная объявляется в вызывающей тег JSP-странице;
  - `scope` — область видимости переменной: `NESTED` (по умолчанию, между открывающим и закрывающим тегами), `AT_BEGIN` (от открывающего тега и до конца JSP-страницы), `AT_END` (от закрывающего тега и до конца JSP-страницы);
- `attribute` — параметры тега, которые должны быть определены до вызова тега. Вложенные элементы:
  - `description` — описание;
  - `name` — имя параметра;
  - `required` — необходимый параметр или необязательный, по умолчанию `false`;
  - `rtexprvalue` — если `true`, тогда значение параметра вычисляется динамически при вызове, если `false` (по умолчанию), тогда значение параметра статическое и определяется при трансляции JSP-страницы;

- type — Java-тип значения параметра, по умолчанию java.lang.Object;
- deferred-value — если значение параметра является отложенным и передается классу тега как javax.el.ValueExpression, определяет с помощью вложенного элемента type Java-тип значения параметра, по умолчанию java.lang.Object;
- deferred-method — если значение параметра является отложенным и передается классу тега как javax.el.MethodExpression, определяет с помощью вложенного элемента method-signature сигнатуру метода, вычисляющего значение параметра, по умолчанию void method();
- fragment — если true, тогда тип атрибута javax.jsp.tagext.JspFragment, по умолчанию false;

- dynamic-attributes — если true, тогда тег имеет параметры с динамическими именами. В этом случае класс тега должен реализовывать интерфейс javax.servlet.jsp.tagext.DynamicAttributes. По умолчанию false;
- example — пример использования тега;
- tag-extension — вложенный элемент extension-element. Содержит элементы, расширяющие JSP-спецификацию;

- tag-file — описывает тег, созданный с использованием JSP-синтаксиса, имеет атрибут id — идентификатор тега в библиотеке. Вложенные элементы:
  - description, display-name и icon — описание, отображаемое имя и значок тега;
  - name — имя тега;
  - path — относительный путь файла тега, начинающийся с /WEB-INF/tags (файл тега находится в каталоге самого Web-приложения) или /META-INF/tags (файл тега упакован в JAR-файл);
  - example — пример использования тега;
  - tag-extension — вложенный элемент extension-element. Содержит элементы, расширяющие JSP-спецификацию;
- function — описывает функции в библиотеки тегов. Вложенные элементы:
  - description, display-name и icon — описание, отображаемое имя и значок функции;
  - function-class — Java-класс, представляющий функцию;
  - function-signature — сигнатура Java-метода функции;
  - example — пример использования функции;
  - function-extension — вложенный элемент extension-element. Содержит элементы, расширяющие JSP-спецификацию;
- taglib-extension — вложенный элемент extension-element. Содержит элементы, расширяющие JSP-спецификацию.

ПРИЛОЖЕНИЕ 12

Библиотеки технологии JavaServer Faces

Библиотека JavaServer Faces API

Пакет *javax.faces*

Пакет `javax.faces` обеспечивает с помощью класса `FactoryFinder` доступ к JSF классам-фабрикам Web-приложения или реализации контейнера, таким как:

- `javax.faces.application.ApplicationFactory` — возвращает экземпляр класса `javax.faces.application.Application`, который представляет JSF Web-приложение в целом и обеспечивает его функциональность;
- `javax.faces.context.ExceptionHandlerFactory` — возвращает экземпляр класса `javax.faces.context.ExceptionHandler`, который управляет исключительными ситуациями JSF (JSR 314);
- `javax.faces.context.ExternalContextFactory` — возвращает экземпляр класса `javax.faces.context.ExternalContext`, который обеспечивает доступ к внешнему для JSF окружению. Например, экземпляр класса `ExternalContext` позволяет получить информацию, содержащуюся в объектах `ServletContext`, `ServletRequest` и `ServletResponse` (JSR 314);
- `javax.faces.context.FacesContextFactory` — возвращает экземпляр класса `javax.faces.context.FacesContext`, который содержит информацию о запросе к JSF-странице и после обработки запроса обеспечивает выдачу ответа;
- `javax.faces.lifecycle.LifecycleFactory` — возвращает экземпляр класса `javax.faces.lifecycle.Lifecycle`, который управляет жизненным циклом обработки запроса к JSF-странице;
- `javax.faces.context.PartialViewContextFactory` — возвращает экземпляр класса `javax.faces.context.PartialViewContext`, который позволяет для выбранных JSF-компонентов ограничиться только обработкой клиентского запроса без выдачи ответа или ограничиться только фазой визуализации ответа клиенту без обработки запроса. Для того чтобы пропустить соответствующие фазы жизненного цикла процесса взаимодействия с клиентом, в клиентский запрос должны быть включены параметры, связанные с полями класса `PartialViewContext` (JSR 314);

- `javax.faces.render.RenderKitFactory` — возвращает экземпляр класса `javax.faces.render.RenderKit`, представляющий набор экземпляров класса `javax.faces.render.Renderer`, который конвертирует внутреннее JSF-представление компонентов в выходной поток ответа клиенту и тем самым обеспечивает визуализацию ответа клиенту;
- `javax.faces.view.facelets.TagHandlerDelegateFactory` — возвращает экземпляр класса `javax.faces.view.facelets.TagHandlerDelegate`, который используется экземплярами классов, реализующих теги (tag handler) (JSR 314);
- `javax.faces.view.ViewDeclarationLanguageFactory` — возвращает экземпляр класса `javax.faces.view.ViewDeclarationLanguage`, который обеспечивает взаимодействие реализации View Declaration Language (VDL) со средой выполнения JSF. View Declaration Language (VDL) — это синтаксис, позволяющий использовать Facelets вместе с JavaServer Pages. Платформа Facelets (JavaServer Faces View Definition) — это дальнейшее развитие технологии JavaServer Faces для создания JSF-приложений, в которых JSF-представление конструируется с использованием HTML-стиля и созданием деревьев JSF-компонентов (JSR 314);
- `javax.faces.component.visit.VisitContextFactory` — возвращает экземпляр класса `javax.faces.component.visit.VisitContext`, который используется методом `visitTree()` класса `javax.faces.component.UIComponent` для прохождения дерева JSF-компонентов и запуска действий для набора компонентов в дереве (JSR 314).

Пакет `javax.faces` также содержит класс `FacesException`, представляющий основные исключительные ситуации JSF и интерфейс `FacesWrapper<T>` (JSR 314) для создания классов-оберток.

Пакет *javax.faces.application*

Пакет `javax.faces.application` обеспечивает связь объектов бизнес-логики приложения с JavaServer Faces, а также механизм для управления выполнением JSF-приложения.

Пакет `javax.faces.application` содержит:

- класс `Application`, представляющий JSF Web-приложение в целом и обеспечивающий его функциональность;
- класс `ApplicationFactory` — возвращает экземпляр класса `Application`;
- класс `ApplicationWrapper` — расширение класса `Application`, которое реализует интерфейс `FacesWrapper<Application>` (JSR 314);
- класс `ConfigurableNavigationHandler` — расширяет класс `NavigationHandler`, который определяет правила навигации страниц приложения (JSR 314);
- класс `FacesMessage` — используется для вывода сообщений клиенту, связанных с определенным JSF-компонентом;
- класс `NavigationCase` — представляет тег `<navigation-case>` (JSR 314);

- класс `NavigationHandler` — управляет навигацией JSF-представлений;
- класс `Resource` — экземпляры класса представляют ресурсы приложения (JSR 314);
- класс `ResourceHandler` — обеспечивает создание экземпляра `Resource`, связывая его с определенным ресурсом приложения. Если клиентский запрос является запросом к ресурсу, тогда создает исходящий поток ресурса в ответ клиенту (JSR 314);
- класс `ResourceHandlerWrapper` — расширение класса `ResourceHandler`, которое реализует интерфейс `FacesWrapper<ResourceHandler>` (JSR 314);
- класс `ResourceWrapper` — расширение класса `Resource`, которое реализует интерфейс `FacesWrapper<Resource>` (JSR 314);
- класс `StateManager` — сохраняет и восстанавливает JSF-представление между запросами;
- класс `StateManagerWrapper` — расширение класса `StateManager`, которое реализует интерфейс `FacesWrapper<StateManager>`;
- класс `ViewHandler` — управляет визуализацией ответа клиенту;
- класс `ViewHandlerWrapper` — расширение класса `ViewHandler`, которое реализует интерфейс `FacesWrapper<ViewHandler>`;
- класс `ProjectStage` — содержит информацию о текущем состоянии приложения в жизненном цикле разработки приложения (JSR 314);
- класс `ViewExpiredException` — представляет исключительные ситуации, возникающие при попытке восстановления JSF-представления;
- аннотация `ResourceDependency` — вставляет ресурсы в JSF-представление (JSR 314);
- аннотация `ResourceDependencies` является контейнером для аннотаций ResourceDependency (JSR 314).

Пакет *javax.faces.component*

Пакет `javax.faces.component` представляет библиотеку JSF-компонентов пользовательского интерфейса.

Пакет `javax.faces.component` содержит:

- интерфейс `ActionSource` — реализуется JSF-компонентом для того, чтобы быть источником событий и иметь возможность присоединения слушателя событий;
- интерфейс `ActionSource2` — расширяет `ActionSource` с поддержкой Expression Language;
- интерфейс `ContextCallback` — запускает действие для определенного JSF-компонента;
- интерфейс `ValueHolder` — реализуется JSF-компонентом для доступа к данным, связанным с данным компонентом, обеспечивает для данных компонента локализацию;

- интерфейс `EditableValueHolder` — расширяет `ValueHolder` и реализуется редактируемыми JSF-компонентами, обеспечивает присоединение слушателя изменения данных компонента и проверку вводимых данных;
- интерфейс `NamingContainer` — реализуется JSF-компонентом, являющимся контейнером для других JSF-компонентов;
- интерфейс `StateHolder` — реализуется классами, сохраняющими свое состояние между запросами;
- интерфейс `PartialStateHolder` — расширяет `StateHolder` и обеспечивает частичное сохранение состояния компонента между запросами. Интерфейс реализуется в том случае, если размер сохраняемого состояния слишком большой и его сохранение связано с большими накладными расходами. Интерфейс `PartialStateHolder` позволяет промаркировать первоначальное состояние компонента и сохранять только его изменения (JSR 314);
- интерфейс `StateHelper` — расширяет `StateHolder` и позволяет сохранять хэш-таблицы связанных с компонентом объектов в виде пар "ключ/значение" объекта (JSR 314);
- интерфейс `UniqueIdVendor` — реализуется JSF-компонентом и создает для него уникальный идентификатор (JSR 314);
- класс `UIComponent` — базовый класс для JSF-компонентов;
- класс `UIComponentBase` — расширение класса `UIComponent` с возможностью присоединения к JSF-компоненту объекта `javax.faces.component.behavior.ClientBehavior`, который добавляет функциональность компоненту с помощью скриптов, выполняющихся на стороне клиента;
- класс `UIData` — JSF-компонент, связанный с таблицей данных;
- класс `UIColumn` — представляет столбец данных для компонента `UIData`;
- класс `UICommand` — JSF-компонент, представляющий команду пользовательского интерфейса (кнопка, элемент меню, гиперссылка);
- класс `UIForm` — форма ввода для клиента;
- класс `UIGraphic` — графическое изображение;
- класс `UIOutput` — отображает клиенту значение, полученное из JavaBeans-компонента;
- класс `UIInput` — расширяет `UIOutput` с обработкой клиентского запроса;
- класс `UIMessage` — отображает сообщение клиенту;
- класс `UIMessages` — отображает набор сообщений клиенту;
- класс `UINamingContainer` — базовый класс для JSF-компонентов, являющихся контейнерами;
- класс `UIOutcomeTarget` — отображает гиперссылку (JSR 314);
- класс `UIOutput` — отображает данные с поддержкой локализации;

- класс `UIPanel` — служит панелью для вложенных компонентов, обеспечивая их компоновку;
- класс `UIParameter` — обеспечивает значения параметров для родительского компонента;
- класс `UISelectBoolean` — представляет логическое значение `true` или `false`. По умолчанию отображается как переключатель;
- класс `UISelectMany` — отображает для клиента список с возможностью выбора нескольких позиций, при этом выбранные значения могут редактироваться клиентом;
- класс `UISelectOne` — отображает для клиента список с возможностью выбора одной позиции, при этом выбранное значение может редактироваться клиентом;
- класс `UISelectItem` — представляет одну позицию для компонентов `UISelectMany` или `UISelectOne`;
- класс `UISelectItems` — представляет набор позиций для компонентов `UISelectMany` или `UISelectOne`;
- класс `UIViewParameter` — представляет параметры клиентского запроса;
- класс `UIViewRoot` — представляет корень для дерева компонентов;
- класс `UpdateModelException` — представляет исключительную ситуацию, связанную с обновлением данных (JSR 314);
- аннотация `FacesComponent` — регистрирует класс как JSF-компонент (JSR 314).

Пакет *javax.faces.component.behavior*

Пакет `javax.faces.component.behavior` (JSR 314) обеспечивает дополнительную функциональность для JSF-компонентов.

Пакет `javax.faces.component.behavior` содержит:

- интерфейс `Behavior` — базовый интерфейс, передающий зарегистрированным слушателям генерируемые события;
- интерфейс `ClientBehavior` — обеспечивает присоединение к компонентам скрипта, исполняемого на стороне клиента;
- интерфейс `ClientBehaviorHolder` — реализуется компонентами, к которым может быть присоединен скрипт, исполняемый на стороне клиента;
- класс `BehaviorBase` — базовый класс, реализующий `Behavior` и обеспечивающий регистрацию слушателей и сохранение состояния;
- класс `ClientBehaviorBase` — расширяет `BehaviorBase` и реализует `ClientBehavior`;
- класс `AjaxBehavior` — расширяет `ClientBehaviorBase` и обеспечивает выполнение кода JavaScript, который содержит AJAX-запрос;
- класс `ClientBehaviorContext` — содержит информацию, необходимую для выполнения скрипта;

- перечисление `ClientBehaviorHint` — содержит информацию, описывающую поведение реализации `ClientBehavior`;
- аннотация `FacesBehavior` — регистрирует класс как `Behavior`.

Пакет *javax.faces.component.html*

Пакет `javax.faces.component.html` содержит классы, представляющие теги библиотеки JavaServer Faces Standard HTML RenderKit Tag Library, использование которой декларируется в JSP-странице с помощью директивы:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
```

Пакет *javax.faces.component.visit*

Пакет `javax.faces.component.visit` (JSR 314) содержит интерфейсы и классы, используемые при обработке дерева компонентов:

- интерфейс `VisitCallback` — содержит метод `visit()`, вызываемый для определенного компонента при прохождении дерева компонентов;
- класс `VisitContext` — содержит информацию о наборе компонентов, используемую методом `visitTree()` класса `UIComponent` при прохождении дерева компонентов;
- класс `VisitContextFactory` — обеспечивает получение объекта `VisitContext`;
- класс `VisitContextWrapper` — класс-обертка для `VisitContext`;
- перечисление `VisitHint` — влияет на прохождение дерева компонентов и используется при создании объекта `VisitContext`;
- перечисление `VisitResult` — определяет результат прохождения дерева компонентов.

Пакет *javax.faces.context*

Пакет `javax.faces.context` состоит из классов, определяющих информацию о состоянии обработки клиентского запроса.

Пакет `javax.faces.context` содержит:

- класс `ExceptionHandler` — управляет исключительными ситуациями, возникающими в жизненном цикле JSF (JSR 314);
- класс `ExceptionHandlerFactory` — обеспечивает получение объекта `ExceptionHandler` (JSR 314);
- класс `ExceptionHandlerWrapper` — класс-обертка для `ExceptionHandler` (JSR 314);
- класс `ExternalContext` — обеспечивает доступ к внешнему для JSF окружению;
- класс `ExternalContextFactory` — обеспечивает получение объекта `ExternalContext`;
- класс `ExternalContextWrapper` — класс-обертка для `ExternalContext` (JSR 314);

- класс `FacesContext` — содержит информацию о состоянии обработки запроса и визуализации ответа. Объект `FacesContext` создается перед обработкой запроса, модифицируется в каждой фазе жизненного цикла обработки запроса и после завершения цикла уничтожается или помещается контейнером в пул;
- класс `FacesContextFactory` — обеспечивает получение объекта `FacesContext`;
- класс `FacesContextWrapper` — класс-обертка для `FacesContext` (JSR 314);
- класс `Flash` — в объект Flash можно поместить объекты, которые будут переданы и визуализированы в следующем JSF-представлении в пределах одной сессии (JSR 314);
- класс `PartialResponseWriter` — обеспечивает исходящий поток ответа на AJAX-запрос (JSR 314);
- класс `PartialViewContext` — позволяет для выбранных JSF-компонентов выполнить фазы жизненного цикла обработки запроса — Apply Request Values, Process Validations и Update Model Values, и/или фазу Render Response (JSR 314);
- класс `PartialViewContextFactory` — обеспечивает получение объекта `PartialViewContext` (JSR 314);
- класс `PartialViewContextWrapper` — класс-обертка для `PartialViewContext` (JSR 314);
- класс `ResponseStream` — представляет бинарный исходящий поток ответа;
- класс `ResponseWriter` — представляет символьный исходящий поток ответа;
- класс `ResponseWriterWrapper` — класс-обертка для `ResponseWriter`.

Пакет *javax.faces.convert*

Пакет `javax.faces.convert` содержит интерфейс `Converter` и классы, реализующие данный интерфейс, которые обеспечивают конвертацию данных, представленных различными типами Java-объектов, в объект `String`, который используется для визуализации клиенту, и наоборот. Пакет содержит аннотацию `FacesConverter` (JSR 314), маркирующую Java-класс как `Converter`.

Пакет *javax.faces.event*

Пакет `javax.faces.event` включает в себя интерфейсы и классы, обеспечивающие обработку событий.

Пакет `javax.faces.event` содержит:

- интерфейс `ActionListener` — интерфейс содержит метод `processAction()`, вызываемый при получении события `ActionEvent`;
- интерфейс `AjaxBehaviorListener` — интерфейс содержит метод `processAjaxBehavior()`, вызываемый при получении события `AjaxBehaviorEvent` (JSR 314);
- интерфейс `BehaviorListener` — базовый интерфейс для слушателей событий `BehaviorEvent` (JSR 314);

- интерфейс `ComponentSystemEventListener` — интерфейс содержит метод `processEvent()`, вызываемый при получении события `ComponentSystemEvent` (JSR 314);
- интерфейс `FacesListener` — базовый интерфейс для слушателей событий `FacesEvent`;
- интерфейс `PhaseListener` — содержит методы `beforePhase()` и `afterPhase()`, вызываемые перед началом и после окончания каждой фазы жизненного цикла обработки запроса;
- интерфейс `SystemEventListener` — содержит метод `processEvent()`, вызываемый при получении события `SystemEvent` (JSR 314);
- интерфейс `SystemEventListenerHolder` — содержит метод `getListenersForEventClass()`, создающий список событий `SystemEventListener`;
- интерфейс `ValueChangeListener` — содержит метод `processValueChange()`, вызываемый при получении события `ValueChangeEvent`;
- интерфейс `ViewMapListener` — содержит метод `processEvent()`, вызываемый при получении события `PostConstructViewMapEvent` или `PreDestroyViewMapEvent` (JSR 314);
- класс `ActionEvent` — представляет событие, связанное с активацией клиентом компонента;
- класс `AjaxBehaviorEvent` — представляет событие, связанное с инициацией клиентом AJAX-запроса (JSR 314);
- класс `BehaviorEvent` — представляет события, генерируемые объектами `Behavior` (JSR 314);
- класс `SystemEvent` — базовый класс для представления системных событий, таких как запуск или завершение приложения, а также возникновение системных исключительных ситуаций (JSR 314);
- класс `ComponentSystemEvent` — базовый класс для представления системных событий, связанных с JSF-компонентом, таких как добавление или удаление компонента из JSF-представления, проверка компонента и др. (JSR 314);
- класс `ExceptionQueuedEvent` — представляет системную исключительную ситуацию (JSR 314);
- класс `ExceptionQueuedEventContext` — обеспечивает информацию о состоянии системы во время возникновения исключительной ситуации `ExceptionQueuedEvent` (JSR 314);
- класс `FacesEvent` — базовый класс для событий, генерируемых JSF-компонентами;
- класс `MethodExpressionActionListener` — при получении события `ActionEvent` выполняется EL-выражение;
- класс `MethodExpressionValueChangeListener` — при получении события `ValueChangeEvent` выполняется EL-выражение;
- класс `PhaseEvent` — представляет событие, связанное с началом или окончанием фазы жизненного цикла обработки запроса;

- класс `PhaseId` — представляет идентификатор фазы жизненного цикла обработки запроса;
- класс `PostAddToViewEvent` — представляет событие, связанное с добавлением компонента в JSF-представление (JSR 314);
- класс `PostAddToViewNonPDLEvent` — представляет событие, связанное с добавлением компонента в JSF-представление в фазе жизненного цикла, отличной от `PhaseId.RESTORE_VIEW` (восстановление представления) (JSR 314);
- класс `PostConstructApplicationEvent` — представляет событие, связанное с запуском приложения (JSR 314);
- класс `PostConstructViewMapEvent` — представляет событие, связанное с первым созданием таблицы данных представления (JSR 314);
- класс `PostRestoreStateEvent` — представляет событие, связанное с восстановлением состояния компонента (JSR 314);
- класс `PostValidateEvent` — представляет событие, связанное с окончанием проверки компонента (JSR 314);
- класс `PreDestroyApplicationEvent` — представляет событие, связанное с завершением приложения (JSR 314);
- класс `PreDestroyViewMapEvent` — представляет событие, связанное с обнулением таблицы данных представления (JSR 314);
- класс `PreRemoveFromViewEvent` — представляет событие, связанное с удалением компонента из JSF-представления (JSR 314);
- класс `PreRenderComponentEvent` — представляет событие, связанное с визуализацией компонента (JSR 314);
- класс `PreRenderViewEvent` — представляет событие, связанное с визуализацией корневого компонента дерева (JSR 314);
- класс `PreValidateEvent` — представляет событие, связанное с началом проверки компонента (JSR 314);
- класс `ValueChangeEvent` — представляет событие, связанное с изменением значения компонента;
- класс `AbortProcessingException` — представляет исключительную ситуацию, возникающую при обработке событий;
- аннотация `ListenerFor` — маркирует класс как слушателя событий (JSR 314);
- аннотация `ListenersFor` — является контейнером для аннотаций `ListenerFor` (JSR 314);
- аннотация `NamedEvent` — регистрирует класс как `ComponentSystemEvent` (JSR 314).

Пакет *javax.faces.lifecycle*

Пакет `javax.faces.lifecycle` содержит класс `Lifecycle`, отвечающий за управление жизненным циклом обработки запроса, а также класс `LifecycleFactory`, обеспечивающий получение объекта `Lifecycle`.

Пакет *javax.faces.model*

Пакет `javax.faces.model` представляет уровень данных для JavaServer Faces и включает в себя:

- интерфейс `DataModelListener` — содержит метод `rowSelected()`, вызываемый в случае выбора столбца таблицы данных для обработки;
- класс `DataModel<E>` — представляет таблицу данных;
- класс `ArrayDataModel<E>` — представляет массив Java-объектов;
- класс `ListDataModel<E>` — представляет список Java-объектов;
- класс `ResultDataModel` — представляет результат выполнения JSTL SQL-запроса;
- класс `ResultSetDataModel` — представляет таблицу данных `java.sql.ResultSet`, полученную в результате запроса к базе данных;
- класс `ScalarDataModel<E>` — представляет строку таблицы данных;
- класс `SelectItem` — представляет позицию в списке выбора `UISelectMany` или `UISelectOne` компонентов;
- класс `SelectItemGroup` — представляет набор позиций (подменю) в списке выбора `UISelectMany` или `UISelectOne` компонентов.

Пакет *javax.faces.render*

Технология JavaServer Faces поддерживает две модели конвертации внутреннего представления JSF-компонентов в исходящий поток ответа клиенту.

Первая модель — direct implementation, в которой JSF-компонент производит конвертацию и визуализацию самостоятельно.

Вторая модель — delegated implementation, в которой за конвертацию и визуализацию JSF-компонента отвечает связанный с ним объект класса `Renderer` пакета `javax.faces.render`. Объекты `Renderer`, поддерживающие визуализацию всех компонентов JSF-представления, упакованы в объект `RenderKit`.

Кроме классов `Renderer` и `RenderKit` пакет `javax.faces.render` содержит классы:

- `RenderKitFactory` — отвечает за регистрацию и получение экземпляров `RenderKit`;
- `RenderKitWrapper` — класс-обертка для `RenderKit` (JSR 314);
- `ResponseStateManager` — отвечает за сохранение состояния JSF-представления между запросами;
- `ClientBehaviorRenderer` — отвечает за выполнение на стороне клиента скрипта, определенного объектом `ClientBehavior` (JSR 314);
- аннотация `FacesBehaviorRenderer` — регистрирует класс как `ClientBehaviorRenderer` (JSR 314);
- аннотация `FacesRenderer` — регистрирует класс как `Renderer` (JSR 314).

Пакет *javax.faces.validator*

Пакет `javax.faces.validator` включает в себя классы и интерфейсы, обеспечивающие проверку вводимых клиентом значений компонентов.

Проверка значений компонентов типа `javax.faces.component.EditableValueHolder` осуществляется в фазе Process Validations жизненного цикла обработки запроса с помощью проверочного класса, реализующего интерфейс `Validator` пакета `javax.faces.validator`. Проверочные классы присоединяются к компоненту методом `addValidator()` интерфейса `EditableValueHolder`.

Пакет `javax.faces.validator` содержит:

- интерфейс `Validator` — имеет метод `validate()`, вызываемый контейнером в фазе Process Validations жизненного цикла обработки запроса и осуществляющий проверку значения компонента;
- класс `BeanValidator` — осуществляет проверку свойства JavaBeanы-компонента, представляющего значение `EditableValueHolder`-компонента (JSR 314);
- класс `DoubleRangeValidator` — осуществляет проверку значения компонента типа `double` относительно максимального и минимального значений;
- класс `LengthValidator` — осуществляет проверку количества символов строки, представляющей значение компонента, относительно максимального и минимального значений;
- класс `LongRangeValidator` — осуществляет проверку значения компонента типа `long` относительно максимального и минимального значений;
- класс `MethodExpressionValidator` — осуществляет проверку значения компонента методом объекта, идентифицируемого EL-выражением;
- класс `RegexValidator` — осуществляет проверку значения компонента относительно регулярного выражения `java.util.regex` (JSR 314);
- класс `RequiredValidator` — осуществляет проверку значения компонента относительно нуля и тем самым обязывает компонент иметь непустое значение (JSR 314);
- класс `ValidatorException` — представляет исключительные ситуации, возникающие при проверке значений компонентов;
- аннотация `FacesValidator` — регистрирует класс как `Validator` (JSR 314).

Пакет *javax.faces.view*

Пакет `javax.faces.view` (JSR 314) содержит классы и интерфейсы, реализующие язык View Declaration Language (VDL) и обеспечивающие совместимость технологии Facelets (JavaServer Faces View Definition) с технологией JavaServer Pages.

Как Facelets, так и JSP являются реализациями VDL. В этом смысле Facelets используется как view declaration language для создания JSF-страниц.

Технология Facelets заменяет технологию JavaServer Pages для создания JSF-страниц и является более быстрой и простой по сравнению с JSP.

Пакет *javax.faces.view.facelets*

Пакет `javax.faces.view.facelets` (JSR 314) представляет библиотеку Java code API of Facelets платформы Facelets (JavaServer Faces View Definition), являющейся заменой JSP для создания JSF-страниц.

Пакет *javax.faces.webapp*

Пакет `javax.faces.webapp` содержит классы, обеспечивающие интеграцию JavaServer Faces в Web-приложения, включая стандартные сервлеты, а также базовые классы для создания пользовательских тегов компонентов и классы, реализующие стандартную библиотеку тегов JSF Core Tags.

Библиотеки тегов технологии JavaServer Faces

Библиотека тегов Standard HTML RenderKit Tag Library

Библиотека тегов Standard HTML RenderKit Tag Library имеет префикс `h` и пространство имен URI **http://java.sun.com/jsf/html**.

Стандартный синтаксис:

```
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
```

XML-синтаксис:

```
<anyxmlelement xmlns:h="http://java.sun.com/jsf/html" />
```

- Тег **<h:dataTable>** визуализирует HTML-таблицу.

  Атрибуты тега:

  - `bgcolor` — цвет фона;
  - `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
  - `bodyrows` — перечень индексов строк, для которых открывается HTML-тег `<tbody>`;
  - `border` — ширина в пикселах рамки таблицы;
  - `captionClass` — определяет атрибут `class` HTML-тега `<caption>`, содержащий список CSS-селекторов `class` для стиля заголовка таблицы;
  - `captionStyle` — определяет атрибут `style` HTML-тега `<caption>`, указывающий стиль для заголовка таблицы;
  - `cellpadding` — отступ от границы ячейки таблицы до ее содержимого;

- `cellspacing` — расстояние между ячейками;
- `columnClasses` — содержит список CSS-селекторов `class` для стиля столбцов таблицы;
- `dir` — направление текста: `"LTR"` (слева направо) или `"RTL"` (справа налево);
- `first` — номер строки, с которой визуализируется таблица;
- `footerClass` — содержит список CSS-селекторов `class` для стиля подстрочника таблицы;
- `frame` — указывает отображение границ вокруг таблицы: `none` (нет границ), `above` (верхняя граница), `below` (нижняя граница), `hsides` (верхняя и нижняя границы), `vsides` (правая и левая границы), `lhs` (левая граница), `rhs` (правая граница), `box` (все границы), `border` (все границы);
- `headerClass` — содержит список CSS-селекторов `class` для стиля заголовка таблицы;
- `id` — идентификатор таблицы;
- `lang` — язык содержимого таблицы (например, `en-US`);
- `onclick` — код JavaScript, выполняемый при щелчке мыши на таблице;
- `ondblclick` — код JavaScript, выполняемый при двойном щелчке мыши на таблице;
- `onkeydown` — код JavaScript, выполняемый при нажатии клавиши на таблице;
- `onkeypress` — код JavaScript, выполняемый при нажатии и освобождении клавиши на таблице;
- `onkeyup` — код JavaScript, выполняемый при освобождении клавиши на таблице;
- `onmousedown` — код JavaScript, выполняемый при нажатии мыши на таблице;
- `onmousemove` — код JavaScript, выполняемый при движении мыши внутри таблицы;
- `onmouseout` — код JavaScript, выполняемый при выходе мыши из таблицы;
- `onmouseover` — код JavaScript, выполняемый при движении мыши вне таблицы;
- `onmouseup` — код JavaScript, выполняемый при освобождении мыши на таблице;
- `rendered` — EL-выражение, определяющее, отображается данный компонент или нет (`true`/`false`);
- `rowClasses` — содержит список CSS-селекторов `class` для стиля строк таблицы;
- `rows` — количество отображаемых строк, начиная с первой;
- `rules` — отображение внутренних рамок в таблице: `none` (нет границ), `groups` (границы между группами строк), `rows` (границы между строками), `cols` (границы между столбцами), `all` (границы между строками и столбцами);

- `style` — CSS-стиль для таблицы;
- `styleClass` — список CSS-селекторов `class` для стиля таблицы;
- `summary` — описание таблицы для невизуального отображения;
- `title` — всплывающая подсказка при наведении курсора на таблицу;
- `value` — текущее значение для компонента;
- `var` — имя параметра, устанавливающего данные для строк при каждой итерации в таблице;
- `width` — ширина таблицы.

❒ Тег **`<h:column>`** визуализирует столбец таблицы. Атрибуты тега: `footerClass`, `headerClass`, `binding`, `id` и `rendered`.

❒ Тег **`<h:commandButton>`** визуализирует кнопку **Submit**.

Атрибуты тега:

- `accesskey` — клавиша доступа к компоненту;
- `action` — EL-выражение, указывающее метод JavaBeans-компонента, который вызывается при нажатии кнопки пользователем и определяет следующее JSF-представление;
- `actionListener` — EL-выражение, определяющее метод JavaBeans-компонента, который является слушателем события, связанного с нажатием кнопки пользователем;
- `alt` — альтернативный текст, если компонент не отображается;
- `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- `dir` — направление текста: `"LTR"` (слева направо) или `"RTL"` (справа налево);
- `disabled` — деактивация кнопки (`true`/`false`);
- `id` — идентификатор компонента;
- `image` — URL-адрес изображения кнопки;
- `immediate` — пропуск фазы проверки JSF-цикла (`true`/`false`);
- `label` — локализованное описание компонента;
- `lang` — язык текста компонента (например, `en-US`);
- `onblur` — код JavaScript, выполняемый при потере фокуса компонентом;
- `onchange` — код JavaScript, выполняемый при изменении значения компонента;
- `onclick` — код JavaScript, выполняемый при щелчке мыши;
- `ondblclick` — код JavaScript, выполняемый при двойном щелчке мыши;
- `onfocus` — код JavaScript, выполняемый при наведении фокуса на компонент;

- `onkeydown` — код JavaScript, выполняемый при нажатии клавиши;
- `onkeypress` — код JavaScript, выполняемый при нажатии и освобождении клавиши;
- `onkeyup` — код JavaScript, выполняемый при освобождении клавиши;
- `onmousedown` — код JavaScript, выполняемый при нажатии мыши;
- `onmousemove` — код JavaScript, выполняемый при движении мыши внутри;
- `onmouseout` — код JavaScript, выполняемый при выходе мыши из компонента;
- `onmouseover` — код JavaScript, выполняемый при движении мыши вне компонента;
- `onmouseup` — код JavaScript, выполняемый при освобождении мыши;
- `onselect` — код JavaScript, выполняемый при выделении текста компонента;
- `readonly` — запрет изменения компонента пользователем (`true`/`false`);
- `rendered` — EL-выражение, определяющее, отображается данный компонент или нет (`true`/`false`);
- `style` — CSS-стиль для компонента;
- `styleClass` — содержит список CSS-селекторов `class` для стиля компонента;
- `tabindex` — с помощью клавиши <Tab> на странице можно осуществлять переход по элементам, которые способны принимать фокус. Атрибут `tabindex` принимает целочисленные неотрицательные значения от 0 до 32 767, задающие порядок, в котором будет совершаться tab-обход;
- `title` — всплывающая подсказка;
- `type` — тип кнопки: `"submit"` (по умолчанию) или `"reset"`;
- `value` — текущее значение компонента.

❒ Тег **`<h:commandLink>`** представляет HTML-гиперссылку `<a href="">` как JSF-команду.

Атрибуты тега:

- `accesskey` — клавиша доступа к компоненту;
- `action` — EL-выражение, указывающее метод JavaBeans-компонента, который вызывается при нажатии ссылки пользователем и определяет следующее JSF-представление;
- `actionListener` — EL-выражение, определяющее метод JavaBeans-компонента, который является слушателем события, связанного с нажатием ссылки пользователем;
- `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- `charset` — кодировка ресурса гиперссылки;

- coords и shape — позиция и форма участка экрана при использовании гиперссылки в HTML-теге <MAP>. Возможные значения атрибута shape: default, rect, circle и poly;
- dir — направление текста: "LTR" (слева направо) или "RTL" (справа налево);
- disabled — деактивация кнопки (true/false);
- hreflang — язык ресурса гиперссылки;
- id — идентификатор компонента;
- immediate — пропуск фазы проверки JSF-цикла (true/false);
- lang — язык текста компонента;
- onblur — код JavaScript, выполняемый при потере фокуса компонентом;
- onclick — код JavaScript, выполняемый при щелчке мыши;
- ondblclick — код JavaScript, выполняемый при двойном щелчке мыши;
- onfocus — код JavaScript, выполняемый при наведении фокуса на компонент;
- onkeydown — код JavaScript, выполняемый при нажатии клавиши;
- onkeypress — код JavaScript, выполняемый при нажатии и освобождении клавиши;
- onkeyup — код JavaScript, выполняемый при освобождении клавиши;
- onmousedown — код JavaScript, выполняемый при нажатии мыши;
- onmousemove — код JavaScript, выполняемый при движении мыши внутри;
- onmouseout — код JavaScript, выполняемый при выходе мыши из компонента;
- onmouseover — код JavaScript, выполняемый при движении мыши вне компонента;
- onmouseup — код JavaScript, выполняемый при освобождении мыши;
- rel — соотношение между текущим документом и связанным (распознаваемые типы ссылок: "index", "next", "prev" и т. д.);
- rendered — EL-выражение, определяющее, отображается данный компонент или нет (true/false);
- rev — обратное соотношение относительно rel между связанным документом и текущим;
- style — CSS-стиль для компонента;
- styleClass — список CSS-селекторов class для стиля компонента;
- tabindex — принимает целочисленные неотрицательные значения от 0 до 32 767, задающие порядок, в котором будет совершаться tab-обход;
- target — имя фрейма, в котором будет отображаться связанный документ;
- title — всплывающая подсказка;

- `type` — тип содержимого связанного документа (например, `type="text/html"`);
- `value` — текущее значение компонента.

❒ Тег **`<h:form>`** визуализирует HTML-форму.

Атрибуты тега:

- `accept` — тип содержимого формы (например, `accept="text/html"`);
- `acceptcharset` — кодировка содержимого формы;
- `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- `dir` — направление текста: `"LTR"` (слева направо) или `"RTL"` (справа налево);
- `enctype` — определяет, как содержимое формы должно быть закодировано перед отправкой на сервер (по умолчанию `"application/x-www-form-urlencoded"`);
- `id` — идентификатор компонента;
- `lang` — язык текста компонента;
- `onclick` — код JavaScript, выполняемый при щелчке мыши;
- `ondblclick` — код JavaScript, выполняемый при двойном щелчке мыши;
- `onkeydown` — код JavaScript, выполняемый при нажатии клавиши;
- `onkeypress` — код JavaScript, выполняемый при нажатии и освобождении клавиши;
- `onkeyup` — код JavaScript, выполняемый при освобождении клавиши;
- `onmousedown` — код JavaScript, выполняемый при нажатии мыши;
- `onmousemove` — код JavaScript, выполняемый при движении мыши внутри;
- `onmouseout` — код JavaScript, выполняемый при выходе мыши из компонента;
- `onmouseover` — код JavaScript, выполняемый при движении мыши вне компонента;
- `onmouseup` — код JavaScript, выполняемый при освобождении мыши;
- `onreset` — код JavaScript, выполняемый при сбросе формы;
- `onsubmit` — код JavaScript, выполняемый при отправке содержимого формы;
- `prependId` — если `true` (по умолчанию), тогда `id` формы будет автоматически вставляться перед `id` элементов формы;
- `rendered` — EL-выражение, определяющее, отображается данный компонент или нет (`true`/`false`);
- `style` — CSS-стиль для компонента;
- `styleClass` — список CSS-селекторов `class` для стиля компонента;
- `target` — имя фрейма, в котором будет отображен ответ на запрос формы;
- `title` — всплывающая подсказка.

- Тег **<h:graphicImage>** визуализирует изображение.

  Атрибуты тега:

  - `alt` — альтернативный текст, если изображение не выводится в браузере;
  - `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
  - `dir` — направление текста: `"LTR"` (слева направо) или `"RTL"` (справа налево);
  - `height` — высота изображения;
  - `id` — идентификатор компонента;
  - `ismap` — если `true`, тогда изображение является серверной картой-изображением, см. HTML-тег `<MAP>`;
  - `lang` — язык текста компонента;
  - `longdesc` — URI-адрес полного описания компонента, дополняет `alt`;
  - `onclick` — код JavaScript, выполняемый при щелчке мыши;
  - `ondblclick` — код JavaScript, выполняемый при двойном щелчке мыши;
  - `onkeydown` — код JavaScript, выполняемый при нажатии клавиши;
  - `onkeypress` — код JavaScript, выполняемый при нажатии и освобождении клавиши;
  - `onkeyup` — код JavaScript, выполняемый при освобождении клавиши;
  - `onmousedown` — код JavaScript, выполняемый при нажатии мыши;
  - `onmousemove` — код JavaScript, выполняемый при движении мыши внутри;
  - `onmouseout` — код JavaScript, выполняемый при выходе мыши из компонента;
  - `onmouseover` — код JavaScript, выполняемый при движении мыши вне компонента;
  - `onmouseup` — код JavaScript, выполняемый при освобождении мыши;
  - `rendered` — EL-выражение, определяющее, отображается данный компонент или нет (`true`/`false`);
  - `style` — CSS-стиль для компонента;
  - `styleClass` — список CSS-селекторов `class` для стиля компонента;
  - `title` — всплывающая подсказка;
  - `url` — URL-адрес изображения;
  - `usemap` — имя клиентской карты-изображения, см. HTML-тег `<MAP>`;
  - `value` — текущее значение компонента;
  - `width` — ширина изображения.

- Тег **<h:inputHidden>** представляет элемент HTML-формы <input type="hidden">.

  Атрибуты тега:

  - binding — EL-выражение, связывающее тег со свойством компонента JavaBeans;
  - converter — указывает экземпляр javax.faces.convert.Converter, осуществляющий конвертацию данных скрытого поля формы;
  - converterMessage — текст сообщения об ошибке при конвертации значения компонента;
  - id — идентификатор компонента;
  - immediate — пропуск фазы проверки JSF-цикла (true/false);
  - rendered — EL-выражение, определяющее, отображается данный компонент или нет (true/false);
  - required — если true, тогда поле формы обязательно должно иметь значение перед тем, как содержимое формы будет отправлено серверу;
  - requiredMessage — текст сообщения об ошибке при нарушении required="true";
  - validator — EL-выражение, определяющее метод экземпляра javax.faces.validator.Validator, который осуществляет проверку значения компонента;
  - validatorMessage — текст сообщения об ошибке при проверке значения компонента;
  - value — текущее значение компонента;
  - valueChangeListener — EL-выражение, определяющее метод экземпляра javax.faces.event.ValueChangeListener — слушателя изменения значения компонента.

- Тег **<h:inputSecret>** представляет элемент HTML-формы <input type="password">.

  Атрибуты тега:

  - accesskey — клавиша доступа к элементу;
  - alt — альтернативный текст;
  - autocomplete — если "off", тогда отключается автозавершение браузером вводимой пользователем информации в поля формы;
  - binding — EL-выражение, связывающее тег со свойством компонента JavaBeans;
  - converter — указывает экземпляр javax.faces.convert.Converter, осуществляющий конвертацию данных поля формы;
  - converterMessage — текст сообщения об ошибке при конвертации значения компонента;
  - dir — направление текста: "LTR" (слева направо) или "RTL" (справа налево);

- `disabled` — деактивация элемента (`true`/`false`);
- `id` — идентификатор компонента;
- `immediate` — пропуск фазы проверки JSF-цикла (`true`/`false`);
- `label` — локализованное описание компонента;
- `lang` — язык текста компонента;
- `maxlength` — максимальное число символов строки элемента;
- `onblur` — код JavaScript, выполняемый при потере фокуса компонентом;
- `onchange` — код JavaScript, выполняемый при изменении значения компонента;
- `onclick` — код JavaScript, выполняемый при щелчке мыши;
- `ondblclick` — код JavaScript, выполняемый при двойном щелчке мыши;
- `onfocus` — код JavaScript, выполняемый при получении фокуса компонентом;
- `onkeydown` — код JavaScript, выполняемый при нажатии клавиши;
- `onkeypress` — код JavaScript, выполняемый при нажатии и освобождении клавиши;
- `onkeyup` — код JavaScript, выполняемый при освобождении клавиши;
- `onmousedown` — код JavaScript, выполняемый при нажатии мыши;
- `onmousemove` — код JavaScript, выполняемый при движении мыши внутри;
- `onmouseout` — код JavaScript, выполняемый при выходе мыши из компонента;
- `onmouseover` — код JavaScript, выполняемый при движении мыши вне компонента;
- `onmouseup` — код JavaScript, выполняемый при освобождении мыши;
- `onselect` — код JavaScript, выполняемый при выделении текста элемента;
- `readonly` — запрет изменения значения компонента пользователем (`true`/`false`);
- `redisplay` — если `false` (по умолчанию), тогда предыдущий введенный пользователем пароль не отображается формой;
- `rendered` — EL-выражение, определяющее, отображается данный компонент или нет (`true`/`false`);
- `required` — если `true`, тогда поле формы обязательно должно иметь значение перед тем, как содержимое формы будет отправлено серверу;
- `requiredMessage` — текст сообщения об ошибке при нарушении `required="true"`;
- `size` — ширина поля формы в символах;
- `style` — CSS-стиль для компонента;
- `styleClass` — список CSS-селекторов `class` для стиля компонента;

- `tabindex` — принимает целочисленные неотрицательные значения от 0 до 32 767, задающие порядок, в котором будет совершаться tab-обход;
- `title` — всплывающая подсказка;
- `validator` — EL-выражение, определяющее метод экземпляра `javax.faces.validator.Validator`, который осуществляет проверку значения компонента;
- `validatorMessage` — текст сообщения об ошибке при проверке значения компонента;
- `value` — текущее значение компонента;
- `valueChangeListener` — EL-выражение, определяющее метод экземпляра `javax.faces.event.ValueChangeListener` — слушателя изменения значения компонента.

❐ Тег **`<h:inputText>`** представляет элемент HTML-формы `<input type="text">`.

Атрибуты тега:

- `accesskey` — клавиша доступа к элементу;
- `alt` — альтернативный текст;
- `autocomplete` — если `"off"`, тогда отключается автозавершение браузером вводимой пользователем информации в поля формы;
- `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- `converter` — указывает экземпляр `javax.faces.convert.Converter`, осуществляющий конвертацию данных поля формы;
- `converterMessage` — текст сообщения об ошибке при конвертации значения компонента;
- `dir` — направление текста: `"LTR"` (слева направо) или `"RTL"` (справа налево);
- `disabled` — деактивация элемента (`true`/`false`);
- `id` — идентификатор компонента;
- `immediate` — пропуск фазы проверки JSF-цикла (`true`/`false`);
- `label` — локализованное описание компонента;
- `lang` — язык текста компонента;
- `maxlength` — максимальное число символов строки элемента;
- `onblur` — код JavaScript, выполняемый при потере фокуса компонентом;
- `onchange` — код JavaScript, выполняемый при изменении значения компонента;
- `onclick` — код JavaScript, выполняемый при щелчке мыши;
- `ondblclick` — код JavaScript, выполняемый при двойном щелчке мыши;
- `onfocus` — код JavaScript, выполняемый при получении фокуса компонентом;

- onkeydown — код JavaScript, выполняемый при нажатии клавиши;
- onkeypress — код JavaScript, выполняемый при нажатии и освобождении клавиши;
- onkeyup — код JavaScript, выполняемый при освобождении клавиши;
- onmousedown — код JavaScript, выполняемый при нажатии мыши;
- onmousemove — код JavaScript, выполняемый при движении мыши внутри;
- onmouseout — код JavaScript, выполняемый при выходе мыши из компонента;
- onmouseover — код JavaScript, выполняемый при движении мыши вне компонента;
- onmouseup — код JavaScript, выполняемый при освобождении мыши;
- onselect — код JavaScript, выполняемый при выделении текста элемента;
- readonly — запрет изменения значения компонента пользователем (true/false);
- rendered — EL-выражение, определяющее, отображается данный компонент или нет (true/false);
- required — если true, тогда поле формы обязательно должно иметь значение перед тем, как содержимое формы будет отправлено серверу;
- requiredMessage — текст сообщения об ошибке при нарушении required="true";
- size — ширина поля формы в символах;
- style — CSS-стиль для компонента;
- styleClass — список CSS-селекторов class для стиля компонента;
- tabindex — принимает целочисленные неотрицательные значения от 0 до 32 767, задающие порядок, в котором будет совершаться tab-обход;
- title — всплывающая подсказка;
- validator — EL-выражение, определяющее метод экземпляра javax.faces.validator.Validator, который осуществляет проверку значения компонента;
- validatorMessage — текст сообщения об ошибке при проверке значения компонента;
- value — текущее значение компонента;
- valueChangeListener — EL-выражение, определяющее метод экземпляра javax.faces.event.ValueChangeListener — слушателя изменения значения компонента.

❒ Тег **<h:inputTextarea>** представляет HTML-тег <textarea> — поле формы для ввода нескольких строк текста.

Атрибуты тега:

- accesskey — клавиша доступа к элементу;
- binding — EL-выражение, связывающее тег со свойством компонента JavaBeans;

- `cols` — ширина поля в символах;
- `converter` — указывает экземпляр `javax.faces.convert.Converter`, осуществляющий конвертацию данных поля формы;
- `converterMessage` — текст сообщения об ошибке при конвертации значения компонента;
- `dir` — направление текста: `"LTR"` (слева направо) или `"RTL"` (справа налево);
- `disabled` — деактивация элемента (`true`/`false`);
- `id` — идентификатор компонента;
- `immediate` — пропуск фазы проверки JSF-цикла (`true`/`false`);
- `label` — локализованное описание компонента;
- `lang` — язык текста компонента;
- `onblur` — код JavaScript, выполняемый при потере фокуса компонентом;
- `onchange` — код JavaScript, выполняемый при изменении значения компонента;
- `onclick` — код JavaScript, выполняемый при щелчке мыши;
- `ondblclick` — код JavaScript, выполняемый при двойном щелчке мыши;
- `onfocus` — код JavaScript, выполняемый при получении фокуса компонентом;
- `onkeydown` — код JavaScript, выполняемый при нажатии клавиши;
- `onkeypress` — код JavaScript, выполняемый при нажатии и освобождении клавиши;
- `onkeyup` — код JavaScript, выполняемый при освобождении клавиши;
- `onmousedown` — код JavaScript, выполняемый при нажатии мыши;
- `onmousemove` — код JavaScript, выполняемый при движении мыши внутри;
- `onmouseout` — код JavaScript, выполняемый при выходе мыши из компонента;
- `onmouseover` — код JavaScript, выполняемый при движении мыши вне компонента;
- `onmouseup` — код JavaScript, выполняемый при освобождении мыши;
- `onselect` — код JavaScript, выполняемый при выделении текста элемента;
- `readonly` — запрет изменения значения компонента пользователем (`true`/`false`);
- `rendered` — EL-выражение, определяющее, отображается данный компонент или нет (`true`/`false`);
- `required` — если `true`, тогда поле формы обязательно должно иметь значение перед тем, как содержимое формы будет отправлено серверу;
- `requiredMessage` — текст сообщения об ошибке при нарушении условия `required="true"`;

- rows — количество строк;
- style — CSS-стиль для компонента;
- styleClass — список CSS-селекторов class для стиля компонента;
- tabindex — принимает целочисленные неотрицательные значения от 0 до 32 767, задающие порядок, в котором будет совершаться tab-обход;
- title — всплывающая подсказка;
- validator — EL-выражение, определяющее метод экземпляра javax.faces.validator.Validator, который осуществляет проверку значения компонента;
- validatorMessage — текст сообщения об ошибке при проверке значения компонента;
- value — текущее значение компонента;
- valueChangeListener — EL-выражение, определяющее метод экземпляра javax.faces.event.ValueChangeListener — слушателя изменения значения компонента.

❒ Тег **<h:message>** визуализирует для клиента сообщения об ошибках, связанные с определенным компонентом. Сообщения имеют различный приоритет FacesMessage.Severity: SEVERITY_INFO (1), SEVERITY_WARN (2), SEVERITY_ERROR (3) и SEVERITY_FATAL (4). Атрибут label связанного с сообщением компонента также включается в сообщение.

Атрибуты тега:

- binding — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- dir — направление текста: "LTR" (слева направо) или "RTL" (справа налево);
- errorClass — CSS-селектор class для стиля сообщения уровня ERROR;
- errorStyle — CSS-стиль для сообщения уровня ERROR;
- fatalClass — CSS-селектор class для стиля сообщения уровня FATAL;
- fatalStyle — CSS-стиль для сообщения уровня FATAL;
- for — идентификатор компонента, для которого отображается сообщение;
- id — идентификатор сообщения;
- infoClass — CSS-селектор class для стиля сообщения уровня INFO;
- infoStyle — CSS-стиль для сообщения уровня INFO;
- lang — язык текста сообщения;
- rendered — EL-выражение, определяющее, отображается данное сообщение или нет (true/false);
- showDetail — если true (по умолчанию), тогда отображаются подробности ошибки;

- `showSummary` — если `false` (по умолчанию), тогда заключение об ошибке не выводится;
- `style` — общий CSS-стиль сообщения;
- `styleClass` — список CSS-селекторов `class` для общего стиля сообщения;
- `title` — всплывающая подсказка;
- `tooltip` — если `true`, тогда подробности сообщения отображаются во всплывающей подсказке;
- `warnClass` — CSS-селектор `class` для стиля сообщения уровня `WARN`;
- `warnStyle` — CSS-стиль для сообщения уровня `WARN`.

❒ Тег **`<h:messages>`** отображает все сообщения, возникающие в течение JSF-жизненного цикла.

Атрибуты тега:

- `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- `dir` — направление текста: `"LTR"` (слева направо) или `"RTL"` (справа налево);
- `errorClass` — CSS-селектор `class` для стиля сообщения уровня `ERROR`;
- `errorStyle` — CSS-стиль для сообщения уровня `ERROR`;
- `fatalClass` — CSS-селектор `class` для стиля сообщения уровня `FATAL`;
- `fatalStyle` — CSS-стиль для сообщения уровня `FATAL`;
- `globalOnly` — если `true`, тогда отображаются только сообщения, не связанные с какими-либо компонентами. По умолчанию `false`;
- `id` — идентификатор тега;
- `infoClass` — CSS-селектор `class` для стиля сообщения уровня `INFO`;
- `infoStyle` — CSS-стиль для сообщения уровня `INFO`;
- `lang` — язык текста сообщения;
- `layout` — указывает, как отображаются сообщения: в виде таблицы `"table"` или списка `"list"`;
- `rendered` — EL-выражение, определяющее, отображаются сообщения или нет (`true`/`false`);
- `showDetail` — если `true` (по умолчанию), тогда отображаются подробности ошибки;
- `showSummary` — если `false` (по умолчанию), тогда заключение об ошибке не выводится;
- `style` — общий CSS-стиль сообщений;
- `styleClass` — список CSS-селекторов `class` для общего стиля сообщений;

- title — всплывающая подсказка;
- tooltip — если true, тогда подробности сообщения отображаются во всплывающей подсказке;
- warnClass — CSS-селектор class для стиля сообщения уровня WARN;
- warnStyle — CSS-стиль для сообщения уровня WARN.

❒ Тег **<h:outputFormat>** позволяет отображать текст, составленный из значений параметров, определенных тегами <f:param value=""/> в теле тега <h:outputFormat>.

Атрибуты тега:

- binding — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- converter — указывает экземпляр javax.faces.convert.Converter, осуществляющий конвертацию данных компонента;
- dir — направление текста: "LTR" (слева направо) или "RTL" (справа налево);
- escape — если true (по умолчанию), тогда символы <, >, &, ', " интерпретируются как символы, а не как элементы синтаксиса HTML или XML;
- id — идентификатор компонента;
- lang — язык текста;
- rendered — EL-выражение, определяющее, отображается компонент или нет (true/false);
- style — CSS-стиль для компонента;
- styleClass — список CSS-селекторов class для стиля компонента;
- title — всплывающая подсказка;
- value — текущее значение компонента.

❒ Тег **<h:outputLabel>** представляет HTML-тег <label>, который связывает текстовую метку с элементом формы.

Атрибуты тега:

- accesskey — клавиша доступа к элементу;
- binding — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- converter — указывает экземпляр javax.faces.convert.Converter, осуществляющий конвертацию данных компонента;
- dir — направление текста: "LTR" (слева направо) или "RTL" (справа налево);
- escape — если true (по умолчанию), тогда символы <, >, &, ', " интерпретируются как символы, а не как элементы синтаксиса HTML или XML;
- for — идентификатор компонента, для которого создается данная метка;
- id — идентификатор компонента;

- `lang` — язык текста;
- `onblur` — код JavaScript, выполняемый при потере фокуса компонентом;
- `onchange` — код JavaScript, выполняемый при изменении значения компонента;
- `onclick` — код JavaScript, выполняемый при щелчке мыши;
- `ondblclick` — код JavaScript, выполняемый при двойном щелчке мыши;
- `onfocus` — код JavaScript, выполняемый при получении фокуса компонентом;
- `onkeydown` — код JavaScript, выполняемый при нажатии клавиши;
- `onkeypress` — код JavaScript, выполняемый при нажатии и освобождении клавиши;
- `onkeyup` — код JavaScript, выполняемый при освобождении клавиши;
- `onmousedown` — код JavaScript, выполняемый при нажатии мыши;
- `onmousemove` — код JavaScript, выполняемый при движении мыши внутри;
- `onmouseout` — код JavaScript, выполняемый при выходе мыши из компонента;
- `onmouseover` — код JavaScript, выполняемый при движении мыши вне компонента;
- `onmouseup` — код JavaScript, выполняемый при освобождении мыши;
- `rendered` — EL-выражение, определяющее, отображается компонент или нет (`true`/`false`);
- `style` — CSS-стиль для компонента;
- `styleClass` — список CSS-селекторов `class` для стиля компонента;
- `tabindex` — принимает целочисленные неотрицательные значения от 0 до 32 767, задающие порядок, в котором будет совершаться tab-обход;
- `title` — всплывающая подсказка;
- `value` — текущее значение компонента.

❒ Тег **`<h:outputLink>`** представляет HTML-тег `<a>`, создающий ссылку.

Атрибуты тега:

- `accesskey` — клавиша доступа к элементу;
- `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- `charset` — кодировка ресурса гиперссылки;
- `converter` — указывает экземпляр `javax.faces.convert.Converter`, осуществляющий конвертацию данных компонента;
- `coords` и `shape` — позиция и форма участка экрана при использовании гиперссылки в HTML-теге `<MAP>`. Возможные значения атрибута `shape`: `default`, `rect`, `circle` и `poly`;

- dir — направление текста: "LTR" (слева направо) или "RTL" (справа налево);
- disabled — деактивация кнопки (true/false);
- hreflang — язык ресурса гиперссылки;
- id — идентификатор компонента;
- id — идентификатор компонента;
- lang — язык текста компонента;
- onblur — код JavaScript, выполняемый при потере фокуса компонентом;
- onclick — код JavaScript, выполняемый при щелчке мыши;
- ondblclick — код JavaScript, выполняемый при двойном щелчке мыши;
- onfocus — код JavaScript, выполняемый при наведении фокуса на компонент;
- onkeydown — код JavaScript, выполняемый при нажатии клавиши;
- onkeypress — код JavaScript, выполняемый при нажатии и освобождении клавиши;
- onkeyup — код JavaScript, выполняемый при освобождении клавиши;
- onmousedown — код JavaScript, выполняемый при нажатии мыши;
- onmousemove — код JavaScript, выполняемый при движении мыши внутри;
- onmouseout — код JavaScript, выполняемый при выходе мыши из компонента;
- onmouseover — код JavaScript, выполняемый при движении мыши вне компонента;
- onmouseup — код JavaScript, выполняемый при освобождении мыши;
- rel — соотношение между текущим документом и связанным (распознаваемые типы ссылок: "index", "next", "prev" и т. д.);
- rendered — EL-выражение, определяющее, отображается компонент или нет (true/false);
- rev — обратное соотношение относительно rel между связанным документом и текущим;
- style — CSS-стиль для компонента;
- styleClass — список CSS-селекторов class для стиля компонента;
- tabindex — принимает целочисленные неотрицательные значения от 0 до 32 767, задающие порядок, в котором будет совершаться tab-обход;
- target — имя фрейма, в котором будет отображаться связанный документ;
- title — всплывающая подсказка;
- type — тип содержимого связанного документа (например, type="text/html");
- value — текущее значение компонента, представляющее значение атрибута href HTML-тега <a>.

- Тег **<h:outputText>** отображает текст.

  Атрибуты тега:

  - `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
  - `converter` — указывает экземпляр `javax.faces.convert.Converter`, осуществляющий конвертацию данных компонента;
  - `dir` — направление текста: `"LTR"` (слева направо) или `"RTL"` (справа налево);
  - `escape` — если `true` (по умолчанию), тогда символы `<`, `>`, `&`, `'`, `"` интерпретируются как символы, а не как элементы синтаксиса HTML или XML;
  - `id` — идентификатор компонента;
  - `lang` — язык текста;
  - `rendered` — EL-выражение, определяющее, отображается компонент или нет (`true`/`false`);
  - `style` — CSS-стиль для компонента;
  - `styleClass` — список CSS-селекторов `class` для стиля компонента;
  - `title` — всплывающая подсказка;
  - `value` — текущее значение компонента.

- Тег **<h:panelGrid>** представляет HTML-тег `<table>`, создающий таблицу. Тег `<h:panelGrid>` используется как менеджер компоновки для JSF-компонентов, в отличие от тега `<h:dataTable>`, который представляет таблицу данных.

  Атрибуты тега:

  - `bgcolor` — цвет фона;
  - `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
  - `border` — ширина в пикселах рамки таблицы;
  - `captionClass` — определяет атрибут `class` HTML-тега `<caption>`, содержащий список CSS-селекторов `class` для стиля заголовка таблицы;
  - `captionStyle` — определяет атрибут `style` HTML-тега `<caption>`, указывающий стиль для заголовка таблицы;
  - `cellpadding` — отступ от границы ячейки таблицы до ее содержимого;
  - `cellspacing` — расстояние между ячейками;
  - `columnClasses` — список CSS-селекторов `class` для стиля столбцов таблицы;
  - `columns` — количество столбцов таблицы;
  - `dir` — направление текста: `"LTR"` (слева направо) или `"RTL"` (справа налево);
  - `footerClass` — список CSS-селекторов `class` для стиля подстрочника таблицы;

- `frame` — указывает отображение границ вокруг таблицы: `none` (нет границ), `above` (верхняя граница), `below` (нижняя граница), `hsides` (верхняя и нижняя границы), `vsides` (правая и левая границы), `lhs` (левая граница), `rhs` (правая граница), `box` (все границы), `border` (все границы);
- `headerClass` — список CSS-селекторов `class` для стиля заголовка таблицы;
- `id` — идентификатор таблицы;
- `lang` — язык содержимого таблицы (например, `en-US`);
- `onclick` — код JavaScript, выполняемый при щелчке мыши на таблице;
- `ondblclick` — код JavaScript, выполняемый при двойном щелчке мыши на таблице;
- `onkeydown` — код JavaScript, выполняемый при нажатии клавиши на таблице;
- `onkeypress` — код JavaScript, выполняемый при нажатии и освобождении клавиши на таблице;
- `onkeyup` — код JavaScript, выполняемый при освобождении клавиши на таблице;
- `onmousedown` — код JavaScript, выполняемый при нажатии мыши на таблице;
- `onmousemove` — код JavaScript, выполняемый при движении мыши внутри таблицы;
- `onmouseout` — код JavaScript, выполняемый при выходе мыши из таблицы;
- `onmouseover` — код JavaScript, выполняемый при движении мыши вне таблицы;
- `onmouseup` — код JavaScript, выполняемый при освобождении мыши на таблице;
- `rendered` — EL-выражение, определяющее, отображается данный компонент или нет (`true`/`false`);
- `rowClasses` — список CSS-селекторов `class` для стиля строк таблицы;
- `rules` — отображение внутренних рамок в таблице: `none` (нет границ), `groups` (границы между группами строк), `rows` (границы между строками), `cols` (границы между столбцами), `all` (границы между строками и столбцами);
- `style` — CSS-стиль для таблицы;
- `styleClass` — список CSS-селекторов `class` для стиля таблицы;
- `summary` — описание таблицы для невизуального отображения;
- `title` — всплывающая подсказка при наведении курсора на таблицу;
- `width` — ширина таблицы.

❐ Тег **`<h:panelGroup>`** является контейнером для JSF-компонентов, который визуализирует их как группу. В частности, использование данного тега в теле тега `<h:panelGrid>` позволяет визуализировать несколько компонентов в одной строке.

Атрибуты тега:

- `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- `id` — идентификатор компонента;
- `layout` — тип компоновки группы. Если значение атрибута `"block"`, тогда используется HTML-тег `<div>`, иначе — HTML-тег `<span>`;
- `rendered` — EL-выражение, определяющее, отображается данный компонент или нет (`true`/`false`);
- `style` — CSS-стиль для компонента;
- `styleClass` — список CSS-селекторов `class` для стиля компонента.

❒ Тег **`<h:selectBooleanCheckbox>`** представляет HTML-тег `<input type="checkbox">` — переключатель.

Атрибуты тега:

- `accesskey` — клавиша доступа к элементу;
- `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- `converter` — указывает экземпляр `javax.faces.convert.Converter`, осуществляющий конвертацию данных компонента;
- `converterMessage` — текст сообщения об ошибке при конвертации значения компонента;
- `dir` — направление текста: `"LTR"` (слева направо) или `"RTL"` (справа налево);
- `disabled` — деактивация элемента (`true`/`false`);
- `id` — идентификатор компонента;
- `immediate` — пропуск фазы проверки JSF-цикла (`true`/`false`);
- `label` — локализованное описание компонента;
- `lang` — язык текста компонента;
- `onblur` — код JavaScript, выполняемый при потере фокуса компонентом;
- `onchange` — код JavaScript, выполняемый при изменении значения компонента;
- `onclick` — код JavaScript, выполняемый при щелчке мыши;
- `ondblclick` — код JavaScript, выполняемый при двойном щелчке мыши;
- `onfocus` — код JavaScript, выполняемый при получении фокуса компонентом;
- `onkeydown` — код JavaScript, выполняемый при нажатии клавиши;
- `onkeypress` — код JavaScript, выполняемый при нажатии и освобождении клавиши;
- `onkeyup` — код JavaScript, выполняемый при освобождении клавиши;

- `onmousedown` — код JavaScript, выполняемый при нажатии мыши;
- `onmousemove` — код JavaScript, выполняемый при движении мыши внутри;
- `onmouseout` — код JavaScript, выполняемый при выходе мыши из компонента;
- `onmouseover` — код JavaScript, выполняемый при движении мыши вне компонента;
- `onmouseup` — код JavaScript, выполняемый при освобождении мыши;
- `onselect` — код JavaScript, выполняемый при выделении текста элемента;
- `readonly` — запрет изменения значения компонента пользователем (`true`/`false`);
- `rendered` — EL-выражение, определяющее, отображается данный компонент или нет (`true`/`false`);
- `required` — если `true`, тогда компонент обязательно должен иметь значение перед тем, как содержимое компонента будет отправлено серверу;
- `requiredMessage` — текст сообщения об ошибке при нарушении `required="true"`;
- `style` — CSS-стиль для компонента;
- `styleClass` — список CSS-селекторов `class` для стиля компонента;
- `tabindex` — принимает целочисленные неотрицательные значения от 0 до 32 767, задающие порядок, в котором будет совершаться tab-обход;
- `title` — всплывающая подсказка;
- `validator` — EL-выражение, определяющее метод экземпляра `javax.faces.validator.Validator`, который осуществляет проверку значения компонента;
- `validatorMessage` — текст сообщения об ошибке при проверке значения компонента;
- `value` — текущее значение компонента;
- `valueChangeListener` — EL-выражение, определяющее метод экземпляра `javax.faces.event.ValueChangeListener` — слушателя изменения значения компонента.

❒ Тег **`<h:selectManyCheckbox>`** визуализирует группу переключателей в виде таблицы.

Атрибуты тега:

- `accesskey` — клавиша доступа к компоненту;
- `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- `border` — ширина в пикселах рамки таблицы;
- `converter` — указывает экземпляр `javax.faces.convert.Converter`, осуществляющий конвертацию данных компонента;

- `converterMessage` — текст сообщения об ошибке при конвертации значения компонента;
- `dir` — направление текста: `"LTR"` (слева направо) или `"RTL"` (справа налево);
- `disabled` — деактивация элемента (`true`/`false`);
- `disabledClass` — CSS-селектор `class` для стиля переключателей в состоянии "выключен";
- `enabledClass` — CSS-селектор `class` для стиля переключателей в состоянии "включен";
- `id` — идентификатор компонента;
- `immediate` — пропуск фазы проверки JSF-цикла (`true`/`false`);
- `label` — локализованное описание компонента;
- `lang` — язык текста компонента;
- `layout` — тип компоновки переключателей: `"pageDirection"` (столбцом), `"lineDirection"` (строкой);
- `onblur` — код JavaScript, выполняемый при потере фокуса компонентом;
- `onchange` — код JavaScript, выполняемый при изменении значения компонента;
- `onclick` — код JavaScript, выполняемый при щелчке мыши;
- `ondblclick` — код JavaScript, выполняемый при двойном щелчке мыши;
- `onfocus` — код JavaScript, выполняемый при получении фокуса компонентом;
- `onkeydown` — код JavaScript, выполняемый при нажатии клавиши;
- `onkeypress` — код JavaScript, выполняемый при нажатии и освобождении клавиши;
- `onkeyup` — код JavaScript, выполняемый при освобождении клавиши;
- `onmousedown` — код JavaScript, выполняемый при нажатии мыши;
- `onmousemove` — код JavaScript, выполняемый при движении мыши внутри;
- `onmouseout` — код JavaScript, выполняемый при выходе мыши из компонента;
- `onmouseover` — код JavaScript, выполняемый при движении мыши вне компонента;
- `onmouseup` — код JavaScript, выполняемый при освобождении мыши;
- `onselect` — код JavaScript, выполняемый при выделении текста элемента;
- `readonly` — запрет изменения значения компонента пользователем (`true`/`false`);
- `rendered` — EL-выражение, определяющее, отображается данный компонент или нет (`true`/`false`);
- `required` — если `true`, тогда компонент обязательно должен иметь значение перед тем, как содержимое компонента будет отправлено серверу;

- requiredMessage — текст сообщения об ошибке при нарушении required="true";
- style — CSS-стиль для компонента;
- styleClass — список CSS-селекторов class для стиля компонента;
- tabindex — принимает целочисленные неотрицательные значения от 0 до 32 767, задающие порядок, в котором будет совершаться tab-обход;
- title — всплывающая подсказка;
- validator — EL-выражение, определяющее метод экземпляра javax.faces.validator.Validator, который осуществляет проверку значения компонента;
- validatorMessage — текст сообщения об ошибке при проверке значения компонента;
- value — текущее значение компонента;
- valueChangeListener — EL-выражение, определяющее метод экземпляра javax.faces.event.ValueChangeListener — слушателя изменения значения компонента.

❒ Тег **<h:selectManyListbox>** представляет HTML-тег <select> — список множественного выбора опций (атрибут multiple).

Атрибуты тега:

- accesskey — клавиша доступа к компоненту;
- binding — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- converter — указывает экземпляр javax.faces.convert.Converter, осуществляющий конвертацию данных компонента;
- converterMessage — текст сообщения об ошибке при конвертации значения компонента;
- dir — направление текста: "LTR" (слева направо) или "RTL" (справа налево);
- disabled — деактивация компонента (true/false);
- disabledClass — CSS-селектор class для стиля опции в состоянии "выключена";
- enabledClass — CSS-селектор class для стиля опции в состоянии "включена";
- id — идентификатор компонента;
- immediate — пропуск фазы проверки JSF-цикла (true/false);
- label — локализованное описание компонента;
- lang — язык текста компонента;
- onblur — код JavaScript, выполняемый при потере фокуса компонентом;
- onchange — код JavaScript, выполняемый при изменении значения компонента;

- `onclick` — код JavaScript, выполняемый при щелчке мыши;
- `ondblclick` — код JavaScript, выполняемый при двойном щелчке мыши;
- `onfocus` — код JavaScript, выполняемый при получении фокуса компонентом;
- `onkeydown` — код JavaScript, выполняемый при нажатии клавиши;
- `onkeypress` — код JavaScript, выполняемый при нажатии и освобождении клавиши;
- `onkeyup` — код JavaScript, выполняемый при освобождении клавиши;
- `onmousedown` — код JavaScript, выполняемый при нажатии мыши;
- `onmousemove` — код JavaScript, выполняемый при движении мыши внутри;
- `onmouseout` — код JavaScript, выполняемый при выходе мыши из компонента;
- `onmouseover` — код JavaScript, выполняемый при движении мыши вне компонента;
- `onmouseup` — код JavaScript, выполняемый при освобождении мыши;
- `onselect` — код JavaScript, выполняемый при выделении текста элемента;
- `readonly` — запрет изменения значения компонента пользователем (`true`/`false`);
- `rendered` — EL-выражение, определяющее, отображается данный компонент или нет (`true`/`false`);
- `required` — если `true`, тогда компонент обязательно должен иметь значение перед тем, как содержимое компонента будет отправлено серверу;
- `requiredMessage` — текст сообщения об ошибке при нарушении `required="true"`;
- `size` — количество отображаемых опций; если атрибут не указан, то все опции отображаются;
- `style` — CSS-стиль для компонента;
- `styleClass` — список CSS-селекторов `class` для стиля компонента;
- `tabindex` — принимает целочисленные неотрицательные значения от 0 до 32 767, задающие порядок, в котором будет совершаться tab-обход;
- `title` — всплывающая подсказка;
- `validator` — EL-выражение, определяющее метод экземпляра `javax.faces.validator.Validator`, который осуществляет проверку значения компонента;
- `validatorMessage` — текст сообщения об ошибке при проверке значения компонента;
- `value` — текущее значение компонента;
- `valueChangeListener` — EL-выражение, определяющее метод экземпляра `javax.faces.event.ValueChangeListener` — слушателя изменения значения компонента.

- Тег **<h:selectManyMenu>** представляет HTML-тег <select> — список множественного выбора опций (атрибут multiple тега <select>) с одной отображаемой опцией (size=1).

  Атрибуты тега:

  - accesskey — клавиша доступа к компоненту;
  - binding — EL-выражение, связывающее тег со свойством компонента JavaBeans;
  - converter — указывает экземпляр javax.faces.convert.Converter, осуществляющий конвертацию данных компонента;
  - converterMessage — текст сообщения об ошибке при конвертации значения компонента;
  - dir — направление текста: "LTR" (слева направо) или "RTL" (справа налево);
  - disabled — деактивация компонента (true/false);
  - disabledClass — CSS-селектор class для стиля опции в состоянии "выключена";
  - enabledClass — CSS-селектор class для стиля опции в состоянии "включена";
  - id — идентификатор компонента;
  - immediate — пропуск фазы проверки JSF-цикла (true/false);
  - label — локализованное описание компонента;
  - lang — язык текста компонента;
  - onblur — код JavaScript, выполняемый при потере фокуса компонентом;
  - onchange — код JavaScript, выполняемый при изменении значения компонента;
  - onclick — код JavaScript, выполняемый при щелчке мыши;
  - ondblclick — код JavaScript, выполняемый при двойном щелчке мыши;
  - onfocus — код JavaScript, выполняемый при получении фокуса компонентом;
  - onkeydown — код JavaScript, выполняемый при нажатии клавиши;
  - onkeypress — код JavaScript, выполняемый при нажатии и освобождении клавиши;
  - onkeyup — код JavaScript, выполняемый при освобождении клавиши;
  - onmousedown — код JavaScript, выполняемый при нажатии мыши;
  - onmousemove — код JavaScript, выполняемый при движении мыши внутри;
  - onmouseout — код JavaScript, выполняемый при выходе мыши из компонента;
  - onmouseover — код JavaScript, выполняемый при движении мыши вне компонента;
  - onmouseup — код JavaScript, выполняемый при освобождении мыши;

- `onselect` — код JavaScript, выполняемый при выделении текста элемента;
- `readonly` — запрет изменения значения компонента пользователем (`true`/`false`);
- `rendered` — EL-выражение, определяющее, отображается данный компонент или нет (`true`/`false`);
- `required` — если `true`, тогда компонент обязательно должен иметь значение перед тем, как содержимое компонента будет отправлено серверу;
- `requiredMessage` — текст сообщения об ошибке при нарушении `required="true"`;
- `style` — CSS-стиль для компонента;
- `styleClass` — список CSS-селекторов `class` для стиля компонента;
- `tabindex` — принимает целочисленные неотрицательные значения от 0 до 32 767, задающие порядок, в котором будет совершаться tab-обход;
- `title` — всплывающая подсказка;
- `validator` — EL-выражение, определяющее метод экземпляра `javax.faces.validator.Validator`, который осуществляет проверку значения компонента;
- `validatorMessage` — текст сообщения об ошибке при проверке значения компонента;
- `value` — текущее значение компонента;
- `valueChangeListener` — EL-выражение, определяющее метод экземпляра `javax.faces.event.ValueChangeListener` — слушателя изменения значения компонента.

❒ Тег **`<h:selectOneListbox>`** представляет HTML-тег `<select>` — список выбора опций (без атрибута `multiple` тега `<select>`).

Атрибуты тега:

- `accesskey` — клавиша доступа к компоненту;
- `binding` — EL-выражение, связывающее тег со свойством компонента JavaBean;
- `converter` — указывает экземпляр `javax.faces.convert.Converter`, осуществляющий конвертацию данных компонента;
- `converterMessage` — текст сообщения об ошибке при конвертации значения компонента;
- `dir` — направление текста: `"LTR"` (слева направо) или `"RTL"` (справа налево);
- `disabled` — деактивация компонента (`true`/`false`);
- `disabledClass` — CSS-селектор `class` для стиля опции в состоянии "выключена";
- `enabledClass` — CSS-селектор `class` для стиля опции в состоянии "включена";

- id — идентификатор компонента;
- immediate — пропуск фазы проверки JSF-цикла (true/false);
- label — локализованное описание компонента;
- lang — язык текста компонента;
- onblur — код JavaScript, выполняемый при потере фокуса компонентом;
- onchange — код JavaScript, выполняемый при изменении значения компонента;
- onclick — код JavaScript, выполняемый при щелчке мыши;
- ondblclick — код JavaScript, выполняемый при двойном щелчке мыши;
- onfocus — код JavaScript, выполняемый при получении фокуса компонентом;
- onkeydown — код JavaScript, выполняемый при нажатии клавиши;
- onkeypress — код JavaScript, выполняемый при нажатии и освобождении клавиши;
- onkeyup — код JavaScript, выполняемый при освобождении клавиши;
- onmousedown — код JavaScript, выполняемый при нажатии мыши;
- onmousemove — код JavaScript, выполняемый при движении мыши внутри;
- onmouseout — код JavaScript, выполняемый при выходе мыши из компонента;
- onmouseover — код JavaScript, выполняемый при движении мыши вне компонента;
- onmouseup — код JavaScript, выполняемый при освобождении мыши;
- onselect — код JavaScript, выполняемый при выделении текста элемента;
- readonly — запрет изменения значения компонента пользователем (true/false);
- rendered — EL-выражение, определяющее, отображается данный компонент или нет (true/false);
- required — если true, тогда компонент обязательно должен иметь значение перед тем, как содержимое компонента будет отправлено серверу;
- requiredMessage — текст сообщения об ошибке при нарушении required="true";
- size — количество отображаемых опций; если атрибут не указан, то все опции отображаются;
- style — CSS-стиль для компонента;
- styleClass — список CSS-селекторов class для стиля компонента;
- tabindex — принимает целочисленные неотрицательные значения от 0 до 32 767, задающие порядок, в котором будет совершаться tab-обход;
- title — всплывающая подсказка;

- `validator` — EL-выражение, определяющее метод экземпляра `javax.faces.validator.Validator`, который осуществляет проверку значения компонента;
- `validatorMessage` — текст сообщения об ошибке при проверке значения компонента;
- `value` — текущее значение компонента;
- `valueChangeListener` — EL-выражение, определяющее метод экземпляра `javax.faces.event.ValueChangeListener` — слушателя изменения значения компонента.

❒ Тег **`<h:selectOneMenu>`** представляет HTML-тег `<select>` — список выбора опций (без атрибута `multiple` тега `<select>`) с одной отображаемой опцией (`size=1`).

Атрибуты тега:

- `accesskey` — клавиша доступа к компоненту;
- `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- `converter` — указывает экземпляр `javax.faces.convert.Converter`, осуществляющий конвертацию данных компонента;
- `converterMessage` — текст сообщения об ошибке при конвертации значения компонента;
- `dir` — направление текста: `"LTR"` (слева направо) или `"RTL"` (справа налево);
- `disabled` — деактивация компонента (`true`/`false`);
- `disabledClass` — CSS-селектор `class` для стиля опции в состоянии "выключена";
- `enabledClass` — CSS-селектор `class` для стиля опции в состоянии "включена";
- `id` — идентификатор компонента;
- `immediate` — пропуск фазы проверки JSF-цикла (`true`/`false`);
- `label` — локализованное описание компонента;
- `lang` — язык текста компонента;
- `onblur` — код JavaScript, выполняемый при потере фокуса компонентом;
- `onchange` — код JavaScript, выполняемый при изменении значения компонента;
- `onclick` — код JavaScript, выполняемый при щелчке мыши;
- `ondblclick` — код JavaScript, выполняемый при двойном щелчке мыши;
- `onfocus` — код JavaScript, выполняемый при получении фокуса компонентом;
- `onkeydown` — код JavaScript, выполняемый при нажатии клавиши;
- `onkeypress` — код JavaScript, выполняемый при нажатии и освобождении клавиши;
- `onkeyup` — код JavaScript, выполняемый при освобождении клавиши;

- `onmousedown` — код JavaScript, выполняемый при нажатии мыши;
- `onmousemove` — код JavaScript, выполняемый при движении мыши внутри;
- `onmouseout` — код JavaScript, выполняемый при выходе мыши из компонента;
- `onmouseover` — код JavaScript, выполняемый при движении мыши вне компонента;
- `onmouseup` — код JavaScript, выполняемый при освобождении мыши;
- `onselect` — код JavaScript, выполняемый при выделении текста элемента;
- `readonly` — запрет изменения значения компонента пользователем (`true`/`false`);
- `rendered` — EL-выражение, определяющее, отображается данный компонент или нет (`true`/`false`);
- `required` — если `true`, тогда компонент обязательно должен иметь значение перед тем, как содержимое компонента будет отправлено серверу;
- `requiredMessage` — текст сообщения об ошибке при нарушении `required="true"`;
- `style` — CSS-стиль для компонента;
- `styleClass` — список CSS-селекторов `class` для стиля компонента;
- `tabindex` — принимает целочисленные неотрицательные значения от 0 до 32 767, задающие порядок, в котором будет совершаться tab-обход;
- `title` — всплывающая подсказка;
- `validator` — EL-выражение, определяющее метод экземпляра `javax.faces.validator.Validator`, который осуществляет проверку значения компонента;
- `validatorMessage` — текст сообщения об ошибке при проверке значения компонента;
- `value` — текущее значение компонента;
- `valueChangeListener` — EL-выражение, определяющее метод экземпляра `javax.faces.event.ValueChangeListener` — слушателя изменения значения компонента.

❒ Тег **`<h:selectOneRadio>`** представляет группу HTML-тегов `<input type="radio">` в виде таблицы.

Атрибуты тега:

- `accesskey` — клавиша доступа к компоненту;
- `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- `border` — ширина в пикселах рамки таблицы;
- `converter` — указывает экземпляр `javax.faces.convert.Converter`, осуществляющий конвертацию данных компонента;

- `converterMessage` — текст сообщения об ошибке при конвертации значения компонента;
- `dir` — направление текста: `"LTR"` (слева направо) или `"RTL"` (справа налево);
- `disabled` — деактивация элемента (`true`/`false`);
- `disabledClass` — CSS-селектор `class` для стиля переключателей в состоянии "выключен";
- `enabledClass` — CSS-селектор `class` для стиля переключателей в состоянии "включен";
- `id` — идентификатор компонента;
- `immediate` — пропуск фазы проверки JSF-цикла (`true`/`false`);
- `label` — локализованное описание компонента;
- `lang` — язык текста компонента;
- `layout` — тип компоновки переключателей: `"pageDirection"` (столбцом), `"lineDirection"` (строкой);
- `onblur` — код JavaScript, выполняемый при потере фокуса компонентом;
- `onchange` — код JavaScript, выполняемый при изменении значения компонента;
- `onclick` — код JavaScript, выполняемый при щелчке мыши;
- `ondblclick` — код JavaScript, выполняемый при двойном щелчке мыши;
- `onfocus` — код JavaScript, выполняемый при получении фокуса компонентом;
- `onkeydown` — код JavaScript, выполняемый при нажатии клавиши;
- `onkeypress` — код JavaScript, выполняемый при нажатии и освобождении клавиши;
- `onkeyup` — код JavaScript, выполняемый при освобождении клавиши;
- `onmousedown` — код JavaScript, выполняемый при нажатии мыши;
- `onmousemove` — код JavaScript, выполняемый при движении мыши внутри;
- `onmouseout` — код JavaScript, выполняемый при выходе мыши из компонента;
- `onmouseover` — код JavaScript, выполняемый при движении мыши вне компонента;
- `onmouseup` — код JavaScript, выполняемый при освобождении мыши;
- `onselect` — код JavaScript, выполняемый при выделении текста элемента;
- `readonly` — запрет изменения значения компонента пользователем (`true`/`false`);
- `rendered` — EL-выражение, определяющее, отображается данный компонент или нет (`true`/`false`);
- `required` — если `true`, тогда компонент обязательно должен иметь значение перед тем, как содержимое компонента будет отправлено серверу;

- requiredMessage — текст сообщения об ошибке при нарушении required="true";
- style — CSS-стиль для компонента;
- styleClass — список CSS-селекторов class для стиля компонента;
- tabindex — принимает целочисленные неотрицательные значения от 0 до 32 767, задающие порядок, в котором будет совершаться tab-обход;
- title — всплывающая подсказка;
- validator — EL-выражение, определяющее метод экземпляра javax.faces.validator.Validator, который осуществляет проверку значения компонента;
- validatorMessage — текст сообщения об ошибке при проверке значения компонента;
- value — текущее значение компонента;
- valueChangeListener — EL-выражение, определяющее метод экземпляра javax.faces.event.ValueChangeListener — слушателя изменения значения компонента.

❐ Теги **<h:button>** и **<h:link>** (JSF 2.0) представляют гиперссылку закладки.

Атрибуты тега:

- outcome — связанный ресурс;
- value — значение компонента.

❐ Тег **<h:body>** (JSF 2.0) представляет HTML-тег <body>.

Атрибуты тега:

- binding — EL-выражение, связывающее тег со свойством компонента JavaBean;
- dir — направление текста: "LTR" (слева направо) или "RTL" (справа налево);
- lang — язык текста компонента;
- onclick — код JavaScript, выполняемый при щелчке мыши;
- ondblclick — код JavaScript, выполняемый при двойном щелчке мыши;
- onkeydown — код JavaScript, выполняемый при нажатии клавиши;
- onkeypress — код JavaScript, выполняемый при нажатии и освобождении клавиши;
- onkeyup — код JavaScript, выполняемый при освобождении клавиши;
- onload — код JavaScript, выполняемый по окончании загрузки браузером окна;
- onmousedown — код JavaScript, выполняемый при нажатии мыши;
- onmousemove — код JavaScript, выполняемый при движении мыши внутри;
- onmouseout — код JavaScript, выполняемый при выходе мыши из компонента;

- `onmouseover` — код JavaScript, выполняемый при движении мыши вне компонента;
- `onmouseup` — код JavaScript, выполняемый при освобождении мыши;
- `onunload` — код JavaScript, выполняемый при удалении браузером документа;
- `style` — CSS-стиль для компонента;
- `styleClass` — список CSS-селекторов `class` для стиля компонента;
- `title` — всплывающая подсказка.

❒ Тег **`<h:head>`** (JSF 2.0) представляет HTML-тег `<head>`.

Атрибуты тега:

- `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- `dir` — направление текста: `"LTR"` (слева направо) или `"RTL"` (справа налево);
- `lang` — язык текста компонента.

❒ Тег **`<h:outputStylesheet>`** (JSF 2.0) представляет ссылку (HTML-тег `<link>`) на внешнюю CSS-таблицу стилей (css-файл).

Атрибуты тега:

- `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- `converter` — указывает экземпляр `javax.faces.convert.Converter`, осуществляющий конвертацию данных компонента;
- `id` — идентификатор компонента;
- `library` — каталог внешней CSS-таблицы стилей;
- `name` — имя внешней CSS-таблицы стилей;
- `rendered` — EL-выражение, определяющее, исполняется данный компонент или нет (`true`/`false`);
- `target` — компонент, с которым связывается данный тег;
- `value` — текущее значение компонента.

❒ Тег **`<h:outputScript>`** (JSF 2.0) представляет ссылку на внешний файл кода JavaScript (HTML-тег `<script>`).

Атрибуты тега:

- `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans;
- `converter` — указывает экземпляр `javax.faces.convert.Converter`, осуществляющий конвертацию данных компонента;
- `id` — идентификатор компонента;

- `library` — каталог JavaScript-файла с расширением js;
- `name` — имя JavaScript-файла;
- `rendered` — EL-выражение, определяющее, исполняется данный компонент или нет (`true`/`false`);
- `target` — указывает компонент, с которым связывается данный тег;
- `value` — текущее значение компонента.

Библиотека тегов JSF Core Tags

Библиотека тегов JSF Core Tags имеет префикс `f` и пространство имен URI **http://java.sun.com/jsf/core**.

Стандартный синтаксис:

```
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core" %>
```

XML-синтаксис:

```
<anyxmlelement xmlns:f="http://java.sun.com/jsf/core" />
```

- Тег **<f:actionListener>** в теле тега компонента регистрирует для него обработчика событий `ActionEvent`. Атрибуты тега — `type` (полное имя Java-класса, реализующего интерфейс `javax.faces.event.ActionListener`) и `binding` (EL-выражение, определяющее экземпляр Java-класса, реализующего интерфейс `javax.faces.event.ActionListener`).
- Тег **<f:attribute>** в теле тега компонента устанавливает для него значение параметра. Атрибуты тега — `name` (имя параметра компонента) и `value` (значение параметра).
- Тег **<f:convertDateTime>** в теле тега компонента устанавливает стиль вывода для временного значения компонента.

  Атрибуты тега:

  - `binding` — EL-выражение, определяющее объект `javax.faces.convert.DateTimeConverter`, который осуществляет конвертацию временного значения компонента;
  - `dateStyle` — стиль для даты согласно `java.text.DateFormat` (`"default"`, `"short"`, `"medium"`, `"long"` и `"full"`);
  - `locale` — локализация (см. `java.util.Locale`);
  - `pattern` — позволяет форматировать согласно шаблонам класса `java.text.SimpleDateFormat`;
  - `timeStyle` — стиль для времени согласно `java.text.DateFormat` (`"default"`, `"short"`, `"medium"`, `"long"` и `"full"`);
  - `timeZone` — временная зона для форматирования (см. `java.util.TimeZone`);
  - `type` — форматируется и отображается время, дата (по умолчанию) или время и дата (`"date"`, `"time"` и `"both"`).

- Тег **<f:convertNumber>** в теле тега компонента используется для отображения данных компонента в виде чисел (по умолчанию), денежных сумм или процентов.

  Атрибуты тега:

  - `binding` — EL-выражение, определяющее объект `javax.faces.convert.NumberConverter`, осуществляющий конвертацию данных компонента;
  - `currencyCode` — в случае значения атрибута `type="currency"` указывает код денежной единицы согласно ISO 4217 Currency Code List (например, EUR);
  - `currencySymbol` — в случае значения атрибута `type="currency"` указывает символ денежной единицы (например, `"$"` для US);
  - `groupingUsed` — если `true` (по умолчанию), тогда знаки перед десятичной запятой группируются определенным образом для установленной локализации, например, в английской версии знаки перед десятичной точкой группируются по три и разделяются запятой;
  - `integerOnly` — если `true`, тогда только целая часть числа конвертируется; по умолчанию `false`;
  - `locale` — локализация;
  - `maxFractionDigit` — максимальное количество знаков после десятичной запятой;
  - `maxIntegerDigits` — максимальное количество знаков перед десятичной запятой;
  - `minFractionDigits` — минимальное количество знаков после десятичной запятой;
  - `minIntegerDigits` — минимальное количество знаков перед десятичной запятой;
  - `pattern` — позволяет форматировать значение согласно шаблонам класса `java.text.DecimalFormat`;
  - `type` — тип отображаемого значения: число (по умолчанию), денежная сумма или проценты (`"number"`, `"currency"` и `"percentage"`).

- Тег **<f:converter>** в теле тега компонента регистрирует экземпляр класса, реализующего интерфейс `javax.faces.convert.Converter` и осуществляющего конвертацию данных, вводимых пользователем, в данные JSF-приложения и наоборот. Атрибуты тега: `converterId` — идентификатор экземпляра `Converter`, соответствует значению элемента `converter-id` JSF-конфигурационного файла; `binding` — EL-выражение, определяющее объект `Converter`.

- Тег **<f:facet>** в теле тега компонента представляет подчиненный компонент, который связан с данным компонентом с помощью атрибута `name`, указывающим роль, которую связанный компонент выполняет. Например, значения `"header"` и `"footer"` атрибута `name` позволяют с помощью компонента `<h:outputText>` (в теле тега `<f:facet name="header"|"footer">`) выводить заголовок или подстрочник для компонента `<h:dataTable>`.

- Тег **<f:loadBundle>** загружает ресурсный файл локализации для текущего JSF-представления. Атрибуты тега — `basename` (базовое имя ресурсного файла локализации) и `var` (имя переменной, хранящей ресурсный файл в виде хэш-таблицы).
- Тег **<f:param>** в теле тега компонента устанавливают, в том числе подставляемые значения параметров компонента ({0} ... {1} и т. д.) в порядке размещения тегов `<f:param>`, если не указан атрибут `name`. Атрибуты тега: `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans; `id` — идентификатор тега; `name` — имя параметра компонента; `value` — значение параметра.
- Тег **<f:phaseListener>** регистрирует экземпляр класса, реализующего интерфейс `javax.faces.event.PhaseListener`. Атрибуты тега — `type` (полное имя Java-класса) и `binding` (EL-выражение, определяющее объект `PhaseListener`).
- Тег **<f:selectItem>** в теле тега компонента представляет опцию выбора (HTML-тег `<option>`). Атрибуты тега: `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans; `id` — идентификатор тега; `itemDescription` — описание опции; `itemDisabled` — если `true`, тогда опция выключена, по умолчанию `false`; `itemLabel` — отображаемый ярлык опции; `itemValue` — значение опции; `value` — EL-выражение, указывающее экземпляр `javax.faces.model.SelectItem` данной опции.
- Тег **<f:selectItems>** в теле тега компонента представляет список опций выбора (HTML-теги `<option>`). Атрибуты тега: `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans; `id` — идентификатор тега; `value` — EL-выражение, указывающее список или массив экземпляров `javax.faces.model.SelectItem`.
- Тег **<f:setPropertyActionListener>** используется для передачи значения компоненту JavaBeans. Атрибуты тега — `value` (EL-выражение, определяющее передаваемое значение) и `target` (EL-выражение, определяющее свойство компонента JavaBeans, которому передается значение).
- Тег **<f:subview>** является контейнером для тегов вложенного JSF-представления. Атрибуты тега: `id` — идентификатор тега; `binding` — EL-выражение, связывающее тег со свойством компонента JavaBeans; `rendered` — EL-выражение, определяющее, отображается вложенная страница или нет (`true`/`false`).
- Тег **<f:validateDoubleRange>** в теле тега компонента регистрирует объект `DoubleRangeValidator`, проверяющий вводимое клиентом значение с плавающей запятой. Атрибуты тега: `binding` — EL-выражение, определяющее объект `javax.faces.validator.DoubleRangeValidator`; `maximum` — максимальное значение, разрешенное для этого компонента; `minimum` — минимальное значение, разрешенное для этого компонента.
- Тег **<f:validateLength>** в теле тега компонента регистрирует объект `LengthValidator`, проверяющий длину вводимой клиентом строки. Атрибуты тега: `binding` — EL-выражение, определяющее объект `javax.faces.validator.LengthValidator`; `maximum` — максимальное значение, разрешенное для этого компонента; `minimum` — минимальное значение, разрешенное для этого компонента.

- Тег **<f:validateLongRange>** в теле тега компонента регистрирует объект `LongRangeValidator`, проверяющий вводимое клиентом значение типа `long integer`. Атрибуты тега: `binding` — EL-выражение, определяющее объект `javax.faces.validator.LongRangeValidator`; `maximum` — максимальное значение, разрешенное для этого компонента; `minimum` — минимальное значение, разрешенное для этого компонента.
- Тег **<f:validator>** в теле тега компонента регистрирует экземпляр класса, реализующего интерфейс `javax.faces.validator.Validator` и осуществляющего проверку данных, вводимых пользователем. Атрибуты тега: `validatorId` — идентификатор создаваемого и регистрируемого экземпляра `Validator`, соответствует значению элемента `validator-id` JSF-конфигурационного файла; `binding` — EL-выражение, определяющее объект `Validator`.
- Тег **<f:valueChangeListener>** в теле тега компонента регистрирует экземпляр класса, реализующего интерфейс `javax.faces.event.ValueChangeListener` — слушателя изменения значения компонента. Атрибуты тега — `type` (полное имя Java-класса) и `binding` (EL-выражение, определяющее объект `ValueChangeListener`).
- Тег **<f:verbatim>** создает и регистрирует компонент `UIOutput`, отображающий содержимое тела тега. Атрибуты тега: `escape` — если `true`, тогда символы `<`, `>`, `&`, `'`, `"` интерпретируются как символы, а не как элементы синтаксиса HTML или XML, по умолчанию `false`; `rendered` — если `true` (по умолчанию), тогда данный компонент отображается.
- Тег **<f:view>** является контейнером для всех остальных тегов JSF-представления. Атрибуты тега:
  - `locale` — локализация JSF-представления (см. `java.util.Locale`);
  - `renderKitId` — идентификатор объекта `javax.faces.render.RenderKit`, обеспечивающего отображение данной страницы;
  - `beforePhase` — EL-выражение, определяющее метод, обрабатывающий событие `javax.faces.event.PhaseEvent` (вызывается перед началом каждой фазы жизненного цикла обработки запроса);
  - `afterPhase` — EL-выражение, определяющее метод, обрабатывающий событие `javax.faces.event.PhaseEvent` (вызывается после окончания каждой фазы жизненного цикла обработки запроса).
- Тег **<f:ajax>** (JSR 314) связывает AJAX-запрос с компонентом или компонентами, способными генерировать события (регистрирует экземпляр `AjaxBehavior`).

  Атрибуты тега:
  - `disabled` — если `false` (по умолчанию), тогда `AjaxBehavior` исполняется;
  - `event` — событие, запускающее AJAX-запрос;
  - `execute` — перечень идентификаторов компонентов, обрабатываемых сервером (символ разделения в списке — пробел);
  - `immediate` — пропуск фазы проверки JSF-цикла (`true`/`false`);

- `listener` — метод слушателя, исполняемый при обработке AJAX-запроса сервером;
- `onerror` — имя JavaScript-функции, обрабатывающей ошибки обработки AJAX-запроса;
- `onevent` — имя JavaScript-функции, обрабатывающей события, связанные с AJAX-запросом (`begin` — запрос послан, `complete` — запрос обработан, но документ не изменен, `success` — документ изменен);
- `render` — перечень идентификаторов компонентов, отображаемых клиенту (символ разделения в списке — пробел).

Ключевые слова, используемые для атрибутов `execute` и `render` — `@all` (все компоненты), `@none` (нет компонентов), `@this` (компонент, запускающий AJAX-запрос), `@form` (компоненты формы, в которой содержится AJAX-запрос).

- Тег **`<f:event>`** (JSR 314) в теле тега компонента регистрирует экземпляр `javax.faces.event.ComponentSystemEventListener`.

  Атрибуты тега:

  - `listener` — EL-выражение, определяющее вызываемый метод компонента JavaBeans;
  - `type` — тип обрабатываемого события (`preRenderComponent`, `PostAddToView`, `preValidate`, `postValidate`, `beforeRender`, `afterAddToParent`, `afterAddToView`).
- Тег **`<f:metadata>`** (JSR 314) содержит метаданные для JSF-представления, является контейнером для тегов `<f:viewParam>`.
- Тег **`<f:viewParam>`** (JSR 314) связывает значения параметров клиентского запроса со свойствами компонентов JavaBeans. Атрибуты тега — `name` (имя параметра) и `value` (EL-выражение, определяющее свойство компонента JavaBeans, в которое заносится значение параметра).
- Тег **`<f:validateBean>`** (JSR 314, JSR 303) в теле тега компонента регистрирует экземпляр `javax.faces.validator.BeanValidator`, который осуществляет проверку значений JSF-компонентов, связанных со свойствами компонентов JavaBeans. Атрибуты тега: `validationGroups` — список проверочных групп (validation group), где проверочная группа представлена полным именем интерфейса, определяющего набор ограничений для значения компонента; `disabled` — деактивация регистрируемого экземпляра (`true`/`false`); `binding` — EL-выражение, определяющее экземпляр `BeanValidator`; `for` — проверяемое значение компонента.
- Тег **`<f:validateRequired>`** (JSR 314) в теле тега компонента регистрирует экземпляр `javax.faces.validator.RequiredValidator`, осуществляющий проверку наличия значения компонента. Атрибуты тега: `disabled` — деактивация регистрируемого экземпляра (`true`/`false`); `binding` — EL-выражение, определяющее экземпляр `RequiredValidator`; `for` — проверяемое значение компонента.
- Тег **`<f:validateRegexp>`** (JSR 314) в теле тега компонента регистрирует экземпляр `javax.faces.validator.RegexValidator`, который осуществляет проверку значения

компонента относительно регулярного выражения `java.util.regex`. Атрибуты тега: `disabled` — деактивация регистрируемого экземпляра (`true`/`false`); `binding` — EL-выражение, определяющее экземпляр `RequiredValidator`; `for` — проверяемое значение компонента; `pattern` — регулярное выражение.

Библиотека тегов Composite

Библиотека тегов Composite (JSR 314) имеет префикс `composite` и пространство имен URI **http://java.sun.com/jsf/composite**.

Стандартный синтаксис:

```
<%@ taglib prefix="composite" uri="http://java.sun.com/jsf/composite" %>
```

XML-синтаксис:

```
<anyxmlelement xmlns:composite="http://java.sun.com/jsf/composite" />
```

Библиотека тегов Composite позволяет создать пользовательский тег в виде отдельного файла, содержащего XHTML-контент и другие JSF-компоненты, при этом xhtml-файл пользовательского тега помещается в каталог ресурсов приложения resources*[name]* (*[name]* — имя библиотеки пользовательских тегов) и его использование объявляется с помощью синтаксиса:

```
<anyxmlelement xmlns:[name]="http://java.sun.com/jsf/composite/[name]" />
```

- Тег **`<composite:interface>`** определяет интерфейс пользовательского компонента.

  Атрибуты тега:

  - `componentType` — имя контейнера компонента;
  - `displayName` — отображаемое имя компонента;
  - `expert` — если `true`, тогда компонент предназначен для продвинутых пользователей;
  - `name` — имя компонента;
  - `preferred` — если `true`, тогда компонент имеет статус `"preferred"`;
  - `shortDescription` — описание компонента.

- Тег **`<composite-implementation>`** определяет реализацию интерфейса пользовательского компонента.

- Тег **`<composite:attribute>`** в теле тега `<composite:interface>` объявляет параметры интерфейса пользовательского компонента.

  Атрибуты тега:

  - `default` — значение атрибута по умолчанию;
  - `displayName` — отображаемое имя атрибута;
  - `expert` — если `true`, тогда компонент предназначен для продвинутых пользователей;

- method-signature — сигнатура метода интерфейса;
- name — имя параметра;
- preferred — если true, тогда компонент имеет статус "preferred";
- required — если true, тогда атрибут обязан иметь значение;
- shortDescription — короткое описание;
- targets — список идентификаторов компонентов, с которыми связан метод интерфейса;
- type — Java-тип атрибута. Если указан атрибут type, атрибут method-signature игнорируется.

- Тег **<composite:facet>** в теле тега <composite:interface> объявляет использование подчиненного компонента, который связан с компонентом страницы с помощью атрибута name, указывающим роль, которую связанный компонент выполняет.

  Атрибуты тега:

  - displayName — отображаемое имя;
  - expert — если true, тогда facet предназначен для продвинутых пользователей;
  - name — роль, которую связанный компонент выполняет;
  - preferred — если true, тогда facet имеет статус "preferred";
  - required — если true, тогда использование данного facet обязательно;
  - shortDescription — короткое описание.

- Тег **<composite:renderFacet>** в теле тега <composite:implementation> указывает использование подчиненного компонента, который связан с компонентом страницы с помощью атрибута name, определяющим роль, которую связанный компонент выполняет. Атрибуты тега — name и required.

- Тег **<composite:insertChildren>** в теле тега <composite:implementation> указывает место, где в странице, использующей пользовательский тег, вставляются подчиненные компоненты.

- Тег **<composite:valueHolder>** в теле тега <composite:interface> объявляет, что пользовательский компонент реализует интерфейс javax.faces.component.ValueHolder и что к пользовательскому компоненту в странице с пользовательским тегом могут присоединяться объекты типа Converter. Атрибуты тега — name (значение соответствует значению атрибута for присоединенного объекта) и targets (перечень идентификаторов компонентов, используемых в теле тега <composite:implementation>, к которым присоединяется объект).

- Тег **<composite:editableValueHolder>** в теле тега <composite:interface> объявляет, что пользовательский компонент реализует интерфейс javax.faces.component.EditableValueHolder и что к пользовательскому компоненту в странице с пользовательским тегом могут присоединяться объекты типа Converter и Validator. Атрибуты тега — name (значение соответствует значению атрибута for присоединенно-

го объекта) и `targets` (перечень идентификаторов компонентов, используемых в теле тега `<composite:implementation>`, к которым присоединяется объект).

- Тег **`<composite:actionSource>`** в теле тега `<composite:interface>` объявляет, что пользовательский компонент реализует интерфейс `javax.faces.component.ActionSource2` и что к пользовательскому компоненту в странице с пользовательским тегом могут присоединяться объекты типа `ActionListener`. Атрибуты тега — `name` (значение соответствует значению атрибута `for` присоединенного объекта) и `targets` (перечень идентификаторов компонентов, используемых в теле тега `<composite:implementation>`, к которым присоединяется объект).
- Тег **`<composite:extension>`** в теле тега `<composite:interface>` содержит XML-контент, не определенный спецификацией (расширяет спецификацию).
- Тег **`<composite:insertFacet>`** в теле тега `<composite:implementation>` указывает место, где в странице, использующей пользовательский тег, вставляется именованный `facet`. Атрибуты тега — `name` (имя вставляемого `facet`) и `required` (требование обязательного наличия вставляемого `facet`, `true`/`false`).

Библиотека тегов Facelets UI

Библиотека тегов Facelets UI (JSR 314) имеет префикс `ui` и пространство имен URI **http://java.sun.com/jsf/facelets**.

Стандартный синтаксис:

```
<%@ taglib prefix="ui" uri="http://java.sun.com/jsf/facelets" %>
```

XML-синтаксис:

```
<anyxmlelement xmlns:ui="http://java.sun.com/jsf/facelets" />
```

- Тег **`<ui:insert>`** вставляет содержимое в шаблон страницы. Страницы приложения могут содержать общие части, такие как заголовок, подстрочник и т. д. Для того чтобы каждый раз их не повторять, можно создать шаблон страницы в виде отдельного файла [name_template].xhtml, содержащего теги `<ui:insert name=""/>`, где атрибут `name` указывает идентификатор вставляемого контента, определяемый в странице, использующей шаблон.
- Тег **`<ui:define>`** в странице, использующей шаблон, в теле тега определяет вставляемый контент. Атрибут тега `name` указывает идентификатор тега `<ui:insert>`.
- Тег **`<ui:composition>`** в странице, использующей шаблон, с помощью атрибута `template` указывает URI-адрес шаблона. Кроме того, тег `<ui:composition>` является контейнером для отображаемого клиенту содержимого. Все, что находится вне тега, клиенту не отображается.
- Тег **`<ui:component>`** является контейнером для отображаемого клиенту содержимого, при этом создается экземпляр `javax.faces.component.UIComponent`, который добавляется в дерево компонентов. Атрибуты тега — `id` (идентификатор компонента) и `binding` (EL-выражение, связывающее тег со свойством компонента JavaBeans).

- Тег **<ui:fragment>** аналогичен тегу <ui:component> за исключением того, что содержимое вне тега не игнорируется. Атрибуты тега — id (идентификатор компонента) и binding (EL-выражение, связывающее тег со свойством компонента JavaBeans).
- Тег **<ui:decorate>** аналогичен тегу <ui:composition> за исключением того, что содержимое вне тега не игнорируется. Атрибут тега template указывает URI-адрес шаблона.
- Тег **<ui:debug>** для отладки используется в шаблонах, создает экземпляр UIComponent, добавляемый в дерево компонентов, при этом с помощью "горячих" клавиш можно открыть окно, отображающее отладочную информацию — текущее состояние дерева компонентов и переменных приложения. Атрибуты тега: hotkey (определяет третью клавишу к <Ctrl> и <Shift>, открывающие окно отладки, по умолчанию <Ctrl>+<Shift>+<D>) и rendered (если false, тогда тег деактивируется, и отладочное окно не открывается).
- Тег **<ui:include>** позволяет включать контент в XHTML-страницы. Атрибут тега src указывает относительный адрес включаемой XHTML-страницы.
- Тег **<ui:param>** передает значения параметров включаемой странице (<ui:include>) или шаблону в теле тегов <ui:include>, <ui:composition>, <ui:decorate>. Атрибуты тега — name (имя параметра) и value (значение параметра).
- Тег **<ui:repeat>** альтернатива тегам <h:dataTable> или <c:forEach>. Атрибуты тега: offset — смещение относительно начала коллекции при итерации; size — размер коллекции; step — шаг итерации; value — имя коллекции; var — имя переменной итерации; varStatus — переменная статуса итерации, обеспечивающая информацией об итерации.
- Тег **<ui:remove>** в теле тега содержит контент, который должен быть удален при компиляции страницы.

  Атрибут jsfc позволяет использовать теги Facelets в стандартных HTML-тегах.

Конфигурационный файл faces-config.xml

По умолчанию JSF-приложение использует конфигурационный файл faces-config.xml, расположенный в каталоге WEB-INF приложения или в папке META-INF JAR-файла каталога \WEB-INF\lib\ Web-приложения. JSF-приложение может иметь также дополнительные конфигурационные файлы, определенные параметром javax.faces.application.CONFIG_FILES файла web.xml. Данный параметр содержит имена конфигурационных файлов, разделенные запятой.

Элементы конфигурационного файла в соответствии с XML-схемой **http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd**:

- faces-config — корневой элемент. Если атрибут metadata-complete (используется только в файле WEB-INF\faces-config.xml) равен true, тогда аннотации, опреде-

ляющие конфигурационную информацию в класс-файлах приложения, игнорируются;

- `application` указывает параметры приложения. Вложенные элементы:
  - `action-listener` — полное имя класса, реализующего интерфейс `javax.faces.event.ActionListener`;
  - `application-extension` — элементы, расширяющие спецификацию;
  - `default-render-kit-id` — экземпляр `javax.faces.render.RenderKit`, используемый вместо стандартного экземпляра по умолчанию;
  - `default-validators` — регистрирует используемый по умолчанию набор экземпляров `Validator`. Вложенный элемент — `validator-id` (идентификатор регистрируемого экземпляра);
  - `el-resolver` — полное имя класса реализации `javax.el.ELResolver`;
  - `locale-config` — объявляет поддерживаемые локализации для приложения. Содержит элементы `default-locale` (локализация по умолчанию, например, `ja_JP_SJIS`) и `supported-locale` (поддерживаемая локализация);
  - `message-bundle` — базовое имя ресурсных файлов локализации сообщений;
  - `navigation-handler` — полное имя класса, расширяющего класс `javax.faces.application.NavigationHandler`;
  - `partial-traversal` — полное имя класса реализации `PartialTraversal`;
  - `property-resolver` — полное имя класса реализации `javax.faces.el.PropertyResolver`;
  - `resource-bundle` содержит элементы: `base-name` — полное имя экземпляра `java.util.ResourceBundle`; `var` — имя, используемое методом `Application.getResourceBundle()`;
  - `resource-handler` — полное имя класса реализации `javax.faces.application.ResourceHandler`;
  - `state-manager` — полное имя класса реализации `javax.faces.application.StateManager`, используемого вместо класса по умолчанию;
  - `system-event-listener` содержит элементы: `system-event-listener-class` — полное имя класса реализации интерфейса `javax.faces.event.SystemEventListener`; `system-event-class` — полное имя класса, расширяющего класс `javax.faces.event.SystemEvent`; `source-class` — полное имя класса — источника событий;
  - `variable-resolver` — полное имя класса реализации `javax.faces.el.VariableResolver`;
  - `view-handler` — полное имя класса, расширяющего класс `javax.faces.application.ViewHandler` и используемого вместо класса по умолчанию;
- `name` — имя конфигурационного файла;

- `ordering` — относительный порядок загрузки конфигурационных файлов приложения. Вложенные элементы:
  - `absolute-ordering` (используется только в WEB-INF\faces-config.xml) — определяет абсолютный порядок загрузки конфигурационных файлов приложения. Если содержит элементы `name`, то данный конфигурационный файл загружается после конфигурационных файлов в порядке, определенном расположением элементов `name`. Если содержит элемент `others`, то данный конфигурационный файл загружается после всех остальных конфигурационных файлов;
  - `after` — если содержит элемент `name`, то данный конфигурационный файл загружается после конфигурационного файла, определенного элементом `name`. Если содержит элемент `others`, то данный конфигурационный файл загружается после всех остальных конфигурационных файлов;
  - `before` — если содержит элемент `name`, то данный конфигурационный файл загружается перед конфигурационным файлом, определенным элементом `name`. Если содержит элемент `others`, то данный конфигурационный файл загружается перед всеми остальными конфигурационными файлами;
- `factory`. Вложенные элементы:
  - `application-factory` — полное имя класса реализации `javax.faces.application.ApplicationFactory`;
  - `exception-handler-factory` — полное имя класса реализации `javax.faces.context.ExceptionHandlerFactory`;
  - `external-context-factory` — полное имя класса реализации `javax.faces.context.ExternalContextFactory`;
  - `faces-context-factory` — полное имя класса реализации `javax.faces.context.FacesContextFactory`;
  - `factory-extension` — элементы, расширяющие спецификацию;
  - `lifecycle-factory` — полное имя класса реализации `javax.faces.lifecycle.LifecycleFactory`;
  - `partial-view-context-factory` — полное имя класса реализации `javax.faces.context.PartialViewContextFactory`;
  - `render-kit-factory` — полное имя класса реализации `javax.faces.render.RenderKitFactory`;
  - `tag-handler-delegate-factory` — полное имя класса реализации `javax.faces.view.facelets.TagHandlerDelegateFactory`;
  - `view-declaration-language-factory` — полное имя класса реализации `javax.faces.view.ViewDeclarationLanguageFactory`;
  - `visit-context-factory` — полное имя класса реализации `javax.faces.component.visit.VisitContextFactory`;

- `component` регистрирует класс реализации `javax.faces.component.UIComponent`. Вложенные элементы:
  - `attribute` — определяет параметры компонента с помощью вложенных элементов: `attribute-name` — имя параметра; `attribute-class` — Java-тип значения параметра; `default-value` — значение параметра по умолчанию; `suggested-value` — предлагаемое значение параметра, используется инструментами разработки; `attribute-extension` — содержит элементы, расширяющие спецификацию;
  - `component-class` — полное имя класса компонента;
  - `component-extension` — элементы, расширяющие спецификацию;
  - `component-type` — имя, под которым компонент регистрируется;
  - `facet` — содержит элементы `facet-name` (если компонент является подчиненным, указывает роль, которую данный компонент играет) и `facet-extension` (содержит элементы, расширяющие спецификацию);
  - `property` — определяет JavaBeans-свойства компонента с помощью вложенных элементов: `property-name` — имя свойства; `property-class` — Java-тип значения свойства; `default-value` — значение свойства по умолчанию; `suggested-value` — предлагаемое значение свойства, используется инструментами разработки; `property-extension` — элементы, расширяющие спецификацию;
- `converter` регистрирует класс реализации `javax.faces.convert.Converter`. Вложенные элементы:
  - `attribute` (используется инструментами разработки) — определяет параметры связанного компонента с помощью вложенных элементов: `attribute-name` — имя параметра; `attribute-class` — Java-тип значения параметра; `default-value` — значение параметра по умолчанию; `suggested-value` — предлагаемое значение параметра; `attribute-extension` — элементы, расширяющие спецификацию;
  - `converter-class` — полное имя регистрируемого класса;
  - `converter-extension` — элементы, расширяющие спецификацию;
  - `converter-for-class` — полное имя класса, для которого данный класс регистрируется;
  - `converter-id` — идентификатор класса;
  - `property` (используется инструментами разработки) — определяет JavaBeans-свойства регистрируемого класса с помощью вложенных элементов: `property-name` — имя свойства; `property-class` — Java-тип значения свойства; `default-value` — значение свойства по умолчанию; `suggested-value` — предлагаемое значение свойства; `property-extension` — содержит элементы, расширяющие спецификацию;
- `managed-bean` представляет JavaBeans-компоненты приложения, свойства и методы которых связываются с помощью EL-выражений с JSF-компонентами. Управляе-

мые JavaBeans-компоненты JSF-приложения, как правило, описываются в отдельном конфигурационном файле \WEB-INF\managed-beans.xml. Если атрибут `eager` элемента равен `true`, тогда экземпляр компонента с областью видимости `application` создается и сохраняется при запуске приложения. Вложенные элементы:

- `list-entries` — описывает свойство компонента типа `java.util.List` или `array` с помощью элементов `value-class` (Java-тип значения) и `null-value | value` (значение свойства);
- `managed-bean-class` — полное имя класса компонента;
- `managed-bean-extension` — элементы, расширяющие спецификацию;
- `managed-bean-name` — имя компонента;
- `managed-bean-scope` — область видимости создаваемого экземпляра компонента (`request|session|application|none`);
- `managed-property` — свойство компонента, определяемое при создании экземпляра. Вложенные элементы: `property-name` — имя свойства; `property-class` — Java-тип значения свойства; `map-entries` — описывает свойство типа `java.util.Map` с помощью элементов `key-class` (Java-тип ключа), `value-class` (Java-тип значения), `map-entry` (элементы `key` и `null-value | value`); `null-value` — значение свойства устанавливается нулевым; `value` — значение свойства; `list-entries` — описывает свойство типа `java.util.List` или `array` с помощью элементов `value-class` (Java-тип значения) и `null-value | value` (значение свойства);
- `map-entries` — описывает свойство компонента типа `java.util.Map` с помощью элементов `key-class` (Java-тип ключа), `value-class` (Java-тип значения), `map-entry` (элементы `key` и `null-value | value`); `null-value` — значение свойства устанавливается нулевым; `value` — значение свойства;

❒ `navigation-rule` определяет правило навигации. Вложенные элементы:

- `from-view-id` — идентификатор JSF-представления (URI-адрес странички), для которого правило применяется. Если элемент отсутствует, тогда правило применимо ко всем представлениям;
- `navigation-case` содержит элементы:
  - `from-outcome` — строка, возвращаемая действием, связанным с компонентом;
  - `from-action` — EL-выражение, определяющее метод-действие JavaBeans-компонента — обработчик событий, возвращающий строку действия;
  - `if` — EL-выражение, определяющее условие, при котором навигация выполняется (`true`/`false`);
  - `to-view-id` — идентификатор JSF-представления (URI-адрес странички), к которому осуществляется переход в результате выполнения условий `from-outcome` или `if`;

    - redirect — контейнер JSP завершает текущий запрос и отправляет клиенту HTTP-ответ, переадресующий его на другую страницу;
  - navigation-rule-extension — элементы, расширяющие спецификацию;
- referenced-bean — указывает ссылку на JavaBeans-компонент для инструментов разработки с помощью элементов referenced-bean-name (имя компонента) и referenced-bean-class (полное имя класса компонента);
- render-kit регистрирует класс реализации javax.faces.render.RenderKit с помощью элементов:
  - render-kit-id — идентификатор;
  - render-kit-class — полное имя класса;
  - renderer — регистрирует класс реализации javax.faces.render.Renderer, отвечающий за визуализацию JSF-компонентов. Каждый объект Renderer отвечает за визуализацию одного или нескольких типов JSF-компонентов, например javax.faces.Button, javax.faces.Text и т. д., при этом каждый JSF-компонент относится к семейству компонентов, например javax.faces.Command, javax.faces.Input и т. д. Семейства объектов Renderer упаковываются в объект RenderKit, отвечающий за визуализацию JSF-представления. Элемент renderer содержит вложенные элементы:
    - component-family — семейство визуализируемых компонентов;
    - renderer-type — тип визуализируемого компонента;
    - renderer-class — полное имя класса реализации Renderer;
    - facet — определяет имя facet, связанного с объектом Renderer с помощью элемента facet-name, вложенный элемент facet-extension содержит элементы, расширяющие спецификацию;
    - attribute — определяет параметры с помощью вложенных элементов: attribute-name (имя параметра), attribute-class (Java-тип значения параметра), default-value (значение параметра по умолчанию), suggested-value (предлагаемое значение параметра, используется инструментами разработки), attribute-extension (содержит элементы, расширяющие спецификацию);
    - renderer-extension — элементы, расширяющие спецификацию;
  - client-behavior-renderer — регистрирует класс реализации javax.faces.render.ClientBehaviorRenderer, отвечающий за воспроизведение скрипта на стороне клиента. Вложенные элементы: client-behavior-renderer-type — идентификатор; client-behavior-renderer-class — полное имя класса;
  - render-kit-extension — элементы, расширяющие спецификацию;
- lifecycle содержит элементы phase-listener (полное имя класса реализации javax.faces.event.PhaseListener) и lifecycle-extension (содержит элементы, расширяющие спецификацию);

- `validator` регистрирует класс реализации `javax.faces.validator.Validator` с помощью вложенных элементов:
  - `validator-id` — идентификатор регистрируемого класса;
  - `validator-class` — полное имя регистрируемого класса;
  - `attribute` (используется инструментами разработки) — определяет параметры связанного компонента с помощью вложенных элементов: `attribute-name` (имя параметра), `attribute-class` (Java-тип значения параметра), `default-value` (значение параметра по умолчанию), `suggested-value` (предлагаемое значение параметра), `attribute-extension` (содержит элементы, расширяющие спецификацию);
  - `property` (используется инструментами разработки) — определяет JavaBeans-свойства регистрируемого класса с помощью вложенных элементов: `property-name` (имя свойства), `property-class` (Java-тип значения свойства), `default-value` (значение свойства по умолчанию), `suggested-value` (предлагаемое значение свойства), `property-extension` (содержит элементы, расширяющие спецификацию);
  - `validator-extension` — элементы, расширяющие спецификацию;
- `behavior` регистрирует класс реализации `javax.faces.component.behavior.Behavior` с помощью вложенных элементов:
  - `behavior-id` — идентификатор;
  - `behavior-class` — полное имя класса;
  - `attribute` (используется инструментами разработки) — определяет параметры связанного компонента с помощью вложенных элементов: `attribute-name` (имя параметра), `attribute-class` (Java-тип значения параметра), `default-value` (значение параметра по умолчанию), `suggested-value` (предлагаемое значение параметра), `attribute-extension` (содержит элементы, расширяющие спецификацию);
  - `property` (используется инструментами разработки) — определяет JavaBeans-свойства регистрируемого класса с помощью вложенных элементов: `property-name` (имя свойства), `property-class` (Java-тип значения свойства), `default-value` (значение свойства по умолчанию), `suggested-value` (предлагаемое значение свойства), `property-extension` (содержит элементы, расширяющие спецификацию);
  - `behavior-extension` — элементы, расширяющие спецификацию;
- `faces-config-extension` содержит элементы, расширяющие спецификацию.

ПРИЛОЖЕНИЕ 13

Описание компакт-диска

Папка	Проект	Описание
Часть_I\Глава_1	StarWar	Апплет, представляющий простую игру "Звездные войны"
Часть_I\Глава_2	AppletTalk	Апплет, представляющий программу-собеседника
Часть_I\Глава_3	SearchInternet	Настольное приложение, позволяющее пользователю осуществлять поиск в Интернете
	SearchInternetWS	Настольное приложение, позволяющее пользователю осуществлять поиск в Интернете, созданное с использованием технологии Java Web Start
Часть_I\Глава_4	JavadocServiceLoader	Пример расширяемого приложения, созданного с использованием библиотеки ServiceLoader API платформы Java SE 6
	JavadocLookupSimple	Пример статически расширяемого приложения, созданного с использованием библиотеки Lookup платформы NetBeans
	JavadocLookupListener	Пример динамически расширяемого приложения, созданного с использованием библиотеки Lookup платформы NetBeans
	JavadocNBMLayer	Пример модульного приложения на базе платформы NetBeans
	JavadocNBM	Пример модуля NetBeans
	JavadocOSGi	Пример OSGi-приложения
Часть_II\Глава_6	JBookReader	MIDP-приложение, позволяющее пользователю читать электронные книги, созданное с использованием высокоуровневой библиотеки API пакета `javax.microedition.lcdui` MIDP-профиля
	JBookReaderDraw	MIDP-приложение, позволяющее пользователю читать электронные книги, созданное с использованием низкоуровневой библиотеки API пакета `javax.microedition.lcdui` MIDP-профиля
Часть_III\Глава_9	WebApplicationSaleStore	Пример приложения Java EE уровня предприятия

Предметный указатель

A

Abstract Window Toolkit (AWT), графическая библиотека 17
AJAX 333, 368
Apache Ant 9
Apache Struts 362

B

Bundles 7

C

Connected Device Configuration (CDC) 220, 241
Connected Limited Device Configuration (CLDC) 220, 222

D

Device fragmentation 246
Doclet 128

E

Echo 364
Eclipse 7
Enterprise JavaBeans 345
Expression Language 464
Extension point 7

F

Foundation Profile 242

G

Google Web Toolkit 368

H

Hibernate 361

I

Inter-MIDlet Communication (IMC) 239

J

jarsigner, инструмент 68, 120
java:
◊ опции:
  ▫ -client 3
  ▫ -Djava.security.manager 67
  ▫ -server 3
Java 3D, графическая библиотека 20
Java Cache Viewer 114
Java Community Process (JCP) 3
Java Data Objects (JDO) 362
Java Data Objects Query Language (JDOQL) 362
Java Message Service (JMS) 348
Java Platform, Enterprise Edition (Java EE) 5, 273
◊ система безопасности 279
Java Platform, Micro Edition (Java ME) 5, 219
Java Platform, Standard Edition (Java SE) 4, 65
◊ for Embedded 4
Java Plug-in 65
Java Runtime Environment (JRE) 2
Java Servlet 330
Java Virtual Machine (JVM) 2
Java Web Start (JWS) 114
java, программа 2
JavaBeans 79
javac, компилятор 2
javacpl 114
javadoc 127
◊ опции:
  ▫ -doclet 128
  ▫ -taglet 128
JavaHelp 436
JavaMail 360
JavaServer Faces (JSF) 332
JavaServer Pages (JSP) 331
JavaServer Pages Standard Tag Library (JSTL) 465
javaws 114, 118
Jini, библиотека 149
jspc, утилита 332
JUnit, библиотека 10, 76, 134
Just-in-time compilation (JIT), компилятор 3

K

keytool, инструмент 68
Knopflerfish OSGi desktop 196

L

Light Weight User Interface Toolkit (LWUIT), графическая библиотека 244
Lookup, библиотека 149

M

MIDlets 220, 225
Mobile Information Device Profile (MIDP) 220, 224
Mobility, платформа 11
MVC 39

N

NetBeans 7
◊ платформа 8
◊ разработка мобильных приложений 246
◊ среда разработки 9

O

OSGi 150

P

Personal Basis Profile 243
Personal Profile 244
policytool 65
Portlet 366
ProGuard 247

R

Reference Implementation (RI) 3
RESTful (JAX-RS) 333

S

Security Manager 65
Spring Framework 364
Standard Widget Toolkit (SWT), библиотека 9, 19
Sun Java Real-Time System (Java RTS) 5
Swing Application Framework 69
Swing, графическая библиотека 8, 9, 19, 37
Synth Look and Feel, библиотека 122

T

Technology Compatibility Kit (TCK) 3

V

Visual Mobile Designer (VMD) 247

W

Web-модуль 280
Web-сервис 333
wscompile, инструмент 336

X

Xenoage JEStart 114
XML-Based RPC (JAX-RPC) 335
XML-Based Web Services (JAX-WS) 340

А

Апплет 15

Б

Байт-код 2
Библиотека 4

В

Виртуальная машина JVM 2

Д

Доклет 128

И

Интернационализация 74

К

Контейнер 273

Л

Локализация 74

М

Манифест 7
Мидлет 220, 225

О

Объект:
◊ сессионный (Session) 346
◊ сущность (Entity) 347, 348
◊ управляемый сообщениями (Message-Driven) 347

П

Песочница 16, 64
Платформа 4
◊ Java 2
Портал 367
Портлет 367
Приложение:
◊ мобильное 246
◊ настольное 16
◊ расширяемое 143
◊ чтение TXT-файлов с помощью мобильного устройства 248
Программа:
◊ апплет:
▫ Звездные войны 20
▫ Собеседник 39
◊ настольное приложение:
▫ поиск в Интернете 71
◊ сервис:
▫ ServiceLoader 153

Р

Расширение 7
Расширяемость приложения 143
Роль безопасности 280

С

Сервер:
◊ Java EE 275
◊ приложений Java EE 275
Сервлет 330
Справка:
◊ интеграция в приложение 440
◊ контекстная 440
Справочная система JavaHelp 436

Т

Теглет 128
Тестирование 134
Точка расширения 7

Ф

Файл:
◊ faces-config.xml 542
◊ java-wsdl-mapping.xml 338
◊ layer.xml 8
◊ WSDL 336
◊ дескриптора библиотеки тегов (TLD) 488
◊ расширение:
▫ class 2
▫ ear 277
▫ hs 438
▫ jar 2
▫ java 2
▫ jdo 362
▫ jhm 437
▫ jnlp 114
▫ jsp 332
▫ jspf 332
▫ jspx 332
▫ nbm 192
▫ war 280
Фрагментация устройств 246